# Variable-Based Fault Localization via Enhanced Decision Tree

JIAJUN JIANG, Tianjin University, China
YUMENG WANG, Tianjin University, China
JUNJIE CHEN*, Tianjin University, China
DELIN LV, Tianjin University, China
MENGJIAO LIU, Tianjin University, China

Fault localization, aiming at localizing the root cause of the bug under repair, has been a longstanding research topic. Although many approaches have been proposed in the last decades, most of the existing studies work at coarse-grained statement or method levels with very limited insights about how to repair the bug (*granularity problem*), but few studies target the finer-grained fault localization. In this paper, we target the *granularity problem* and propose a novel finer-grained variable-level fault localization technique. Specifically, the basic idea of our approach is that fault-relevant variables may exhibit different values in failed and passed test runs, and variables that have higher discrimination ability have a larger possibility to be the root causes of the failure. Based on this, we propose a program-dependency-enhanced decision tree model to boost the identification of fault-relevant variables via discriminating failed and passed test cases based on the variable values. To evaluate the effectiveness of our approach, we have implemented it in a tool called VarDT and conducted an extensive study over the Defects4J benchmark. The results show that VarDT outperforms the state-of-the-art fault localization approaches with at least 268.4% improvement in terms of bugs located at Top-1, and the average improvement is 351.3%.

Besides, to investigate whether our finer-grained fault localization result can further improve the effectiveness of downstream APR techniques, we have adapted VarDT to the application of patch filtering, where we use the variables located by VarDT to filter incorrect patches. The results denote that VarDT outperforms the state-of-the-art PATCH-SIM and BATS by filtering 14.8% and 181.8% more incorrect patches, respectively, demonstrating the effectiveness of our approach. It also provides a new way of thinking for improving automatic program repair techniques.

CCS Concepts: • **Software and its engineering → Search-based software engineering**; **Software maintenance tools**.

Additional Key Words and Phrases: Fault localization, Program debugging, Decision tree

**ACM Reference Format:**
Jiajun Jiang, Yumeng Wang, Junjie Chen, Delin Lv, and Mengjiao Liu. 2023. Variable-Based Fault Localization via Enhanced Decision Tree. *ACM Trans. Softw. Eng. Methodol.* 1, 1, Article 1 (January 2023), 32 pages. https://doi.org/10.1145/3624741

*Corresponding author.

Authors' addresses: Jiajun Jiang, College of Intelligence and Computing, Tianjin University, Tianjin, China, 300350, jiangjiajun@tju.edu.cn; Yumeng Wang, College of Intelligence and Computing, Tianjin University, Tianjin, China, 300350, jazz244008@tju.edu.cn; Junjie Chen, College of Intelligence and Computing, Tianjin University, Tianjin, China, 300350, junjiechen@tju.edu.cn; Delin Lv, College of Intelligence and Computing, Tianjin University, Tianjin, China, 300350, ldlmntq@tju.edu.cn; Mengjiao Liu, College of Intelligence and Computing, Tianjin University, Tianjin, China, 300350, mengjiaoliu@tju.edu.cn.


1049-331X/2023/1-ART1 $15.00
https://doi.org/10.1145/3624741

## 1 INTRODUCTION

Program bugs are inevitably introduced in programs, which will potentially cause great financial losses and even disasters. Therefore, fixing bugs timely when they occur is critical. In particular, the first stage of program debugging is to locate the root cause of bugs under repair, which is an expensive and labor-intensive process. To facilitate this process, many automatic fault localization techniques have been proposed [2, 4, 5, 16, 31–33, 44–46, 51, 90] in the last decades, aiming at providing a list of candidate locations that are most possibly faulty to aid the subsequent program repair process.

Although great success has been achieved, the mainstream fault localization techniques still suffer from two major limitations. First, the fault localization precision is low, the state-of-the-art techniques can only locate about 21% genuine faulty statements as the top-1 returned results [85]. Inaccurate fault localization results can be misleading and increase the risk of generating incorrect patches due to the incomplete specification [55, 61, 78]. Second, the granularity of existing fault localization results is still coarse-grained at statement or method levels, which provide few insights beyond locations related to the root cause for repairing the bug. As a result, even given the genuine faulty locations, the patch space is still large, which aggravates the problem of generating incorrect patches. As reported in existing studies [47, 91], when providing genuine faulty statements, the state-of-the-art automatic program repair (APR) techniques can still repair a small number of bugs with generating many incorrect patches, significantly affecting the usability of APR techniques. In this paper, we call these two limitations *precision* and *granularity* problems, respectively, in fault localization.

Over the years, the vast majority of existing studies mainly focus on the *precision* problem, and have adopted different techniques, such as mutation testing [52], machine learning [80], deep learning [41, 42, 48], etc., and incorporated diverse information like test coverage [53], program dependency [3, 88], code changes [62] and program invariants [6], to improve the precision. However, most of the studies work at statement or method levels, but few works target the *granularity* problem, especially in the scenario of APR. Although some techniques have been designed at a finer-grained level (e.g., variable level), they are either requiring the intervention of developers [83, 84] or targeting a particular type of variables [37, 43, 44], making them infeasible to further promote the effectiveness of downstream APR techniques.

Aiming at significantly improving the effectiveness of fault localization and thus boosting the subsequent program repair process, in this paper we propose a novel and general fault localization technique, named VarDT, addressing the *granularity* problem by effectively identifying the fine-grained fault-relevant variables via leveraging a program-dependency-enhanced decision tree model. Intuitively, the basic idea of VarDT is that fault-relevant variables may exhibit different values in failed and passed test runs, and variables that have higher discrimination ability have a larger possibility to be the root causes of the failure. According to this intuition, we adopt the decision tree model to aid the identification of the most fault-relevant variables by building discrimination models for failed and passed runs using candidate variables. However, since the number of variables and their value space are usually large in real-world programs, especially in industry-grade programs, VarDT further incorporates the static program analysis to improve its scalability and effectiveness, including program slicing and dependency analysis. We will introduce our approach detailedly in Section 3.

To evaluate the effectiveness of our approach, we have implemented a prototype of it as an automatic fault localization tool, also named VarDT, and conducted an extensive experiment on the widely-used Defects4J [34] benchmark. The results show that VarDT successfully located the fault-relevant variables at Top-1 position for 23.5% of bugs, which significantly outperformed

seven state-of-the-art baseline approaches. Particularly, the improvement is at least 268.4%, and on average 351.3% regarding the bugs located at Top-1. Moreover, to investigate whether our approach can further promote the effectiveness of downstream APR techniques, we also adapted VarDT to the application of patch filtering, where it correctly filtered out 67.4% incorrect patches. Although not designed as a comprehensive and standalone patch filtering technique, it improves the state-of-the-art PATCH-SIM and BATS by 14.8% and 181.8%. The results indicate that our finer-grained fault localization technique is indeed effective and promising to further improve the effectiveness of downstream APR techniques.

In summary, this paper makes the following contributions:

- We propose a novel variable-based fault localization technique, named VarDT, which identifies fault-relevant variables by constructing program-dependency-enhanced decision tree models using variables for discriminating passed and failed test cases.
- We conduct an extensive study on the widely-used Defects4J benchmark in two distinct application scenarios, i.e., fault localization and patch filtering. The results demonstrate the effectiveness of our approach by comparing it with existing state-of-the-art approaches.
- We provide a new way of thinking for improving APR techniques – providing finer-grained fault localization results to refine the patch space of APR tools.
- We have published all our experimental results and implementation to facilitate future research for replication and comparison. **https://github.com/ssmingz/VarDT**.

In the following, we first motivate our approach through a running example in Section 2, and then introduce the details of our approach in Section 3. Section 4 and 5 present the setup and result analysis of our experiment, while Section 6 and 7 qualitatively analyze the results of our approach and discuss the threats to validity, respectively. Finally, we present the related work in Section 8 and conclude the paper in Section 9.

## 2 MOTIVATING EXAMPLE

In this section, we will motivate our approach with a running example. Listing 1 presents the patch code of Lang-27 in the widely-used Defects4J benchmark [34], where the lines leading by "+" denote code to be added while "-" to be deleted.

```
452 Number createNumber(String str) throws Exception {
  ...
473     int decPos = str.indexOf('.');
474     int expPos = str.indexOf('e') + str.indexOf('E') +1;
475
476     if (decPos > -1) {
477
478         if (expPos > -1) {
479-          if (expPos < decPos) {
    +         if (expPos < decPos || expPos > str.length()){
480               throw new NumberFormatException(str + " is not a valid number.");
481           }
482           dec = str.substring(decPos + 1, expPos);
483         } else {
484           dec = str.substring(decPos + 1);
485         }
486         mant = str.substring(0, decPos);
487     } else {
488         if (expPos > -1) {
    +           if (expPos > str.length()) {
    +               throw new NumberFormatException(str + " is not a valid number.");
    +           }
489           mant = str.substring(0, expPos);
  ...
```

Listing 1. Patch code of Lang-27 from the Defects4J benchmark.

| Test | str | expPos | str.length() | Result |
|---|---|---|---|---|
| $t_1$ | “1l” | -1 | 2 | PASS |
| $t_2$ | “1111 ” | -1 | 5 | PASS |
| $t_3$ | “-1.1E200” | 4 | 8 | PASS |
| $t_4$ | “1eE” | 4 | 3 | FAIL |

Test samples of Lang-27 and partial variable values in the faulty method when running the corresponding test.

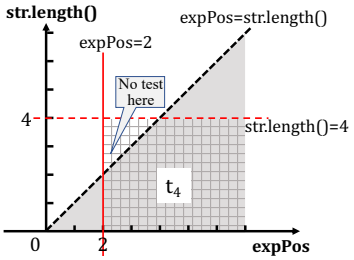



Fig. 1. Visualization of variable constraint estimation for the example shown in Listing 1.

In this example, when providing an input string `str`, the method `createNumber(*)` from the class of `NumberUtils` will transform it into a `java.lang.Number` object, e.g., transforming “10” into an `Integer` of 10. In this process, the method will automatically check the validity of the input and then decide which type of number should be created. For example, when the input string contains the character “e” (or “E”), an exponential number is always expected. However, due to the faulty code, when taking the illegal input “1eE”, a `StringIndexOutOfBoundsException` was incurred at line 489 (line 479 can be triggered by other inputs), while actually a `NumberFormatException` was expected (see Listing 1). The reason is that the method failed to check the validity of the input when multiple “e/E”s exist.

To locate the root cause of the failure, existing approaches typically return a ranked list of suspicious code lines (or methods), such as the widely-used coverage-based fault localization techniques [2, 92]. However, existing approaches can hardly locate the accurate faulty code in this example due to their inherent limitations that they cannot distinguish code elements appearing in the same basic blocks. In fact, even providing the genuine faulty code line, there is still a large search space (i.e., any syntax-valid expressions) to repair the bug due to the coarse-grained fault localization results, where incorrect patches may also be easily produced. On the contrary, if the finer-grained fault-relevant variables `expPos` and `str.length()` were known, the patch space would be significantly reduced and thus incorrect patches would also be effectively avoided.

However, accurately identifying the fault-relevant variables is indeed challenging since the variable values can be diverse in different test runs (see the left table in Figure 1). Besides, it is also common that we are required to capture the complex constraints among multiple variables for isolating failed from passed runs and understanding the root cause, e.g., `expPos>str.length()`. Checking all possible variable combinations is indeed time-consuming and even impossible in practice. Targeting this challenge, we hereby propose a novel variable-level fault localization technique based on the *decision tree model*. The basic intuition of our approach is that complex variable constraints can be estimated (or even constructed) by combining multiple primitive constraints, where only one variable is used in each individual constraint. The reason is that each primitive constraint can discriminate the failed test run from at least a subset of the passed runs and their combinations may approximate the desired complex constraint. For example, the primitive constraints `expPos>=2` and `str.length()<4` can discriminate $t_4$ from $\{t_1, t_2\}$ and $\{t_2, t_3\}$ respectively, and their combination can estimate the constraint of `expPos>str.length()` in the running example as shown in Figure 1 (right-side figure). In the figure, the **shaded area** denotes the constraint of `expPos>str.length()`, while the **gridded area** represents the combination of the two primitive constraints. Therefore, the failed ($t_4$) and passed test cases ($t_1$, $t_2$ and $t_3$) can also be distinguished by the two primitive constraints. In this way, the variables used, e.g., `expPos`, in those primitive constraints have large possibility to be the indicator of the test failure since it has the ability to isolate the failed tests from the passed ones, and thus are potentially the fault-relevant variables

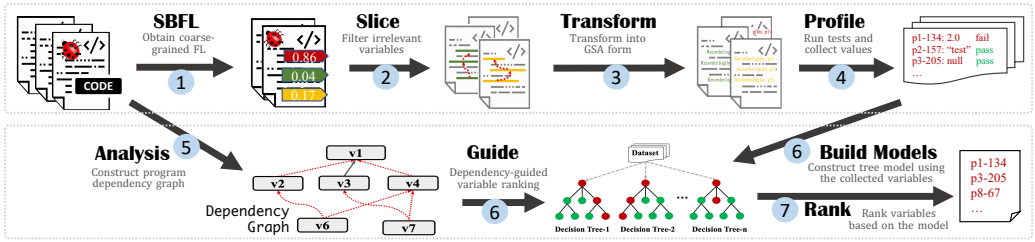



Fig. 2. Overview of our approach VarDT

(defined in Section 4.3). However, since there are usually many available variables that may produce a large number of primitive constraints, how to combine them and accurately locate the desired fault-relevant variables is still *non-trivial*. According to the characteristics of this task, we propose an enhanced decision tree model to aid the variable identification and constraint building process since the decision tree in nature performs a similar process to our task, i.e., using multiple primitive constraints (branch conditions) to estimate complex constraints for classification. We will introduce more details in Section 3 by taking this bug as the running example.

## 3 OVERVIEW

This section introduces the details of our approach. As aforementioned, the basic idea of our approach is to use variables to build constraints for distinguishing failed and passed runs, where the variables that have higher discrimination ability have larger possibilities to be the fault-relevant variables. Figure 2 shows the overview of our approach (named VarDT). In general, it consists of two stages. When given a program with at least one test case failed on it, VarDT first *collects the values of a set of variables* in both failed and passed test runs at some program checkpoints. Then, it *builds decision tree models* using those collected variables to distinguish failed and passed test runs, after which it identifies the fault-relevant variables from those used for constructing the branch conditions (i.e., constraints) in the models since they exhibit the ability to discriminate failed and passed tests.

