SOFTWARE TESTING, VERIFICATION AND RELIABILITY
*Softw. Test. Verif. Reliab.* 0000; **00**:1–36
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/stvr

# On Subsumption Relationships in Data Flow Testing

Marcos Lordello Chaim[1]* Kesina Baral[2], Jeff Offutt[2], Mario Concilio Neto[1], and Roberto Paulo Andrioli de Araujo[1]

[1]*University of Sao Paulo, Sao Paulo, SP, Brazil*
[2]*George Mason University, Fairfax, VA, USA*

SUMMARY

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, there are roadblocks to its application. Finding data flow subsumptions correctly and efficiently has been an elusive goal, the savings provided by data flow subsumptions and the cost to find them need to be assessed, and the fault detection ability of a reduced set of DU-associations and the advantages of data flow testing over node and edge coverage need to be verified. This paper presents novel solutions to these problems. We present algorithms that correctly find data flow subsumptions and are asymptotically less costly than previous algorithms. We present empirical data that shows that data flow subsumption is effective at reducing the number of DU-associations to be tested and can be found at scale. Furthermore, we found that using reduced DU-associations decreased the fault detection ability by less than 2%, and data flow testing adds testing value beyond node and edge coverage.


Received ...

KEY WORDS: Software testing, Structural testing, Data flow testing, Subsumption relationship, Data flow analysis frameworks, Algorithms, Empirical analysis

## 1. 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 [8,22], graphs [39,49], and the software's input space [16]. *Coverage criteria* provide a systematic way to generate test requirements [3] 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

*Correspondence to: Marcos Lordello Chaim, Rua Arlindo Bettio 1000, Vila Guaraciaba, CEP 03828-000, Sao Paulo, SP, Brazil. E-mail: chaim@usp.br

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*. These are also called *du-associations*, *def-use associations*, *du-pairs*, and *def-pairs*. This paper uses *DU-associations*, or simply DUA.

Studies have shown that data flow testing (DFT) is comparable to that with control flow testing (CFT) [14, 18, 21]. Hemmati showed that DUA coverage finds faults that control flow coverage criteria do not [18]. 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 [43, 47].

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 number of lines of source code (LOC) and number of DUAs, along with other metrics. Consider Math, the eighth from the bottom of the table, which has 54,518 LOC. The all-uses criterion [42], which requires every use to be reached at least once, has 87,603 DUA requirements on Math, which is 60% more than the LOC.

Table I. Program metrics and data flow data

| Program | LOC | **Methods w. DUAs** | Methods executed | DUAs |
|---|---|---|---|---|
| Csv | 602 | 40 | 37 | 929 |
| Cli | 1107 | 60 | 54 | 1291 |
| **Codec** | 1946 | 109 | 92 | 4446 |
| Jsoup | 2046 | 136 | 119 | 1866 |
| JacksonXml (J-Xml) | 3084 | 179 | 124 | 3402 |
| **Compress** | 4974 | 217 | 116 | 6286 |
| Gson | 3840 | 226 | 191 | 3281 |
| Mockito | 7468 | 400 | 366 | 4236 |
| JacksonCore (J-Core) | 10,978 | 563 | 383 | 17,653 |
| JxPath | 14,699 | 800 | 691 | 20,178 |
| Lang | 15,270 | 1157 | 715 | 22,290 |
| Time | 19,672 | 1182 | 1010 | 18,160 |
| Collections | 18,156 | 1311 | 1094 | 16,937 |
| JacksonDatabind (J-DataBind) | 27,274 | 1737 | 1263 | 31,797 |
| **Math** | 54,518 | 2415 | 1999 | 87,603 |
| Chart | 68,346 | 3219 | 2151 | 81,847 |
| Closure | 61,177 | 3696 | 3241 | 78,068 |
| Elki | 47,799 | 5820 | * | 180,675 |
| SystemDS | 60,121 | 8516 | * | 255,382 |
| Weka | 216,781 | 11,315 | 1964 | 337,063 |
| CoreNLP | 101,910 | 13,626 | * | 444,792 |
| **Total** | 531,938 | 56,720 | 15,610 | 1,618,182 |

This paper presents a novel algorithm (Algorithm 1) that implements data flow analysis without having to check each individual DUA requirement. This in turn allows DFT to be cheaper. The general approach is to exploit the subsumption relationship among testing requirements (TR) [23, 35, 46]. A TR $tr_1$ (e.g., a DUA $D_1$) *subsumes* another test requirement $tr_2$ (e.g., a DUA $D_2$) if



DOI: 10.1002/stvr

every test path that satisfies $tr_1$ also satisfies $tr_2$ [35]. 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 [35]. 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 [46] 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.

Thus, we focus on the subsumption relationship to identify the DUAs that are not covered by node and edge coverage or other DUAs to get the most value from data flow testing with the least cost. However, identifying and using such test requirements has several issues:

1. *Data flow subsumption discovery*. How best to discover data flow testing subsumption relationships (DFTS) has been an elusive goal. The current algorithms are costly, being linear [46] in the number of DUAs for DUA-edge and quadratic [23, 35] for DUA-DUA subsumption. This cost hampers its application for industry-size systems. Furthermore, some algorithms [23, 35] can miss paths that would block the subsumption of DUAs, leading to incorrect results.

2. *Effectiveness*. The number of unconstrained DUAs should be substantially smaller than the original set of DUAs to reduce the cost of data flow testing. Several papers that explored reduction in the number of DUAs used small programs, and algorithms that miss paths that can block the subsumption relationship [35] [23]

3. *Scalability*. No prior work assesses the scalability of data flow subsumption algorithms. Therefore, we do not know if the time taken by the algorithm is feasible for industrial use.

4. *Fault detection ability*. The fault detection ability of unconstrained DUAs may be affected by infeasible unconstrained DUAs, and program interruptions such as exceptions.

5. *Yield of data flow testing*. For practitioners to use data flow testing, the faults it reveals must supplement faults revealed by control flow testing.

This paper addresses the issues of subsumption relationship in data flow testing. We address both cost and omissions regarding *data flow subsumption discovery* by presenting a novel and efficient approach to tackle data flow subsumptions. This approach models the problem of finding *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, 26]. Using the local DUA-node subsumption, one can efficiently discover the subsumption of DUAs with respect to nodes, edges, and other DUAs.

We conducted an empirical study to assess the effectiveness and scalability of data flow subsumption. We used 21 programs (listed in Table I) from the Defects4J repository [25] and four extra programs, namely, Elki, SystemDS, Weka, and CoreNLP. The program size ranged from 602 source lines of code (LOC) to more than 200,000 LOC. We used the approach presented in this paper to identify unconstrained DUAs, DUA-edge and DUA-node subsumptions in the programs, with the purpose of evaluating the effectiveness and scalability of data flow subsumptions. Our findings show that data flow subsumption is effective at reducing the number of DUAs to be tested and can be found at scale for programs similar to those developed in the industry.

We also assessed the fault detection ability of unconstrained DUAs and the result of DUA-node and DUA-edge subsumption. We used probabilistic coupling [7] as a proxy for the fault detection ability of DUAs. Our results indicate that the likelihood of missing a fault by using only unconstrained DUAs is less than 2%. Happily, these missed faults can be determined either statically or at run-time. Furthermore, the top fault revealing DUAs were not subsumed by node coverage for around 20% of the bugs and around 7.5% by the edge coverage. Subsumption relationship can




DOI: 10.1002/stvr

highlight which DUAs are most likely to reveal faults and though modest, data flow testing adds value.

Our results suggest that 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 represents a major extension to the paper presented at the 14th IEEE International Conference on Software Testing, Verification, and Validation [5]. The most significant extensions are (1) this paper adds an in-depth analyses of the fault detection ability of unconstrained DUAs and the value added by data flow testing, and (2) this paper includes extensive additional empirical work. The empirical work is reflected in Section 6, and includes several additional research questions and empirical results on fault detection and effectiveness of data flow testing using unconstrained DUAs. This paper also has significant more background, details of the algorithms, and related work. The paper starts with background in data flow testing in Section 2. We then describe data flow subsumptions in Section 3, followed by our solution to solve the local DUA-node subsumption in Section 4. Algorithms to find other data flow subsumptions are described in Section 5, followed by experimental analysis in Section 6. Related work is discussed in Section 7, and conclusions are in Section 8.

## 2. BACKGROUND

Graph based testing criteria use graph abstractions of the software under test to generate tests. A directed 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 [6] 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 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 [32, 38, 42, 48]. In this paper, we focus on the *all-uses* criterion proposed by Rapps and Weyuker [42].




DOI: 10.1002/stvr

```
/* 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 */ }
```

Figure 1. Example program Max

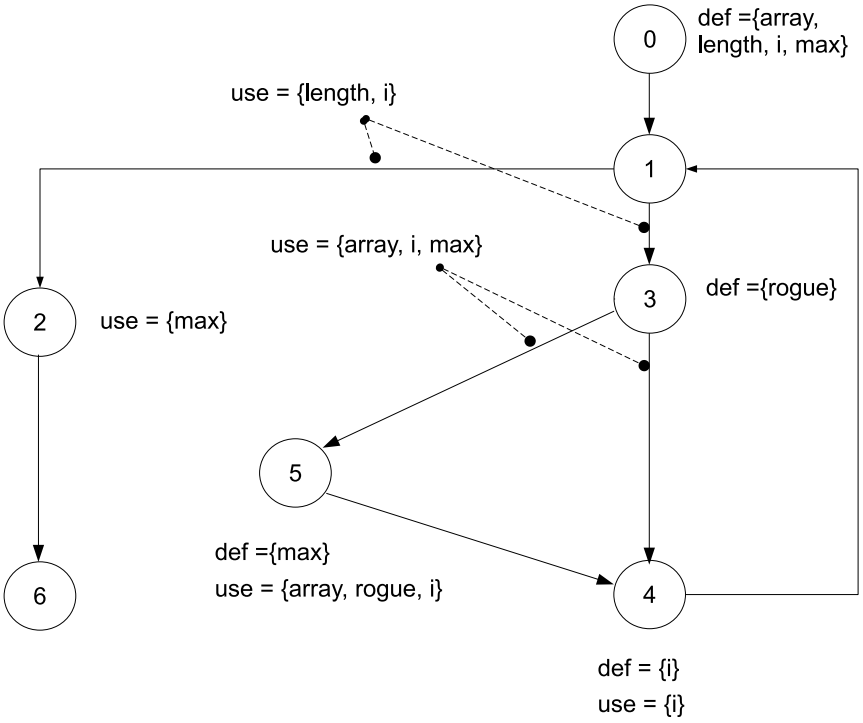


Figure 2. Annotated flow graph for Max

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 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




DOI: 10.1002/stvr

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) |

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 [35].

## 3. 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 $C_2$ [9, 42]. 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 [15].

Marré and Bertolino [35] 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 [34]. 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 [35]. Thus, testers can save resources by only covering TRs that are guaranteed to cover other TRs. Santelices and Harrold [46] 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 [2, 28–31]. Killing a minimal set of mutants 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 [2, 28] introduced the theoretical concept, presented the mutant subsumption graph (MSG), and showed how to approximate the "true" MSG dynamically. Subsequent papers showed how to approximate the MSG statically [29], showed that redundant (constrained) mutants effect the mutation score [31], and used minimal mutation to identify a significant weakness in selective mutation [30]. 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.

### *3.1. 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



*Prepared using* stvrauth.cls
DOI: 10.1002/stvr

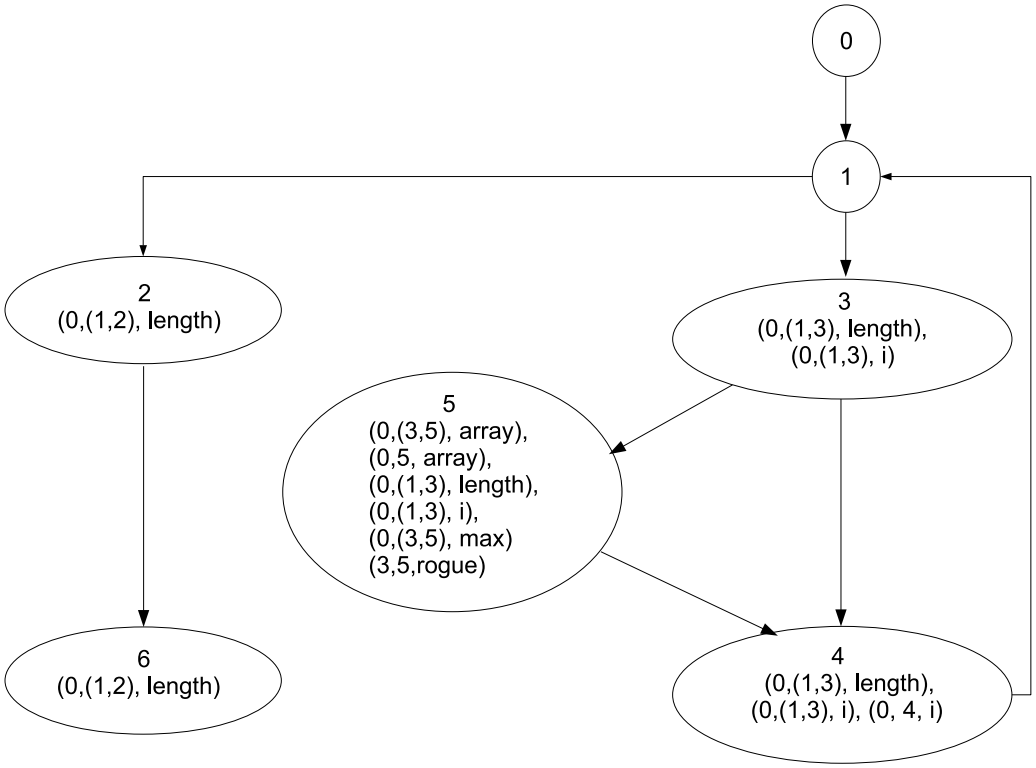


Figure 3. Local DUA-node subsumption for program Max

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 program 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 $n_i$ to the exit node includes $n_j$. A node dominates itself but does not post-dominate itself [23]. 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%.

