Improving Fault Localization by Integrating Value and Predicate Based Causal Inference Techniques

Yiğit Küçük,
Department of Computer and Data Sciences
Case Western Reserve University
Cleveland, OH, USA
yxk368@case.edu

Tim A. D. Henderson,
Google Inc.
Mountain View, CA, USA
tadh@google.com

Andy Podgurski
Department of Computer and Data Sciences
Case Western Reserve University
Cleveland, OH, USA
podgurski@case.edu

***Abstract*—Statistical fault localization (SFL) techniques use execution profiles and success/failure information from software executions, in conjunction with statistical inference, to automatically score program elements based on how likely they are to be faulty. SFL techniques typically employ one type of profile data: either coverage data, predicate outcomes, or variable values. Most SFL techniques actually measure correlation, not causation, between profile values and success/failure, and so they are subject to confounding bias that distorts the scores they produce. This paper presents a new SFL technique, named *UniVal*, that uses causal inference techniques and machine learning to integrate information about both predicate outcomes and variable values to more accurately estimate the true failure-causing effect of program statements. *UniVal* was empirically compared to several coverage-based, predicate-based, and value-based SFL techniques on 800 program versions with real faults.**

I. Introduction

There has been a vast amount of research on *automated software fault localization* (AFL) [1]–[4], which seeks to automate all or part of the process of locating the software faults that are responsible for observed software failures. *Statistical fault localization* (SFL) or *spectrum-based fault localization* [1], [2], [4], [5] comprises a large body of AFL techniques that apply statistical measures of association — some computed with machine learning or data mining techniques — to execution data (execution profiles or *spectra*) and to failure data (e.g., pass/fail labels) in order to compute putative measures, called *suspiciouness scores*, of the likelihood that individual program statements or other program elements are responsible for observed failures. These scores are used to guide developers to faults, e.g., by using them to highlight suspicious statements in graphical displays of code [1] or to rank statements for inspection [4]. Statistical fault localization is also the first step in a number of automated program repair techniques [6], [7].

SFL techniques are applicable when the data they require are readily and cheaply available in sufficient quantity. In particular, they generally require data from a diverse and penetrating set of tests or operational executions that includes significant numbers of both failures and successes. Such data might be available, for instance, from deployed software that is instrumented to record profile data and that is equipped with a mechanism by which users may report failures they encounter.

It seems fair to say that statistical fault localization research stands at a crossroads. Although a large number of SFL techniques have been proposed, to our knowledge none of them consistently locates faults with enough precision to justify its widespread use in industry. In part, this is because most such techniques rely on just one source of information about internal program behavior: code coverage profiles, or, similarly, recorded outcomes of program predicates such as those that control the execution of conditional branches and loops. Recent work has sought to overcome this limitation, e.g., by employing *value profiles/spectra* (profiles of variable values) [8]–[10] or by combining different kinds of information pertinent to fault localization, such as coverage-based SFL scores, textual similarity measures, and fault-proneness predictions based on static program analysis [11]–[14]. Another problem with existing SFL techniques is that many of them suffer from *confounding bias*, which can cause a correct statement to appear suspicious because of another, faulty statement that influences its execution. Recent work has also sought to overcome this problem by employing *causal inference* techniques [15]–[19].

Existing SFL techniques are based on analysis of code coverage profiles or predicate profiles, on one hand, or of value profiles, on the other hand. To our knowledge, no previous technique is both coverage-based (or predicate-based) and value-based. While two techniques, one of each kind, can be combined simply by taking the average or maximum of the scores they produce for each potential fault location, this will not in itself properly address confounding bias.

This paper proposes a new approach to statistical fault localization that integrates information about both predicate outcomes and variable values, and that does so in a principled way that controls for confounding bias. The key to this approach, which we call *UniVal*, is to transform the program under analysis so that branch and loop predicate outcomes become variable values, so that one causally sound value-based SFL technique can be applied to both variable assignments and predicates. This paper reports on an large-scale empirical evaluation of *UniVal* involving the latest version (2.0.0) of the widely used Defects4J evaluation framework [20], which contains seventeen software projects and 835 program versions containing actual faults. In this study, *UniVal* was compared to several coverage-based, predicate-based, and value-based SFL techniques. *UniVal* substantially out-performed each of these

techniques. To the best our knowledge this is the only fault localization study that has used all the programs in the latest version of Defects4J, and it uses the largest number of real programs and faults of any study. We also report empirical results characterizing the relationship between the cost of fault localization and an important property of data in a causal inference study called *covariate balance*. These results help explain the effectiveness of *UniVal*.

II. Motivating Example

To illustrate our technique, we now apply *UniVal* to locate a real software fault. This example is from the 62nd faulty version of Google's Closure Compiler (`Closure 62`) from the widely used Defects4J [20] repository (v2.0). This fault exists in the `format` method, which is located between lines 66-111 in the `LightweightMessageFormatter` class. The faulty segment of the code is depicted in Listing 1 with some elements omitted for brevity, and with the lines of code renumbered.

The method `format` is intended to compose an error message for a script written in Javascript. As input parameters, `format` takes a `JSError` object named `error`, which contains information about the nature of the error, and a boolean value named `warning`, which indicates whether the error message should be formatted as a warning message. However, this method has a faulty predicate at line 5 ("$<$" is used instead of "$\leq$") that prevents the error column number, `charno`, from being displayed correctly for some inputs to the method. The fix for the faulty predicate expression is shown in a comment on line 4.

```
1  private String format(JSError error, boolean warning) {
2    ...
3    int charno = error.getCharno(); // Column number within line
4    // Fix: && charno <= sourceExcerpt.length()
5    if (excerpt.equals(LINE) && 0 <= charno && charno <
         sourceExcerpt.length()) {
6      for (int i = 0; i < charno; i++) {
7        char c = sourceExcerpt.charAt(i);
8        if (Character.isWhitespace(c)) {
9          b.append(c);
10       } else {
11         b.append(' ');
12       }
13     }
14     b.append("^\n");
15   }
16   ...
```

Listing 1: Original code for the motivating example

```
1  private String format(JSError error, boolean warning) {
2    ...
3    int charno = error.getCharno();
4    boolean P3_0 = false;
5    boolean P3_1 = false;
6    boolean P3_2 = false;
7    if ((P3_2 = (excerpt.equals(LINE)))
8         && (P3_1 = (0 <= charno))
9         && (P3_0 = (charno < sourceExcerpt.length()))) {
10     boolean P4_0 = false;
11     for (int i = 0; (P4_0 = (i < charno)); i++) {
12       char c = sourceExcerpt.charAt(i);
13       boolean P5_0 = false;
14       if ((P5_0 = (Character.isWhitespace(c)))) {
15         b.append(c);
16       } else {
17         b.append(' ');
18       }
19     }
20     b.append("^\n");
21   }
22   ...
```

Listing 2: Predicates in Listing 1 extracted to boolean variables.

The code in Listing 1 illustrates how confounding bias [21] may arise in fault localization: the outcome of the faulty predicate at line 5 confounds the causal effects on program failure of the statements at lines 6-14. This implies that unless we adjust for confounding, the suspiciousness scores calculated for the variables and predicates located at those lines are likely to be biased and distorted.

In order to apply *UniVal* and other fault localization techniques to the code in Listing 1, we first used a tool we developed named *PredicateTransformer* to transform the predicates in branch and loop conditions into assignment statements that assign the results of evaluating the predicates to new boolean variables. Note that each predicate in a compound boolean expression is transformed in this way. The transformed code is depicted in Listing 2.