However, it is hard and even impossible to examine the complete space of all program variables since it is usually huge, especially for large-scale programs, which may involve tens of thousands of variables. To overcome this challenge, VarDT combines existing lightweight method-level fault localization and adopts program slicing technique to identify a subset of covered statements for inspection, which can improve the efficiency and scalability of our approach. Specifically, in the current implementation of VarDT, we utilize the widely-used spectrum-based fault localization (SBFL) technique to locate a list of the most suspicious methods. The reason is that SBFL is very efficient compared with other methods since it only requires the coverage information of test cases. Particularly, we adopt the implementation published by Jiang et al. [29]. Please note that our approach is independent of this localization process, and it can be easily replaced by other methods as long as the output is an ordered list of suspicious faulty methods, such as the latest deep-learning-based techniques [42, 48], which can produce much better results than SBFL and potentially can further improve the performance of VarDT. In the following, we will introduce the core components of VarDT in more detail.

### 3.1 Dynamic Program Slicing

By using the coarse-grained fault localization techniques, we can obtain an ordered list of methods that are most likely to contain bugs. In this way, we can just focus on the variables used in these

methods. However, it is intuitive that not all statements and variables in these methods affect the output of the failing tests. In order to further reduce the search space of candidate variables for inspection and increase accuracy, VarDT leverages dynamic program slicing techniques [3] to filter out statements that are indeed irrelevant.

Specifically, when given a slicing criterion and a certain test input, VarDT performs an intra-procedure slicing process based on the data and control dependency relations along the execution trace backwardly. Although less accurate compared with the inter-procedure slicing, the intra-procedure slicing is much more efficient without the need of heavy inter-procedure analysis. As a consequence, the slicing process in VarDT will not be affected by the scale of programs under debugging but only affected by the size of a single method. Hence, our approach has good scalability on large-scale projects. Regarding the slicing criterion, we pick *the line of code that was lastly executed by the failed test in the method* because it is usually the location of failures or the indicator of finishing the complete functionality of the method and may produce variables affecting the subsequent program execution (e.g., `return` statements in many cases). Indeed, taking the failing assertions as the slicing criteria should produce more accurate results. However, it needs to perform an inter-procedure analysis, which is time-consuming as mentioned above. Instead, we take the intra-procedure analysis to approximate the accurate slicing for balancing efficiency and effectiveness. It will be presented in Sections 5.2 and 5.4, our slicing process is indeed effective. However, we also plan to investigate more effective and efficient slicing strategies in our future work for improving the reliability and usability of our approach. For instance, recall the example shown in Listing 1, the failed test run crashed at line 489 (lastly executed), which directly depends on the fault-relevant variables `str` and `expPos`, and thus they will be included in the slicing while the variable `mant` in line 486 will be filtered out. In this way, a subset of statements will be identified for further checking, highlighted in the gray color ■ in Listing 1 (Lines 473-476,488,489), while the other statements and associated variables will be ignored.

In our evaluation, we will also conduct an experiment to discuss the impact of the slicing process on the effectiveness of our approach in Section 5.

### 3.2 Program Transformation and Profiling

By program slicing, a subset of statements that are most likely to be the root cause of the test failure can be obtained. Next, VarDT will collect the variable values in those statements during the running of test cases by automatically instrumenting output statements to the source code.

Particularly, in order to tackle programs of any forms, VarDT further incorporates a program transformation process that can transform source code into a GSA form [37], where compound expressions will be implicitly decomposed into TAC (Three Address Code) format. For example, the expression `(a>b&&c>d)` will be transformed into `(v=((`$v_1$`=(a>b))&&(`$v_2$`=(c>d))))` by inserting corresponding temporary variable declarations on demand (i.e., `v`, $v_1$ and $v_2$). In this way, the intermediate computation results of compound expressions can also be collected through these temporary variables, such as the result of `a>b`. Specifically, VarDT transforms expressions in three types of code structures, i.e., *conditional expressions*, *return expressions* and *arguments of method calls*. The reason is that conditional expressions are widely-used (e.g., in `if` and loop conditions, etc.) and error-prone in practice and many bugs are caused by incorrect sub-conditions [30, 47, 71], while the expressions in the latter two types take the responsibility of value transmission and thus may potentially spread faulty variable values to a broader range outside the method, and the misuses of them are prevalent in practice [47, 58]. As a result, checking the values of these expressions is indeed necessary for locating the root causes of program failures.

After program transformation, VarDT can only focus on the variables (including temporary variables of expressions) used by the statements in the slicing. However, since the failures are

Table 1. A description for variables and predicates included by VarDT

| Type | Target Value | Description |
|---|---|---|
| Primitive | Actual Value | Primitive values, or ASCII code for variables of `char` type. |
| Object | Null Check | Output `true` if the variable is `null`, otherwise output `false`. |
| | Type Check | The dynamic value type of the variable during program running. e.g., `String` |
| | Fields | Unfold variables of `Object` type and output field values, e.g., user-defined classes. |
| | Size/length | Access the size attribute of an object by invoking `size()`/`length()`/`length` (if has). |
| | Elements | Output the element values of primitive types in variables of `Collection` type. |

usually closely correlated with certain conditions. For instance, the failure shown in the running example (see Figure 1) is closely correlated with the length of the `String` object `str`. Therefore, only considering the values of the variables is often not sufficient. Inspired by existing studies [4, 16, 24, 31, 43, 45, 90], we have summarized a set of predicates that are closely correlated with test failures according to the fix patterns collected by existing automated program repair techniques and studies [13, 30, 38, 47, 58, 63, 71]. In other words, apart from the concrete values exhibited by the (temporary) variables used in the program, we have also defined several common predicates that may be highly related to test failures, whose details are presented in Table 1. Please note that VarDT can be easily extended by incorporating more types of predicates on demand, e.g., predicates that are specific to particular application domains. In this way, the values that will be collected at line 489 in Listing 1 include not only the primitive variable values of `expPos` and `mant`, but also the predicate values of `str == null` and `str.length()`. In particular, when a variable has more than one value in a test run, e.g., the variable in a loop statement, we only collect its value in the last iteration. The reasons are twofold: (1) The number of values is undecidable for different test runs and may be larger than one thousand. Considering all values tends to involve noise and thus decreases the accuracy of fault localization. Additionally, it will also cause heavier computation and storage overhead. (2) The latest value (i.e., obtained in the last iteration) usually has a larger possibility to affect the final test result. Therefore, it may have a better capacity for discriminating different test cases. For a more thorough investigation of its impact to the effectiveness of our approach, we leave it as our future work.

To collect the above variable values, we have implemented a lightweight value profiling process in VarDT. Specifically, given a line of code, it can automatically parse the types of all variables associated to the line and insert the output statements by using the Eclipse Java Development Toolkit (https://www.eclipse.org/jdt/) for recording the corresponding predicate values defined in Table 1 during the running of test cases.

### 3.3 Tree Model Construction

As aforementioned, the basic idea of our approach is using variables to construct (multiple) primitive constraints and their combinations to distinguish passed and failed test runs, where the variables that have higher discrimination ability may have larger possibility to be fault-relevant. Based on this, we propose a novel fault-relevant variable identification technique by leveraging the decision tree model, which has been well studied to be effective in many applications [19, 25, 67, 79]. Particularly, the reasons that we decide to use the decision tree model to aid the construction of constraints in our approach are twofold: (1) Our application scenario actually can be viewed as a binary classification task, where the labels are “PASS” and “FAIL”, representing the testing results of test cases. (2) The decision tree model has good interpretability, where the branch conditions in the model explain how a given input is classified to the particular class. The conditions in the same tree path can be combined to form a more complex constraint that is only satisfied by the data belonging to the corresponding leaf node in the tree. That is, why an input is classified to the corresponding class is

Table 2. The details of test cases in different benchmarks of real-world bugs.

| Benchmark | Language | #Projects | # Bugs | Program Size | #All Tests | #Failing Tests | | |
|---|---|---|---|---|---|---|---|---|
| | | | | | | Avg. | Min | Max |
| **Bugs.jar** | Java | 8 | 1,158 | large | 1,536 | 2 | 1 | 3 |
| **ManyBugs** | C | 7 | 185 | large | 1,190 | 13 | 1 | 79 |
| **Bears** | Java | 72 | 251 | small-large | 750 | 2 | 1 | 10 |
| **Defects4J** | Java | 17 | 835 | middle | 2,292 | 2 | 1 | 84 |

traceable. Recall that our ultimate target is to identify the variables that can discriminate failed and passed tests, the interpretability and traceability properties of the model satisfy our requirements.

Next, we introduce the details of our tree model construction process in VarDT. In general, it includes two sub-processes, named *Enhanced Variable Selection* (Section 3.3.1) and *Tree Model Building* (Section 3.3.2). The former takes the responsibility to select proper variables for branch condition building, while the latter then uses the selected variables to construct concrete conditions and divides test runs into different groups. For each group, the same process will proceed until the tests in all groups cannot be further divided, where a decision tree model is built successfully.

*3.3.1 Enhanced Variable Selection.* Unlike the features used in traditional classification problems, variables collected by VarDT naturally have clear and strong correlations, i.e., control dependency and data dependency, which reflect the influence of different variables to the execution results. For example, in the patch code shown in Listing 1, the crashed line 489 depends on the variable `expPos` defined in line 474, which further depends on the input argument `str`. In other words, though the program crashed due to the incorrect value of `expPos`, the input `str` may also be the root cause of the failure in practice. However, the general variable selection algorithm in decision tree models does not consider such dependency information, and may significantly affect the overall effectiveness of fault-relevant variable localization since it may cause the irrelevant variables located and decrease the fault localization precision (will be presented in Section 5). To overcome this limitation, we propose an *enhanced variable selection algorithm* depending on a novel variable prioritization strategy which takes the program dependency factor into consideration.

Intuitively, when a variable is depended on by more other variables, its value will have higher possibility to affect the final execution results in different execution paths, and thus potentially affect more test cases. In fact, previous studies [23] also demonstrated that the variables depended on by more other variables tend to have stronger impacts on the execution output, and test cases will have higher possibilities to fail if those variables are faulty. Moreover, we observe that usually a small number of test cases, e.g., one or two, will be affected and failed in real-world buggy programs. For example, Table 2 shows the statistics of test cases in four widely-used benchmarks of real-world bugs, including Bugs.jar [57], ManyBugs [39], Bears [50], and Defects4J [34]. The size of these programs varies from small to large [27]. The percentages of average failing tests over all the tests on these four benchmarks are 0.13%, 1.09% , 0.27% , and 0.09% respectively. In particular, the absolute number of failing test cases is indeed small. 88.6% bugs are only triggered by less than three test cases in all the benchmarks. In other words, the fault-relevant variables tend to affect test cases in a small scale [51, 52, 55, 78]. Therefore, we introduce a *dependency penalty* to incorporate such an observation through static analysis. That is, a variable depended on by more other variables is less likely to be faulty since it tends to affect more test cases, and thus should have smaller probability to be located. Formula 1 defines the computation of the penalty for variable $v$ when providing the dependency graph $g$ and a list of interested variables $l$ in $g$.

$$\begin{aligned} depScore(v, g, l) &= DEP\_FACTOR^{|S|} \\ s.t.\ S &= \{x | x \in l \wedge g \vdash x \hookrightarrow v\} \end{aligned} \tag{1}$$

**Algorithm 1:** Enhanced Variable Selection for Model Building

**Input:** *varList:* a list of variables to be ranked. *data:* a set of program states for each test case. *graph:* program dependency graph.
**Output:** *var:* the variable with the largest score

```
1  Function selectVariable(varList, data, graph):
2      foreach var in varList do
3          var.score ← gainRatio(var) + correlation(var, data.labels)
4          dependency ← depScore(var, graph, varList)          // calculating the dependency penalty
5          var.score ← var.score × dependency
6      end
7      foreach var in varList do
8          foreach v in var.getEqualVars(graph) do
9              var ← aggregate(var, v)                         // aggregate equivalent variables into one
10             if v.score > var.score then
11                 var.score ← v.score
12             end
13         end
14     end
15     return varList.sort().first                             // return the variable with the largest score
```

In the formula, we use $g \vdash x \hookrightarrow v$ to represent that variable $x$ depends on variable $v$ according to $g$, i.e., the node of $v$ in graph $g$ is reachable from that of $x$. $DEP\_FACTOR \in (0, 1.0]$ is a constant penalty factor, indicating how much the dependency affects the importance of variables.