### *3.2. 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




DOI: 10.1002/stvr

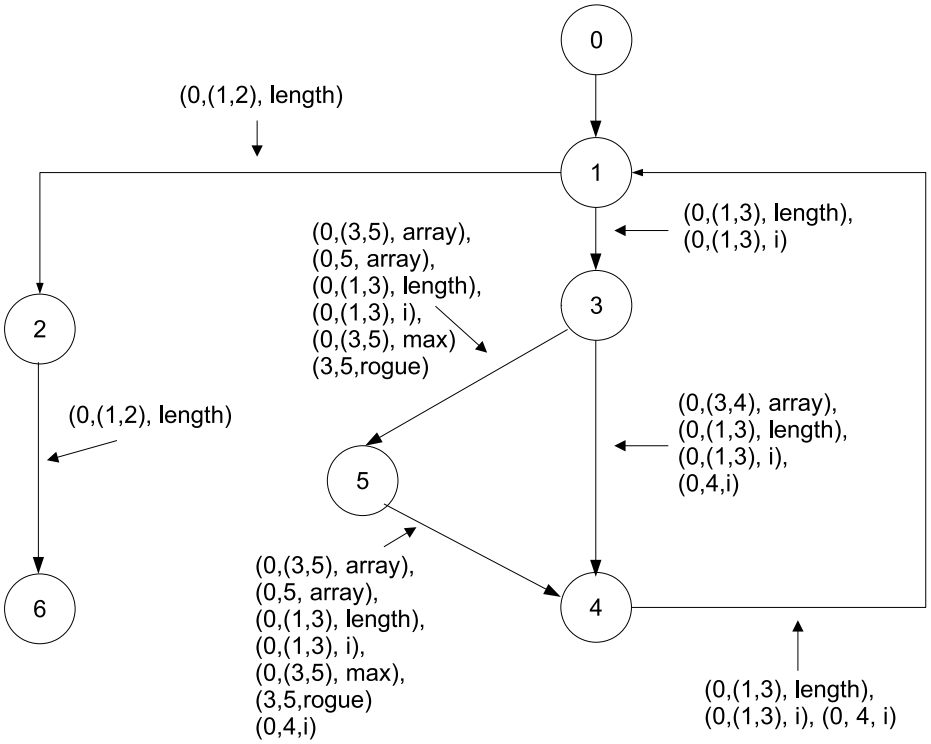


Figure 4. Local DUA-edge subsumption for program Max

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).

### *3.3. DUA-DUA subsumption*

DUA-node and DUA-edge subsumption relate different criteria–node coverage with all-uses 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 $D_2$.

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 [28].



*Prepared using* stvrauth.cls
DOI: 10.1002/stvr

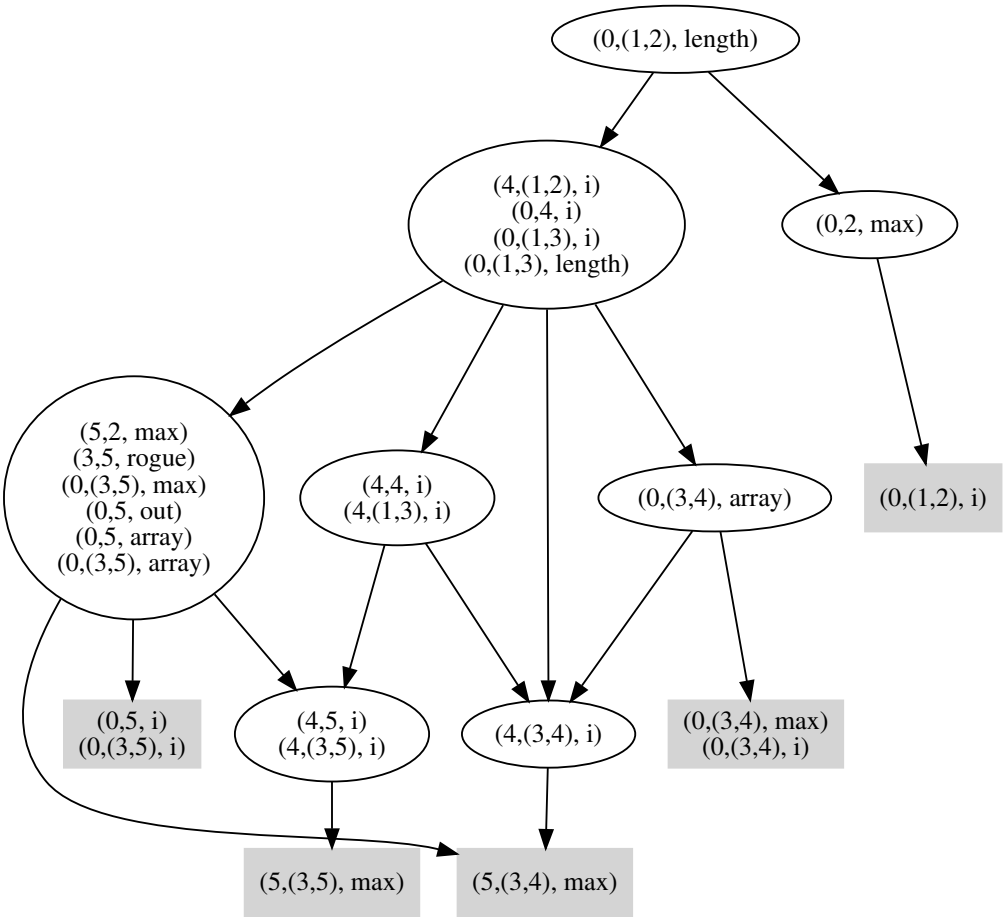


Figure 5. DUA-DUA subsumption for example program Max

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), $i$)$\}$.

## 4. 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 [26]. They determine *facts* that are valid at the entry or exit of a program point $p$ whenever $p$ is reached [20]. We use the data flow analysis framework *Data Flow Subsumption Framework* (DSF) [4] 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.



*Prepared using* stvrauth.cls
DOI: 10.1002/stvr

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

### *4.1. 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.

**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 1-6 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 [6]. 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. $u = n$




DOI: 10.1002/stvr

**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 [1]. 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.

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 construct 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 $p$ [1]. 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, represent the DUAs that are 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**($p$).

To exemplify how SA works, consider node 5 from program Max in Figure 2. Figure 6 shows how sets **IN**(5), **Covered**(5), and **OUT**(5) evolve during SA's execution in three distinct points (*Point 1*, *Point 2*, and *Point 3*). Point 1, before Line 8 in Algorithm 1, is located immediately inside the body of the while-loop, which repeats until all sets **OUT**($n$) stop changing. For program Max, the while-loop repeats three times until sets **OUT**($n$) stop changing. Figure 6 shows **IN**(5), **Covered**(5), and **OUT**(5) at Point 1 in these three iterations. Point 2, before Line 13 in Algorithm 1, is immediately



*Prepared using* stvrauth.cls
DOI: 10.1002/stvr

after the execution of the while-loop and Point 3, before Line 17 in Algorithm 1, is just before returning the **Covered**(n) sets.

| **Point 1 – Before line 8 in Algorithm 1** | | |
|---|---|---|
| **Iter.** | **Sets** | **DUAs** |
| 1 | **IN**(5) | — |
| | **Covered**(5) | All DUAs |
| | **OUT**(5) | All DUAs |
| 2 | **IN**(5) | (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), (3,5,rogue) |
| | **Covered**(5) | (0,(3,5),array), (0,5,array), (0,(1,3),length), (0,(1,3),i),(0,(3,5),i), (0,5),i), (0,(3,5),max), (3,5,rogue) |
| | **OUT**(5) | (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,(3,5),max), (3,5,rogue), (5,2,max), (5,(3,5),max), (5,(3,4),max) |
| 3 | **IN**(5) | (0,(3,5),array), (0,(3,4),array), (0,5,array), (0,(1,3),length), (0,(1,2),length), (0,(1,3),i), (0,4),i), (0,(3,5),max), (3,5,rogue) |
| | **Covered**(5) | (0,(3,5),array), (0,5,array), (0,(1,3),length), (0,(1,3),i), (0,(3,5),max), (3,5,rogue) |
| | **OUT**(5) | (0,(3,5),array), (0,(3,4),array), (0,5,array), (0,(1,3),length), (0,(1,2),length), (0,(1,3),i), (0,4),i), (0,(3,5),max), (3,5,rogue), (5,2,max), (5,(3,5),max), (5,(3,4),max) |
| **Point 2 – Before line 13 in Algorithm 1** | | |
| | **Sets** | **DUAs** |
| | **IN**(5) | (0,(3,5),array), (0,(3,4),array), (0,5,array), (0,(1,3),length), (0,(1,2),length), (0,(1,3),i), (0,4),i), (0,(3,5),max), (3,5,rogue) |
| | **Covered**(5) | (0,(3,5),array), (0,5,array), (0,(1,3),length), (0,(1,3),i), (0,(3,5),max), (3,5,rogue) |
| | **OUT**(5) | (0,(3,5),array), (0,(3,4),array), (0,5,array), (0,(1,3),length), (0,(1,2),length), (0,(1,3),i), (0,4),i), (0,(3,5),max), (3,5,rogue), (5,2,max), (5,(3,5),max), (5,(3,4),max) |
| **Point 3 – Before line 17 in Algorithm 1** | | |
| | **Sets** | **DUAs** |
| | **IN**(5) | (0,(3,5),array), (0,(3,4),array), (0,5,array), (0,(1,3),length), (0,(1,2),length), (0,(1,3),i), (0,4),i), (0,(3,5),max), (3,5,rogue) |
| | **Covered**(5) | (0,(3,5),array), (0,5,array), (0,(1,3),length), (0,(1,3),i), (0,(3,5),max), (3,5,rogue) |
| | **OUT**(5) | (0,(3,5),array), (0,(3,4),array), (0,5,array), (0,(1,3),length), (0,(1,2),length), (0,(1,3),i), (0,4),i), (0,(3,5),max), (3,5,rogue), (5,2,max), (5,(3,5),max), (5,(3,4),max) |

Figure 6. Sets of program Max's node 5 during the execution of Algorithm 1

At Point 1, in the first iteration†, **IN**$(5)^1$, **Covered**$(5)^1$ and **OUT**$(5)^1$ have the values that were assigned during the initialization on Lines 1-6. **IN**$(5)^1$ will still receive the results of the transfer function on Line 9. **Covered**$(5)^1$ and **OUT**$(5)^1$ contain all DUAs as assigned on Lines 1-6. In the second iteration, Point 1 shows **IN**$(5)^2$, **Covered**$(5)^2$, and **OUT**$(5)^2$ having the result of the application of DSF transfer functions (Lines 9-12) during the first iteration. **IN**$(5)^2$ is a subset of DUAs because it is calculated at Line 9 using **OUT**$(3)^1$ (node 3 is the only predecessor of node

†We use exponents to differentiate the values of sets **IN**(5), **Covered**(5) and **OUT**(5) in the three iterations at Point 1. For instance, **IN**$(5)^2$ refers to the value of the **IN**(5) in the second iteration; that is, it refers to the set of DUAs presented in the sixth row from the top of the table of Figure 6.



*Prepared using* stvrauth.cls
DOI: 10.1002/stvr

5). **Covered**(5)$^2$ and **OUT**(5)$^2$ also contain a restricted set of the DUAs due to the application of DSF transfer functions on Lines 11 and 12. Point 1 in the third iteration shows the results of the second iteration where **IN**(5)$^3$ has six fewer DUAs than **IN**(5)$^2$, **Covered**(5)$^3$ two fewer DUAs than **Covered**(5)$^2$, and **OUT**(5)$^3$ four fewer DUAs than **OUT**(5)$^2$.

At Point 1, **OUT**(5) starts with all DUAs and as the DSF transfer functions are applied, its value converges to a fixed point, which is the set of those DUAs that are covered or available to be covered at node 5. When sets **OUT** converge for every node $n$ of Max's flow graph, SA leaves the while-loop at Line 7 and goes to Point 2.

Point 2 shows **IN**(5), **Covered**(5), and **OUT**(5) as having the same values as Point 1 at iteration 3, showing that **OUT**(5) had already converged to its fixed point after iteration 2. However, not all **OUT**($n$) had converged, so a third iteration was needed. In the next section, we analyze the number of iterations required to leave while-loop at Line 7 for most of the programs.