Next, we used our instrumentation tool *GSA_Gen* on the predicate-transformed program to generate our own variant of the *gated static single-assignment (GSA) form* [22] representation of the program. This tool collects information about data dependencies between variables, and it also instruments the program to record the runtime values of assignment targets in a dictionary data structure that is maintained by a Java class we created named *CollectOut*. The variable values are recorded by inserting calls to a static method *CollectOut.record()* (referred to as `record()` in Listing 3), which takes the following

```
1  private String format(JSError error, boolean warning) {
2    ...
3    int charno = error.getCharno();
4    record("Formatter", "format", 88, 2, "charno_0", charno, 0);
5    boolean P3_0 = false;
6    record("Formatter", "format", 92, 2, "P3_0_0", P3_0, 0);
7    boolean P3_1 = false;
8    record("Formatter", "format", 93, 2, "P3_1_0", P3_1, 0);
9    boolean P3_2 = false;
10   record("Formatter", "format", 94, 2, "P3_2_0", P3_2, 0);
11   if ((P3_2 = (excerpt.equals(LINE)))
12        && (P3_1 = (0 <= charno))
13        && (P3_0 = (charno < sourceExcerpt.length())))) {
14     record("Formatter", "format", 109, 2, "charno_2", charno, 2);
15     record("Formatter", "format", 109, 2, "charno_3", charno, 3);
16     record("Formatter", "format", 109, 2, "P3_2_1", P3_2, 1);
17     record("Formatter", "format", 109, 2, "P3_1_1", P3_1, 1);
18     record("Formatter", "format", 109, 2, "P3_0_1", P3_0, 1);
19     boolean P4_0 = false;
20     record("Formatter", "format", 98, 3, "P4_0_0", P4_0, 0);
21     for (int i = 0; (P4_0 = (i < charno)); i++) {
22       record("Formatter", "format", 99, 3, "P4_0_1", P4_0, 1);
23       record("Formatter", "format", 99, 3, "charno_1", charno, 1);
24       char c = sourceExcerpt.charAt(i);
25       record("Formatter", "format", 100, 4, "c_0", c, 0);
26       boolean P5_0 = false;
27       record("Formatter", "format", 101, 4, "P5_0_0", P5_0, 0);
28       if ((P5_0 = (Character.isWhitespace(c)))) {
29         record("Formatter", "format", 106, 4, "P5_0_1", P5_0, 1);
30         record("Formatter", "format", 106, 4, "c_1", c, 1);
31         b.append(c);
32       } else {
33         b.append(' ');
34       }
35     }
36     b.append("^\n");
37   }
38   ...
```

Listing 3: Instrumented version of the code that belongs to the `format` method from Listing 1

parameters in the given order: package name, class name, method name, line number encountered in the original code, code block number, name in GSA form, value to be recorded, and the version of the program variable. The GSA version of the code in Listing 1 is shown in Listing 3.

Consider what happens when *GSA_Gen* encounters the assignment statement `P3_1 = (0 <= charno)` at line 8 of Listing 2. The tool then: inserts the proper GSA variable version increments; inserts a call to *CollectOut.record()* to record the value of `P3_1` and the other parameters mentioned above; and adds a key-value pair `[P3_1_i,charno_j]` to the dictionary, where `i` and `j` are the respective variable versions in the assignment statement. The value(s) for key `P3_1_i` are later used for confounding adjustment (see Section IV).

We tested the GSA transformed program with the developer-written tests in the Defects4J suite, using the *test* command to run these tests. Out of the 6050 tests, two failed with assertion errors. We applied two of the coverage-based statistical fault localization metrics that performed the best in recent studies [23]–[26], namely, Ochiai [5] and DStar [25], to the code coverage and failure data for all of the tests. We then ranked the variables and predicates of the faulty class, `LightweightMessageFormatter`, in nonincreasing order of their suspiciousness scores (with ties receiving the average rank for all tied variables). The Ochiai and DStar metrics produced *identical* rankings. The faulty predicate `P3_0` at line 13 in Listing 3 (line 5 in Listing 1) received the same score as 23 other variables and ranked in 17th place in the resulting suspiciousness list, which included all of the variables and predicates within the **if** branch in the faulty method. The low rank for the faulty predicate is likely due to confounding, which the Ochiai and DStar metrics don't adjust for.

We next applied *UniVal* to the same transformed program. We input the data recorded by our run-time library *CollectOut* together with the values of a binary outcome variable Y, which indicates whether the program executions failed (1) or succeeded (0), to our method for calculating suspiciousness scores. This code executes the *analysis phase* of *UniVal*, which is described in Section IV. Finally, we mapped these scored statements back to the original program.

The suspiciousness list produced with *UniVal* contained unique scores for each variable and predicate. The faulty predicate `P3_0`, at line 13 in Listing 3 (line 5 in Listing 1), was ranked highest among all the predicates with a score of 0.086, and it ranked 5th among all assignment targets. The variables `charno_2` and `charno_3` recorded at lines 14 and 15 of Listing 3, which are recorded for possible use in simulating a GSA-form ϕ_{if} function (see Section III-B), represent the value of the variable `charno` in lines 12 and 13. They would both be ranked 1st overall, with a score of 0.49, if we included them in the ranking.[1] The non-faulty predicate `P4_0`, at line 21 in Listing 3 (line 6 in Listing 1), was ranked 18th with a score of 0.001.

We performed an extra step to the experiment to illustrate that the adjustment for confounding in *UniVal* indeed makes a difference. We replicated the steps described previously for using *UniVal* with a minor change for the correct predicate `P4_0`, which is located at line 21 in Listing 3. We removed `P3_0` (at line 13 in Listing 3) from the set of adjustment variables (covariates) in the random forest model for `P4_0`. Consequently, the score for `P4_0` increased to 0.01, which changed its rank to 14th. The new score and rank of `P4_0` were higher than those of the faulty predicate `P3_0`, whose score and rank declined to 0.005 and 16th. Therefore, the suspiciousness scores for the variables were in fact distorted by the lack of adjustment for confounding bias.

III. Background

A. Causal Inference

In this section we provide background on statistical causal inference required to understand our technique, *UniVal*. Statistical causal inference is concerned with estimating without bias the average causal effect of a treatment variable upon an outcome variable. A *treatment variable* or *exposure variable* T is a variable that an investigator could, at least in principle, intervene upon to change its value. For example, in SFL T might represent the outcome of a branch predicate, which a developer could change in a debugger. An *outcome variable* Y is an observable variable of interest to an investigator, such as an indicator of whether a program execution fails or not. In statistical causal inference, one is concerned with outcomes for individuals or units in a population, and it is often convenient to denote the outcome for a unit i by Y_i. In SFL, the units are typically program executions.

A key concept in causal inference is that of a *counterfactual outcome*, which is a value that the outcome variable Y could take on if, counter to the facts, the treatment variable T took on a value t' different from the value t that it actually took on. A very similar concept is that of a *potential outcome*, which is a value that the outcome variable could take on if the treatment variable were assigned a particular value. As is common in the causal inference literature, we shall use the terms "counterfactual outcome" and "potential outcome" interchangeably. In the influential Neyman-Rubin causal modeling framework [27], a distinct *potential outcome random variable* $Y^{T=t}$ exists for each possible value t of the outcome variable.

In principle, for each unit i and each pair of treatment values t and t' there is an *individual causal effect* $\delta_i = y_i^t - y_i^{t'}$, where y_i^t and $y_i^{t'}$ are values of the potential outcome random variables $Y_i^{T=t}$ and $Y_i^{T=t'}$, respectively. Conceptually, the *average causal effect* (ACE) of T on Y over a population is the expected value $\mathrm{E}[\delta]$ of the individual causal effects δ_i over that population. However, in general it is not possible to compute the δ_i, because at most one of $Y_i^{T=t}$ and $Y_i^{T=t'}$ is observed for each i, namely, the potential outcome under the actual treatment. This is known as the *Fundamental Problem of Causal Inference* [28]. In this sense, causal inference is

[1] In our empirical evaluation, we assign scores to only the predicates in branch conditions.

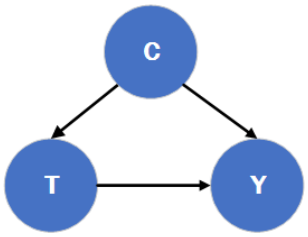

Fig. 1: Example Confounding DAG

a *missing data problem*. In statistical causal inference, this problem is circumvented by estimating average causal effects from a study sample of units and by "borrowing information" from each unit in the sample.

In a randomized experiment, which is an interventional study in which the the investigator assigns treatments randomly to units, the missing potential outcomes are missing completely at random, and the average causal effect $\mathrm{E}[\delta] = \mathrm{E}[Y^{T=t} - Y^{T=t'}] = \mathrm{E}[Y^{T=t}] - \mathrm{E}[Y^{T=t'}]$ can be estimated by computing the average outcomes in the treatment groups with $T = t$ and $T = t'$ and taking their difference [29]. However, in observational studies generally and in SFL applications in particular, treatment assignment is usually *not* randomized, and hence the difference of the treatment group averages is often a biased estimate of the average causal effect. In SFL, for example, whether a particular statement is executed during a program run or whether a given variable is assigned a specific value depends on the behavior of other statements, which may also affect whether the program fails.

Different types of bias can affect a causal effect estimate [29]. The best known form of systematic bias and the one that will be addressed in this paper is *confounding bias* (or simply *confounding*). Confounding is bias due to the presence of a variable that is a *common cause* of the treatment variable and the outcome variable. Confounding bias is best explained in terms of a causal graph, and such graphs are also used to identify *confounders*, which are variables that can be statistically adjusted for during causal effect estimation in order to reduce or eliminate confounding bias. A *causal directed acyclic graph* or *causal DAG* is a DAG in which the vertices represent causal variables and in which there is a directed edge (A, B) or $A \to B$ between two variables A and B only if A is known or assumed to be a cause of B.