Based on this definition, we present our variable selection algorithm in Algorithm 1. Specifically, for each variable *var*, its priority is determined by three parts (Line 3-5). The *dependency penalty* has been defined in Formula 1, while the function of *correlation(*)* returns the general *Pearson correlation coefficient* [10] between variables and the testing results. Finally, the *gainRatio(var)* is a builtin function in the C4.5 decision tree model [56] for computing how much confidence can be gained by choosing the variable *var* to distinguish the given data. We also present the definitions of the functions *gainRatio(var)* (i.e., Formulas 2-4) and *correlation(*)* (i.e., Formula 5) as follows to make the paper self-contained.

$$gainRatio(v) = \frac{\Delta_{info}(v)}{-\sum_{i=1}^{m} P(v_{si}) log_2 P(v_{si})} \tag{2}$$

$$\Delta_{info}(v) = Entropy(t_v.p) - \sum_{i=1}^{m} P(v_{si}) Entropy(t_v.c(v_{si})) \tag{3}$$

$$Entropy(t) = -\sum_{i=1}^{c} p(i|t) log_2 p(i|t) \tag{4}$$

$$\rho_{A,C} = \frac{N \sum AC - \sum A \sum C}{\sqrt{N \sum A^2 - (\sum A)^2} \sqrt{N \sum C^2 - (\sum C)^2}} \tag{5}$$

In the Formulas 2-4, $m$ represents the number of distinct values for variable $v$, $P(v_{si})$ denotes the probability of $v$ taking the value $v_{si}$, $t_v$ denotes the tree node using variable $v$ for constructing the branching predicate, $t_v.p$ denotes the parent node of $t_v$ in the tree, while $t_v.c(v_{si})$ denotes the child node of $t_v$ where the value $v_{si}$ belongs. Finally, $p(i|t)$ indicates the probability that a randomly selected instance (i.e., test case in our application) in the data associated with node $t$ belongs to class $i$, and $c$ is the total number of classes (i.e., 2 in our application, "PASS" and "FAIL"). Regarding the Formula 5, $A$ represents the value of the variable currently concerned, $N$ represents the number

**Algorithm 2:** Tree Model Building

**Input:** *data:* a set of program states for each test case. *graph:* program dependency graph.
**Output:** *trees:* a set of decision trees.

```
1  Function buildModel(data, graph):
2      trees ← ∅
3      varSet ← {var | var is recorded in data}
4      while varSet is not empty do
           /* build one decision tree model using subset of variables in each round */
5          tree ← buildTree(data, toList(varSet), graph)
6          if tree is not a leaf node then
7              trees ← trees ∪ tree
8          end
9          varSet ← varSet \ {var | var is used by tree}
10     end
11     return trees
12 Function buildTree(data, varList, graph):
13     tree ← leafNode(data)
       /* size(data)>2 ∧ data include different labels */
14     if data can be classified then
15         var ← selectVariable(varList, data, graph)   // select variables for condition construction
16         cond ← calculateCondition(data, var)
17         groups ← divide(data, var, cond)             // divide data into groups based on cond
18         tree ← internalNode(data)                    // root node of subtree
19         foreach g in groups do
20             tree.children.add(buildTree(g, varList))
21         end
22     end
23     return tree
```

of test cases that make the concerned variable to be $A$, while $C$ Represents the test result ("0" for "PASS" and "1" for "FAIL").

In other words, we use both the two components, i.e., *Pearson correlation coefficient* and *gain ratio*, to measure the importance of candidate variables from different angles, where the former reflects the linear correlation between variable and test results directly, while the later indicates the confidence of choosing a certain variable for data partition. In summary, a variable that has a smaller influence to the program semantics (i.e., larger *depScore(*)*), a larger correlation to the test results, and more confidence to be the discriminator, will gain higher priority.

After computation, each variable will be assigned a priority score (refer to Line 2-6 in Algorithm 1). Next, we aggregate the equivalent variables appearing at different locations into one as the representative by removing the others according to the dependency relation (Line 8-9). In this paper, we define two variables are equivalent *iff* they are defined by the same assignment according to the define-use relation [26]. Finally, the score of the representative variable will be the maximal one of all its equivalent variables (Line 10-11). The reason is that they are always having the same value in a run, which may cause duplicate selection of the same variable. For example, the variables of `expPos` appearing at lines 474, 488 and 489 in Listing 1 are equivalent since they are all defined by the same assignment in line 474, then two of them will be removed by Algorithm 1. Finally, the variable with the largest score will be selected during the tree model building process, and thus returned (Line 15).

**Algorithm 3:** Calculate Condition in Tree Building

```
   Input: data: a set of program states for each test case. var: target variable selected.
   Output: value: a value of var used for condition construction.
1  Function calculateCondition(data, var):
2      curSplit ← data.firstInstance.value(var) // set the first value of var as the default one of curSplit
3      bestGain ← MIN_VALUE                     // set MIN_VALUE as an initial value
4      foreach instance in data do
           /* only traversing the instance whose value of var is not missing */
5          if instance.notMissing(var) then
6              attrVal ← instance.value(var)
7              if attrVal>curSplit then
8                  curGain ← entropyGain(attrVal, data)   // compute gain for every attribute value
                   /* find the attribute value with the highest gain */
9                  if curGain>bestGain then
10                     bestGain ← curGain
11                     splitPoint ← (attrVal + curSplit ) / 2   // in a bisection way
12                 end
13                 curSplit ← attrVal
14             end
15         end
16     end
17     return splitPoint
```

*3.3.2 Tree Model Building.* According to the above variable selection algorithm, we present the details of our model building process, which is shown in Algorithm 2. In general, when providing a set of candidate variables that may be fault-relevant, we try to build decision tree models to classify test cases into different groups (i.e., "Passed" and "Failed") by using all of them. In this way, every variable will have the possibility to be located since the fault-relevant variables can be multiple in practice. Specifically, given the values of a set of variables by running each test case (i.e., *data*) and the dependency graph of the program, the tree model building process (i.e., *buildModel(*)*) is iteratively proceeded. That is, VarDT each time chooses a subset of variables to construct a tree model (Line 5) until using up all candidate variables (Line 4). As shown in Line 9, the variables that have already been used by constructed tree models will be removed from the candidate list to avoid repeated selection in the follow-up model building process, i.e., each variable can be used in no more than one decision tree. Therefore, the tree model building process is guaranteed to terminate. As a consequence, the output of the model building process is a set of decision trees, each of which can independently isolate the failed test runs from the passed ones by using a subset of variables.

Particularly, in each iteration, the tree model is recursively constructed from the top down using the provided variables by invoking the function of *buildTree(*)*. Specifically, each time the variable with the highest priority (i.e., *var* returned by *selectVariable(*)*) will be selected to construct a predicate for dividing the given *data* into different groups (Line 15-17). If the selected variable *var* is nominal, the predicate will be a switch-case-like multi-way condition, while if numeric, a binary predicate, such as "$\geq$" and "$<$" will be generated (Line 16). According to the predicate, *data* will be divided into different groups. VarDT recursively performs the above construction process (Line 19-21) for each group until the input data do not require further discrimination (Line 14). In particularly, Algorithm 3 (a built-in algorithm in Weka [74]) presents the process for computing the values for predicate construction. In general, given the targeted variable *var* and all the values of it (i.e., *data*), the algorithm searches for a value *splitPoint* iteratively in a greedy manner by traversing

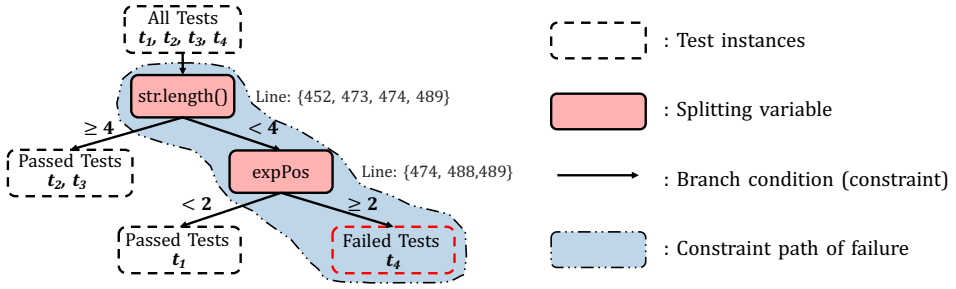



Fig. 3. A sketch of a constructed tree for Listing 1

all the possible values of *var*. Specifically, in each iteration, it chooses the value *attrVal* to update the *splitPoint* (Line 11) if it produces higher information gain[1] (Line 8-9), which is a commonly-used metric in decision tree models for measuring the discrimination ability of certain variables. We use the function *entropyGain(attrVal, data)* to represent the information gain when given *data* and variable *attrVal* for *var*. Finally, the returned value *attrVal* will be directly used to construct the predicate, such as *var>attrVal*, etc.

Up to now, when providing the required data, tree models can be constructed according to Algorithms 1 and 2. For example, recall the example shown in Listing 1, according to Algorithm 1, the temporary variable representing `str.length()` will receive the highest priority, and thus will be first selected for building the branch condition as shown in Figure 3. Specifically, according to its values in different test runs (see Figure 1), a branch condition `str.length()<4` will be constructed and divide tests into different groups[2], i.e., $\{t_2, t_3\}$ and $\{t_1, t_4\}$. Recursively, in the second round variable `expPos` will be selected and further divide the test set $\{t_1, t_4\}$ into $\{t_1\}$ and $\{t_4\}$. By now, the failed test run ($t_4$) is completely isolated from the passed runs. From Figure 3 we can also see that the constraints only satisfied by the failed test runs are highly related to the root cause of the failure.

In particular, to improve the scalability and efficiency, VarDT builds tree models for different methods independently. That is, VarDT each time takes the profiled variable data and the intra-procedure dependency graph within a single method as the input and outputs a set of constructed models, based on which it identifies the most fault-relevant variables by a ranking strategy (to be presented in Section 3.4).

### 3.4 Variable Ranking

According to the previous sections, when providing a buggy program, VarDT can construct a set of tree models for each candidate faulty method using the associated variables. In this section, we further introduce the variable ranking strategy, which provides a protocol to rank variables in different models of different methods and obtain the list of the most suspicious variables that are fault-relevant. Please note that this ranking strategy is different from the variable selection process shown in Algorithm 1, where the latter aims to make the most suspicious variables be chosen for model building by estimating their capability of discriminating failed and passed tests, while the former ranking strategy to be introduced in this section is to assign a global suspicious score to each chosen variable according to the built models.

[1]https://en.wikipedia.org/wiki/Information_gain_(decision_tree)
[2]Please note that the constant value "4" is automatically computed by the default builtin function of decision tree model in Weka ( https://www.cs.waikato.ac.nz/ml/index.html), which is also presented in Algorithm 3.

Specifically, we define a decision tree as a tuple of $M = (t, p, D, C)$, where $t$ denotes the root node of the tree, $p$ denotes the predicate associated with the node $t$, and $D$ is a set of data (including tests and corresponding variable records) associated with the node $t$. Finally, $C$ is the set of subtrees of $t$. Then, when providing the tree $M$ built for a particular method, we define the posterior discriminative score of variable $v$ used in the predicate $p$ by Formula 6.

$$DS(v) = \frac{(1 - Gini(v) \times \sqrt{|D|}}{failNodeDist} + depScore(v, g, l) \tag{6}$$

$$Gini(v) = 1 - ({\mathcal{P}_p}^2 + {\mathcal{P}_f}^2) \tag{7}$$

where $|D|$ denotes the number of test cases in $D$, we use its square root to shrink the discrepancy of test numbers since it can range from several to hundreds. $Gini(v)$ represents the general *Gini index* [11] of variable $v$, denoting the *impurity* of the tree rooted $t$. Specifically, $\mathcal{P}_p$ ($\mathcal{P}_f$) represents the probability that a randomly selected test case in $D$ is a passed (failed) one. For the given example shown in Listing 1 and Figure 1, $Gini(str.length)$ will be calculated as $1 - (0.75^2 + 0.25^2) = 0.375$ and $Gini(expPos)$ will be $1 - (0.5^2 + 0.5^2) = 0.5$. Finally, *failNodeDist* denotes the length of the tree path from the root node $t$ to the leaf node containing the failed tests in $M$. The smaller the length is, the more specific to the failed test the variable will be, and thus will be more fault-relevant. For example, variable `expPos` will be more fault-relevant than `str.length()` according to Figure 3 since $failNodeDist(str.length) = 2$ and $failNodeDist(expPos) = 1$. As a result, the corresponding score of the variable `str.length` will be $\frac{(1-0.375)\times\sqrt{1}}{1} = 0.625$, and the score of the variable `expPos` will be $\frac{(1-0.5)\times\sqrt{2}}{1} = 0.707$. The second part $depScore(v, g, l)$ is the *dependency penalty* of variable $v$ defined by Formula 1. To sum up, the score of variable $v$ is determined by the discrimination ability of the variable in the decision tree (the first part), and its impact on the program semantics (the second part).

Then, the global ranking score of variable $v$ from method $m_v$ is computed by Formula 8, where $methodScore(m_v)$ denotes the suspiciousness of method $m_v$ computed in the first step of VarDT, i.e., the method-level fault localization. In particular, we use the quadratic value of the suspiciousness to weaken its impact on the final rank and thus strengthen the importance of the variable discrimination ability (i.e., $DS(v)$). The bigger the $FS(v)$ is, the higher the variable $v$ will rank.

$$FS(v) = DS(v) \times methodScore(m_v)^2 \tag{8}$$

According to this ranking strategy, the fault-relevant variable `expPos` at lines $\{474, 488, 489\}$ was successfully ranked at the Top-1 position. As shown in Figure 3, the built constraints related to the failure can indeed estimate the desired complex constraints as presented in Figure 1.

## 4 EXPERIMENT SETUP

To evaluate the effectiveness of our approach, we have implemented it in a tool named VarDT, and conducted an extensive study by comparing it with state-the-the-art fault localization approaches. Besides, to investigate whether our finer-grained fault localization results can further improve the effectiveness of downstream APR approaches, we also adapted our approach to the application of patch filtering. Specifically, we address the following research questions in our evaluation.

- **RQ1:** How effective is VarDT for identifying fault-relevant variables in real-world programs?
- **RQ2:** Does each component contribute to the effectiveness of VarDT?
- **RQ3:** Can VarDT help to improve the results of automatic program repair?
- **RQ4:** How efficient is VarDT in fault localization?

Table 3. Subjects for fault localization

| Project | #Bugs | Project | #Bugs | Project | #Bugs | Project | #Bugs |
|---|---|---|---|---|---|---|---|
| Commons-math (**Math**) | 23 | Commons-jxPath (**JxPath**) | 8 | Commons-compress (**Compress**) | 9 | **Gson** | 7 |
| JFreeChart (**Chart**) | 12 | Commons-cli (**Cli**) | 8 | Jackson-dataformat-xml (**JXml**) | 4 | **Jsoup** | 18 |
| Commons-lang (**Lang**) | 23 | Commons-codec (**Codec**) | 10 | Jackson-core (**JCore**) | 16 | **Mockito** | 21 |
| Closure-compiler (**Closure**) | 94 | Commons-csv (**Csv**) | 5 | Jackson-databind (**JDatabind**) | 28 | Joda-time (**Time**) | 12 |
| **TOTAL** | | | | **298** | | | |

### 4.1 Subjects

In the evaluation of fault localization (RQ1&RQ2), we adopt the Defects4J (version 2.0) benchmark [34], which is widely-used in previous studies [29, 30, 47, 53, 71, 92]. Specifically, we conducted our experiment on a subset of the benchmark according to the following two constraints. First, the genuine faulty method can be located within the Top-10 returned results by the method-level fault localization in the first step of VarDT as shown in Figure 2. The reason is that the state-of-the-art approach can correctly locate more than 80% bugs in Top-10 [48]. Therefore, targeting this portion of bugs can be significant for practical use and also reduce the overhead of variable profiling (we will also discuss its impact in Section 6). Second, the faulty method has to be covered by at least three (including at least one failed and one passed) test cases so as to the decision tree model in VarDT can work normally. The details of the subjects used in our experiment are listed in Table 3.