Values of **IN**(5), **Covered**(5), and **OUT**(5) at Point 2 are used to calculate the final value of **Covered**(5) on Lines 13-16. Point 3 shows the result of SA. The final value of **Covered**(5) is equal to that of Point 2 and contains the DUAs {(0,(3,5),array), (0,5,array), (0,(1,3),length), (0,(1,3),i), (0,(3,5),max), (3,5,rogue)}, as shown in node 5 of Figure 3. That is so because **OUT**(5) converged after iteration 2, however, sometimes the **OUT** set converges in the last iteration, which makes it necessary to apply transfer function at Line 11 again at Line 16 to achieve the correct values for **Covered**($n$).

### *4.2. 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 [1] plus 2. These loops tend to be limited to a small constant [27, 45]. 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.

### *4.3. Memory requirements*

The subsumption algorithm SA receives as input the flow graph $G(N, E, s, e)$ of a program $P$; and sets **Disabled**($n$), **Sleepy**($n$), **PotCovered**($n$), and **Born**($n$). SA returns the sets **Covered**($n$) as output. Additionally, SA uses two working sets for each node $n$ of $G$, **IN**($n$) and **OUT**($n$), and the working set **CurSleepy**. All these sets have the size $U$, where $U$ is the set of all DUAs required to test $P$. As a result, SA uses 7 $\times$ |N| + 1 sets of size |U|. Thus, the memory cost for all SA's sets is O(|N| $\times$ |U|). The memory needed for flow graph $G$ costs O(|N| + |E|). Since we can assume the number of edges as O(|N|), the flow graph $G$ costs O(|N|) to store.

Therefore, all SA memory requirements total O(|N|) + O(|N| $\times$ |U|), that is, O(|N| $\times$ |U|). Though this amount of memory should not stress the RAM memory of modern computers, it can be reduced even further if one implements SA's sets as bit vectors. Memory was not a problem in our empirical assessment.



*Prepared using* stvrauth.cls
DOI: 10.1002/stvr

### *4.4. SA's correctness*

We verified the correctness of SA by proving that the data flow subsumption framework (DSF) is monotone and distributive and that sets **OUT**($n$) contain the covered and available nodes for coverage DUAs at a node $n$. As a result, by applying an iterative algorithm, SA finds sets **OUT** and then the final value of **Covered**($n$). We detail the proof in our technical report [4].

Therefore, SA finds the local DUA-subsumption at every node $n$; that is, every DUA that is subsumed whenever node $n$ is reached from any path from the start node $s$. However, SA only works and has the costs as discussed above when the following conditions hold.

1. A program $P$'s flow graph $G$ has to be *well-formed*: (a) there is a single start node $s$ with zero incoming edges; (b) there is a single exit node $e$ with zero outgoing edges; (c) for every edge ($n'$,$n$), $n' \neq n$ (there are no self loops).

2. Sets **Disabled**($n$), **Sleepy**($n$), **PotCovered**($n$), and **Born**($n$) should have been calculated previously.

## 5. 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.

### *5.1. DUA-node subsumption*

Section 3 presented the DUA-node subsumption algorithm informally. First, we find the local DUA-node subsumption using SA and the post-dominance relationship. Then, we union the **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.

### *5.2. 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')) ⋂ PotCovered(n)] ⋃ Covered(n);
4     else
5         Covered(n', n) = Covered(n') ⋃ 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



DOI: 10.1002/stvr

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 $\approx O(|\mathrm{N}|)$.

### *5.3. 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 [34, 35] 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, in principle, 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 $D_2$.

To find DUAs $D_2$ that are subsumed by $D_1$, we first calculate the graph graphdua$(D_1)$ using the algorithm described in our previous paper [37]. Figure 7 shows the graphdua for $D_1$(3, 5, rogue). A graphdua consists of the composition of five sub-graphs; each encompassing sub-paths required to cover $D_1$.

The first sub-graph, *SG1*, encompasses paths from $s$ to $d_1$. In Figure 7, this sub-graph is composed of nodes whose identifiers are pairs *n(1)* where $n$ is the node of the original flow graph $G$ and 1 is the identifier of *SG1*. There is an edge $(n'(1),n(1))$ in a graphdua if there is an edge $(n',n)$ in $G$. The graphdua also includes paths from $d_1$ to $d_1$; they are represented by sub-graph *SG2*, which in turn is composed of nodes *n(2)* and edges connecting them. *SG3* represents def-clear paths wrt $X_1$ from $d_1$ to $u_1$, *SG4* paths from $u_1$ to $u_1$, and *SG5* paths from $u_1$ to $e$. In Figure 7, these sub-graphs are indicated by nodes identified by pairs *n(3)*, *n(4)*, and *n(5)*, respectively. As proposed by Marré and Bertolino, $G*$ does not include paths encompassed by sub-graphs *SG2* and *SG4*, which might lead to incorrect subsumptions since possible redefinition paths are not blocked. We compare and contrast $G*$ and the graphdua in related work (Section 7).

We can run SA on graphdua$(D_1)$ since it is a regular graph with a different node identification schema. 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 $D_1$. In Figure 7, the **Covered** set at node 6(5) gives all DUAs subsumed by $D_1$(3, 5, rogue).

As far as cost goes, $G*$ and graphdua$(D_1)$ cost $O(|\mathrm{E}|)$ to calculate, where $|\mathrm{E}|$ is the number of edges [34, 37]; 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 [33]. Implementing the sets of subsumed DUAs as bit vectors and scanning them with machine instructions reduces this cost.

## 6. EXPERIMENTAL ANALYSIS

Previous sections showed how data flow subsumptions can be calculated at asymptotically lower cost than previous approaches. However, Section 1 raises additional questions related to identifying and using data flow subsumption. We present them as research questions below:

**Subsumption discovery** :




DOI: 10.1002/stvr

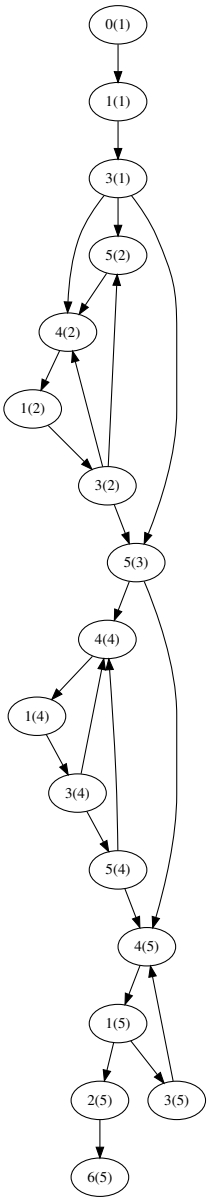


Figure 7. Graphdua for $D_1(3, 5, \text{rogue})$

**RQ1:** Does our approach using SA and graphduas correctly find data flow subsumptions?

**Effectiveness** :

**RQ2** : Is the number of unconstrained DUAs (substantially) smaller than the total number of DUAs?

**RQ3** : How many DUAs are subsumed by node and edge coverage?

**Scalability** :

**RQ4:** How long does it take to calculate the data flow subsumption?

**Fault detection ability** :

**RQ5** : How much fault detection ability is lost by using unconstrained DUAs to create tests?




DOI: 10.1002/stvr

**Yield of data flow testing** :

**RQ6** : Are the top fault revealing DUAs subsumed by node and edge coverage?

The rest of this section presents the subject programs, the pool of faults used to assess questions RQ5 and RQ6, the concept of probabilistic coupling used as proxy of the fault detection ability and, for each RQs, the results, and discussion. We conclude the section with threats to validity. We have deployed a replication package on github: `https://github.com/icst2021satool`.

### *6.1. Subject programs*

For our study, we chose 17 programs from the Defects4J repository [25], plus machine learning programs Elki (`https://elki-project.github.io/`), SystemDS (`https://systemds.apache.org/`), and Weka (`https://www.cs.waikato.ac.nz/ml/weka/`) and natural language processing program CoreNPL (`https://stanfordnlp.github.io/CoreNLP/`) 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 Elki's commit 6465675, SystemDS version 3.0.0, Weka's version 3.8, and CoreNLP version 4.4.0. The programs' purposes vary: Compress, Csv, Gson, JacksonCore, JacksonDataBind, JacksonXml, and JSoup manipulate text in compressed and binary files; Cli, Closure, and JxPath parse and compile; Collections and Lang are utilities for data structure manipulation and languages; Elki, SystemDS, Math, and Weka are mathematics, statistics, and data mining packages; Time manipulates date and time objects; Mockito supports software testing; and CoreNPL supports natural language processing (CoreNPL). The programs also vary in size: ranging from small programs such as Csv (602 LOC) to larger programs such as Weka (216,781 LOC).

Table 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. We used `javancss` (`www.kclee.de/clemens/java/javancss/`) to calculate the LOC, and `SAtool` (`github.com/icst2021satool/source-code`) to find the methods with DUAs, the DUAs themselves, and data flow subsumptions. We modified `Jaguar` [44] to collect DUA coverage for every JUnit method of the developers' tests included in the programs' repository.

As a result, we use `SAtool` to calculate DUAs and subsumption relationships and then `Jaguar` to find DUA coverage. To ensure the integrity of the data from `SAtool` and `Jaguar`, we checked whether both tools generate the same set of DUAs[‡]. We found no difference for programs in the Defects4J repository, but found 66 (out of 509) Weka's classes with different `SAtool` and `Jaguar` DUA sets. These classes were removed before evaluating our research questions. Additionally, `SAtool` could not analyze a few classes from Elki (2), Weka (1), SystemDS (11), and CoreNPL (6). We were not able to collect coverage data for Elki, SystemDS, and CoreNLP, because `Jaguar` was not able to properly process them.

### *6.2. Pool of faults*

We used most of the faulty versions of the programs present in Defects4j repository, but could not use all due to the following `Jaguar` limitations:

1. `Jaguar` uses `BA-DUA` [11] to collect data flow coverage. `BA-DUA` does not generate data flow coverage for methods with uncaught exceptions, so we could not use those faults.

2. JVM limits the size of generated Java bytecode for each method in a class to a maximum of 64K bytes. If the method's bytecode plus the instrumentation added by `BA-DUA` exceeds 64K bytes, the method is not instrumented and no coverage is generated.

[‡]See `https://github.com/icst2021satool/subsumption-experiment/blob/main/scripts/comparison.py`




DOI: 10.1002/stvr

3. We ran the test classes using the command line interface (CLI) of JUnit (`http://junit.org`), Defects4J ran the tests using tools such as Maven (`https://maven.apache.org/`). We could run the majority of Defects4J's faulty versions with JUnit, excepting Lang's versions 35 to 65. These versions use reflection, which caused more test cases to fail.

Thus, we could not use 51 of the original 835 faults, leaving 784 to form the pool for our study. Table III details the Defects4J programs, number of bugs removed, and the reason for removal. In the fourth column (**Removed bugs**), the number between parenthesis refers to the Jaguar limitation from the list above that caused us to remove that bug.

Table III. Faulty versions used in the pool of faults

| Program | # of bugs in the pool | Total # of bugs | Removed bugs |
|---|---|---|---|
| Chart | 23 | 26 | 6(1,2),8(1,2),10(1,2) |
| Cli | 39 | 39 | — |
| Closure | 172 | 174 | 137(1,2), 143(1,2) |
| Codec | 18 | 18 | — |
| Collections | 4 | 4 | — |
| Compress | 46 | 47 | 44(1,2) |
| Csv | 16 | 16 | |
| Gson | 17 | 18 | 8(1,2) |
| JacksonCore | 22 | 26 | 5(1,2),19(1,2),23(1,2),25(1,2) |
| JacksonDatabind | 109 | 112 | 44(1,2), 45(1,2), 95(1,2) |
| JacksonXml | 6 | 6 | |
| Jsoup | 93 | 93 | |
| JxPath | 22 | 22 | |
| Lang | 30 | 64 | 6(1,2), 11(1,2), 35-65(3) |
| Math | 103 | 106 | 5(1,2),55(1,2),89(1,2) |
| Mockito | 38 | 38 | — |
| Time | 26 | 26 | — |
| **Total** | 784 | 835 | |

The pool's data for a particular bug is comprised of matrices for which the rows represent the tests and the columns represent the DUAs required by the methods of a class. There is one matrix for each class loaded during testing. A cluster of 128 servers with 20 cores, 512 GB of RAM, and Intel(R) Xeon(R) CPU E7-2870 @ 2.40GHz processor collected the data for most programs, with the exception of Cli, Collections, Compress, Csv, JacksonDatabind, Lang, Mockito, and Time. For these programs, we used `Jaguar` with a modified `BA-DUA` library to circumvent a problem with class loading. Some versions use libraries (e.g., PowerMock: `https://powermock.github.io/`) that overrides the application's class loader so that `Jaguar` loses access to the data structures needed to track data flow coverage.

For RQ1, we used data coverage on one buggy version (1b) for every Defects4J programs and Weka. We restricted the analysis to one buggy version of the programs because: (a) we did not expect significant differences from one bug to another, and (b) we manually checked parts of the results. For RQ2 to RQ4, we added Elki, SystemDS, and CoreNPL because subsumption discovery, effectiveness, and cost seemed particularly sensitive to complex mathematical programs.

RQ5 and RQ6 used the pool of faults described in Table III and did not include Elki, SystemDS Weka, and CoreNPL because these programs do not have faulty versions available.