Figure 1 is a very simple example of a DAG involving three variables, T, C, and Y. The edge $T \to Y$ represents the direct causal effect of T on Y. The DAG indicates that C confounds this effect, because C is a common cause of T and Y. The noncausal path $T \leftarrow C \to Y$, which is called a *backdoor path* because it begins with an arrow into the treatment variable, represents a biasing flow of statistical association from T to Y via C. As a result of this flow, a naive, unadjusted estimate of the ACE would mix the causal effect of T on Y with the association "carried" by the backdoor path. This implies that to obtain an unbiased estimate of the ACE, C must be adjusted for.

One way to obtain an unbiased estimate of the ACE is to *block* (d-separate [21]) the backdoor path $T \leftarrow C \to Y$ by conditioning on the value of C during the analysis, e.g., by computing causal effect estimates separately for each level or stratum of C and then combining them via a weighted average to obtain an estimate of the population ACE. Causal inference theory provides a number of results that characterize the sets of variables that may be used for confounding adjustment, in terms of the structure of a causal DAG. For example, the *Backdoor Adjustment Theorem* [21] states that a set Z of variables that blocks every backdoor path between the treatment variable and the outcome variable is sufficient for confounding adjustment.

Another way to estimate the average causal effect, which we employ in this paper, is to interpolate or predict the missing counterfactual outcome $Y_i^{T=t}$ or $Y_i^{T=t'}$ for each unit based on the data for all the units in the study sample and then to plug in the predicted value in the formula for the individual causal effect δ_i.

B. Gated Static Single Assignment Form

Gated single assignment form (GSA form) [22] is an extension of *static single assignment form* (SSA form) [30]. SSA form is a specialized intermediate program representation that makes the data flow of a program explicit by ensuring that each variable is defined in exactly one location (hence the name static *single* assignment form). When two or more definitions for a variable reach a single use a new definition is created to merge the reaching definitions by using a special "pseudo-function" called ϕ which "picks" the correct definition to use at runtime.

GSA replaces the ϕ function of SSA form with three gating functions ϕ_{if}, ϕ_{entry}, and ϕ_{exit} (alternatively, γ, μ, and η, respectively). ϕ_{if} represents the merging of control flow after an if-statement and takes as an argument the *predicate* in the controlling if-statement, allowing ϕ_{if} to choose the correct value. ϕ_{entry} is similar to ϕ_{if} except it merges loop-carried variables at the top of iterative control structures. Finally, ϕ_{exit} merges variables that are both *live* at the exit of a loop and are *modified* by that loop. Taken together these three new gating functions effectively embed the control dependence graph [31] into the intermediate representation by linking the choice of the variable definition to use to the predicate which controls the computation.

The instrumentation used by *UniVal* is inspired by (and its placement is guided by) GSA form. If GSA form was directly converted into program instrumentation it would be expensive due to the extra runtime overhead implied by the gating functions ϕ_{if}, ϕ_{entry}, and ϕ_{exit} for choosing the right definitions. Instead instrumentation is inserted to record values (and their controlling predicates) at the locations where the gating functions would have appeared in GSA form. (See Section II.) This permits us to determine causal parents needed to control for confounding in *UniVal*.

IV. Method

In this section, we describe the *UniVal* fault localization technique in detail. *UniVal* is based on predicting the values

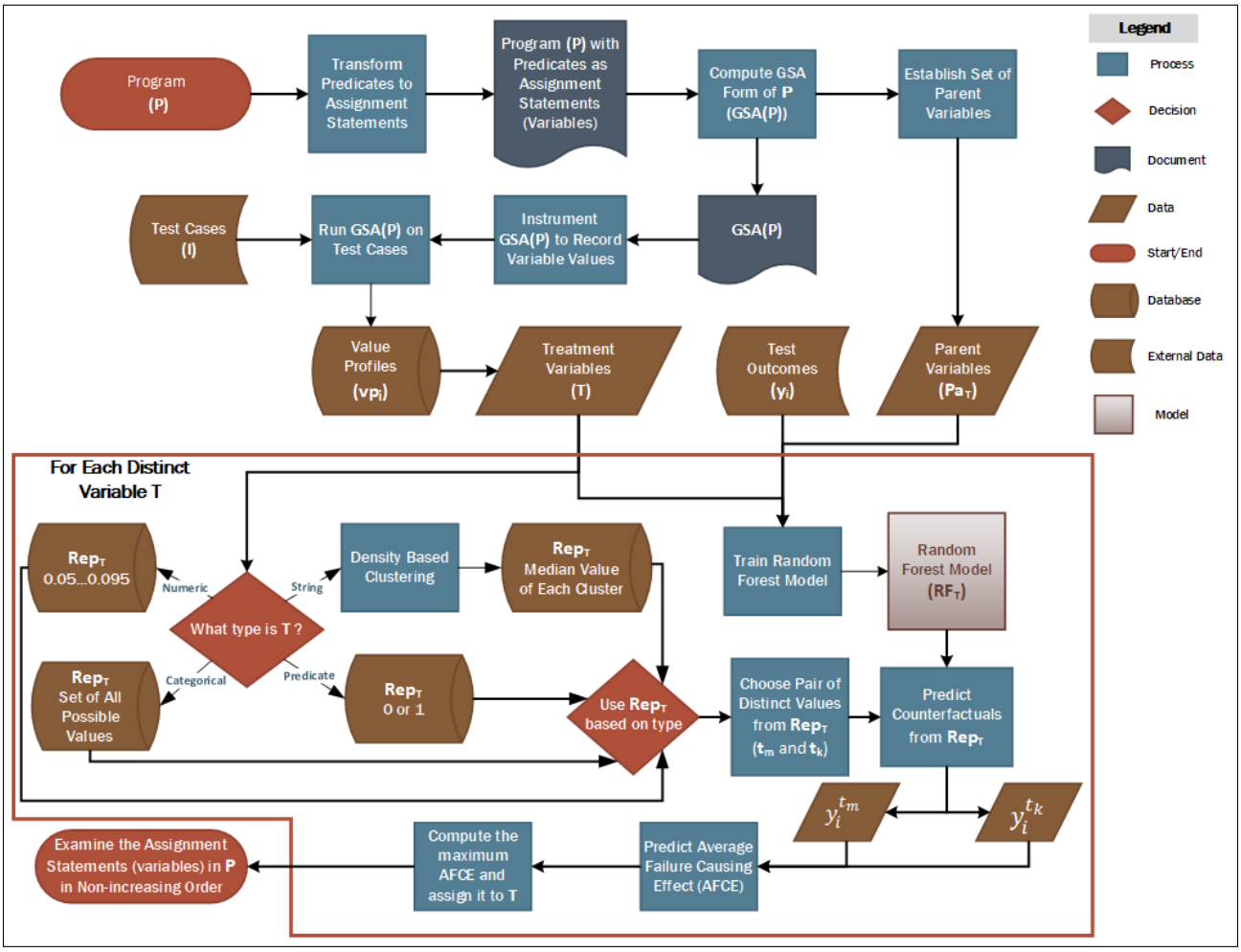

Fig. 2: Causal fault localization method *UniVal*

of individual counterfactual outcomes, using machine learning models that are trained on data from a sample consisting (ideally) of significant numbers of both passing and failing executions. There is a separate model for each assignment to a program variable, including assignments inserted into branch predicates (e.g., lines 11-13 in Listing 3). The data required for each execution and each assignment includes the values of the treatment variable T, the outcome variable Y, and the set of covariates X. The treatment variable is the assignment target, the outcome variable indicates whether the execution passed or failed, and the covariates are the parents of the treatment variable (if any exist) in the causal DAG for the program being debugged. For example, the recorded data for line 12 of Listing 3 includes the values of `P3_1` and `charno` as well as the program outcome (pass or fail). Note that each backdoor path (see Section III) in the causal DAG begins with an arrow $T \leftarrow P$, where P is a parent of T. Thus, by the Backdoor Adjustment Theorem (see Section III) the set X of parents of T blocks all backdoor paths from T to Y, and therefore X is sufficient for confounding adjustment.