Regarding the patch filtering application (RQ3), we adopt the datasets constructed by Xiong et al. [78] and Tian et al. [66], and use all the patches for bugs included in Table 3.

### 4.2 Baseline and Configuration

In the experiment of fault localization, following the latest research [37], we compare the effectiveness of our approach with six state-of-the-art variable-level fault localization techniques listed as follows, and adopt their open-source implementation published by Küçük et al. [37] to perform the experiment.

**UniVal** [37]: It is the latest approach that uses causal inference and machine learning to integrate information about both predicate outcomes and variable values to estimate the effects of variables to test failures.

**NUMFL** [8]: Specifically, we employed its two variants **NUMFL-QRM** and **NUMFL-DLRM**, which locate variables by combining causal and statistical analyses to characterize the causal effects of individual numerical expressions on output errors.

**ESP** [24]: This method locates variables via measuring the difference between an assigned variable in the failed run and its average value in all test runs.

**Baah2010** [7]: It locates variables by using a linear regression model to measure the *confounding bias* among variables.

**IsoVar** [72]: It identifies a set of suspicious variables based on variable execution matrices, and then performs mutation analysis at the bytecode level to isolate fault-correlated variables.

**Q-SFL** [54]: It splits software components into a set of qualitative states and considers them as SFL components to be ranked using traditional fault-localization methodologies.

Besides, by following existing work for variable-level fault localization [37], we also adapt two representative and most widely-used spectrum-based fault localization techniques to work at variable level for comparison, i.e., **Ochiai** [1] and **DStar** [75], which were proved to perform well [49, 53]. More concretely, we first collect the coverage of each variable (i.e., whether a variable in a line is executed or not.) during the running of test cases and then compute the suspicious scores for variables according to the predefined formulas by following previous studies [37]. That is, given the total number of failed test cases as $f$, and the numbers of failed and passed test cases

that cover the variable $v$ as $f_v$ and $p_v$, respectively, the suspicious score of the variable $v$ will be $O(v) = f_v/\sqrt{f(f_v + p_v)}$ when adopting Ochiai and $O(v) = f_v^2/(p_v + f - f_v)$ when adopting DStar (star=2). In particular, IsoVar was only evaluated on the bugs from Defects4J v1.2 (including Math, Chart, Lang, Closure, Time and Mockito.) while Q-SFL only works with bugs whose fault-relevant variables are method parameters and return values. To make a fair comparison, we have also compared our approach with them on the corresponding bugs, respectively.

Regarding the configurations of VarDT, we set the default value of *DEP_FACTOR* as 0.8 for computing the *dependency penalty* in Formula 1, whose impact will be further investigated in our evaluation. In addition, to evaluate the effectiveness of each component in our approach, we also create a set of variants of VarDT, which are listed as follows.

**VarDT$_{slice}$** : removes the dynamic program slicing component in VarDT and considers variables in all statements covered by the failed tests within the interested methods.

**VarDT$_{tree}$** : removes the tree model in VarDT. As a result, the variables are basically ranked according to the *dependency penalty* and the method suspicious score.

**VarDT$_{dep}$** : removes the *dependency penalty* used for variable ranking from both model building and variable ranking processes, while keeps the others unchanged.

**VarDT$_{ms}$** : removes the method score computed by SBFL in the final variable ranking process of VarDT, i.e., $FS(v) = DS(v)$.

As aforementioned, in our experiment we further investigate whether our finer-grained fault localization approach can promote downstream APR techniques. As it is well known that APR approaches are easy to produce plausible (i.e., can make all test cases pass) but incorrect patches due to the problem of weak test suite [55, 77, 78], which significantly affects the usability of APR techniques in practice. Therefore, we evaluate the performance of VarDT in the application of patch filtering, aiming at avoiding such *plausible* patches and improving the precision of APR patches. However, since our approach is not designed as a standalone patch filtering tool, we adapt it to this scenario. Specifically, we use the fault-relevant variables located by VarDT to judge the correctness of patches generated by APR approaches: If no fault-relevant variable is removed, inserted or replaced by the repair patch, it will be regarded as incorrect and deleted, otherwise the patch will be regarded as correct. In this study, we compare the results of our approach with two state-of-the-art patch filtering approaches, i.e., PATCH-SIM [78] and BATS [66]. PATCH-SIM filters patches through generating new test cases, whose oracles are estimated by measuring the similarity of executions before and after applying the patches. BATS employs an unsupervised learning-based system to predict patch correctness by measuring the similarity between the candidate patch and historical patches whose failed test cases are similar to the one associated to the candidate patch. However, PATCH-SIM and BATS were originally evaluated over different datasets. In order to clearly compare the performance of different approaches, we have re-run BATS on the bugs that are commonly used by both PATCH-SIM and our approach. Specifically, we adopted the open-source implementation of BATS published by the authors in our experiment. In particular, we have conducted an extensive study on BATS under the configurations of Embedding=CC2Vec and Cut-off Similarity$\in${0.0, 0.6, 0.7, 0.8} by following the original paper [66]. Finally, we adopted the experimental results when "Cut-off Similarity=0.0" in our comparison, since this configuration made BATS filter the least number of correct patches.

To ease the replication of our experimental results and promote future studies in this research area, we have published all our experimental data and the implementation of VarDT at **https://github.com/ssmingz/VarDT**.

### 4.3 Measurement

Although several variable-level fault localization techniques have been proposed as introduced in the Introduction, there is still no clear definition of fault-relevant variables, the ground truth employed by different studies may also be diverse. For example, Küçük et al. [37] only focus on numerical assignments and predicates, while Liblit et al. [44] locates the variables in return statements or on the left side of an assignment. To provide a fair comparison and make our results reproducible in future studies, we first provide a definition of *fault-relevant variables* from the perspective of program repair by partially referring to the variables used in existing fault localization studies [35, 37, 44, 72], such as the variables directly modified in a patch [72], the temporary predicate variables [37], etc. Specifically, we define a variable (which can be a temporary variable of a predicate expression in the GSA form) as fault-relevant if it satisfies one of the following conditions:

(1) Variables that are directly modified (i.e., replaced or deleted) or inserted to the code for repairing the bug, such as the variable `v` in the code change of "`v>0`→`v'>0`" or the temporary predicate variable `v=exp'` in the code change of "`if(exp||exp'){}`→`if(exp){}`".
(2) Variables whose values are directly affected by the newly inserted statement, such as the variable `v` in the code change of "`v=exp;`→`if(cnd){v=exp;}`".
(3) If a data-flow-breaking statement (e.g., `return`) is deleted or inserted, the temporary variable corresponding to the surrounding branch condition (if exists) since it is the indicator of the bug, such as variable `v` in the code change of "`if(v=cnd){}`→`if(v=cnd){return;}`"
(4) If all the statements in the body of an `if` statement are modified/deleted or an `If` statement is deleted, indicating a special condition is incurred, the temporary variable of the condition, such as variable `v` in the code changes of "`if(v=cnd){exp;}`→`if(v=cnd){exp';}`" and "`if(v=cnd){exp;}`→`exp;`". Please note that if only a portion of statements are modified in the body, the failures are more likely to be caused by the incorrect statements themselves but unlikely related to the condition. In such cases, the first rule can be applied.

The basic intuition of our definition is to locate variables that will be directly modified or are indicators that produce the bug. For example, the variables `expPos` and `str.length()` are both fault-relevant in the running example. Particularly, a buggy program may have multiple fault-relevant variables. On the basis of this definition, we have manually identified the fault-relevant variables for each bug used in our experiment, which will play as the ground truth and may also provide a standard for promoting future research (published in our open-source repository). Specifically, during the manual analysis, two authors identified the fault-relevant variables independently by checking the source code and the developer patches provided in the Defects4J dataset and we adopted the Cohen's Kappa coefficient [68] to measure the inter-rater agreement between them by following the existing work [15, 59]. Since the conditions of fault-relevant variables are clear, the Cohen's Kappa coefficient was already over 95% for the first 10% of analysis results, and thus the two authors identified the fault-relevant variables independently in the subsequent analysis process and we calculated the Cohen's Kappa coefficients after analyzing each 10% bugs. As a result, the Cohen's Kappa coefficients were always over 95% throughout the complete analysis process. All the inconsistencies were discussed with a third author to reach a consensus. Finally, there are on average 4 fault-relevant variables per each bug in our studied dataset. As it will be presented in Section 5.3 that correctly locating these fault-relevant variables indeed can boost existing automatic program repair techniques, further demonstrating the reliability of the ground truth.

*4.3.1 Metrics.* Following previous studies [1, 42, 48, 51, 53, 92], we employ three metrics in the fault localization experiment. **Recall at Top-N:** computes the number of bugs that have at least one fault-relevant variable correctly located within the Top-N position in the ranked list (aka.,

precision). We set N$\in \{1, 3, 5, 10\}$ like existing studies [29, 92]. **Mean First Rank (MFR):** denotes the average rank of the first located fault-relevant variables for multiple bugs. **Mean Average Rank (MAR):** When a bug has multiple fault-relevant variables, the MAR denotes the average rank of all these variables, while for multiple bugs, this metric denotes the average MAR of them. Following existing studies [29, 92], we adopt the average rank for variables in a tie. Please note that if the candidate variable list of an approach includes no fault-relevant variable, the corresponding bug will be removed when calculating the MAR and MFR for the approach to mitigate the bias due to the difference of employed predicates by different approaches. For example, NUMFL only locates numerical variables, and thus the variables of object type will never appear in its located variables, causing a larger MAR and MFR. In other words, the performance of NUMFL will be underestimated regarding the metrics of MAR and MFR if the fault-relevant variables are only objects. However, this performance degradation is caused by the incomplete variables considered but not the fault localization algorithms. Therefore, we remove the corresponding bugs for computing MAR and MFR for avoiding the bias induced by such cases. Besides, we do not use the metric of *Exam Score* used in statement-level fault localization [29, 53, 92]. The reason is that it requires the total number of candidate variables in different approaches to the same for a fair comparison, which cannot be satisfied in our experiment as aforementioned.

In the application of patch filtering, we adopt two metrics following previous studies [66, 77, 78], i.e., **Precision**=$N_{fi}/(N_{fi} + N_{fc})$ and **Recall**=$N_{fi}/(N_{fi} + N_{ni})$, where $N_{fi}$ denotes the number of incorrect patches filtered, $N_{fc}$ denotes the number of correct patches filtered, and $N_{ni}$ denotes the number of incorrect patches not filtered.

## 5 RESULT ANALYSIS

### 5.1 RQ1: Overall Effectiveness of VarDT

As introduced, we conducted our experiment over 298 real-world bugs from Defects4J benchmark and compared the results with seven baseline approaches. Table 4 presents the experimental results of different approaches regarding the percentage of bugs localized within Top-N positions and the metric of MFR and MAR. Furthermore, we also performed a paired sample Wilcoxon signed-rank test [76] at the significance level of 0.05 to investigate whether our results significantly differ from those of the comparative baseline approaches. The *p-values* are presented below the corresponding results. From the table, we can see that our approach significantly outperforms the baselines (all *p-values* are smaller than 0.5). Specifically, VarDT successfully located the desired fault-relevant variables for 23.5%, 38.9%, 47.3% and 62.1% of bugs within the Top-1/3/5/10 positions, respectively. The second best approach in each metric located the desired variables for 6.4%, 12.1%, 16.4%, 19.8% of the bugs within Top-1/3/5/10 positions. Compared with it, the improvement of our approach is 268.4%, 222.2%, 187.8% and 213.6% regarding the recall at Top-1/3/5/10. When comparing the Top-1 recall, the improvement of our approach over the baseline approaches ranges from 268.4% to 483.3%, and the average improvement is 351.3%. Particularly, VarDT significantly outperforms the latest state-of-the-art UniVal by 268.4% regarding the bugs located at Top-1. The Top-1 recall is important since it indicates that the first returned variable will be the desired one, which will waste no effort for manual inspection. Regarding the metrics of MAR and MFR, our approach also outperforms the competitors with at least 79.9% and 74.6% improvement, and the average improvement is 81.3% and 77.1%, respectively. In addition, we further analyzed the types of bugs that can be accurately located by our approach, which is presented in Figure 4. In the figure, we present the number of bugs whose fault-relevant variables can be located at Top-1 by VarDT (i.e., 70 bugs in total). From the figure, we can see that VarDT is effective in locating different types of bugs, especially for

Table 4. Overall experimental result comparison regarding all metrics with statistical significance test

| Metric | VarDT | UniVal | Baah2010 | ESP | NUMFL-DLRM | NUMFL-QRM | Ochiai | DStar(Star=2) |
|---|---|---|---|---|---|---|---|---|
| **Top-1** | **23.5%** | 6.4% | 5.7% | 5.7% | 4.0% | 4.0% | 5.7% | 6.0% |
| | | 1.21E-03 | 8.03E-04 | 6.47E-04 | 5.20E-04 | 6.48E-04 | 1.21E-03 | 1.22E-03 |
| **Top-3** | **38.9%** | 12.1% | 9.4% | 9.1% | 8.7% | 11.1% | 11.7% | 12.1% |
| | | 5.30E-04 | 5.29E-04 | 4.37E-04 | 6.53E-04 | 6.52E-04 | 7.97E-04 | 7.97E-04 |
| **Top-5** | **47.3%** | 14.4% | 14.4% | 13.4% | 11.7% | 14.8% | 16.4% | 16.4% |
| | | 6.50E-04 | 9.76E-04 | 4.37E-04 | 4.36E-04 | 9.70E-04 | 9.67E-04 | 9.67E-04 |
| **Top-10** | **62.1%** | 19.5% | 17.8% | 17.4% | 17.1% | 19.8% | 19.8% | 19.8% |
| | | 4.34E-04 | 6.52E-04 | 4.37E-04 | 4.37E-04 | 6.52E-04 | 6.52E-04 | 6.52E-04 |
| **MFR** | **7.5** | 29.6 | 33.1 | 45.6 | 32.0 | 31.4 | 31.3 | 31.0 |
| | | 9.35E-04 | 7.76E-04 | 6.43E-04 | 5.31E-04 | 5.31E-04 | 7.76E-04 | 7.76E-04 |
| **MAR** | **10.7** | 53.1 | 57.4 | 68.5 | 56.9 | 54.5 | 56.7 | 55.7 |
| | | 4.38E-04 | 4.38E-04 | 6.43E-04 | 5.31E-04 | 5.30E-04 | 5.31E-04 | 5.31E-04 |

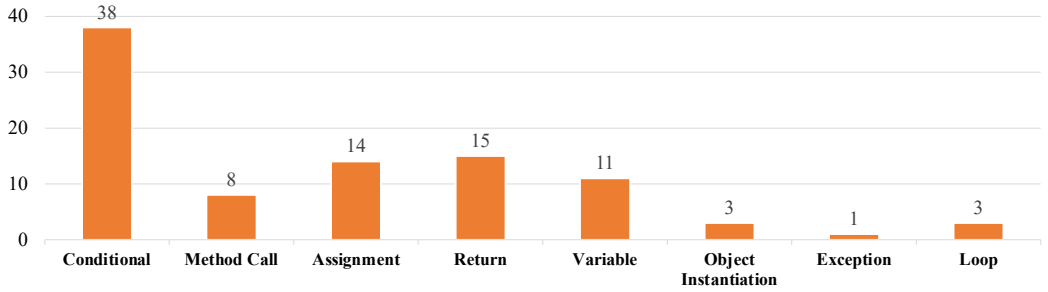



Fig. 4. The number of bugs whose fault-relevant variables can be accurately located at Top-1 by VarDT (70 bugs in total). The *x-axis* denotes the types of source code that need to be changed (delete/insert/replace) for repairing the bug. Particularly, one buy may change multiple locations related to the same variable.

those that need to repair conditional expressions. The results demonstrate that our approach is much more effective.