DOI: 10.1002/stvr

### 6.3. Probabilistic coupling

Recently, Chen et al. [7] introduced a new measure to assess the sensitivity of node, edge, DUA, and mutant coverage test requirements in revealing faults. *Probabilistic coupling* ($PC$) is defined, for a real fault $f$ and a test requirement $tr$, by the conditional probability $p = P(\text{detect} f \mid tr \text{ is detected})$. Thus, $PC$ captures the likelihood of detecting a fault given that a particular test requirement is satisfied during testing. Note that a $tr$ with $p = 1$ is perfectly coupled with fault $f$; if $p = 0$, it is perfectly decoupled; and if $0 < p < 1$, it is probabilistic coupled.

The *maximum probabilistic coupling* ($PC_{max}$) for a set of test requirements indicates the sensitivity of the test requirements in revealing a known real fault and is agnostic to noise caused by unrelated test goals (e.g., test set size). Though probabilistic coupling does not capture the complex inter-dependencies of test requirements, it can approximate fault detection ability of testing requirements.

We calculated the $PC$ for each DUA for every bug in the pool as a proxy for their fault detecting ability. Let $f$ be a fault in the pool and $D$ a DUA required by all uses criterion to test a program $P$; we calculate $PC(D)$ using the formula below.

$$PC(D) = \frac{\#\text{ of tests that covered } D \text{ and failed for fault} f}{\#\text{ of tests that covered } D}$$

Then we selected those DUAs with $PC_{max}$ as an approximation of the data flow testing ability in revealing the bugs. Among the DUAs with $PC_{max}$ there might be unconstrained DUAs, subsumed DUAs, and DUAs subsumed by nodes and edges. We make use of this information to assess the fault detection ability of unconstrained DUAs and the value of data flow testing.

### 6.4. RQ1: Data flow subsumption calculation

Section 5 presents algorithms to find data flow subsumptions, local DUA-edge subsumption and, especially, global DUA-node, DUA-edge, and DUA-DUA subsumptions, which are built upon SA and our new data structure, graphdua. We use DUA coverage and global DUA-DUA subsumption data to verify our data flow subsumption approach using SA and graphduas. We verify them by checking two properties associated with DUA-DUA subsumption.

The subsumption relationship is reflexive; that is, every test requirement (in our case, a DUA) of a program subsumes itself. The reflexive property was checked in 1,618,182 DUAs of 56,720 methods of all programs and failed for 25 methods and 72 DUAs. The 25 failures were based on two issues with not well-formed 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`, which might generate ill-well-formed graphs. Once we removed 447 (of 56,720) 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 and found that it did not hold for 21 of 15,610 executed methods. The subsumption property was disrupted in 20 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 our approach using SA graphduas correctly finds data flow subsumption, provided SA's assumptions hold (Section 4). Additionally, it shows that execution interruptions (e.g., exceptions in `try` clauses) can disrupt global data flow subsumptions as discussed in Section 3.



DOI: 10.1002/stvr

### *6.5. RQ2 and RQ3: Data flow subsumption effectiveness*

Table IV 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 in the table presents the mean, median, minimum, maximum, and standard deviation values for their columns.

Table IV. Effectiveness and scalability data

| Program | DUAs | %-DUA -Node | %-DUA -Edge | %-Unc. DUAs | t-DUA -Node (ms) | t-DUA -Edge (ms) | t-Unc. DUAs (ms) |
|---|---|---|---|---|---|---|---|
| Csv | 929 | 57.3 | 70.3 | 31.5 | 331.3 | 323.6 | 664.0 |
| Cli | 1291 | 69.6 | 83.0 | 29.4 | 426.4 | 426.6 | 877.0 |
| **Codec** | 4446 | 38.0 | 48.4 | 37.9 | 640.1 | 634.8 | 6268.0 |
| Jsoup | 1866 | 80.0 | 92.9 | 26.8 | 582.7 | 567.3 | 828.1 |
| J-Xml | 3402 | 75.7 | 87.9 | 26.5 | 686.2 | 665.8 | 1187.5 |
| **Compress** | 6286 | 58.7 | 67.0 | 28.3 | 1109.5 | 1182.2 | 5801.6 |
| Gson | 3281 | 76.6 | 88.4 | 29.7 | 767.1 | 741.9 | 1140.3 |
| Mockito | 4236 | 74.6 | 90.0 | 32.1 | 1104.2 | 1047.9 | 1482.0 |
| J-Core | 17,653 | 59.5 | 72.7 | 28.1 | 1484.0 | 1469.2 | 6199.1 |
| JxPath | 20,178 | 66.9 | 84.4 | 29.7 | 1685.0 | 1661.8 | 6201.6 |
| Lang | 22,290 | 62.8 | 73.9 | 32.0 | 1996.5 | 1937.8 | 5717.0 |
| Time | 18,160 | 69.2 | 80.9 | 31.1 | 2101.5 | 2046.9 | 5143.0 |
| Collections | 16,937 | 68.8 | 81.9 | 30.0 | 2107.3 | 2028.8 | 4319.1 |
| J-DataBind | 31,797 | 73.8 | 85.6 | 26.4 | 2845.2 | 2774.8 | 6963.8 |
| **Math** | 87,603 | 57.9 | 64.2 | 25.3 | 11,625.4 | 11,220.4 | 100,184.0 |
| Chart | 81,847 | 72.8 | 82.2 | 24.5 | 6131.7 | 5776.9 | 17,165.8 |
| Closure | 78,068 | 67.9 | 83.7 | 31.8 | 6106.6 | 5817.2 | 31,206.8 |
| Elki | 180,675 | 63.4 | 72.4 | 24.9 | 12,544.0 | 12,367.6 | 39,454.3 |
| SystemDS | 255,382 | 66.0 | 79.6 | 26.5 | 14,012.5 | 13,943.9 | 42,679.6 |
| Weka | 337,063 | 59.9 | 70.9 | 27.4 | 19,647.2 | 19,164.1 | 84,613.3 |
| CoreNLP | 255,382 | 54.6 | 65.2 | 36.7 | 32,163.9 | 30,831.3 | 1,114,919.0 |
| **Mean** | 68,037 | 65.4 | 77.4 | 29.4 | 5719.0 | 5553.8 | 70,620.0 |
| **Median** | 18,160 | 66.90 | 80.9 | 29.4 | 1996.5 | 1937.8 | 6199.1 |
| **Min.** | 929 | 38.0 | 48.4 | 24.5 | 331.3 | 323.6 | 664.0 |
| **Max.** | 337,063 | 80.0 | 92.9 | 37.9 | 32,163.9 | 30,831.3 | 1,114,919.0 |
| **Std Dev.** | 100,723 | 9.5 | 10.7 | 3.5 | 8148.5 | 7873.3 | 240,877.6 |

Effectiveness data show that node and edge coverage can lead to significant data flow coverage; 65.4% and 77.4% on average. Though the minimum values can be as low as 38% (**%DUA-node**) and 48.4% (**%DUA-edge**), the mean and median values are close, showing that many DUAs are covered when nodes and edges are plentiful in the program. However, many small methods have 100% **%DUA-node** and **%DUA-edge** coverage (18,761 and 33,9191 methods, respectively).

To assess the effectiveness on harder to test methods, Figure 8 gives the histograms of **%DUA-node** and **%DUA-edge** for 2,604 methods (out of 56,720 of all programs) with more than 100 DUAs. The number of methods with **%DUA-node** coverage equal or below 40% is 544 and for **%DUA-edge**, 326. A few methods have as many as thousands of DUAs; in our data set we have 50 methods with more than 1,000 DUAs. So, many DUAs may still need to be tested after achieving node and edge coverage.

About 30% of all DUAs are unconstrained (Table IV). Interestingly, the relative amount of unconstrained DUAs is quite similar in all programs: they comprise between 24.5% and 37.9% of all



*Prepared using* stvrauth.cls
DOI: 10.1002/stvr

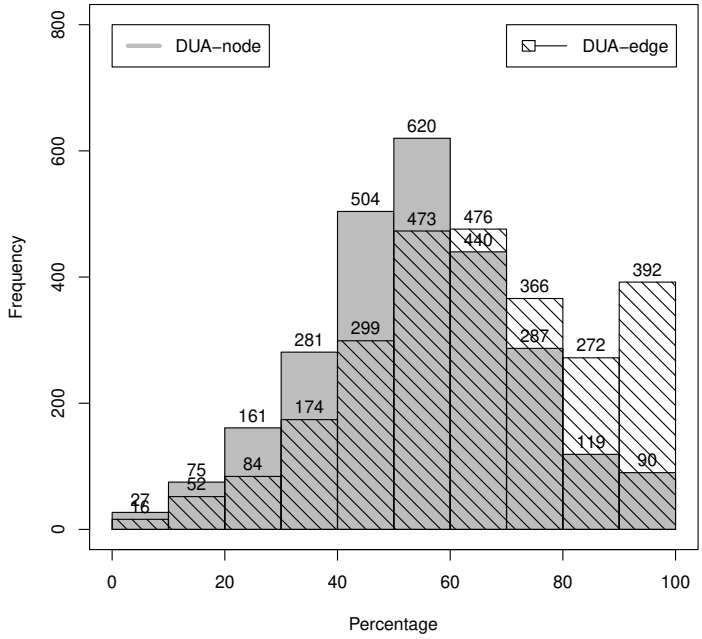


Figure 8. DUA coverage due to node and edge coverage for methods with more than 100 DUAs

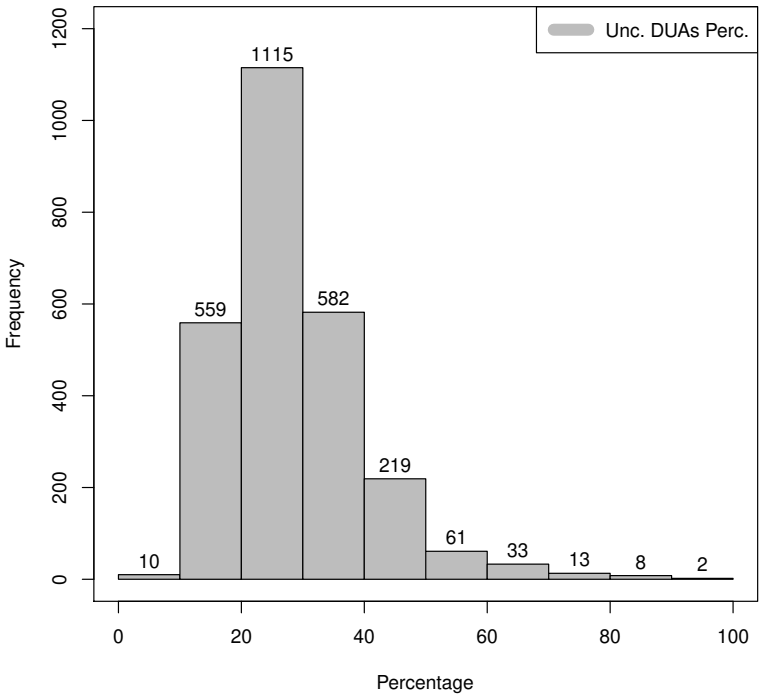


Figure 9. Unconstrained DUAs for methods with more than 100 DUAs



*Prepared using* stvrauth.cls
DOI: 10.1002/stvr

DUAs. Figure 9 illustrates methods with more than 100 DUAs. Unconstrained DUAs represent less than or equal to 30% of all DUAs for 1,684 out of the 2,604 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 need to test unconstrained DUAs not covered by edge or node coverage.

### *6.6. RQ4: Data flow subsumption cost*

We ran `SAtool` to find data flow subsmptions on a MacAir, 1.8 GHz Dual-Core Intel Core i5, 8 GB 1600 MHz DDR3. Columns **t-DUA-node** and **t-DUA-edge** in Table IV 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 took only around 30s to find the local DUA-node and DUA-edge subsumption for the more demanding program, CoreNLP. As expected by the asymptotic analysis, **t-DUA-node** and **t-DUA-edge** values are very similar.

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 a 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. 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. The slowest, by far, is CoreNP, which took 18 min, followed by Math with 100s and Weka with 85s. Figure 10 plots the number of DUAs and the time in milliseconds for unconstrained sets calculation to assess the relation with the number of DUAs. For all but a few programs, the number of **t-Unc. DUAs** is a fraction of the number of **DUAs**. Exceptions are Codec, Compress, Math, and CoreNLP.

The effect of the methods' complexity is magnified for unconstrained DUAs calculation. SA is applied on each DUA's graphdua; thus, any increase to SA's cost will be multiplied by the number of DUAs. This is true for Math, but even more so for CoreNLP, which took about 18 minutes to find the unconstrained DUAs. The second most demanding program, Math, only took about 1 min and 40s. CoreNLP has 33 methods with more than 1,000 DUAs. However, CoreNLP's overall cost is due to a single method that uses about two thirds of the total time. That method's control flow graph has 1,222 nodes, 2,132 edges, 15 back arcs, and 22,501 DUAs! Excepting for methods as complex as this one, SA finds unconstrained DUAs very quickly.

This single method brings up several questions. Should a method whose control flow graph has 1,222 nodes be refactored? Is there any way to assess or ensure the correctness of such a method? Is data flow the best testing approach for such a method? These questions, while intriguing, are beyond the scope of this paper.

### *6.7. RQ5: Fault detection ability of unconstrained DUAs*

We used the DUAs with the highest $PC_{max}$ as an approximation of the fault detection ability of data flow testing.