Figure 2 depicts the operation of *UniVal*. The first two steps in *UniVal*, which together we call the *instrumentation phase*, are source-to-source transformations of the program $\mathcal{P}$ to be debugged. First, each predicate in a branch conditions of $\mathcal{P}$ is transformed into an assignment statement that assigns the value of the predicate to a new boolean variable. (See for example lines 7-9 of Listing 2, which correspond to line 5 of Listing 1.) We created a prototype tool named the *PredicateTransformer* for this task. This tool also records information about each predicate, including the type of control statement it belongs to (e.g. while, for, if-else, if), the predicate expression, and the line number where it was encountered in the original Java file. Second, the resulting program is transformed by another prototype tool we created, named *GSA_Gen*, into the specialized version of Gated Single Assignment form described in Section III-B. This entails inserting calls to a function that records the values of treatment variables and covariates as well as other information needed by our implementation. At the same time, the causal parents of assignment targets (the variables whose values are used in the assignment) are determined.

The next phase of *UniVal*, which we call the *profiling phase*, involves executing the instrumented GSA version $GSA(\mathcal{P})$ of the program on a set $\mathcal{I}$ of test cases or operational inputs

in order to record the variable values and other information mentioned above. Note that for a treatment variable T assigned to in a loop, only the *last* value assigned to T is recorded, along with the corresponding covariate values. It is assumed that the program outcomes (pass or fail) for these executions are already known or are determined prior to analysis.

In *UniVal*, assignment targets will have missing values in particular program runs (which we call NAs) if their assignment statements are not executed. If the treatment variable has no recorded values or just one unique value, it is omitted from *UniVal*'s consideration. Note that parts of a compound predicate expression might have missing values due to short-circuited evaluation.

The third phase of *UniVal*, which we call the *analysis phase*, involves fitting a counterfactual prediction model for each assignment target in $GSA(\mathcal{P})$. We adopted the counterfactual prediction approach to analysis described in [32], although, unlike *UniVal*, that work does not address predicates or strings. In the current implementation of *UniVal*, we employ random forest learners [33] as prediction models, because they are flexible enough to non-parametrically model a wide variety of relationships between the treatment variable and covariates, on one hand, and the outcome variable, on the other hand. Specifically, we used the *Ranger* random forest package [34]. The model for a given variable assignment includes the treatment variable (the assigned variable) and the covariates (the used variables) as predictors, and the model is used to predict the counterfactual program outcomes (pass or fail) under different values of the treatment variable.

For each treatment variable T, a set Rep_T of representative treatment values is chosen, and counterfactual outcomes are predicted for these treatments. Rep_T is chosen differently based on the type of T. For a boolean or categorical treatment variable T, Rep_T contains all recorded values of T. For a string variable, a clustering algorithm is first used to cluster the recorded treatment values, and then Rep_T becomes the set of cluster IDs, which are treated like categorical values. We use the *stringdist* [35] package to obtain a matrix of distances between values. This matrix is input to the distance-based clustering algorithm *DBSCAN* [36] (with MinPts = $2 * dim$, where dim is the dimension of the dataframe). For a numeric treatment variable T, Rep_T consists of the $0.05, 0.15, \ldots, 0.95$ quantiles of the empirical distribution of the recorded values of T. *UniVal* does not currently handle other data types.

For each representative treatment value $t \in Rep_T$ and each complete input i to $\mathcal{P}$, *UniVal* predicts the counterfactual outcome y_i^t. This is done by plugging t into the model together with the covariate values x_i recorded at the assignment to T during execution of $\mathcal{P}$ on i. Note that *UniVal* predicts counterfactual outcomes even for actual, recorded treatment-covariate combinations, that is, even when the true counterfactual outcome is known. Given the set of predicted counterfactual outcomes for T, *UniVal* computes for each $t \in Rep_T$ an estimate $\hat{\mathrm{E}}[Y^{T=t}]$ of the counterfactual mean $\mathrm{E}[Y^{T=t}]$, by averaging the predictions $\{\hat{y}_i^t\}$. The suspiciousness score for T is set to the maximum, over all pairs $t, t' \in Rep_T$ of $\hat{\mathrm{E}}[Y^{T=t}] - \hat{\mathrm{E}}[Y^{T=t'}]$. That is, the score is the maximum, over all pairs of representative treatment values for T, of the average failure-causing effect of assigning t instead of t' to T.

The final phase of *UniVal*, which we call the *localization phase*, involves employing the suspiciousness scores to assist developers in finding the cause or causes of failures observed when $\mathcal{P}$ was executed on the set of inputs $\mathcal{I}$. Traditionally, this is done by ranking statements in non-increasing order of their suspiciousness scores and then having developers inspect statements in that order [4]. Although this is convenient for evaluating SFL techniques — and it is used for that purpose in this paper — it has been argued that this is a simplistic approach to fault localization [37], [38], which programmers are in many cases unlikely to follow (e.g., when many statements in a program get very high scores). We envision *UniVal* being used in combination with other sources of information, including developer knowledge and intuition, to effectively localize faults.

V. Empirical Evaluation

A. Study Setup

We empirically evaluated the fault localization performance of *UniVal* in a substantial empirical study involving subject programs from the latest, expanded version (2.0.0) of the popular Defects4J evaluation framework [20]. We compared the fault localization costs of *UniVal* and several competing techniques: the non-interventional value-based techniques Elastic Predicates (ESP) [10] and NUMFL (specifically the two variants NUMFL-DLRM and NUMFL-QRM) [19]; the non-interventional coverage-based technique of Baah *et al.* [15], which employs linear regression for causal inference; the interventional technique Predicate Switching [39], which alters conditional branching; and finally two well-known coverage-based SFL (CB-SFL) metrics that performed well in recent comparative studies [5], [24], [26], namely, Ochiai [5] and D-Star (with star = 2) [25].

There is one important note concerning the implementation of Baah *et al.*'s original technique based on linear regression [15]. With that technique, the only covariate in the regression model was a coverage indicator for the forward control dependence predecessor of the target statement. For this study, we have modified Baah *et al.*'s technique by including the variables used at the target statement as covariates. We believe this is a notable improvement on the original technique. The modified version almost always performs second best among the studied methods.

Defects4J (Version 2.0.0) [20] is a collection of 17 programs and 835 faulty program versions containing a wide spectrum of real software faults. The number of subject programs and the total number of faulty program versions have nearly doubled in this release of Defects4J. Although the very low failure rates of many Defects4J programs make them non-ideal for statistical fault localization [40], its user friendly scripts and continuous support and evolution makes it an invaluable source for empirical studies.

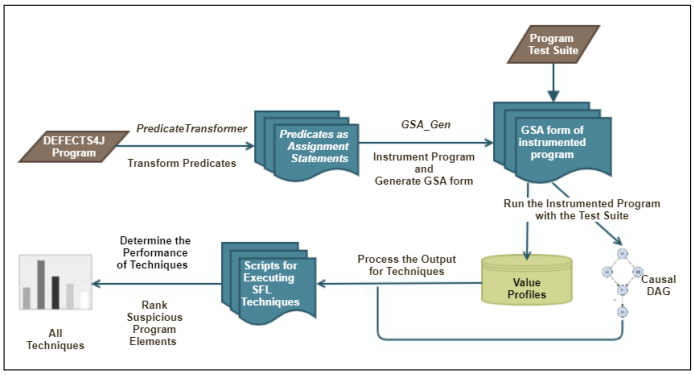

Fig. 3: Process of Empirical Evaluation

Defects4J Subject Programs			
Program ID	KLOC	Average # of tests	# of faulty versions
Chart	96	220	24
Cli	4	147	37
Csv	2	153	16
Math	85	172	103
Time	28	2525	25
Lang	22	94	60
Closure	90	3448	170
Mockito	23	671	35
Codec	10	132	16
JxPath	21	272	20
Gson	12	566	16
Collections	46	180	3
Compress	11	158	45
Jsoup	14	344	90
JacksonCore	31	248	25
JacksonXml	6	143	6
JacksonDatabind	4	1343	109

TABLE I: Summary of Subject Programs

In our study, the techniques we compare assign suspiciousness scores to different program elements. *UniVal* assigns scores to numeric, string, and categorical assignment statements and to predicates, ESP assigns scores to numeric assignment statements and to predicates, and NUMFL assigns scores to each subexpression of numeric assignment statements. Predicate Switching assigns scores only to predicates. Baah *et al's* linear regression technique [15] and the other coverage-based SFL techniques assign scores to all the statements, although each statement within a control dependence region receives the same score. To enable a fair comparison between techniques, we confined the comparison to only predicates and numeric assignment statements. (Note that to reduce overhead in this study only the faulty classes are instrumented.) With Predicate Switching, however, we reported the cost of finding the nearest predicate to the fault, since the technique does not assign scores to non-predicates. As a result of these restrictions, our study did not evaluate *UniVal*'s ability to handle string variables. We intend to do so in a future study.