Though effective, the absolute number of bugs located at Top-1 by VarDT is still small (i.e., 23.5%). A major reason is the inaccuracy of the coarse-grained fault localization results used by VarDT. As it will be presented later (see Figure 5), when providing the accurate method-level FL results, the effectiveness of VarDT can be significantly improved. Please note that the results of the compared approaches in our experiment are much worse than those results reported in the previous study [37]. To ensure the correctness of the results, we further carefully checked them manually. In addition, since the results were produced using the virtual machine published by the authors, we believe they should be reliable. By further analyzing the results we found that the reasons for their poor performance are three-fold. First, they mainly measure the correlation between variables and test outcomes while tend to overlook the internal relationships among different variables. Although some of them have tried to capture such relationships in an indirect way (e.g., UniVal and Baah2010 take the confounding bias among variables into consideration), but the weak test suite makes them less effective due to the lack of sufficient training data. Second, the baseline approaches mainly consider variables of primitive types while the fault-relevant variables of `Object` type will be missed without unfolding the fields of them. Finally, their ranking strategies incline to numerical variables. Furthermore, the decline may also be partially caused by the different definitions of ground-truth variables, but they were not published by the authors, and thus we

Table 5. Comparison for predicates and numeric assignment bugs (230 bugs, *p-value*<0.05 for all.)

| Metric | VarDT | UniVal | Baah2010 | ESP | NUMFL-DLRM | NUMFL-QRM | Ochiai | DStar(Star=2) |
|---|---|---|---|---|---|---|---|---|
| **Top-1** | **26.1%** | 8.3% | 7.0% | 7.4% | 5.2% | 4.8% | 7.0% | 7.4% |
| **Top-3** | **40.4%** | 15.2% | 11.3% | 11.3% | 10.4% | 13.0% | 13.9% | 14.3% |
| **Top-5** | **49.1%** | 17.8% | 17.4% | 16.5% | 13.9% | 17.8% | 20.0% | 20.0% |
| **Top-10** | **63.0%** | 23.9% | 21.7% | 20.9% | 20.0% | 23.9% | 24.3% | 24.3% |
| **MFR** | **7.2** | 26.5 | 31.3 | 44.6 | 29.3 | 28.5 | 29.1 | 29.2 |
| **MAR** | **10.6** | 51.3 | 57.2 | 69.7 | 56.2 | 53.3 | 56.0 | 55.2 |

Table 6. Top-1 results of all approaches over different projects

| Project | VarDT | UniVal | Baah2010 | ESP | NUMFL-DLRM | NUMFL-QRM | Ochiai | DStar(Star=2) |
|---|---|---|---|---|---|---|---|---|
| Compress | **11.1%** | **11.1%** | **11.1%** | 0.0% | **11.1%** | 0.0% | **11.1%** | **11.1%** |
| Gson | **42.9%** | 14.3% | 14.3% | 0.0% | 0.0% | 0.0% | 14.3% | 14.3% |
| Codec | **20.0%** | 0.0% | 0.0% | 0.0% | 10.0% | 10.0% | 0.0% | 0.0% |
| Csv | **20.0%** | **20.0%** | **20.0%** | **20.0%** | 0.0% | 0.0% | **20.0%** | **20.0%** |
| Lang | **26.1%** | 13.0% | 4.4% | 17.4% | 8.7% | 4.4% | 0.0% | 0.0% |
| JXml | **25.0%** | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% |
| Chart | **25.0%** | 8.3% | 8.3% | 8.3% | 0.0% | 0.0% | 0.0% | 0.0% |
| JCore | **6.3%** | **6.3%** | 0.0% | 0.0% | 0.0% | 0.0% | **6.3%** | **6.3%** |
| Jsoup | **16.7%** | 5.6% | 0.0% | 5.6% | 5.6% | 0.0% | 0.0% | 0.0% |
| JxPath | **12.5%** | 0.0% | 0.0% | 0.0% | 0.0% | **12.5%** | 0.0% | 0.0% |
| Math | **26.1%** | 13.0% | 13.0% | 17.4% | 13.0% | 13.0% | 13.0% | 13.0% |
| JDatabind | **32.1%** | 3.6% | 10.7% | 3.6% | 0.0% | 0.0% | 17.9% | 17.9% |
| Time | **25.0%** | 0.0% | 8.3% | 0.0% | 0.0% | 8.3% | 0.0% | 8.3% |
| Cli | **37.5%** | 12.5% | 0.0% | 12.5% | 12.5% | 12.5% | 0.0% | 0.0% |
| Mockito | **28.6%** | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 4.8% | 4.8% |
| Closure | **22.3%** | 5.3% | 5.3% | 4.3% | 3.2% | 4.3% | 4.3% | 4.3% |
| **TOTAL** | **23.5%** | **6.4%** | **5.7%** | **5.7%** | **4.0%** | **4.0%** | **5.7%** | **6.0%** |

cannot reproduce their results. On the other hand, most of the compared baselines were designed to locate predicates and numeric assignment statements. To investigate whether it would cause the performance decline of them, we further confined the comparison only to these bugs in Table 5, which contains 230 bugs in total. Based on the results shown in Table 4 and 5, the performance of the baseline approaches was indeed improved when only considering the bugs related to predicates and numeric assignments. Nevertheless, our approach still significantly outperformed all the baselines by localizing 214.5%~443.8% more bugs at Top-1, indicating the effectiveness of our approach.

To investigate the performance of difference approaches on different projects, we further reported the detailed Top-1 and MAR/MFR results of them on each project in Table 6 and 7, respectively. Since the number of bugs from individual project is relatively small, we do not perform the statistical test due to insufficient support samples. According to the results, VarDT performed consistently well over different projects, and always outperforms the baselines regarding Top-1 recall, indicating the generalizability and superiority of our approach. Apart from VarDT, UniVal also performed relatively stable regarding Top-1 over different projects, owing to its effective modeling of confounding bias between variables and test results, whereas the performance of other approaches is easier to be affected by the targeted projects. Besides, the results also show that most of approaches consistently performed well on the projects of Compress and Csv. The reason is that the commonly located fault-relevant variables are either producing null pointer errors or the only ones in the candidate methods, both of which are easy to be located. When comparing the results of MAR and MFR, VarDT still outperforms the baseline approaches on most of the

Table 7. MAR and MFR results of all approaches over different projects

| Project | VarDT | | Unival | | Baah2010 | | ESP | | NUMFL-DLRM | | NUMFL-QRM | | Ochiai | | Dstar(Star=2) | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | MAR | MFR | MAR | MFR | MAR | MFR | MAR | MFR | MAR | MFR | MAR | MFR | MAR | MFR | MAR | MFR |
| Compress | **14.9** | **11.5** | 54.6 | 45.0 | 38.0 | 29.4 | 48.0 | 34.2 | 44.3 | 27.4 | 45.3 | 27.4 | 41.0 | 30.2 | 34.8 | 24.0 |
| Gson | **7.2** | **6.0** | 22.0 | 22.0 | 51.5 | 51.5 | 54.5 | 54.5 | 62.0 | 62.0 | 62.0 | 62.0 | 58.5 | 58.5 | 58.5 | 58.5 |
| Codec | **22.8** | **17.4** | 60.0 | 33.0 | 60.5 | 38.5 | 66.7 | 54.5 | 44.4 | 26.0 | 45.3 | 27.2 | 49.3 | 28.2 | 48.8 | 27.6 |
| Csv | **8.3** | **2.0** | 53.8 | 52.3 | 26.7 | 24.7 | 60.9 | 60.0 | 32.5 | 28.3 | 28.5 | 24.0 | 28.9 | 25.7 | 28.9 | 25.7 |
| Lang | **9.6** | **6.1** | 106.2 | 82.0 | 101.2 | 58.3 | 173.9 | 108.3 | 103.7 | 49.7 | 95.7 | 47.3 | 93.4 | 68.2 | 93.3 | 68.2 |
| JXml | **4.3** | **4.3** | 9.0 | 9.0 | 5.0 | 5.0 | 41.0 | 41.0 | 17.0 | 17.0 | 18.0 | 18.0 | 7.0 | 7.0 | 7.0 | 7.0 |
| Chart | **6.5** | **4.4** | 24.7 | 7.5 | 45.7 | 19.3 | 46.2 | 39.8 | 54.3 | 36.5 | 49.2 | 34.5 | 44.3 | 19.3 | 45.3 | 19.3 |
| JCore | **16.7** | **12.6** | 124.4 | 34.4 | 168.7 | 104.8 | 164.5 | 74.8 | 120.4 | 54.0 | 89.7 | 38.5 | 134.1 | 35.7 | 125.8 | 35.5 |
| Jsoup | **12.4** | **10.6** | 37.3 | 30.5 | 41.9 | 23.0 | 37.4 | 30.0 | 51.4 | 36.4 | 53.5 | 38.0 | 59.2 | 40.5 | 59.5 | 41.5 |
| JxPath | 15.6 | 12.4 | 30.3 | 30.0 | 24.3 | 24.0 | **8.3** | **8.0** | 14.8 | 14.0 | 12.3 | 12.0 | 30.3 | 30.0 | 30.3 | 30.0 |
| Math | **12.9** | **5.8** | 87.1 | 17.6 | 100.2 | 22.2 | 97.9 | 27.4 | 101.5 | 27.9 | 105.3 | 32.8 | 104.3 | 25.9 | 104.3 | 25.9 |
| JDatabind | **10.9** | 8.2 | 34.3 | 6.8 | 37.3 | **3.4** | 47.2 | 20.5 | 36.3 | 12.7 | 36.8 | 13.7 | 30.8 | 5.9 | 30.8 | 5.9 |
| Time | **7.8** | **5.1** | 68.1 | 43.8 | 64.0 | 27.5 | 91.1 | 63.0 | 57.6 | 29.0 | 58.5 | 30.2 | 75.6 | 41.7 | 75.5 | 41.5 |
| Cli | **5.1** | **2.6** | 18.9 | 3.7 | 20.6 | 8.0 | 37.7 | 37.7 | 30.6 | 16.3 | 31.0 | 17.0 | 23.2 | 6.7 | 23.2 | 6.7 |
| Mockito | 7.0 | 5.8 | 8.3 | 5.0 | 11.7 | 9.0 | 9.2 | 7.0 | 9.7 | 5.0 | 10.8 | 7.0 | 7.0 | **4.0** | **4.7** | **4.0** |
| Closure | **8.6** | **5.6** | 110.7 | 51.7 | 121.6 | 81.9 | 110.9 | 69.3 | 130.6 | 69.7 | 130.1 | 72.3 | 120.6 | 73.1 | 119.9 | 75.4 |
| **Total** | **10.7** | **7.5** | **53.1** | **29.6** | **57.4** | **33.1** | **68.5** | **45.6** | **56.9** | **32.0** | **54.5** | **31.4** | **56.7** | **31.3** | **55.7** | **31.0** |

Table 8. Comparison with IsoVar (185 bugs)

| Metric | Top-1 | Top-3 | Top-5 | Top-10 | MFR | MAR |
|---|---|---|---|---|---|---|
| **VarDT** | **24.3%** | **40.5%** | **48.6%** | **63.8%** | **5.5** | **8.7** |
| **IsoVar** | 3.2% | 10.3% | 14.1% | 24.9% | 78.5 | 90.6 |

Table 9. Comparison with Q-SFL (92 bugs)

| Metric | Top-1 | Top-3 | Top-5 | Top-10 | MFR | MAR |
|---|---|---|---|---|---|---|
| **VarDT** | **28.3%** | **45.7%** | **51.1%** | **70.7%** | **4.5** | **7.2** |
| **Q-SFL** | 8.8% | 17.6% | 23.1% | 23.1% | 2.7 | 2.7 |

projects. Specifically, VarDT achieved the best MAR on 14 out of 16 projects, and the best MFR on 13 projects. In particular, VarDT performed slightly worse than ESP on JxPath and than DStar on Mockito. The reason is that ESP failed to locate any fault-relevant variables for all bugs except JxPath-8 from JxPath (please refer to the calculation of MFR/MAR in Section 4.3.1). In other words, the MFR/MAR for ESP on JxPath was calculated based on only one bug, while for our approach, these metrics were calculated based on multiple bugs. In fact, our approach achieved better results than ESP when only considering the bug of JxPath-8. Regarding the worse result compared with DStar on Mockito, our approach achieved relatively poor performance on the bug of Mockito-34, which greatly affected its MFR/MAR results on Mockito. In general, our approach consistently outperforms baseline approaches on the vast majority of projects.