Figures 11 and 12 give violin plots for $PC_{max}$ for all DUAs and only unconstrained DUAs, for the bugs in our pool. The triangle indicates the median and the star indicates the average in both figures. Violin plots indicate the regions where the distribution of data is denser. In both plots, the most often $PC_{max}$ value is 1.0—378 for DUAs and 374 for unconstrained DUAs out of 784. There is very little difference between Figures 11 and 12. The average $PC_{max}$ for DUAs is 0.651 and 0.643 for unconstrained DUAs with median 0.667 for both.

Table V presents the estimated probability of losing fault detection ability by using unconstrained DUAs (uncDUAs) instead of DUAs for the bug pool. Unconstrained DUAs lost fault detection ability for 15 out of 784 bugs, which gives an estimated probability of 1.91%. Due to the large sample size, we assume the convergence of sample mean to a Normal distribution; thus, the 95% confidence interval is 0.95% and 2.87%.




DOI: 10.1002/stvr

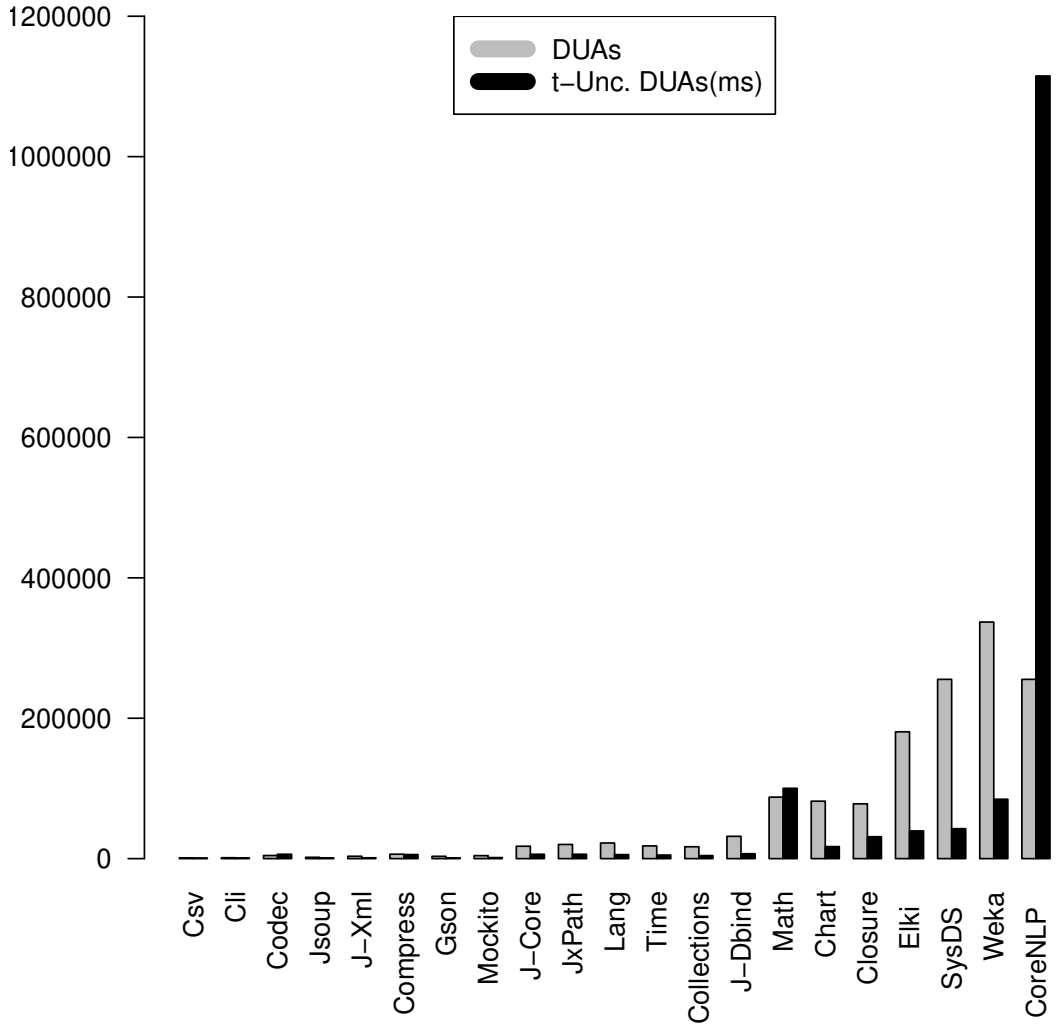


Figure 10. Number of DUAs and time to find the unconstrained DUAs

Table V. Probability of fault detection ability loss

| | |
|---|---|
| # Bugs with $PC_{max}$(UncDUAs) $<$ $PC_{max}$(DUAs) $=$ True | 15 |
| # Bugs with $PC_{max}$(UncDUAs) $<$ $PC_{max}$(DUAs) $=$ False | 769 |
| Probability of $PC_{max}$(UncDUAs) $<$ $PC_{max}$(DUAs) $=$ True | 1.91% |
| Inferior Confidence Interval | 0.95% |
| Superior Confidence Interval | 2.87% |

Figures 11 and 12 suggest that data flow testing is quite effective for the faults. Tests covering data flow requirements were able to reveal almost half of the faults in our pool, that is, $PC_{max}$ equals to 1.0 for 378 faults. Additionally, they indicate that unconstrained DUAs and DUAs, with the exception of a few bugs, have identical $PC_{max}$ distributions.

This finding shows that unconstrained DUAs are able to capture fault revealing data flows as well as reduce the number of test requirements. Table V further shows that the likelihood of losing fault detection ability by using unconstrained DUAs is very low.



*Prepared using* stvrauth.cls
DOI: 10.1002/stvr

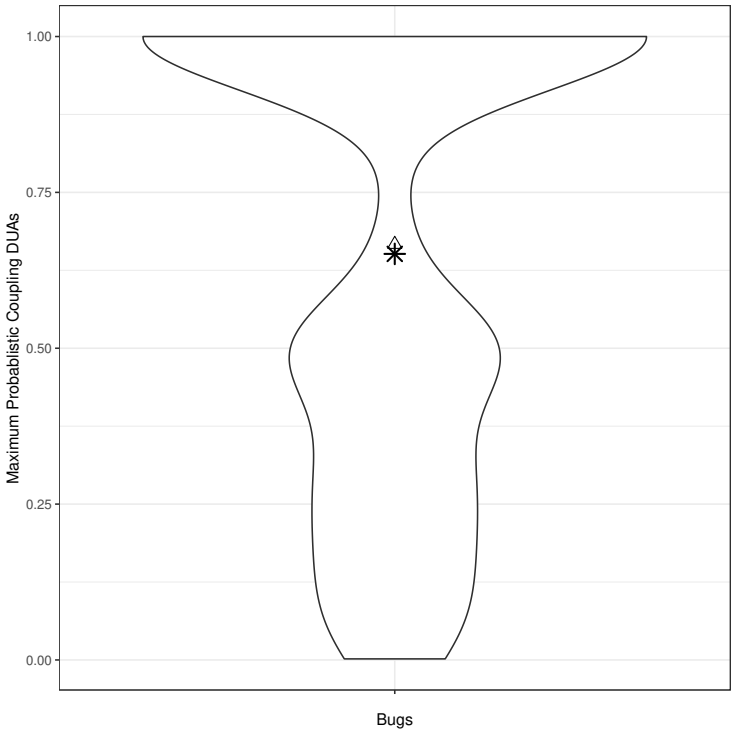


Figure 11. Maximum probabilistic coupling for DUAs

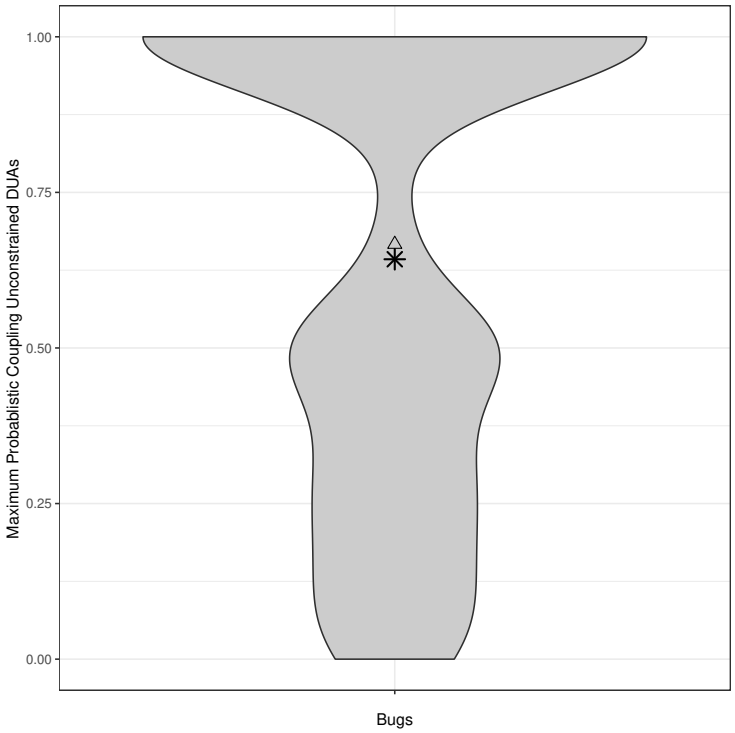


Figure 12. Maximum probabilistic coupling for unconstrained DUAs

In most cases, the fault detection ability of unconstrained DUAs is due to interference in DUA-DUA subsumption. We also identified cases where DUA-DUA subsumption fails at run-time or




DOI: 10.1002/stvr

cannot be calculated. The subsumption relationship fails when: a program *interruption occurs* (Section 6.4); or a subsumed DUA is covered by a test, but its subsumers are not. The latter situation happens because the subsumption relationship is *not symmetric*[§]. Additionally, we cannot find data flow subsumptions when the method under analysis has an *ill-formed flow graph* (Section 6.4). We investigated these situations and their impact on the unconstrained DUAs ability to reveal faults. Table VI gives the causes of data flow subsumption disruption, the number of bugs associated with each cause, and the loss of fault detection ability ($PC_{max}(\text{UncDUAs}) < PC_{max}(\text{DUAs}) = \text{True}$).

Table VI. Data flow subsumption disruption and fault detection ability loss

| Bugs with failed data flow subsumption | | | |
|---|---|---|---|
| Cause of Disruption | Loss of fault detection ability | | # Bugs |
| | Yes | No | |
| Program interruption | 0 | 4 | 4 |
| Ill-formed flow graph | 8 | 0 | 8 |
| Non-symmetric | 7 | 0 | 7 |
| Total | 15 | 4 | 19 |

The most fault revealing DUAs for four bugs were located in methods in which data flow subsumption failed due to program interruption (row **Program interruption**). However, there was no loss of fault detection ability because at least one unconstrained DUA was among the top revealing DUAs. The unconstrained DUAs failed to subsume all DUAs at run-time, but they were still as prone to reveal the faults as all DUAs.

For eight bugs, the top revealing DUAs were located in methods with ill-formed graphs (row **Ill-formed flow graph**). Strictly speaking, the subsumption relationship was not disrupted, but it could not be calculated. Once the ill-formed flow graphs are fixed, the subsumption relationship will be found. The fact that DUA-DUA subsumption is not symmetric (row **Non-symmetric**) causes fault detection loss when it occurred in top revealing DUAs. Though detrimental to the fault detection ability of unconstrained DUAs, this situation rarely occurs (7 out of 784).

Methods with ill-formed flow graphs are easily identified statically by checking the flow graph structure. Testers can use all DUAs to test these methods. Program interruption and non-symmetric situations were identified in our experiment by checking whether the subsumed DUAs were covered when the unconstrained DUAs were. A testing tool could implement the same verification and inform the tester when a method has disrupted subsumption relationships. Therefore, the disruptions of the data flow subsumption can be identified and circumvented to avoid fault detection loss.

Considering the top revealing DUAs, a practitioner will seldom lose fault detection ability using unconstrained DUAs to create tests. Even in the rare situations when it can occur, the loss is preventable with appropriate tools.

### *6.8. RQ6: Value of data flow and control flow testing*

PC measures how sensitive a bug is to the coverage of a particular testing requirement ($tr$); that is, $PC$ gives the likelihood of revealing a fault given that $tr$ is covered in a test set. We investigated the DUAs with highest $PC$ ($PC_{max}$) and local DUA-node and local DUA-edge subsumptions. Our findings indicate that data flow testing will be especially valuable when DUAs with $PC_{max}$ are not subsumed either by nodes or by edges.

Let $D$ be a DUA required by a program $P$ such that $PC(D) = p$ and $D$ is subsumed by a control flow testing requirement (node or edge) $c$. As discussed in the previous section, data flow subsumptions are not symmetric. As result, covering $c$ implies covering $D$, but $D$ might be covered without $c$ being covered. However, considering only the top revealing DUAs in our pool, data flow subsumption disruption was a rare event–occurring only in seven bugs out of 784. Additionally, these events are unlikely when using local DUA-node and DUA-edge subsumptions. This is because

[§] $D_1$ subsuming $D_2$ ($D_2 \rightarrow D_1$) does not imply $D_2$ subsuming $D_1$ ($D_1 \rightarrow D_2$)



*Prepared using* stvrauth.cls
DOI: 10.1002/stvr

DUAs subsumed between the visit of a node or a edge to the end of the program are not taken into account.