For a few program versions NUMFL failed to assign a suspiciousness score to any assignment statement and the suspiciousness list only contained zero values. We suspect this is caused by using a binary outcome variable rather than the absolute difference between the actual and expected output as NUMFL is intended to do. We did not include these versions in our comparison. We also did not include a program version if it had fewer than 20 "relevant" test cases, which cover a faulty class [20]. The remaining subject program versions are summarized in Table I.

We used the EXAM score measure [41] and Hit@N measure (sometimes referred to as Recall@N or Top-N) [42] to report the cost of fault localization for each technique. EXAM score is the percentage of program statements a developer must examine, in non-increasing order of their suspiciousness scores, before finding the fault. If there are ties in the scoring of statements, we assumed that half of the tied statements will have to be examined before a programmer localizes any fault among them. Hit@N is the number of program versions for which a fault was found within the top N ranked statements in non-increasing order of suspiciousness scores. Because our comparison involves assignment statements and predicates, if the fault was not directly related to an assignment statement or predicate (e.g. a fault of omission) we determined a set of fault localization candidates with an approach similar to the one described by Pearson *et al.* [24]. Finally, We used a linux Ubuntu 20.04 LTS machine that runs on an Intel i5-930H quad core CPU at 2.4-4.1 GHz and that has 8GB of RAM for our experiments. The time measurements are calculated by inserting a timestamp at the beginning and at the end of each step of the pipeline script and reporting the averages for all programs. The EXAM and Hit@N metrics involve simple average or count calculations; hence, we used Microsoft Excel to compute them. We ran all the program versions in our comparison 10 times and averaged the results over all runs in case techniques displayed random variation.

B. Results

Table II provides an summary of the results of the empirical evaluation showing the average performance of each technique on each program using the three evaluation methods: EXAM, Hit@10, and Hit@5. Lower EXAM scores are better than high ones, while higher Hit@10 and Hit@5 scores are better than low ones. Average runtimes for the methods in our empirical evaluation were: 55.6 seconds for *UniVal*, 78.6 seconds for NUMFL (cumulative time to run both models), 660 seconds for Predicate Switching, 44.8 seconds for Baah2010, 13.2 seconds for ESP, and 4.4 seconds for the coverage based techniques. The average overhead of our instrumentation is about 25% (e.g. Math-1 takes 8 seconds to execute without the instrumentation and 10 seconds with the instrumentation).

We make two observations about *UniVal*'s overall performance as shown in Table II. First, *UniVal* usually had the best score for a given program/cost metric combination. Second, as can be seen in the EXAM Score table, *UniVal*'s scores display less variation than other methods. This indicates not only that *UniVal* is capable of relatively precise localization (as shown in the Hit@5 table) but also that it provides more consistent results. In contrast, the well known Ochiai method provides precise localization somewhat frequently, but it exhibits much more variation.

EXAM Score								
Program	*UniVal*	NUMFL-QRM	NUMFL-DLRM	Elastic Predicates (ESP)	Baah2010	Predicate-Switching	Ochiai	D-Star
Chart	5%	11%	18%	14%	10%	47%	17%	17%
Cli	10%	21%	21%	24%	20%	35%	20%	24%
Codec	11%	15%	20%	10%	16%	22%	16%	19%
Csv	10%	25%	39%	24%	29%	53%	31%	26%
Math	8%	15%	12%	18%	22%	58%	11%	17%
Time	7%	15%	18%	14%	11%	27%	9%	6%
Lang	4%	9%	22%	17%	9%	30%	8%	10%
Mockito	9%	9%	14%	13%	16%	45%	8%	11%
Closure	11%	22%	20%	11%	11%	59%	25%	20%
JxPath	17%	22%	35%	26%	22%	36%	25%	24%
Gson	9%	18%	30%	37%	13%	25%	18%	20%
Jsoup	9%	14%	13%	15%	11%	31%	20%	21%
JacksonCore	15%	29%	27%	17%	25%	31%	25%	28%
JacksonDatabind	19%	35%	23%	22%	23%	39%	15%	21%
JacksonXml	17%	17%	31%	23%	16%	34%	18%	18%
Compress	8%	16%	15%	15%	8%	52%	11%	14%
Collections	9%	14%	15%	20%	7%	22%	11%	10%
Average	**10%**	**18%**	**22%**	**19%**	**16%**	**38%**	**17%**	**18%**
Standart Deviation	**4%**	**7%**	**8%**	**7%**	**7%**	**12%**	**7%**	**6%**
Wilcox. Sig. Test (p-value)	***N/A***	**4.38E-04**	**1.96E-04**	**3.52E-04**	**1.96E-04**	**1.96E-04**	**1.40E-03**	**3.27E-04**
Hit@ 10 (Top-10)								
Program	*UniVal*	NUMFL-QRM	NUMFL-DLRM	Elastic Predicates (ESP)	Baah2010	Predicate-Switching	Ochiai	D-Star
Chart	19	17	14	17	18	9	14	14
Cli	30	26	20	24	26	11	22	22
Codec	16	13	10	14	15	7	15	15
Csv	11	9	6	9	11	6	7	8
Math	92	78	68	70	70	51	71	74
Time	20	17	12	14	15	9	15	14
Lang	58	53	48	58	55	25	50	50
Mockito	28	23	14	24	27	12	16	18
Closure	134	110	100	110	120	57	72	72
JxPath	17	14	12	15	15	9	14	14
Gson	12	9	9	8	11	9	11	11
Jsoup	79	74	67	70	72	46	69	67
JacksonCore	20	17	12	15	17	12	16	16
JacksonDatabind	91	83	71	72	84	45	81	82
JacksonXml	4	3	2	2	4	2	4	4
Compress	42	41	36	40	44	25	39	38
Collections	2	2	1	2	2	1	2	2
Overall: 800	**675**	**589**	**502**	**564**	**606**	**336**	**518**	**521**
Wilcox. Sig. Test (p-value)	***N/A***	**1.96E-04**	**1.96E-04**	**4.38E-04**	**2.09E-03**	**1.96E-04**	**4.38E-04**	**4.38E-04**
Hit@ 5 (Top-5)								
Program	*UniVal*	NUMFL-QRM	NUMFL-DLRM	Elastic Predicates (ESP)	Baah2010	Predicate-Switching	Ochiai	D-Star
Chart	17	14	13	15	16	8	14	14
Cli	27	23	16	23	22	10	20	20
Codec	12	9	9	12	12	6	13	13
Csv	7	6	2	7	10	3	4	7
Math	89	74	67	69	67	50	67	72
Time	18	15	8	9	11	8	12	11
Lang	53	49	43	51	50	23	49	48
Mockito	26	21	12	23	25	11	15	16
Closure	129	108	89	105	119	54	70	69
JxPath	15	10	10	10	14	8	13	11
Gson	8	8	5	5	10	8	7	7
Jsoup	78	69	65	68	70	44	67	64
JacksonCore	16	13	10	13	14	10	15	13
JacksonDatabind	88	81	67	69	81	44	78	81
JacksonXml	4	2	2	2	4	2	4	4
Compress	39	38	34	35	37	20	39	37
Collections	1	1	0	0	2	1	2	2
Overall: 800	**627**	**541**	**452**	**516**	**564**	**310**	**489**	**489**
Wilcox. Sig. Test (p-value)	***N/A***	**4.38E-04**	**2.93E-04**	**4.60E-04**	**3.19E-02**	**4.38E-04**	**2.71E-03**	**1.92E-03**

TABLE II: Comparison of *UniVal* and competitive metrics for all Defects4J programs

To evaluate the statistical significance of our results, we conducted Wilcoxon signed-rank tests [43] for the difference in performance between *UniVal* and other techniques, for each part of Table II. The resulting p-values are reported in the bottom row of each part of the table. Each difference in performance between *UniVal* and another technique is significant at the 0.05 level.

Figure 4 visually compares *UniVal*'s performance versus the other techniques using the EXAM score. There are only seven instances in which *UniVal* does worse than a competing technique (out of 119 total). In general, this study indicates that of all the techniques considered, *UniVal* has the best performance on the the Defects4J dataset.

C. The Effect of Covariate Balance

In causal inference, covariates are variables other than the treatment variable or the outcome variable that may be associated with either variable. They are used for confounding adjustment or for reducing the variance of estimates. In the absence of confounding, as in a randomized experiment, the joint distributions of the covariates should be similar in both (or in all) treatment groups [29], [44]. This condition is called *covariate balance*. It typically does not occur in observational studies. There are techniques, such as matching [16], [45], that attempt to achieve covariate balance in the analysis sample by removing certain units from the original study sample. However, *UniVal* statistically adjusts for covariates rather than modifying the study sample.