Finally, we compare our approach with other two baseline approaches, i.e., IsoVar and Q-SFL. As introduced in Section 4.2, we compare VarDT with them over a subset of bugs used in our experiment to make a fair comparison. Specifically, 185 bugs are used by both IsoVar and VarDT, while 92 bugs are used by both Q-SFL and VarDT. Tables 8 and 9 present the corresponding experimental results. From the results, we can see that VarDT outperformed IsoVar on all the metrics. Specifically, our approach outperformed IsoVar by localizing 659.4% more bugs at Top-1. The reason for the poor performance of IsoVar is mainly because it only considers individual variables but ignores complex expressions, e.g, `if` conditions. However, expression errors are indeed common in real practice, such as misusing a logic operator in a `if` condition [30], misusing an API [40], etc. In summary, VarDT outperformed IsoVar by correctly localizing 6.6x more bugs at Top-1. When comparing our approach with Q-SFL, VarDT still performed much better regarding Top-1, and the improvement is about 221.6%. However, Q-SFL achieved better results than VarDT regarding MAR and MFR. The reason is that Q-SFL only considers method parameters and return values, which leaves Q-SFL a small number of candidate variables. Therefore, the ranking of fault-relevant variables tends to be small. To sum up, the result comparison with IsoVar and Q-SFL demonstrates that our approach is effective in localizing fault-relevant variables.

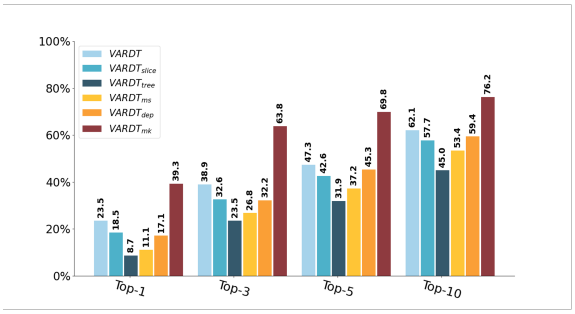



Fig. 5. Effects of different components in VarDT

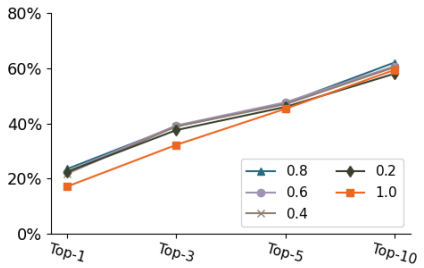



Fig. 6. Effects of dependency penalty factors

## 5.2 RQ2: Contribution of Each Component

In order to evaluate the effectiveness of each component in VarDT, we have conducted an ablation study with a set of variants of VarDT, which have been introduced in Section 4.2. Figure 5 presents the results of each variant regarding the metrics of Top-N recall, MFR and MAR. According to the results, all components in VarDT largely contributed to the overall effectiveness of VarDT since a large drop on the Top-N recall was incurred when removing any one of them. Specifically, regarding the metric of Top-1 recall, the dynamic program slicing contributed 27.0% higher effectiveness (*vs* VarDT$_{slice}$), the tree model contributed 170.1% (*vs* VarDT$_{tree}$), the *dependency penalty* contributed 37.4% (*vs* VarDT$_{dep}$), and the use of method score for variable ranking contributed 111.7% (*vs* VarDT$_{ms}$), respectively. However, all of them still outperform the baseline approaches. In particular, the core novel component (i.e., tree model) in VarDT makes the largest contribution. In summary, the ranking of component contributions is *tree model* > *method score* > *dependency penalty* > *program slicing* regarding Top-N.

Since the method score largely affects the effectiveness of VarDT, a question may naturally raise: Whether VarDT can be further improved by providing a more accurate fault localization result (e.g., providing the genuine faulty method). Therefore, we empirically evaluated the fault localization results of VarDT in the circumstance where the faulty method was known, for which we created another variant VarDT$_{mk}$. The results of VarDT$_{mk}$ are also presented in Figure 5. By providing the accurate method-level fault localization results, VarDT$_{mk}$ successfully located the desired variables for 39.3% of bugs at Top-1, the improvement over VarDT is about 67.2%. Moreover, the Top-10 recall (the maximum number of candidates that developers are willing to inspect according to the previous study [36]) is as high as 76.2%, indicating the promise of incorporating a more effective method-level fault localization technique into VarDT.

Then, we further investigated the impact of the configuration for *DEP_FACTOR*, which represents the strength of *dependency penalty*. According to Formula 1, the smaller the value is, the larger the penalty will be (i.e., the variable will be less likely to be selected). Figure 6 presents the results when taking different values, where 0.8 is the default value. From the figure, we can see that VarDT achieved the best overall result when taking the value in $[0.6, 0.8]$, and the impact of this configuration is relatively small. Specifically, when taking 0.8, VarDT achieved the highest Top-1 recall as 23.5%, whereas it achieved the lowest Top-1 recall as 17.1% when taking 1.0. The result indicates that VarDT is insensitive to this configuration although it is indeed important according to the result of VarDT$_{dep}$, which completely removes the *dependency penalty* component.

Finally, we also investigated the impact of program slicing in depth on the **space reduction** of candidate variables. The result shows that the reduction ratio ranges from 18.5% to 42.2% over different projects, and on average is 31.9%, denoting the necessity of it for improving efficiency.

Table 10. Patch filtering result comparison with PATCH-SIM and BATS

| Project | #Plausible(Correct) | VarDT$_{Top\text{-}1}$ | VarDT$_{Top\text{-}3}$ | VarDT$_{Top\text{-}5}$ | VarDT$_{Top\text{-}10}$ | PATCH-SIM | BATS |
|---|---|---|---|---|---|---|---|
| Math | 24(3) | 22(2) | 22(2) | 20(1) | 14(0) | 15(0) | 7(3) |
| Chart | 14(2) | 14(2) | 8(1) | 7(1) | 6(1) | 4(0) | 5(1) |
| Lang | 10(4) | 10(4) | 10(4) | 10(4) | 9(3) | 2(0) | 4(1) |
| Time | 8(1) | 7(1) | 6(0) | 6(0) | 6(0) | 6(0) | 0(0) |
| **Total** | 56(10) | 53(9) | 46(7) | 43(6) | 35(4) | 27(0) | 16(5) |
| **Precision** | | 83.0% | 84.8% | 86.0% | 88.6% | **100.0%** | 68.8% |
| **Recall** | | **95.7%** | 84.8% | 80.4% | 67.4% | 58.7% | 23.9% |

### 5.3 RQ3: Performance in Patch Filtering

To evaluate whether our finer-grained variable-level fault localization results can further promote the effectiveness of downstream APR techniques, we adapted VarDT to the task of patch filtering and compared the result with the state-of-the-art PATCH-SIM and BATS. The details have been introduced in Section 4.2.

Table 10 presents the experimental result comparison between our approach and the two baseline approaches. Specifically, we list the number of all plausible and correct patches per each project in the left part of the table, while list the filtering results in the right part. Particularly, VarDT$_{Top\text{-}N}$ represents that we use the Top-N variables located by VarDT to filter patches according to the process introduced in Section 4.2. In each cell, *X(Y)* denotes the corresponding approach in total filtered *X* patches, in which *Y* patches were correct patches. According to the result, although our approach was not designed as a comprehensive and standalone patch filtering technique, it can still achieve comparable and even better results compared with the state-of-the-art patch filtering approaches. Specifically, on the dataset from PATCH-SIM (removing bugs not included in Table 3), VarDT could filter out about 67.4% incorrect patches using the Top-10 results of VarDT, while PATCH-SIM only filtered 58.7% and BATS only filtered 23.9%. That is, VarDT outperforms PATCH-SIM by 14.8% and BATS by 181.8%. Particularly, the patch precision (the percentage of correct patches over all patches) increased to 28.6% and 34.5% from 17.9% by VarDT$_{Top\text{-}10}$ and PATCH-SIM respectively after filtering. However, the patch precision decreased to 12.5% by BATS after filtering. The result indicates the performance of VarDT and the feasibility of boosting automatic program repair techniques by filtering incorrect patches using a finer-grained fault localization. It also reflects the reliability of our ground truth since it is indeed closely related to the repair of the bug. Besides, designing new automatic program repair techniques based on the variable-level fault localization potentially can further improve the number of correct patches since many incorrect patches can be avoided to be generated in the online repair process, and thus correct patches will have more possibility to be generated. More studies can be conducted in this direction.

Though effective, our approach tends to incorrectly filter out correct patches compared with PATCH-SIM. For example, 4 correct patches were filtered out by VarDT$_{Top\text{-}10}$ while none by PATCH-SIM. Particularly, with the decrease of variable numbers (i.e., from Top-10 to Top-1), although more incorrect patches can be filtered out, the *precision* of filtering will also sharply drop. When using the Top-1 result (i.e., only one candidate variable for each bug), 9 out of 10 correct patches will be filtered due to the inaccuracy of VarDT. Specifically, 9-4=5 correct patches were mistakenly filtered by VarDT$_{Top\text{-}1}$ because the genuine fault-relevant variables were located within Top-10 but not Top-1, and 2/4 correct patches filtered by VarDT$_{Top\text{-}10}$ because the fault-relevant variables were ranked out of Top-10. As for the other two correct patches filtered by VarDT$_{Top\text{-}10}$, the reason is that they are not syntactically similar to the developer patches. However, the fault-relevant variables were identified based on the developer patches as mentioned in Section 4.3. As a result, even though VarDT can correctly locate the fault-relevant variables, the patches may still be filtered. For instance,

to repair the bug Chart-3, developers introduced two new statements `copy.minY=Double.NaN` and `copy.maxY=Double.NaN` to reset the variable `copy` in the faulty method `createCopy(*)`, and VarDT indeed correctly located the variable `copy` at Top-1. However, the APR patch fixed this bug by inserting a new function call `findBoundsByIteration()` in a different method (i.e., `add(*)`) from the developer patch. As a consequence, the APR patch was filtered since it changed no variable in the method `createCopy(*)`. Additionally, since there may be multiple fault-relevant variables for a bug, depending on the Top-1 variable may also potentially harm the efficacy of patch filtering. According to the analysis above, we can notice that it may be not adequate to filter incorrect patches by simply identifying whether any fault-relevant variable is changed in the patch since a bug can be fixed in different ways, a deeper analysis may be needed in some cases, such as considering the side effects [9, 17] of the patch. The preliminary study in this paper is our first attempt at filtering incorrect patches by using fault-relevant variables, which already shows the promise of it. We plan to conduct a more systematic study in our future work.

Particularly, after further analyzing the results of different approaches, we found that there were only 19 incorrect patches that were commonly filtered out by both VarDT$_{Top\text{-}10}$ and PATCH-SIM. In other words, (35+27)-19=43 incorrect patches and 4 correct patches could be filtered by combining these two, leaving the *precision* and *recall* of filtering as 91.5% and 93.5%, respectively, and the patch precision will also increase to 66.7%. The result indicates that our approach complements PATCH-SIM and thus they can be used together to further promote the performance of APR techniques. In contrast, when combining the results of VarDT$_{Top\text{-}10}$ and BATS, (31+11)-10=32 incorrect patches and (4+5)-0=9 correct patches could be filtered, leaving the *precision* and *recall* of filtering as 78.0% and 69.6%, respectively, and the patch precision will decrease to 6.7%. The major reason is that both our approach and BATS tend to filter correct patches.

### 5.4 RQ4: Efficiency of VarDT

In order to analyze the efficiency of VarDT, we have recorded the time cost of each process in our experiment. Table 11 lists summarized results. According to the results, we can see that most of the processes are indeed efficient. For example, the slicing process takes in average 28.5s, while the program instrumentation and the tree building processes are much more efficient. The most time-consuming process is to collect variable values, which needs to run the test cases. However, in order to improve the efficiency of VarDT, we have optimized the collecting process by only running the test cases that cover the variables we target. As a result, the average time cost for this process is about 2 minutes. In summary, the overall running time of our approach is acceptable for practical use.

## 6 DISCUSSION

**Interpretability**: According to the experimental results shown in Section 5, our approach can indeed promote the effectiveness of downstream APR techniques. In this section, we further explore whether the variables located by VarDT are also useful in the task of manual debugging, or rather whether the built tree models by VarDT have good interpretability and can provide debugging insights. Therefore, we have performed a small user study. Specifically, we sampled 39 bugs that can be located by VarDT within Top-3 locations from our dataset and conducted a manual analysis on them. After carefully analyzing the patch code, we conclude that the located variables can indeed provide insightful information to promote the understanding of the bugs for 38 out of 39 bugs. The only one for which we think the located variables by VarDT do not help much is the bug of JacksonCore-15 (the patch code is shown in Listing 3). The reasons are twofold. On one hand, the bug is naturally complex to understand as shown in Listing 3; On the other hand, the located variables by VarDT are those like `_headContext` and `_currToken`, which are still far from the

Table 11. Timecost of each process in VarDT (in seconds)

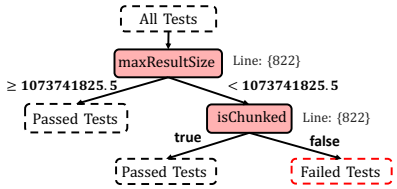
| Project | Tracing | Slicing | Instrumentation | Collect Values | Dependency Analysis | Tree Building | Total |
|---|---|---|---|---|---|---|---|
| **Chart** | 42.0 | 11.3 | 4.0 | 12.5 | 9.0 | 0.4 | 79.1 |
| **Cli** | 10.0 | 1.2 | 2.3 | 2.3 | 1.0 | 3.5 | 20.3 |
| **Codec** | 16.8 | 3.3 | 2.2 | 8.4 | 2.1 | 0.6 | 33.3 |
| **Compress** | 21.1 | 131.5 | 3.1 | 11.2 | 2.3 | 0.3 | 169.5 |
| **Csv** | 11.2 | 2.7 | 2.1 | 22.5 | 2.1 | 0.3 | 40.8 |
| **Gson** | 18.5 | 3.4 | 3.2 | 7.2 | 2.7 | 0.7 | 35.6 |
| **JCore** | 66.3 | 8.2 | 11.4 | 33.3 | 4.7 | 4.5 | 128.3 |
| **JDatabind** | 105.1 | 32.4 | 21.2 | 80.0 | 24.3 | 6.9 | 269.9 |
| **JXml** | 22.5 | 0.3 | 0.3 | 15.5 | 2.3 | 1.5 | 42.4 |
| **Jsoup** | 50.0 | 6.6 | 3.1 | 33.5 | 5.8 | 3.9 | 103.0 |
| **JxPath** | 20.0 | 15.8 | 5.0 | 10.2 | 10.9 | 3.3 | 65.3 |
| **Lang** | 40.1 | 2.6 | 2.3 | 47.8 | 2.2 | 0.3 | 95.3 |
| **Math** | 136.2 | 89.9 | 3.1 | 532.3 | 23.4 | 0.5 | 785.5 |
| **Time** | 73.8 | 39.0 | 11.0 | 48.2 | 32.0 | 1.2 | 205.1 |
| **Mockito** | 349.1 | 4.1 | 33.6 | 1112.8 | 4.7 | 1.3 | 1505.5 |
| **Closure** | 113.2 | 104.4 | 21.7 | 243.5 | 70.6 | 40.8 | 594.3 |
| **AVERAGE** | **68.5** | **28.5** | **8.1** | **138.8** | **12.5** | **4.4** | **260.8** |

patch as the expected values require the calls to additional functions, i.e., `isScalarValue()` and `isStartHandled()`. In the current implementation, we do not include such function calls since it may cause additional errors due to unknown side-effects [9, 64].