Thus, considering local data flow subsumptions, a fault $f$ in our pool and DUAs $D$ with $PC_{max}$, we can approximate that $PC(c) \approx PC(D) = p$, if $c$ subsumes $D$. Figure 13 shows a bar chart One, subsumption **True** means that at least one top revealing DUA of the bug is subsumed by a node or edge. Two, subsumption **False** means that no node or edge subsumes the top revealing DUAs. Three, subsumption **not clear** means there are top revealing DUAs not subsumed by nodes and edges and the other top revealing DUAs are located in ill-formed methods, so we could not determine whether they are subsumed by nodes or edges.

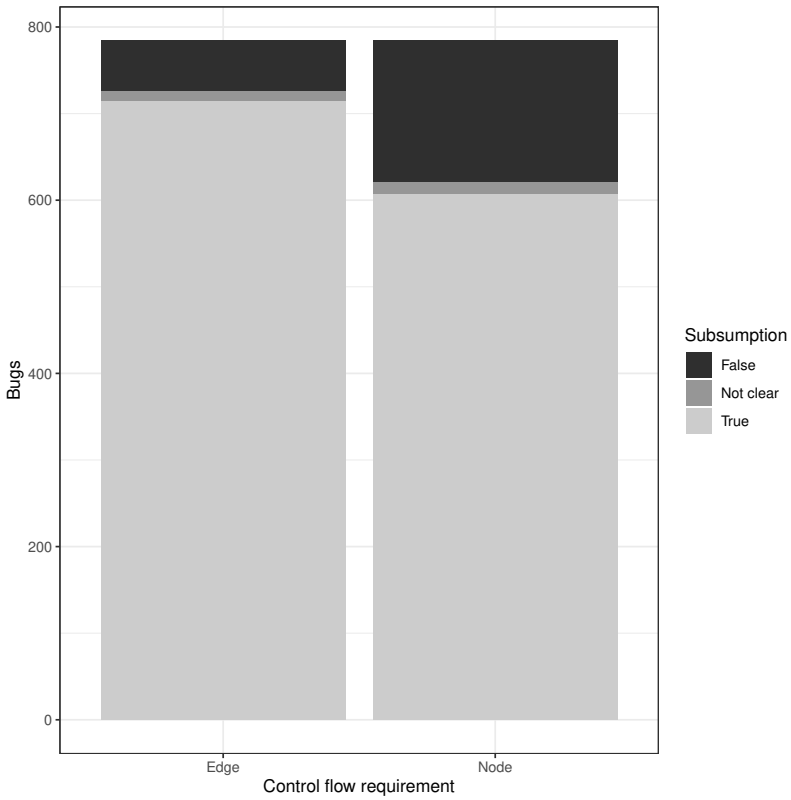


Figure 13. Bar chart of bugs whose top revealing DUAs are subsumed by node or edges

Black rectangles in Figure 13 give the bugs for which data flow testing may generate gains in terms of fault detection ability when compared with control flow testing. They indicate that about one fifth of bugs have top revealing DUAs that are not subsumed by nodes, and about 1/20 are not subsumed by edges.

Table VII gives the number of occurrences in each category for DUA-node and DUA-edge subsumptions, the estimated probability of data flow testing gains with respect to node and edge coverage, and additional statistical data. The estimated probability of data flow improving over node coverage is 21.14$\pm$2.88% and over edge coverage is 7.50$\pm$1.86%

The results suggest that data flow testing possible gains are modest, especially with respect to edge coverage. These gains are consistent with the effectiveness data in Section 6.5. We found that node and edge coverage subsume, on average, 65.4% and 77.4% of all DUAs of the programs. As most of the methods have a limited number of DUAs, many of which are subsumed by nodes and edges, bugs are more likely to be located in one of these methods than in methods with more than 100 DUAs, which are fairly rare. Nevertheless, the results also indicate that data flow testing can reveal faults that would go undetected if only control flow testing was used.



*Prepared using* stvrauth.cls
DOI: 10.1002/stvr

In this context, algorithms to find data flow subsumptions are useful to focus the practitioner's attention on relevant data flows; in particular, data flows not subsumed by the coverage of nodes and edges.

Table VII. Data flow testing probability of improving fault detection ability

| | Subsumption by Node | Subsumption by Edge |
|---|---|---|
| # Bugs with **no** top DUA subsumed (subsumption **False**) | 163 | 58 |
| # Bugs with at least one top DUA subsumed (subsumption **True**) | 608 | 715 |
| # Bugs with no clear subsumption (subsumption **not clear**) | 13 | 11 |
| Estimated probability | 21.14% | 7.50% |
| Inferior Confidence Interval | 18.26% | 5.65% |
| Superior Confidence Interval | 24.02% | 9.36% |

### *6.9. Limitations and threats to validity*

This section discusses the conclusion risks, and internal and external threats to the validity of our work. We addressed the external threat to validity by using the bugs in the Defects4J repository. Defects4J contains open-source programs that are comparable to industry programs, thus reducing external threat. To further reduce that threat, we also studied Elki, SystemDS, Weka, and CoreNLP because the mathematical nature of those software systems challenged the scalability of our algorithm.

An internal threat to validity is the correctness of our tools and algorithm. To address that threat, we verified the SA both formally with a proof and empirically. We use `SAtool` to compute the DUAs and `Jaguar` to check the coverage of the DUAs. The two tools use the same library to calculate the DUAs and to further ensure consistency between them, we wrote a script to compare the DUAs generated by the two tools. The data structures used in our SA implementation are mostly based on Java APIs. However, they may have inefficiencies that we are not aware of. Despite that, the algorithm's 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 those DUAs will be subsumed by the unconstrained DUAs. The programs and scripts created to generate probabilistic coupling data and to check data flow subsumption disruptions were verified for only a handful of bugs, which constitutes another internal threat to validity.

One conclusion threat is due to the use of $PC$ as a proxy of the fault detection ability of DUAs. $PC$ is an approximation because it does not capture the complex inter-dependencies between test requirements. Nevertheless, it has two advantages: (1) it does not require costly simulations to estimate fault-detection probabilities; (2) it is robust to noise introduced by irrelevant test goals and tests [7].

Additionally, the test pools used are in themselves a threat to the conclusion validity. Ideally, a test pool should include tests that cover 100% of the feasible DUAs several times. Even so, though less likely, an extra (failing or passing) test to this "ideal" test pool might change $PC_{max}$. In this sense, DUA coverage can serve as a rough assessment of test pool's adequacy.

Table VIII gives DUA coverage percentage for faulty versions used in our pool calculated using `Jaguar` and `BA-DUA`. It presents the following data flow coverage values per Defects4J project: minimum, first quartile, median, mean, third quartile, maximum, and the standard deviation. Research questions RQ5 and RQ6 are assessed using these faulty versions. The coverage did not vary much among the faulty versions of a project: the median and median values are similar and the standard deviation is small. The median and mean values were below 50% only for Chart, between 50% and 70% for three projects, and above 70% for 13. Although the test pools are away from the "ideal" pool, considering the size of the projects, a DUA coverage above 70% indicates a quite thorough test pool since we are taking into account all (feasible and infeasible) 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 used local subsumption, the impact



*Prepared using* stvrauth.cls
DOI: 10.1002/stvr

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. Nevertheless, the top revealing DUAs suggest that these are rare events and should have a limited impact on the results. It could be useful to experiment further with software that has a great deal of exception handling.

## 7. RELATED WORK

This paper addresses several issues regarding data flow testing and subsumption, including identifying, effectiveness, scalability, the fault detection ability of unconstrained DUAs, and yield of data flow testing. We discuss work related to each issue below.

### *7.1. Data flow subsumption discovery*

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 [46]. 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. A node $n_k$ is *control-dependent* on $n_i$ if and only if there exists a directed path $P$ from $n_i$ to $n_k$ and any $n_j$ in $P$ (excluding $n_i$ and $n_k$) is post-dominated by $n_k$ and $n_i$ is not post-dominated by $n_k$. If $n_k$ is control dependent on $n_i$, then $n_i$ has two outgoing edges: one that leads to $n_k$ execution and other that does not. CD($n_k$) represents the edge on which $n_k$ is control dependent [23].

$D$ is covered if one required edge for $d$ (that is, $d$ is control-dependent on this edge) 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. Local DUA-edge subsumption, in turn, provides similar information, DUAs inferable after edge coverage, at a much lower cost, $\approx$ O(|N|).

We have identified three previous algorithms to find DUA-DUA subsumption. Marré and Bertolino suggested two algorithms (referred to as M&B I [33] and M&B II [34, 35]), and Jiang et al. proposed another [23]. M&B I is based on rules establishing conditions under which whenever a test path covers $D_1(d_1, u_1, X_1)$, it also covers $D_2(d_2, u_2, X_2)$. Figure 14 displays the rules for subsumption of DUA $D_2$ by DUA $D_1$. The rules are based on the possible alignments of nodes $s$ (start node), $d_1$, $d_2$, $u_1$, $u_2$ and $e$ (exit node), on restrictions over the paths connecting these nodes, and on the definitions allowed to occur in nodes $d_1$ and $u_1$.

As an example of a subsumption rule, consider Rule 2 (item **2.** in Figure 14). Roughly speaking, to fulfill it, nodes $s$, $d_2$, and $d_1$ must be aligned and so must $d_1$, $u_2$, and $u_1$. Three nodes $n_i$, $n_j$, and $n_k$ are *aligned* if every path from node $n_i$ to $n_k$ contains $n_j$, denoted by AL($n_i$, $n_j$, $n_k$) [33]. Such a requirement aims imposes the ordering of Rule 2. Furthermore, all paths connecting $d_2$ and $d_1$ must be def-clear wrt $X_2$ and the paths connecting $d_1$ and $u_2$ that are def-clear wrt $X_1$ must also be def-clear wrt $X_2$. The final requirement for $D_1$ to subsume $D_2$ is that a definition of $X_2$ is not allowed to occur in node $d_1$.

By fulfilling these conditions, $D_1$ should in principle subsume $D_2$. Some conditions must be verified algorithmically:

- *Alignment of nodes*. Check whether three nodes $n_i$, $n_j$ e $n_k$ are aligned, denoted by AL($n_i$, $n_j$, $n_k$). For Rule 2, AL($s$, $d_2$, $d_1$) and AL($d_1$, $u_2$, $u_1$) should hold.



*Prepared using* stvrauth.cls
DOI: 10.1002/stvr

1. $s \ \ldots \ d_2 \ \ldots \ u_2 \ \ldots \ d_1 \ \ldots \ u_1 \ \ldots \ e$

2. $s \ \ldots \ d_2 \ \ldots \ \ldots \ d_1 \ \ldots \ u_2 \ \ldots u_1 \ \ldots \ e$

3. $s \ \ldots \ d_1 \ \ldots \ d_2 \ \ldots \ u_2 \ \ldots u_1 \ \ldots \ e$

4. $s \ \cdots \ d_1 \ \ldots \ u_1 \ \ldots \ d_2 \ \ldots \ u_2 \ \ldots \ e$

5. $s \ \ldots \ d_1 \ \ldots \ d_2 \ \ldots \ \ldots \ u_1 \ \ldots u_2 \ \ldots \ e$

6. $s \ \cdots \ d_2 \ \ldots \ \ldots \ d_1 \ \ldots \ \ldots \ u_1 \ \ldots u_2 \ \ldots \ e$

Figure 14. Subsumption rules for $D_1$= ($d_1$, $u_1$, $X_1$) and $D_2$ = ($d_2$, $u_2$, $X_2$) [33].

- *All paths def-clear wrt $X$*. Check whether every path from a node $n_i$ to node $n_j$ is def-clear wrt variable $X$, denoted AllDefClear($n_i$, $n_j$, $X$). Regarding Rule 2, AllDefClear($d_2$, $d_1$, $X_2$) should be true.
- *Any path simultaneously def-clear wrt $X$ and $Y$*. Check whether any path from $n_i$ to node $n_j$ that is def-clear wrt $X$ is def-clear wrt to $Y$ as well, denoted AnySimDefClear($n_i$, $n_j$, $X$,$Y$). Considering rule 2, AnySimDefClear($d_1$, $u_2$, $X_1$,$X_2$) should hold.

The formal description of all six rules [33] for the subsumption of $D_1$($d_1$, $u_1$, $X_1$) by $D_2$($d_2$, $u_2$, $X_2$) is presented below.