To better understand the effect of covariate balance and imbalance on the performance of *UniVal* and coverage-based fault localization, we conducted a sub-study in which we measured the degree of covariate imbalance in the data sets for faulty Defects4J [46] versions and we related it to the cost of fault localization, as measured by EXAM score [41], for *UniVal* and the coverage-based SFL metric Ochiai [5]. To simplify the comparison, we considered only program versions that contain a faulty predicate in a branch condition. Thus, for both the *UniVal* and Ochiai techniques the treatment value corresponded to the outcome of that predicate (*true* or *false*). Additionally, we considered only predicates that have covariates with numerical values. To measure covariate imbalance, we calculated the mean, over all covariates, of the difference in the mean values of individual covariates for the two treatment groups.

We located the program versions in release 1.4 of the Defects4J repository that contained faults in branch predicates by consulting a recent study that details the fault types in that release [46]. For release 2.0 of Defects4J, we searched among the fault-fix patches included with the release. We found a total of 228 program versions fitting our criteria.

The results of our study of the effect of covariate imbalance on fault localization are depicted in Figure 5. The figure shows a scatter plot in which the X-axis represents the mean, over all covariates, of the difference in the mean values of individual covariates for the two treatment groups and in which the Y-axis represents the EXAM scores for *UniVal* and *Ochiai* for data sets with given levels of covariate imbalance. Note that along the X-axis, larger values represent greater imbalance, and along the Y-axis, larger values represent higher costs for fault localization.

The results indicate that higher fault localization costs are associated with greater imbalance. This is consistent with previous results indicating that covariate balance is associated with lower estimation bias in observational studies [47]. However, it is evident in Figure 5 that even when the covariates are imbalanced, the cost of fault localization with *UniVal* is usually lower than with Ochiai. This is due to the fact that *UniVal* adjusts for confounding and therefore mitigates the effects of covariate imbalance.

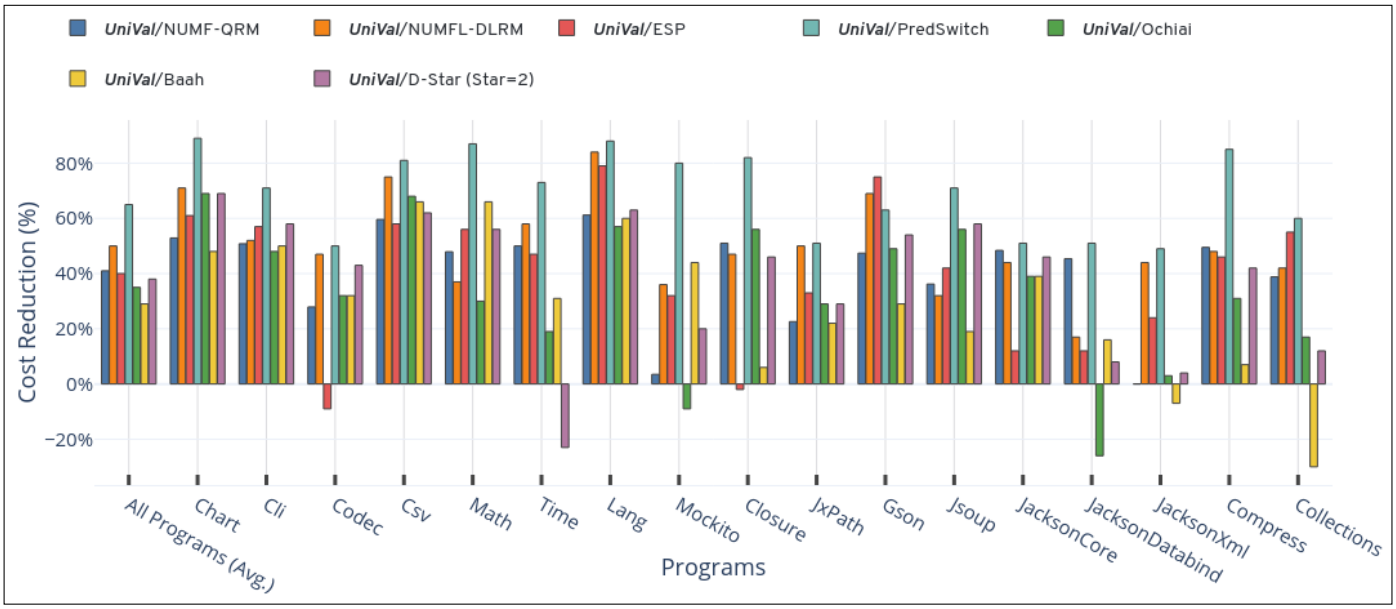

Fig. 4: *EXAM* score reductions achieved by UniVal over other techniques, for all Defects4J programs

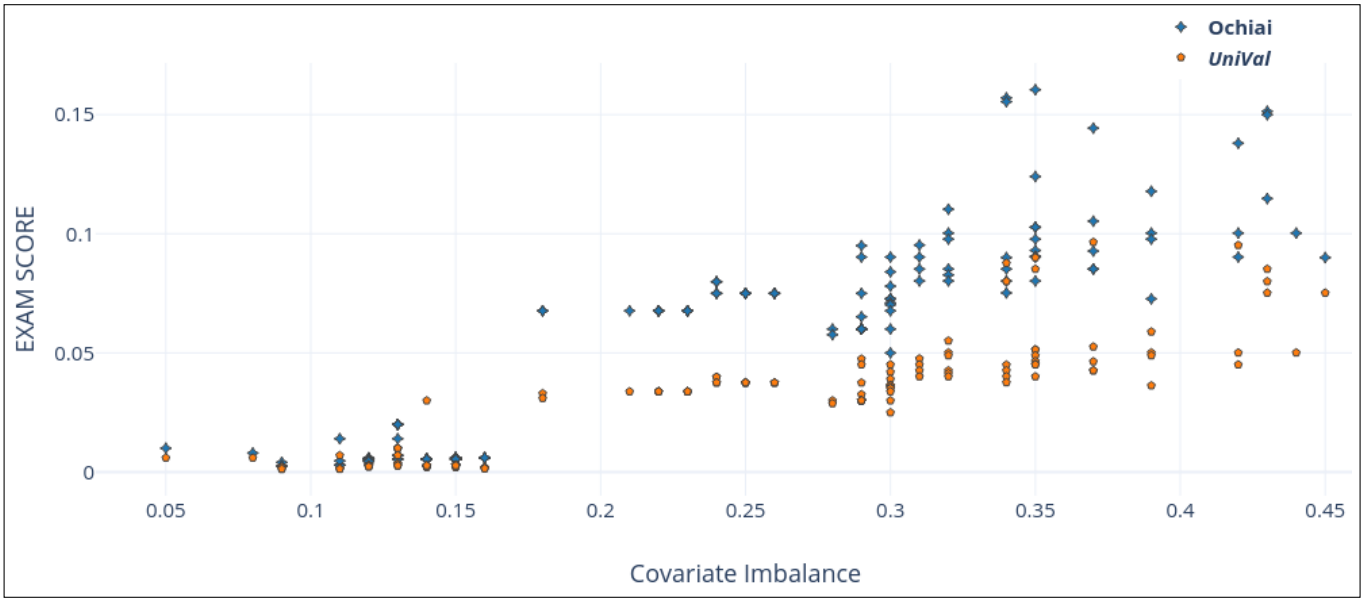

Fig. 5: Relationship Between Covariate Imbalance and EXAM Score

D. Threats to Validity

1) Internal Validity: As previously noted in the literature the EXAM score and Hit@N metrics are imperfect evaluation methods for automatic fault localization [37], [38]. In particular they assume a programmer will always start at the top of the ranked list of program elements and move monotonically down the list ignoring the surrounding program structure. They also assume that the programmer is equally likely to examine program elements with the same suspiciousness score. But, programmers use their intuition, background knowledge of the program, and other methods of debugging when using automatic fault localization tools. Hence, this evaluation method does not fully account for programmer behavior.

The EXAM and Hit@N cost metrics handle program elements with the same suspiciousness scores by giving each such element the same (average) rank score (see Standard Rank Score in [38]). This creates evaluation bias in favor of fault localization techniques that are more likely than others to assign different elements the same suspiciousness score (such as the coverage-based metrics Ochiai and DStar). However, some of the techniques we studied — including the proposed technique *UniVal* — don't tend give the same scores to different program elements.

2) External Validity: The Defects4J dataset [20] is a very useful collection of programs, bugs, and programmer-written tests. However, no collection can be all encompassing, and

this collection is still only a small sample of real programs and their bugs. A technique performing well on these 17 programs might not perform particularly well on any other program (and vice versa). Second, the test cases are provided by the developers and are generally in the form of unit tests. Arguably, end-to-end system tests would be a better choice for evaluating fault localization techniques [40]. In particular the tests fail at very low rates on the faulty program versions in the dataset. Ideally, there would be more balance between failing and passing tests. Finally, because the tests are generally unit tests they do not simulate operational input by end-users the programs.

In general, despite the weaknesses outlined above, the authors feel this study is a state-of-the-art empirical evaluation of fault localization techniques. The study employs only real bugs, real tests, and substantially sized programs. It does not use the often criticized "injected faults" or "generated test cases." Nor was it conducted on toy programs constructed for the purpose. Finally, every effort was made to put the previous work in the best light: from improvements to Baah's method to filtering out faults which NUMFL could not localize.

VI. Related Work

Cleve and Zeller [48] proposed an interventional approach to fault localization based on *cause transitions*, points in time where a variable starts to become the cause of failure, and they showed how delta debugging [3] can be used to find them. Jeffrey *et al.* [9] presented a value-based fault localization technique called *value replacement*, which searches for program statements whose execution can be intervened upon to change incorrect program output to correct output. Similarly, earlier work by Zhang *et al.* introduced *predicate switching* [39], which searches predicates that are executed by failing tests and whose outcomes can be altered to make the program succeed. Johnson *et al.* [49] present an approach to "causal testing", in which input fuzzing is used to generate passing and failing tests that are similar to an original failing test. The generated tests are then used to pinpoint failures. The aforementioned techniques each require a complete or partial test oracle. Furthermore, they also entail possibly costly repeated runs of subject programs and searches among them, while *UniVal* does not require any oracle or repeated runs of subject programs.

Fariha *et al.* [50] propose a technique called Automated Interventional Debugging (AID) that seeks to pinpoint the root cause of an application's intermittent failures and to generate an explanation of how the root cause triggers them. AID approximates causal relationships between events in terms of their occurrence times and intervenes to change the value of predicates in order to prune a causal DAG and extract a causal path from the root cause to a failure. Although AID makes use of counterfactuals, it does not adjust for confounding bias.

Baah *et al.* [15] pointed out that the conventional SFL techniques are susceptible to confounding bias, and they employed causal inference methodology to localize faults. A linear regression model was fitted for test outcomes with a coverage indicator for the target statement as the treatment and a coverage indicator for its forward control dependence predecessor as a covariate. Bai *et al.* presented two variants of a value-based causal statistical fault localization technique called NUMFL [19]. This technique makes use of *generalized propensity scores*, which are used to achieve covariate balance. Although these causal inference based techniques adjust for confounding like *UniVal*, they employ parametric regression and thus lack the modeling flexbility of random forests. Elastic predicates (ESP) [10], which were presented by Gore *et al.*, involve measuring how different, in standard deviations, an assigned variable's value is from its average value. This difference is used as a suspiciousness score. In contrast to *UniVal*, ESP does not control for confounding bias.

Feyzi *et al.* proposed an approach [51] to reducing the number of statements that must be considered in fault localization. They use backward slicing techniques and information theory to find candidate cause-effect chains (introduced by Zeller *et al.* [3]), before applying causal inference based techniques (such as Baah2010) on these candidates. Their method is still subject to the limitations of the techniques it combines, which are mentioned throughout this section.

Recent studies have investigated combinations of SFL metrics or sources of information for use in fault localization. Xuan *et al.* [12] presented a technique that uses learning-to-rank methods [52] to combine multiple SFL metrics. Sohn and Yoo [13] combined SFL results with source code metrics from static analysis. Zou *et al.* [14] found that combining a variety of SFL techniques was beneficial. Li *et al.* [11] uses deep learning with neural networks to integrate multiple sources of information that vary from SFL metrics to textual similarity measures. Although these approaches to combining different sources of information in fault localization are promising, they do not adjust for confounding or other biases and therefore their results are prone to bias. [53] Mutation-based SFL techniques [54]–[56] measure the suspiciousness of a program statement by how much a mutation to it can change the number of program failures that occur on a test set. These techniques often generate a very large number of mutations, hence they may be very costly to apply. Unlike *UniVal*, they do not employ established causal inference methodology.

Model-based debugging (MBD) techniques apply model-based diagnosis to software fault localization [23], [57]–[59]. In MBD a logical model of a program is derived automatically from its source code and a search algorithm finds minimal sets of statements whose faultiness can explain incorrect behavior observed on test cases. Abreu *et al.* [23] presented a new model-based approach, named BARINEL, to diagnosing multiple intermittent faults, which uses a Bayesian method for estimating the probability that a faulty component exhibits correct behavior. Recent studies [60]–[62] have applied model-based debugging to spreadsheet programs, which are a special type of numerical program. We believe *UniVal* might be an efficient algorithm for spreadsheet programs. None of the aforementioned MBD techniques addresses confounding bias.

VII. Conclusion

This paper has presented *UniVal*, which is a novel approach to statistical fault localization that is based on statistical causal inference methodology and that integrates value-based and predicate-based fault localization by transforming predicates into assignment statements. It uses a machine learning model to estimate, with minimal bias, the average causal effect of counterfactual assignments to program variables, without actually changing the program or its executions. *UniVal* currently handles the values of numeric, boolean, categorical, and string values. We reported the results of an extensive empirical evaluation of *UniVal*, in which it outperformed a variety of competing techniques. In future work, we intend to expand the range of program elements and data types to which *UniVal* applies.

Data Availability

Our implementation for the prototype tools and experiments presented in this paper is publicly available [63].

Acknowledgement

This work was partially supported by NSF award CCF-1525178 to Case Western Reserve University. The authors would also like to thank Zhoufu Bai for providing scripts we used for including NUMFL into our evaluation.

References

[1] J. A. Jones, M. J. Harrold, and J. Stasko, "Visualization of test information to assist fault localization," in *Proceedings of the 24rd International Conference on Software Engineering, 2002. ICSE 2002.* IEEE, 2002, pp. 467–477.

[2] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, "Scalable statistical bug isolation," *Acm Sigplan Notices*, vol. 40, no. 6, pp. 15–26, 2005.

[3] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Transactions on Software Engineering*, vol. 28, no. 2, pp. 183–200, 2002.

[4] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," *IEEE Transactions on Software Engineering*, vol. 42, no. 8, pp. 707–740, 2016.

[5] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. J. Van Gemund, "A practical evaluation of spectrum-based fault localization," *Journal of Systems and Software*, vol. 82, no. 11, pp. 1780–1792, 2009.

[6] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "Semfix: Program repair via semantic analysis," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 772–781.

[7] F. Y. Assiri and J. M. Bieman, "Fault localization for automated program repair: effectiveness, performance, repair correctness," *Software Quality Journal*, vol. 25, no. 1, pp. 171–199, 2017.

[8] T. Xie and D. Notkin, "Checking inside the black box: Regression testing by comparing value spectra," *IEEE Transactions on software Engineering*, vol. 31, no. 10, pp. 869–883, 2005.

[9] D. Jeffrey, N. Gupta, and R. Gupta, "Fault localization using value replacement," in *Proceedings of the 2008 international symposium on software testing and analysis*. ACM, 2008, pp. 167–178.

[10] R. Gore, P. F. Reynolds, and D. Kamensky, "Statistical debugging with elastic predicates," in *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 2011, pp. 492–495.

[11] X. Li, W. Li, Y. Zhang, and L. Zhang, "Deepfl: integrating multiple fault diagnosis dimensions for deep fault localization," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 2019, pp. 169–180.

[12] J. Xuan and M. Monperrus, "Learning to combine multiple ranking metrics for fault localization," in *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 2014, pp. 191–200.

[13] J. Sohn and S. Yoo, "Fluccs: Using code and change metrics to improve fault localization," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 2017, pp. 273–283.

[14] D. Zou, J. Liang, Y. Xiong, M. D. Ernst, and L. Zhang, "An empirical study of fault localization families and their combinations," *IEEE Transactions on Software Engineering*, 2019.

[15] G. K. Baah, A. Podgurski, and M. J. Harrold, "Causal inference for statistical fault localization," in *Proceedings of the 19th international symposium on Software testing and analysis*. ACM, 2010, pp. 73–84.

[16] ——, "Mitigating the confounding effects of program dependences for effective fault localization," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on foundations of software engineering*. ACM, 2011, pp. 146–156.

[17] R. Gore and P. F. Reynolds, "Reducing confounding bias in predicate-level statistical debugging metrics," in *34th International Conference on Software Engineering (ICSE), 2012*. IEEE, 2012, pp. 463–473.