1. AllDefClear($d_2$, $u_2$, $X_2$) *AND* AL($s$,$u_2$,$d_1$) *AND* AL ($s$, $d_2$, $u_2$); ***OR***
2. (a) $X_1 \neq X_2$ *AND* AllDefClear($d_2$, $d_1$, $X_2$) *AND* AnySimDefClear($d_1$, $u_2$, $X_1$, $X_2$) *AND* $X_2$ is not defined in $d_1$ *AND AND* AL ($d_1$, $u_2$, $u_1$); ***OR***
   (b) $X_1 = X_2$ *AND* $d_1$ = $d_2$ *AND* AL ($d_1$, $u_2$, $u_1$); ***OR***
3. (a) $X_1 \neq X_2$ *AND* AnySimDefClear($d_2$, $u_2$, $X_1$, $X_2$) *AND* AL($d_1$, $d_2$, $u_1$) *AND* AL($d_2$,$u_2$,$u_1$); ***OR***
   (b) $X_1 = X_2$ *AND* $d_1 = d_2$ *AND* AL($d_1$,$u_2$,$u_1$); ***OR***
4. AllDefClear($d_2$, $u_2$, $X_2$) *AND* AL($u_1$,$d_2$,$e$) *AND* AL ($d_2$, $u_2$, $e$); ***OR***
5. (a) $X_1 \neq X_2$ *AND* AnySimDefClear($d_2$, $u_1$, $X_1$, $X_2$) *AND* AllDefClear($u_1$, $u_2$, $X_2$) *AND* $X_2$ is not defined in $u_1$ *AND* AL($d_1$, $d_2$, $u_1$) *AND* AL($u_1$, $u_2$, $e$); ***OR***
   (b) $X_1 = X_2$ *AND* $d_1 = d_2$ *AND* $X_2$ is not defined $u_1$ *AND* AllDefClear($u_1$, $u_2$, $X_2$) *AND* AL($u_1$,$u_2$,$e$); ***OR***
6. (a) $X_1 \neq X_2$ *AND* AllDefClear($d_2$, $d_1$, $X_2$) *AND* AnySimDefClear($d_1$, $u_1$, $X_1$, $X_2$) *AND* $X_2$ is not defined in $d_1$ *AND* AllDefClear($u_1$, $u_2$, $X_2$) *AND* $X_2$ is not defined in $u_1$ *AND* AL($e_0$,$d_2$,$d_1$) *AND* AL($u_1$, $u_2$, $e$); ***OR***
   (b) $X_1 = X_2$ *AND* $d_1$ = $d_2$ *AND* $X_2$ is not defined in $u_1$ *AND* AllDefClear($u_1$, $u_2$, $X_2$) *AND* AL($u_1$,$u_2$,$e$).

Marré and Bertolino [33] show that AL($n_i$, $n_j$, $n_k$), AllDefClear($n_i$, $n_j$, $X$), and AnySimDefClear($n_i$, $n_j$, $X$, $Y$) can be checked at a cost of O(|E|) where E is the number of edges



*Prepared using* stvrauth.cls
DOI: 10.1002/stvr

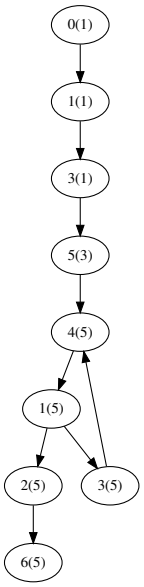


Figure 15. $G*$ for DUA (3, 5, rogue)

of the program flow graph. Since each DUA is verified against every other DUA, the total cost for DUA-DUA subsumption verification is O($|\mathrm{U}|^2|\mathrm{E}|$), where U is the number of duas.

Unfortunately, the rules described above miss paths that might occlude DUA-DUA subsumptions. Consider the subsumption of $D_2$(0, 5, i) by $D_1$(3, 5, rogue) of the Max program example. For the example, $s = 0$, $d_1 = 3$, $u_1 = 5$, $X_1$ = *rogue*, $d_2 = 0$, $u_2 = 5$, and $X_2 = i$. All conditions of Rule 2(a) are fulfilled as shown below.

- $X_1 \neq X_2$ is true since variable $i$ is different from variable *rogue*
- AllDefClear($d_2$, $d_1$, $X_2$) and AnySimDefClear($d_1$, $u_2$, $X_1$, $X_2$) are true because AllDefClear(0, 3, $rogue$) and AllDefSimClear(3, 5, $rogue$, $i$) are, respectively, true
- AL($s$,$d_2$,$d_1$) and AL($d_1$, $u_2$, $u_1$) are both true since AL(0, 0, 3) and AL(3, 5, 5) are true
- Finally, $X_2$ (variable $i$) is not defined in $d_1$ (node 3)

Thus, according to Rule 2(a), $D_2$(0, 5, i) is subsumed by $D_1$(3, 5, rogue). However, Rule 2(a) allows test path (0,1,3,4,1,3,5,4,1,2,6) when it should not. Such a path covers $D_1$ but does not $D_2$. In this path, $d_1$ (in the example, node 3) occurs twice, but that possibility is not discarded by condition AnySimDefClear($d_1$, $u_2$, $X_1$, $X_2$) of Rule 2(a). Thus, M&B I does not account for *back paths*, such as (3,4,1,3), to avoid finding incorrect DUA-DUA subsumptions.

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 $G*$. Figure 15 shows $G*$ $D_1$(3, 5, rogue) as proposed by Marré and Bertolino [34, 35]. 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 II runs in time $O(|\mathrm{U}|^2|\mathrm{N}|)$ [33, 34].

$G*$ 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 $u_1$. That is, it does not include the sub-paths encoded by graphdua's sub-graphs *SG2* and *SG4*. By comparing graphdua for $D_1$(3, 5, rogue) (Figure 7) and $G*$ for $D_1$(3, 5, rogue) (Figure 15), one can observe that a graphdua encodes many more paths than $G*$.




DOI: 10.1002/stvr

Consider again 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 $G*$ also 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 [23] algorithm for DUA-DUA subsumption is based on the concepts of *def-use order* and *control dependency* [46]. 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) CD($d_1$) $\bigcup$ CD($u_1$) $\supseteq$ CD($d_2$) $\bigcup$ CD($u_2$), where 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), which subsumes $D_2$(0, 5, i) according to this technique. 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). Jiang et al. did not discuss complexity, but it is at least O(|U|$^2$|N|, since every DUA is checked against every other and one has to find dominance relationship of nodes.

All three prior approaches do not find DUA-DUA subsumptions because they cannot encode back paths (encoded by graphdua's *SG2* and *SG4*) that might block the subsumption relationships. However, they can be fixed. M&B I and Jiang et al. can include back paths in their rules and M&B II will work if a graphdua is used instead of a $G*$.

Still, the "fixed" algorithms could not find DUA-DUA subsumption as efficiently as our algorithm can. We apply SA, the algorithm that finds local DUA-node subsumption, on graphduas. In doing so, we find all DUAs subsumed by a particular DUA $D_1$ at once. As a result, we find all DUAs subsumed by $D_1$ in $\approx O(|\mathrm{N}|)$ time, and all DUA-DUA subsumptions in $\approx O(|\mathrm{U}||\mathrm{N}|)$ time. On the other hand, M&B I, M&B II, and Jiang et al. cannot find DUA-DUA subsumptions in time less than O(|U|$^2$|N| because they find subsumption relationships on a DUA-by-DUA basis. Our experimental data suggests that costs increase significantly with algorithms that are quadratic in the number of DUAs.

We have one final remark on how SA allows efficient calculation of data flow subsumptions. We developed the *Data Flow Subsumption Framework* (DFS) [4], which assigns to a node the DUAs subsumed or available to be subsumed whenever it is reached. SA uses DFS to quickly calculate local DUA-node subsumption ($\approx O(|\mathrm{N}|)$). Using local DUA-node subsumption, we can then efficiently find local DUA-edge ($\approx O(|\mathrm{N}|)$) and DUA-DUA subsumption ($\approx O(|\mathrm{U}||\mathrm{N}|)$). Our approach efficiently calculates data flow subsumptions because SA processes several DUAs at once, whereas prior techniques find subsumption relationships in a DUA-by-edge [46] or DUA-by-DUA basis [23, 34–36].

### *7.2. Effectiveness*

Jiang et al. presents an evolutionary approach to generating input data for data flow testing called Evolutionary Data flow Testing Generation (EDTG) [23]. Their algorithm for data flow subsumption first reduces the number of DUAs. EDTG then uses a genetic algorithm (GA) with a fitness function tailored for DUA coverage to generate input data for the reduced collection of DUAs. Comparison between EDTG and classic data flow test generation (CDTG) shows that they both use the same fitness function, but EDTG uses unconstrained DUAs for input data generation.

Jiang et al.'s results are similar to those presented in Section 6.5, but the EDTG algorithm does not include paths that could block the subsumption relationship. They found a reduction on the number of DUAs varying from 19% to 63%, in classes requiring a relatively small number of DUAs (a couple of hundreds on average). Our assessment uses industry-like programs (Table I) that implement mathematical functions, few of them with more than 2000 DUAs. It further corroborates the evidence that unconstrained DUAs substantially reduce the number of DUAs to be tested:



*Prepared using* stvrauth.cls
DOI: 10.1002/stvr

Table VIII. DUA coverage percentage for Defect4J's faulty versions per program

| Program | Min | 1st Qu. | Median | Mean | 3rd Qu. | Max. | Std. dev. |
|---|---|---|---|---|---|---|---|
| Chart | 41.74 | 45.12 | 46.20 | 45.91 | 46.49 | 48.17 | 1.27 |
| Cli | 76.10 | 85.77 | 90.44 | 87.36 | 90.98 | 93.50 | 5.71 |
| Closure | 59.22 | 69.59 | 74.10 | 73.49 | 76.55 | 79.94 | 4.06 |
| Codec | 65.05 | 73.59 | 75.29 | 74.91 | 77.57 | 78.83 | 3.85 |
| Collections | 75.79 | 75.80 | 76.22 | 76.32 | 76.74 | 77.05 | 0.63 |
| Compress | 59.18 | 66.86 | 70.30 | 69.87 | 73.87 | 75.82 | 4.22 |
| Csv | 81.47 | 85.65 | 86.23 | 85.70 | 86.37 | 86.75 | 1.32 |
| Gson | 71.61 | 75.85 | 76.13 | 75.95 | 76.56 | 76.69 | 1.13 |
| J-Core | 55.99 | 57.58 | 58.97 | 59.39 | 60.38 | 63.86 | 2.36 |
| J-Databind | 61.71 | 62.86 | 63.44 | 64.14 | 63.78 | 69.54 | 2.38 |
| J-Xml | 63.20 | 68.19 | 70.06 | 71.43 | 74.57 | 84.08 | 5.11 |
| Jsoup | 63.20 | 68.19 | 70.06 | 71.43 | 74.57 | 84.08 | 5.11 |
| JxPath | 56.39 | 56.67 | 56.77 | 58.49 | 56.91 | 66.69 | 3.84 |
| Lang | 85.24 | 85.92 | 86.27 | 86.30 | 86.95 | 87.08 | 0.67 |
| Math | 76.40 | 77.95 | 79.76 | 80.29 | 82.86 | 83.73 | 2.60 |
| Mockito | 70.36 | 73.50 | 80.92 | 78.64 | 83.03 | 83.75 | 4.76 |
| Time | 79.12 | 79.65 | 79.84 | 79.76 | 79.95 | 80.21 | 0.34 |

around 30% of the total. Additionally, we evaluated the savings from DUA-node and DUA-edge subsumptions: 65.4% of DUAs are subsumed by node coverage and 77.4% by edge coverage.

### *7.3. Scalability*

Previous papers on data flow subsumptions [23, 35, 46] do not address the cost of calculating the subsumption relationship. Santelices and Harroldd [46] address the overhead reduction on program instrumentation by using edge coverage to infer data flow coverage. They calculate the asymptotic complexity of their algorithm ($O(|\mathrm{U}|)$), but do not provide experimental data. Marré and Bertolino's [35] implementation of their M&B II algorithm included a manual step, making algorithm complexity less relevant. Jiang et at. [23] do not provide scalability data either. Furthermore, the programs they used are not complex (hundreds of DUAs) enough to evaluate whether their algorithm scales.

### *7.4. Fault detection ability of unconstrained DUAs*

Marré and Bertolino [35] ran an experiment to assess the fault detection ability of test sets created using unconstrained DUAs. The experiment used five programs from the Siemens test suite [21], Replace, Schedule, Schedule 2, Tcas, and Totinfo. An ad hoc initial test set that covered 50% of the DUAs[¶] was developed for all selected programs. New test sets were then created by adding test cases to the initial test set to obtain coverage increases of 5% until reaching 95% coverage. Two strategies are used to increase the coverage: random selection (RA) of test cases; and addition of a new test case only if it covers a new unconstrained DUA (UD) until reaching the desired coverage in both cases.

The fault detection ability of the new test cases were assessed by the number of faults detected (ND), the number of test cases needed for each coverage level (TS), and the fault density (FD)

[¶]An unconstrained set of edges was also assessed in [35], although we report only the data regarding data flow testing.



*Prepared using* stvrauth.cls
DOI: 10.1002/stvr

given by the ratio between ND and TS. The results suggest that there is no significant difference up to 85% of coverage in terms of number of faults detected. However, as coverage grows, the difference of faults detected becomes evident in favor of random selection (RA) test sets, with statistical significance for test sets with coverage over 85%. On the other hand, RA needs more test cases to reach the same coverage as unconstrained DUA (UD). UD is more efficient in terms of FD than RA for test sets covering 85% or over, which means that UD detects more faults with fewer test cases.

Marré and Bertolino assess the fault detection ability of test sets developed using unconstrained DUAs and test sets created using random selection to improve data flow coverage. They found that random selection impacted the size of the test sets, making the comparison difficult. Chen et al. [7] claim that random selection is not a good approximation of the test creation process in practice.

We used probabilistic coupling to avoid confounding test goals to compare the fault detection ability of unconstrained DUAs and all DUAs. Additionally, the subjects of the two assessments differ largely in scale: they used five small programs of the Siemmens test suite [21], whereas we used 784 larger programs from the Defects4J repository.