[18] G. Shu, B. Sun, A. Podgurski, and F. Cao, "Mfl: Method-level fault localization with causal inference," in *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. IEEE, 2013, pp. 124–133.

[19] Z. Bai, G. Shu, and A. Podgurski, "Causal inference based fault localization for numerical software with numfl," *Software Testing, Verification and Reliability*, vol. 27, no. 6, p. e1613, 2017.

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

[21] J. Pearl, *Causality*. Cambridge University Press, 2009.

[22] K. J. Ottenstein, R. A. Ballance, and A. B. MacCabe, "The program dependence web: a representation supporting control-, data-, and demand-driven interpretation of imperative languages," in *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation - PLDI '90*, vol. 20-22-June. New York, New York, USA: ACM Press, 1990, pp. 257–271. [Online]. Available: http://portal.acm.org/citation.cfm?doid=93542.93578

[23] R. Abreu, P. Zoeteweij, and A. J. Van Gemund, "A new bayesian approach to multiple intermittent fault diagnosis," in *Twenty-First International Joint Conference on Artificial Intelligence*, 2009.

[24] S. Pearson, J. Campos, R. Just, G. Fraser, R. Abreu, M. D. Ernst, D. Pang, and B. Keller, "Evaluating and improving fault localization," in *Proceedings of the 39th International Conference on Software Engineering*. IEEE Press, 2017, pp. 609–620.

[25] W. E. Wong, V. Debroy, R. Gao, and Y. Li, "The dstar method for effective software fault localization," *IEEE Transactions on Reliability*, vol. 63, no. 1, pp. 290–308, 2014.

[26] L. Lucia, D. Lo, L. Jiang, F. Thung, and A. Budi, "Extended comprehensive study of association measures for fault localization," *Journal of software: Evolution and Process*, vol. 26, no. 2, pp. 172–219, 2014.

[27] G. W. Imbens and D. B. Rubin, *Causal inference in statistics, social, and biomedical sciences*. Cambridge University Press, 2015.

[28] P. W. Holland, "Statistics and causal inference," *Journal of the American statistical Association*, vol. 81, no. 396, pp. 945–960, 1986.

[29] M. Hernán and J. Robins, *Causal Inference*. Chapman Hall/CRC, forthcoming, 2018.

[30] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Transactions on Programming Languages and Systems*, vol. 13, no. 4, pp. 451–490, oct 1991. [Online]. Available: http://portal.acm.org/citation.cfm?doid=115372.115320

[31] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 9, no. 3, pp. 319–349, 1987.

[32] A. Podgurski and Y. Küçük, "Counterfault: Value-based fault localization by modeling and predicting counterfactual outcomes," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2020, pp. 382–393.

[33] L. Breiman, "Random forests," *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.

[34] M. N. Wright and A. Ziegler, "Ranger: a fast implementation of random forests for high dimensional data in c++ and r," *arXiv preprint arXiv:1508.04409*, 2015.

[35] M. van der Loo, J. van der Laan, R. C. Team, N. Logan, and C. Muir, "Package 'stringdist'," *CRAN, June*, vol. 6, 2019.

[36] M. Hahsler, M. Piekenbrock, and D. Doran, "dbscan: Fast density-based clustering with r," *Journal of Statistical Software*, vol. 91, no. 1, pp. 1–30, 2019.

[37] C. Parnin and A. Orso, "Are automated debugging techniques actually helping programmers?" in *Proceedings of the 2011 international symposium on software testing and analysis*. ACM, 2011, pp. 199–209.

[38] T. A. Henderson, A. Podgurski, and Y. Kucuk, "Evaluating automatic fault localization using markov processes," in *2019 19th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2019, pp. 115–126.

[39] X. Zhang, N. Gupta, and R. Gupta, "Locating faults through automated predicate switching," in *Proceedings of the 28th international conference on Software engineering*, 2006, pp. 272–281.

[40] Y. Küçük, T. A. Henderson, and A. Podgurski, "The impact of rare failures on statistical fault localization: the case of the defects4j suite," in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, pp. 24–28.

[41] E. Wong, T. Wei, Y. Qi, and L. Zhao, "A crosstab-based statistical method for effective fault localization," in *Software Testing, Verification, and Validation, 2008 1st International Conference on*. IEEE, 2008, pp. 42–51.

[42] S. Wang and D. Lo, "Version history, similar report, and structure: Putting them together for improved bug localization," in *Proceedings of the 22nd International Conference on Program Comprehension*. ACM, 2014, pp. 53–63.

[43] F. Wilcoxon, "Individual comparisons by ranking methods," in *Breakthroughs in statistics*. Springer, 1992, pp. 196–202.

[44] B. B. Hansen and J. Bowers, "Covariate balance in simple, stratified and clustered comparative studies," *Statistical Science*, pp. 219–236, 2008.

[45] P. C. Austin, "Balance diagnostics for comparing the distribution of baseline covariates between treatment groups in propensity-score matched samples," *Statistics in medicine*, vol. 28, no. 25, pp. 3083–3107, 2009.

[46] V. Sobreira, T. Durieux, F. Madeiral, M. Monperrus, and M. A. Maia, "Dissection of a bug dataset: Anatomy of 395 patches from defects4j," in *Proceedings of SANER*, 2018.

[47] J. J. Sauppe and S. H. Jacobson, "The role of covariate balance in observational studies," *Naval Research Logistics (NRL)*, vol. 64, no. 4, pp. 323–344, 2017.

[48] H. Cleve and A. Zeller, "Locating causes of program failures," in *Proceedings of the 27th international conference on Software engineering*. ACM, 2005, pp. 342–351.

[49] B. Johnson, Y. Brun, and A. Meliou, "Causal testing: Understanding defects' root causes," in *Proceedings of the 2020 International Conference on Software Engineering*, 2020.

[50] A. Fariha, S. Nath, and A. Meliou, "Causality-guided adaptive interventional debugging," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2020, pp. 431–446.

[51] F. Feyzi and S. Parsa, "Inference: effective fault localization based on information-theoretic analysis and statistical causal inference," *Frontiers of Computer Science*, vol. 13, no. 4, pp. 735–759, 2019.

[52] T.-Y. Liu *et al.*, "Learning to rank for information retrieval," *Foundations and Trends® in Information Retrieval*, vol. 3, no. 3, pp. 225–331, 2009.

[53] C. Liu, C. Gao, X. Xia, D. Lo, J. Grundy, and X. Yang, "On the replicability and reproducibility of deep learning in software engineering," *arXiv preprint arXiv:2006.14244*, 2020.

[54] M. Papadakis and Y. Le Traon, "Metallaxis-fl: mutation-based fault localization," *Software Testing, Verification and Reliability*, vol. 25, no. 5-7, pp. 605–628, 2015.

[55] L. Zhang, L. Zhang, and S. Khurshid, "Injecting mechanical faults to localize developer faults for evolving software," in *ACM SIGPLAN Notices*, vol. 48, no. 10. ACM, 2013, pp. 765–784.

[56] S. Moon, Y. Kim, M. Kim, and S. Yoo, "Ask the mutants: Mutating faulty programs for fault localization," in *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*. IEEE, 2014, pp. 153–162.

[57] W. Mayer and M. Stumptner, "Evaluating models for model-based debugging," in *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 2008, pp. 128–137.

[58] W. Mayer, R. Abreu, M. Stumptner, A. J. Van Gemund *et al.*, "Prioritising model-based debugging diagnostic reports," in *Proceedings of the 19th International Workshop on Principles of Diagnosis*. Citeseer, 2008, pp. 127–134.

[59] F. Wotawa, M. Nica, and I. Moraru, "Automated debugging based on a constraint model of the program and a test case," *The journal of logic and algebraic programming*, vol. 81, no. 4, pp. 390–407, 2012.

[60] D. Jannach, T. Schmitz, B. Hofer, K. Schekotihin, P. Koch, and F. Wotawa, "Fragment-based spreadsheet debugging," *Automated software engineering*, vol. 26, no. 1, pp. 203–239, 2019.

[61] R. Abreu, B. Hofer, A. Perez, and F. Wotawa, "Using constraints to diagnose faulty spreadsheets," *Software Quality Journal*, vol. 23, no. 2, pp. 297–322, 2015.

[62] R. Abreu, A. Riboira, and F. Wotawa, "Debugging spreadsheets: A csp-based approach," in *2012 IEEE 23rd International Symposium on Software Reliability Engineering Workshops*. IEEE, 2012, pp. 159–164.

[63] Y. Kucuk, T. Henderson, and A. Podgurski, "Improving Fault Localization by Integrating Value and Predicate Based Causal Inference Techniques," Jan. 2021. [Online]. Available: https://doi.org/10.5281/zenodo.4441439