### *7.5. Value of data flow testing and control flow testing*

Comparing fault detection ability of testing criteria has proved a challenging task, which has often let to contradictory results [7]. Several studies compared data flow and control flow testing [13,14,18,21]. Most of them were conducted years ago, used small programs [13,21], and differed in experimental design, specific data flow criteria used, and tools (ASSET [12], ATAC [19], Tactic [41], DUAforensics [46]). With the exception of Hemmati's study [18], none of the previous studies establish were conclusive in terms of which criteria was strong.

Hemmati [18] compared control flow criteria (statement, branch, loop, and MCDC coverage) against definition-use pair coverage with respect to their ability to detect faults. He found that out of 274 faults in a subset of the Defects4J repository bugs, only 76 (28%) were found by control flow coverage criteria. Definition-use pair coverage detected 79% of the faults not detected by control flow criteria. He only applied data flow analysis on bugs where no control flow criteria were effective in detecting the faults, we do not know if control flow criteria could detect bugs that data flow could not.

Our assessment of the yield of data flow testing is based on DUA coverage, and DUA-node and DUA-edge subsumption and probabilistic coupling as an approximation of the fault detection ability. Hemmati's is based on actual control flow coverage, but uses fewer than half the number of bugs we use. While our assessment suggests modest gains for data flow over control flow, Hemmati's study indicates significant improvement in fault detection ability. Despite the differences of bug sets and comparison measures used, we hypothesize that the use of inter-procedural data flow testing implemented by DUAforensics might have played a role in leveraging data flow against control flow since we only used intra-procedural data flow testing. Though the issue regarding data flow and control testing value is not settled, the impact of inter-procedural data flow testing [24] on fault detection ability needs further investigation.

## 8. CONCLUSIONS

Some studies indicate that data flow testing (DFT) can detect faults missed by control flow testing (CFT) [18]. 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 and effective test suites [35, 46]. Thus, the subsumption relationship can help make data flow testing practical. Nevertheless, several issues remained to be addressed: including subsumption discovery, effectiveness, scalability, fault detection ability of unconstrained DUAs, and yield of data flow testing.



*Prepared using* stvrauth.cls
DOI: 10.1002/stvr

Reliably identifying subsumption for definition-use associations is a difficult problem, whose early solutions have subtle flaws and inefficient algorithms. We tackled the data flow subsumption problem by modeling it as a 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, we can then find other data flow (DUA-node, DUA-edge, and DUA-DUA) subsumptions at costs that are substantially lower than other algorithms.

We investigated the effectiveness and scalability of the data flow subsumptions by applying it on programs similar to those developed in the industry. Our experimental data suggest that DFT costs can be reduced significantly by identifying data flow subsumption and then eliminating redundant requirements. Additionally, the cost to calculate the subsumption relationship using the algorithms presented in this paper was less than 2 minutes for programs as large as 200,000 lines of code. Very large and complex programs, however, can take as much as 18 min to calculate data flow subsumptions due to particular program units. These outliers can be easily identified using static metrics (for example, number of DUAs, cyclomatic complexity, nested loops, etc.). Despite that, our data suggest that data flow subsumption can be cost-effective in practice and incorporated directly into interactive development environments.

We used a pool of 784 faults to investigate the loss of fault detection ability when using the unconstrained DUAs—the subset of DUAs most relevant for testing identified using subsumption—to develop tests. Our results suggest that the chance of a practitioner losing fault detection ability is less than 2%. Furthermore, one can prevent these situations with appropriate tooling. We found that data flow testing is more effective than control flow testing, although by a modest amount: around 20% over node coverage and 7.5% over edge coverage. Nevertheless, these unique fault revealing data flows are found at low cost, even for large programs.

Data flow testing was introduced to add precision to software testing [42]. The subsumption relationship captures both the strengths and weaknesses of data flow testing. More importantly, the practitioner can make use of it to focus on data flows that add more value to testing.

We have several plans for extending this research. An important question is which coverage criteria add the most value to the practitioner. We hypothesize that the edge-pair coverage [3, 40] would subsume more data flows at low cost in the intra-procedural context. We plan to address different use-case scenarios when comparing edge, edge-pairs, unconstrained DUAs, and all DUAs in terms of fault detection ability. Another line of research is to explore the subsumption and fault detection ability of inter-procedural data flows. This might be particularly important since the real value of data flow testing might reside in the inter-procedural realm. Spectrum-based fault localization techniques can benefit from data flow subsumptions to select relevant spectra. Finally, we hope to carry out an empirical comparison of our subsumption algorithm with the algorithms by M&B I [33], M&B II [34, 35], and Jiang et al. [23] to further evaluate the differences between the subsumption algorithms.

## ACKNOWLEDGMENT

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

## REFERENCES

1. Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools*. Pearson Addison-Wesley, Boston, 2 edition, 2007.
2. Paul Ammann, Marcio E. Delamaro, and Jeff Offutt. Establishing theoretical minimal sets of mutants. In *7th IEEE International Conference on Software Testing, Verification, and Validation (ICST)*, pages 21–30, Cleveland, OH, March 2014.
3. Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, Cambridge, UK, 2nd edition, 2017.



DOI: 10.1002/stvr

4. Marcos Lordello Chaim, Kesina Baral, and Jeff Offutt. A data flow analysis framework for data flow subsumption. *CoRR*, abs/2101.05962, 2021.
5. Marcos Lordello Chaim, Kesina Baral, Jeff Offutt, Mario Concilio, and Roberto P. A. Araujo. Efficiently finding data flow subsumptions. In *14th IEEE Conference on Software Testing, Verification and Validation, ICST 2021, Porto de Galinhas, Brazil, April 12-16, 2021*, pages 94–104. IEEE, 2021.
6. Marcos Lordello Chaim and Roberto Paulo Andrioli de Araujo. An efficient bitwise algorithm for intra-procedural data-flow testing coverage. *Information Processing Letters*, 113(8):293–300, 2013.
7. Yiqun T. Chen, Rahul Gopinath, Anita Tadakamalla, Michael D. Ernst, Reid Holmes, Gordon Fraser, Paul Ammann, and René Just. Revisiting the relationship between fault detection, test adequacy criteria, and test set size. In *35th IEEE/ACM International Conference on Automated Software Engineering (ASE '20)*, ASE, 2020.
8. J. J. Chilenski and S. P. Miller. Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal*, 9(5):193–200, 1994.
9. Lori A. Clarke, Andy Podgurski, Debra J. Richardson, and Steven J. Zeil. A comparison of data flow path selection criteria. In *Proceedings of the 8th International Conference on Software Engineering*, ICSE '85, page 244–251, Washington, DC, USA, 1985. IEEE Computer Society Press.
10. Thanh-Binh Dao and Etsuya Shibayama. Security sensitive data flow coverage criterion for automatic security testing of web applications. In *Engineering Secure Software and Systems*, ESSoS, pages 101–113, 2011.
11. Roberto Paulo Andrioli de Araujo and Marcos Lordello Chaim. Data-flow testing in the large. In *Seventh IEEE International Conference on Software Testing, Verification and Validation, ICST*, pages 81–90, Cleveland, Ohio, USA, April 2014.
12. F. G. Frankl, S. N. Weiss, and E. J. Weyuker. ASSET: A system to select and evaluate tests. In *Proceedings of the IEEE Conference on Software Tools*, pages 72–79. IEEE Computer Society Press, Los Alamitos, California, USA, 1985.
13. P. Frankl and S. Weiss. An experimental comparison of the effectiveness of branch testing and data flow testing. *IEEE Transaction on Software Engineering*, 19(8):774–787, 1993.
14. P. G. Frankl and O. Iakounenko. Further empirical studies of test effectiveness. In *ACM SIGSOFT Foundations of Software Engineering Conference*, FSE 1998, pages 153–162, 1998.
15. P. G. Frankl and E. J. Weyuker. An applicable family of data flow testing criteria. *IEEE Transactions on Software Engineering*, 14(10):1483–1498, October 1988.
16. Mats Grindal, Jeff Offutt, and Sten F. Andler. Combination testing strategies: A survey. *Software Testing, Verification, and Reliability*, 15(3):167–199, 2005.
17. M. S. Hecht. *Flow analysis of computer programs*. Elsevier North-Holland, New York, 1977.
18. H. Hemmati. How effective are code coverage criteria? In *International Conference on Software Quality, Reliability, and Security*, pages 151–156. IEEE, August 2015.
19. J. R. Horgan and Saul London. A data flow coverage testing tool for C. In *Proc. of Symposium on Assessment of Quality Software Development Tools*, pages 2–10, 1992.
20. S. Horwitz, A. Demers, and T. Teitelbaum. An efficient general iterative algorithm for dataflow analysis. *Acta Informatica*, 24:679–694, 1987.
21. Monica Hutchins, Herb Foster, Tarak Goradia, and Thomas Ostrand. Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *16th International Conference on Software Engineering*, ICSE, pages 191–200, 1994.
22. Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5):649–678, 2011.
23. 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*, 12(4):318–323, 2018.
24. Zhenyi Jin and Jeff Offutt. Integration testing based on software couplings. In *Proceedings of the Tenth Annual Conference on Computer Assurance (COMPASS 95)*, pages 13–23, Gaithersburg, MD, June 1995. IEEE Computer Society Press.
25. René Just, Darioush Jalali, and Michael 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*, pages 437–440, July 2014.
26. J. B. Kam and J. D. Ullman. Monotone data flow frameworks. *Acta Informatica*, 7:305–317, 1977.
27. Donald E. Knuth. An empirical study of FORTRAN programs. *Software: Practice and Experience*, 1(2):105–133, 1971.
28. Bob Kurtz, Paul Ammann, Marcio E. Delamaro, Jeff Offutt, and Lin Deng. Mutant subsumption graphs. In *Tenth IEEE Workshop on Mutation Analysis (Mutation)*, Cleveland, OH, March 2014.
29. Bob Kurtz, Paul Ammann, and Jeff Offutt. Static analysis of mutant subsumption. In *Eleventh IEEE Workshop on Mutation Analysis (Mutation)*, Graz, Austria, April 2015.
30. Bob Kurtz, Paul Ammann, Jeff Offutt, Marcio E. Delamaro, Mariet Kurtz, and Nida Gökće. Analyzing the validity of selective mutation with dominator mutants. In *24th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*, Seattle Washington, USA, November 2016.
31. Bob Kurtz, Paul Ammann, Jeff Offutt, and Mariet 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.
32. J. Laski and B. Korel. A data flow oriented program testing strategy. *IEEE Transactions on Software Engineering*, SE-9(3):347–354, 1983.
33. 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*, pages 147–157, New York, USA, 1996. ACM Press.
34. Martina Marré. *Program Flow Analysis for Reducing and Estimating the Cost of Test Coverage Criteria*. PhD thesis, Dep. de Computacion, FCEyN – Universidad de Buenos Aires, Argentina, 1997.




DOI: 10.1002/stvr

35. Martina Marré and Antonia Bertolino. Using spanning sets for coverage testing. *IEEE Transactions on Software Engineering*, 29(11):974–984, 2003.
36. D. I. S. Marx and P. G. Frankl. The path-wise approach to data flow testing with pointer variables. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 135–146, New York, USA, 1996. ACM Press.
37. Mario Concilio Neto, Roberto P. A. Araujo, Marcos Lordello Chaim, and Jeff Offutt. Graph representation for data flow coverage. In *IEEE 45th Annual Computers, Software, and Applications Conference, COMPSAC 2021, Madrid, Spain, July 12-16, 2021*, pages 952–961. IEEE, 2021.
38. Simeon C. Ntafos. On required element testing. *IEEE Transactions on Software Engineering*, 10(6):795–803, 1984.
39. Jeff Offutt and Aynur Abdurazik. Generating tests from UML specifications. In *Proceedings of the 2nd International Conference on The Unified Modeling Language: Beyond the Standard*, UML'99, page 416–429, Berlin, Heidelberg, 1999. Springer-Verlag.
40. Jeff Offutt, Shaoying Liu, Aynur Abdurazik, and Paul Ammann. Generating test data from state-based specifications. *Software Testing, Verification, and Reliability, Wiley*, 13(1):25–53, March 2003.
41. T. J. Ostrand and E. J. Weyuker. Data flow-based test adequacy analysis for languages with pointers. In *Proceedings of the Symposium on Software Testing, Analysis and Verification – TAV4*, pages 74–86, 1991.
42. Sandra Rapps and Elaine Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, 11(4):367–375, April 1985.
43. Henrique Lemos Ribeiro, Roberto Paulo Andrioli de Araujo, Marcos Lordello Chaim, Higor Amario de Souza, and Fabio 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*, pages 1–11. IEEE, 2019.
44. Henrique Lemos Ribeiro, Higor Amario de Souza, Roberto Paulo Andrioli de Araujo, Marcos Lordello Chaim, and Fabio 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*, pages 404–409. IEEE Computer Society, 2018.
45. Barbara G. Ryder and Marvin C. Paull. Elimination algorithms for data flow analysis. *ACM Computing Surveys*, 18(3):277–316, September 1986.
46. Raul Santelices and Mary Jean Harrold. Efficiently monitoring data-flow test coverage. In *22nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 343–352, 2007.
47. Raul Santelices, James A. Jones, Yanbing Yu, and Mary Jean Harrold. Lightweight fault-localization using multiple coverage types. In *Proc. of the 31st International Conference on Software Engineering*, ICSE, pages 56–66, 2009.
48. H. Ural and B. Yang. A structural test selection criterion. *Information Processing Letters*, 28:157–163, 1988.
49. Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4):366–427, December 1997.





DOI: 10.1002/stvr