As for the large portion of bugs that VarDT can provide insights for manual debugging, besides the running example shown in Figure 3, we further present another example. Listing 2 presents the patch code. For fixing this bug, the second argument (i.e., `MINE_CHUNK_SIZE`) was replaced by a conditional expression. To locate the fault-relevant variables, VarDT constructed the decision tree and obtained the final candidate variables as shown in Figure 7. From the figure we can see that the constructed branch condition in the tree model actually closely correlated with the failure (or rather the same condition as that to be inserted in the patch), based on which developers will be much easier to fix the bug. However, as it also can be seen that although VarDT is effective and useful, it may also provide misleading messages. For example, the located variable `maxResultSize` may guide developers to check the fault-irrelevant `if` condition in Line 828 in Listing 2, which may wast manual efforts. Besides, since the equivalent variables (present in the same execution path without re-assignment between them as explained in Section 3.3.1) are merged by VarDT and thus the located variables may not always tell the desirable lines of code to be fixed accurately. As a consequence, developers may be confused by the results. For instance, the located fault-relevant variable `isChunked` appears at line 822 whereas the code change exactly takes place at line 827. However, we argue that it is natural and reasonable since the line of code to be fixed may originally not use such variables. In such cases, the traditional line-level fault localization potentially can be further combined. In the future, we plan to further investigate the performance of VarDT by combining with existing fault localization approaches, and incorporating more kinds of predicates, e.g., function calls and global variables. Furthermore, we also plan to conduct a more comprehensive study on the performance of VarDT in the real manual debugging process for assisting developers.

```
822 byte[] encodeBase64(byte[] binaryData,boolean isChunked,boolean urlSafe,int maxRsltSize) {
823     if (binaryData == null || binaryData.length == 0) {
824         return binaryData;
825     }
826
827-    long len = getEncodeLength(binaryData, isChunked ? MIME_CHUNK_SIZE : 0, CHUNK_SEPARATOR);
827+    long len = getEncodeLength(binaryData, MIME_CHUNK_SIZE, CHUNK_SEPARATOR);
828     if (len > maxResultSize) {
829         throw new IllegalArgumentException("Input array too big, the output ... (" +
```

Listing 2. Patch code of Codec-9 from Defects4J benchmark.




| Variables | Score |
|---|---|
| Base64#byte[]#encodeBase64#byte[],boolean,boolean,int#isChunked-822 | 0.432 |
| Base64#long#getEncodeLength#byte[],int,byte[]#pArray-969 | 0.240 |
| Base64#void#encode#byte[],int,int#in-437 | 0.148 |
| Base64#byte[]#encodeBase64#byte[],boolean,boolean,int#maxResultSize-822 | 0.125 |
| ... | |

In the table, the associated classes, methods (including return and argument types) and line numbers of the variables are also presented.

Fig. 7. The built tree model and the final rank of candidate variables for Codec-9

```
222 public JsonToken nextToken() throws IOException {
  -     if(!_allowMultipleMatches && _currToken != null && _exposedContext == null){
  -         if((_currToken.isStructEnd() && _headContext.isStartHandled()) ){
  -             return (_currToken = null);
  -         }
  -         else if(_currToken.isScalarValue() && !_headContext.isStartHandled() && !_includePath
  -                 && _itemFilter == TokenFilter.INCLUDE_ALL) {
  -             return (_currToken = null);
  -         }
  -     }
231       TokenFilterContext ctxt = _exposedContext;
          ...
272       JsonToken t = delegate.nextToken();
```

Listing 3. Patch code of JacksonCore-15 from Defects4J benchmark.

**Limitation and future work**: As explained in Section 4.1, VarDT requires that at least three test cases (including at least one failed and one passed) cover the faulty method, which may affect the usability of our approach in practice since the accompanied test suite tend to be weak [55, 78]. In such cases, the state-of-the-art test generation approaches [21, 22, 86] may be potentially combined to overcome this limitation. Moreover, besides collecting the variable values like existing approaches, our approach will also introduce the extra overhead for program dependency analysis and tree model construction. However, since the overhead of these two processes is relatively small, i.e., 16.9s on average, it can be acceptable in practice.

In addition, our approach locates fault-relevant variables by leveraging the enhanced decision tree to construct the constraints and relations between variables. As presented in Algorithm 3, the constructed conditions largely depend on the quality of given test cases. As a result, even though the localized variables are correct, the constructed conditions (or variable constraints) may be still meaningless, especially for bugs that involve complex constraints but weak tests. Additionally, the predicates included in our current implementation are still limited, and VarDT estimates the relationship between variables by catching the constraints of each variable separately, which may also confine the effectiveness of our approach since some relationships may be hard to be estimated by assembling standalone constraints, such as $a\%b = 0$ and so on. On the contrary, VarDT tends to perform better on conditional bugs (refer to Figure 4) due to their simple value scope (i.e., false and true). To further improve the capability of VarDT for constructing complex relations, a more effective tree condition construction algorithm that can incorporate domain knowledge will help, such as program invariant inference [69, 89], which potentially can assist VarDT in constructing more complex constraints and better catching the relationships between variables. We plan to explore more effective tree construction algorithms and combine invariant inference in the future.

Finally, in our experiment, we only consider the bugs that can be located within Top-10 by the method-level fault localization. When considering all the bugs, we have to collect variable values in all methods covered by failing test cases, which is indeed time-consuming based on the time cost shown in Table 11. However, in order to investigate the performance of our approach in such a case, we have conducted an extra experiment over a subset of projects. Specifically, we employed all the

Table 12. Results of VarDT on all methods

| Project | #Bugs | Top-1 | Top-3 | Top-5 | Top-10 | MFR | MAR |
|---|---|---|---|---|---|---|---|
| **Chart** | 16 | 3 | 8 | 9 | 11 | 9.0 | 11.3 |
| **Csv** | 13 | 1 | 5 | 5 | 6 | 12.9 | 19.5 |
| **Gson** | 14 | 3 | 4 | 4 | 6 | 10.4 | 20.1 |
| **Jsoup** | 73 | 5 | 11 | 18 | 23 | 77.4 | 88.8 |
| **JxPath** | 15 | 1 | 2 | 3 | 5 | 11.4 | 17.8 |
| **Lang** | 46 | 11 | 14 | 18 | 28 | 8.9 | 11.8 |
| **Math** | 57 | 9 | 17 | 22 | 31 | 12.0 | 20.6 |
| **Time** | 18 | 3 | 7 | 8 | 14 | 19.3 | 25.4 |
| **TOTAL** | **252** | **14.3%** | **27.0%** | **34.5%** | **49.2%** | **20.2** | **26.9** |

bugs (i.e. 252 bugs) that have at least three test cases covering the buggy method regardless of the method-level fault localization. In other words, the bugs used in this study include those bugs whose genuine faulty methods were not located within Top-10 by the method-level fault localization, and VarDT collected the variable values in all the methods covered by the failed test cases. Then, VarDT ranked all the variables based on their scores computed by Formula 8. Table 12 presents the experimental results. As shown, Top-1/3/5/10 results of VarDT decreased by 39.1%, 30.6%, 27.1%, 20.8% respectively compared with the results over Top-10 methods. However, VarDT can locate the fault-relevant variables at Top-1 for 14.1% bugs, which still outperforms the baselines. The results further confirmed the effectiveness of VarDT. However, improving the running efficiency and the performance of VarDT without the aid of method-level fault localization can further improve its usability in real practice, we leave it as our future work.

## 7 THREATS TO VALIDITY

The threats to internal validity mainly lie in the implementation and ground truth used in our experiment. In order to ensure the reliability of our implementation, two authors have carefully checked its correctness through code review, which can mitigate this threat to some degree. Regarding the ground truth, we have provided a clear definition of fault-relevant variables, based on which we manually analyzed the source code and measured the consistency of analysis results by the Cohen's Kappa coefficient. Therefore, we believe it is reliable. Additionally, the evaluation result of VarDT in the patch filtering application also improves our confidence. However, there may still exist fault-relevant variables that we were not aware of and thus were not included in our definition, which may affect the results of our study. To facilitate the replication of our results and provide a method for more comprehensive comparisons under different conditions in future studies, we have published all our data and implementation.

The threats to external validity mainly lie in the used subjects. In our experiment, we only adopted a subset of the bugs from the Defects4J benchmark according to the constraints introduced in Section 4.1. However, since the studied bugs are from 15 different real-world projects, we believe it can be representative to some extent. The effectiveness of VarDT on a wider range of projects beyond Defects4J remains to be studied.

## 8 RELATED WORK

### 8.1 Variable-based Fault Localization

Our approach targets the variable-based fault localization, the most related techniques are Uni-Val [37], NUMFL [8], ESP [24], and Baah2010 [7], which have been introduced in Section 4.2. Different from them, our approach locates fault-relevant variables by leveraging decision trees to build variable constraints for discriminating failed and passed test runs, which is the first time as

far as we are aware. Most recently, Wen et al. [72] proposed to employ statistical and mutation analysis to isolate fault-relevant variables (named IsoVar). IsoVar is largely different from VarDT. Specifically, VarDT locates fault-relevant variables by constructing isolation constraints using variables while IsoVar depends on variable coverage; Moreover, VarDT considers intermediate variables in compound expressions (e.g., conditional expressions in `if` statements) besides individual variables whereas IsoVar does not. Besides, the statistical debugging [43] and its following work [4, 16, 24, 31, 45, 90] are also related to our approach, which depends on test coverage to compute the importance of a set of pre-defined predicates. On the contrary, our approach uses a dependency-enhanced tree model to identify fault-relevant variables, but not simply their coverage. In addition, existing studies also employed decision tree [12] or random forest models [73] in fault localization. However, they were designed for improving the statement-level fault localization. Similarly, Perez and Abreu [**?** ] proposed Q-SFL, which employs qualitative reasoning over method arguments of primitive types and return values to refine the coarse-grained (e.g, line-level) fault localization results, whereas our approach targets the variable level and additionally incorporates the dependency factor for model building. In addition, invariant inference [18, 20, 69, 89] is also related to our approach since both of them endeavor to identify interested variables by constructing the variable constraints based on the dynamic execution of the program. However, they are also significantly different. First, invariant inference identifies variable constraints by a set of pre-defined templates. In contrast, our approach catches the constraints by a decision tree model. Second, invariant inference takes only the variable values in the dynamic execution as inputs without considering the static program dependency, which is an important component in our approach.

Besides locating fault-relevant variables directly, several studies use variable/value profiles to boost statement-level fault localization. For example, a set of studies devoted to improving statement-level fault localization by replacing the values of certain expressions with alternative values in order to make the failed test pass [14, 28, 87]. Recent studies [51, 52] incorporated mutation analysis to improve fault localization. Shen et al. [60] combined statistical localization with directed fuzzing to overcome the over-fitting and estimation bias problem in fault localization. Different from them, our work aims at locating the finer-grained fault-relevant variables directly.

Finally, there are also some interventional fault localization approaches depending on variable values [83, 84], Compared with them, our approach is fully automatic. The latest studies also employed different models to combine the strength of multiple techniques [29, 42, 80, 92]. These techniques can be further combined with our approach and boost its effectiveness by providing a more precise method-level fault localization result. In turn, our approach may also improve existing techniques by integrating it into them.

### 8.2 Automatic Patch Filtering

In order to improve the patch quality (i.e., precision) in automatic program repair, researchers have proposed a series of patch filtering techniques. Among them, test-generation-based techniques are the most widely studied, and the core challenge is the lack of test oracles. Facing this challenge, existing studies employed different strategies. Yang et al. [81] proposed Opad, which filters patches that cause program crashes or produce memory errors. Xin and Reiss [77] proposed DiffTGen which depends on human experts to provide the test oracle. While Xiong et al. [78] proposed PATCH-SIM that estimates the test results by measuring the execution similarity of test cases before and after repair. On the contrary, Tan et al. [65] pre-defined a set of anti-patterns that easily produce incorrect patches for patch filtering. Recently, Ye et al. [82] proposed to employ a machine learning method to classify the correctness of patches, while Wang et al. [70] and Tian et al. [66] proposed to leverage deep-learning techniques to aid the identification of incorrect patches by learning from historical data. Different from existing approaches, our work focuses on improving

the fault localization effectiveness by providing finer-grained results, which can also aid the patch filtering process but from a different perspective, and thus is orthogonal to them.

## 9 CONCLUSION

Fault localization is a hot research topic and many approaches have been proposed in the last decades. However, most of existing approaches targeted the *precision problem*, whereas the *granularity problem* was paid much less attention. In this paper, we targeted the *granularity problem* and proposed a variable-level fault localization technique, named VarDT, in which we designed a novel program-dependency-enhanced decision tree model to aid the identification of fault-relevant variables. We have evaluated the effectiveness of VarDT in both fault localization and patch filtering applications by comparing with the state-of-the-art techniques. The results demonstrate that VarDT significantly outperformed the baseline fault localization approaches with at least 268.4% improvement regarding the bugs located at Top-1, and also outperformed the state-of-the-art patch filtering techniques by filtering 14.8%-181.8% more incorrect patches. The experimental results demonstrate the effectiveness of our approach.

## ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive suggestions to help improve the quality of this paper. This work was supported by the National Natural Science Foundation of China under Grant Nos. 62202324, 62322208 and 62232001.

## REFERENCES

[1] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. 2006. An Evaluation of Similarity Coefficients for Software Fault Localization *(Pacific Rim International Symposium on Dependable Computing)*. 39–46. https://doi.org/10.1109/PRDC.2006.18

[2] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. 2007. On the accuracy of spectrum-based fault localization. In *TAICPART-MUTATION*. 89–98.

[3] Hiralal Agrawal and Joseph R. Horgan. 1990. Dynamic Program Slicing *(Programming Language Design and Implementation)*. 246–256. https://doi.org/10.1145/93542.93576

[4] Piramanayagam Arumuga Nainar, Ting Chen, Jake Rosin, and Ben Liblit. 2007. Statistical Debugging Using Compound Boolean Predicates *(International Symposium on Software Testing and Analysis)*. 5–15. https://doi.org/10.1145/1273463.1273467

[5] Piramanayagam Arumuga Nainar and Ben Liblit. 2010. Adaptive Bug Isolation *(International Conference on Software Engineering)*. 255–264. https://doi.org/10.1145/1806799.1806839

[6] Tien-Duy B. Le, David Lo, Claire Le Goues, and Lars Grunske. 2016. A learning-to-rank based fault localization approach using likely invariants. In *International Symposium on Software Testing and Analysis*. 177–188.

[7] George K. Baah, Andy Podgurski, and Mary Jean Harrold. 2010. Causal Inference for Statistical Fault Localization. In *International Symposium on Software Testing and Analysis*. 12 pages. https://doi.org/10.1145/1831708.1831717

[8] Zhuofu Bai, Gang Shu, and Andy Podgurski. 2015. NUMFL: Localizing Faults in Numerical Software Using a Value-Based Causal Model. In *International Conference on Software Testing, Verification and Validation*. 1–10. https://doi.org/10.1109/ICST.2015.7102597

[9] John P Banning. 1979. An efficient way to find the side effects of procedure calls and the aliases of variables. In *ACM SIGPLAN Symposium on Principles of Programming Languages*. 29–41.

[10] Jacob Benesty, Jingdong Chen, Yiteng Huang, and Israel Cohen. 2009. Pearson correlation coefficient. In *Noise reduction in speech processing*. 1–4.

[11] Leo Breiman, Jerome H. Friedman, Richard A. Olshen, and Charles J. Stone. 1984. *Classification And Regression Trees*.

[12] Lionel C. Briand, Yvan Labiche, and Xuetao Liu. 2007. Using Machine Learning to Support Debugging with Tarantula. In *IEEE International Symposium on Software Reliability Engineering*. 10 pages.

[13] E. C. Campos and M. d. A. Maia. 2017. Common Bug-Fix Patterns: A Large-Scale Observational Study. In *ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. 404–413. https://doi.org/10.1109/ESEM.2017.55

[14] Satish Chandra, Emina Torlak, Shaon Barman, and Rastislav Bodik. 2011. Angelic Debugging. In *International Conference on Software Engineering*. 10 pages. https://doi.org/10.1145/1985793.1985811

[15] Junjie Chen, Yihua Liang, Qingchao Shen, Jiajun Jiang, and Shuochuan Li. 2022. Toward understanding deep learning framework bugs. *ACM Transactions on Software Engineering and Methodology* 32, 6 (2022).
[16] Trishul M. Chilimbi, Ben Liblit, Krishna Mehra, Aditya V. Nori, and Kapil Vaswani. 2009. HOLMES: Effective Statistical Debugging via Efficient Path Profiling *(International Conference on Software Engineering)*. 34–44. https://doi.org/10.1109/ICSE.2009.5070506
[17] K. D. Cooper and K. Kennedy. 1988. Interprocedural Side-Effect Analysis in Linear Time. In *Programming Language Design and Implementation*. 57–66. https://doi.org/10.1145/53990.53996
[18] Christoph Csallner, Nikolai Tillmann, and Yannis Smaragdakis. 2008. DySy: Dynamic Symbolic Execution for Invariant Inference. In *International Conference on Software Engineering*. 281–290. https://doi.org/10.1145/1368088.1368127
[19] Xuelian Deng, Yuqing Li, Jian Weng, and Jilian Zhang. 2019. Feature selection for text classification: A review. *Multimedia Tools and Applications* 78, 3 (2019), 3797–3816.
[20] Michael D Ernst, William G Griswold, Yoshio Kataoka, and David Notkin. 1999. Dynamically discovering pointer-based program invariants. In *International Conference on Software Engineering*, Vol. 373.
[21] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++ : Combining Incremental Steps of Fuzzing Research. In *USENIX Workshop on Offensive Technologies*.
[22] Gordon Fraser and Andrea Arcuri. 2011. Evosuite: automatic test suite generation for object-oriented software. In *European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*. 416–419.
[23] Tarak Goradia. 1993. Dynamic Impact Analysis: A Cost-Effective Technique to Enforce Error-Propagation *(International Symposium on Software Testing and Analysis)*. 171–181. https://doi.org/10.1145/154183.154269
[24] Ross Gore, Paul F. Reynolds, and David Kamensky. 2011. Statistical debugging with elastic predicates. In *International Conference on Automated Software Engineering*. 492–495. https://doi.org/10.1109/ASE.2011.6100107
[25] Aashish Gupta, Shilpa Sharma, Shubham Goyal, and Mamoon Rashid. 2020. Novel xgboost tuned machine learning model for software bug prediction. In *International Conference on Intelligent Engineering and Management*. 376–380.
[26] A. Hajnal and I. Forgacs. 2002. A precise demand-driven definition-use chaining algorithm. In *European Conference on Software Maintenance and Reengineering*. 77–86. https://doi.org/10.1109/CSMR.2002.995792
[27] Thomas Hirsch and Birgit Hofer. 2022. A Systematic Literature Review on Benchmarks for Evaluating Debugging Approaches. *Journal of Systems and Software* 192, C (2022), 17 pages. https://doi.org/10.1016/j.jss.2022.111423
[28] Dennis Jeffrey, Neelam Gupta, and Rajiv Gupta. 2008. Fault Localization Using Value Replacement. In *International Symposium on Software Testing and Analysis*. 12 pages. https://doi.org/10.1145/1390630.1390652
[29] Jiajun Jiang, Ran Wang, Yingfei Xiong, Xiangping Chen, and Lu Zhang. 2019. Combining Spectrum-Based Fault Localization and Statistical Debugging: An Empirical Study. In *International Conference on Automated Software Engineering*. 502–514. https://doi.org/10.1109/ASE.2019.00054
[30] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. 2018. Shaping Program Repair Space with Existing Patches and Similar Code. In *International Symposium on Software Testing and Analysis*.
[31] Lingxiao Jiang and Zhendong Su. 2007. Context-aware Statistical Debugging: From Bug Predictors to Faulty Control Flow Paths *(International Conference on Automated Software Engineering)*. 184–193. https://doi.org/10.1145/1321631.1321660
[32] James A. Jones and Mary Jean Harrold. 2005. Empirical Evaluation of the Tarantula Automatic Fault-localization Technique. In *International Conference on Automated Software Engineering*. 273–282.
[33] James A. Jones, Mary Jean Harrold, and John Stasko. 2002. Visualization of Test Information to Assist Fault Localization. In *International Conference on Software Engineering*.
[34] René Just, Darioush Jalali, and Michael D Ernst. 2014. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *International Symposium on Software Testing and Analysis*. 437–440.
[35] Jeongho Kim, Jindae Kim, and Eunseok Lee. 2019. VFL: Variable-based fault localization. *Information and Software Technology* 107 (2019), 179 – 191. https://doi.org/10.1016/j.infsof.2018.11.009
[36] Pavneet Singh Kochhar, Xin Xia, David Lo, and Shanping Li. 2016. Practitioners' Expectations on Automated Fault Localization *(International Symposium on Software Testing and Analysis)*. 165–176. https://doi.org/10.1145/2931037.2931051
[37] Yiğit Küçük, Tim A. D. Henderson, and Andy Podgurski. 2021. Improving Fault Localization by Integrating Value and Predicate Based Causal Inference Techniques. In *International Conference on Software Engineering*. 12 pages. https://doi.org/10.1109/ICSE43902.2021.00066
[38] Xuan-Bach D Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. 2017. S3: syntax-and semantic-guided repair synthesis via programming by examples. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 593–604.
[39] Claire Le Goues, Neal Holtschulte, Edward K. Smith, Yuriy Brun, Premkumar Devanbu, Stephanie Forrest, and Westley Weimer. 2015. The ManyBugs and IntroClass Benchmarks for Automated Repair of C Programs. *IEEE Transactions on*

*Software Engineering* 41, 12 (2015), 1236–1256. https://doi.org/10.1109/TSE.2015.2454513

[40] Xia Li, Jiajun Jiang, Samuel Benton, Yingfei Xiong, and Lingming Zhang. 2021. A Large-scale Study on API Misuses in the Wild. In *International Conference on Software Testing, Verification and Validation*. 241–252. https://doi.org/10.1109/ICST49551.2021.00034

[41] Xia Li and Lingming Zhang. 2017. Transforming Programs and Tests in Tandem for Fault Localization. Object-Oriented Programming Systems, Language, and Applications (2017), 92:1–92:30. https://doi.org/10.1145/3133916

[42] Yi Li, Shaohua Wang, and Tien N. Nguyen. 2021. Fault Localization with Code Coverage Representation Learning. In *International Conference on Software Engineering*. 661–673. https://doi.org/10.1109/ICSE43902.2021.00067

[43] Ben Liblit, Alex Aiken, Alice X. Zheng, and Michael I. Jordan. 2003. Bug Isolation via Remote Program Sampling. In *Programming Language Design and Implementation*. 141–154. https://doi.org/10.1145/781131.781148

[44] Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken, and Michael I. Jordan. 2005. Scalable Statistical Bug Isolation. In *Programming Language Design and Implementation*. 15–26. https://doi.org/10.1145/1065010.1065014

[45] Chao Liu, Long Fei, Xifeng Yan, Jiawei Han, and S. P. Midkiff. 2006. Statistical Debugging: A Hypothesis Testing-Based Approach. *IEEE Transactions on Software Engineering* 32, 10 (2006), 831–848. https://doi.org/10.1109/TSE.2006.105

[46] Chao Liu, Xifeng Yan, Long Fei, Jiawei Han, and Samuel P. Midkiff. 2005. SOBER: Statistical Model-based Bug Localization *(European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering)*. 286–295. https://doi.org/10.1145/1081706.1081753

[47] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. 2019. TBar: Revisiting Template-Based Automated Program Repair. In *International Symposium on Software Testing and Analysis*. 12 pages.

[48] Yiling Lou, Qihao Zhu, Jinhao Dong, Xia Li, Zeyu Sun, Dan Hao, Lu Zhang, and Lingming Zhang. 2021. Boosting Coverage-Based Fault Localization via Graph-Based Representation Learning *(European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering)*. 13 pages. https://doi.org/10.1145/3468264.3468580

[49] Lucia Lucia, David Lo, Lingxiao Jiang, Ferdian Thung, and Aditya Budi. 2014. Extended comprehensive study of association measures for fault localization. *Journal of Software: Evolution and Process* 26, 2 (2014), 172–219.

[50] Fernanda Madeiral, Simon Urli, Marcelo Maia, and Martin Monperrus. 2019. Bears: An Extensible Java Bug Benchmark for Automatic Program Repair Studies. In *IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER '19)*. https://arxiv.org/abs/1901.06024

[51] S. Moon, Y. Kim, M. Kim, and S. Yoo. 2014. Ask the Mutants: Mutating Faulty Programs for Fault Localization. In *International Conference on Software Testing, Verification and Validation*. 153–162. https://doi.org/10.1109/ICST.2014.28

[52] Mike Papadakis and Yves Le Traon. 2015. Metallaxis-FL: Mutation-based Fault Localization. *Softw. Test. Verif. Reliab.* 25, 5-7 (Aug. 2015), 605–628. https://doi.org/10.1002/stvr.1509

[53] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D. Ernst, Deric Pang, and Benjamin Keller. 2017. Evaluating and Improving Fault Localization *(International Conference on Software Engineering)*. 609–620.

[54] Alexandre Perez and Rui Abreu. 2018. Leveraging Qualitative Reasoning to Improve SFL. In *International Joint Conference on Artificial Intelligence*. 1935–1941.

[55] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. 2015. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *International Symposium on Software Testing and Analysis*. 257–269.

[56] J Ross Quinlan. 2014. *C4.5: programs for machine learning*.

[57] Ripon Saha, Yingjun Lyu, Wing Lam, Hiroaki Yoshida, and Mukul Prasad. 2018. Bugs.jar: A Large-Scale, Diverse Dataset of Real-World Java Bugs. In *International Conference on Mining Software Repositories (MSR)*. 10–13.

[58] Ripon K. Saha, Yingjun Lyu, Hiroaki Yoshida, and Mukul R. Prasad. 2017. ELIXIR: Effective Object Oriented Program Repair. In *International Conference on Automated Software Engineering*. http://dl.acm.org/citation.cfm?id=3155562.3155643

[59] Qingchao Shen, Haoyang Ma, Junjie Chen, Yongqiang Tian, Shing-Chi Cheung, and Xiang Chen. 2021. A comprehensive study of deep learning compiler bugs. In *European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*. 968–980.

[60] Shiqi Shen, Aashish Kolluri, Zhen Dong, Prateek Saxena, and Abhik Roychoudhury. 2021. Localizing Vulnerabilities Statistically From One Exploit. In *ASIA CCS*. 537–549. https://doi.org/10.1145/3433210.3437528

[61] Edward K Smith, Earl T Barr, Claire Le Goues, and Yuriy Brun. 2015. Is the cure worse than the disease? overfitting in automated program repair. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering*. 532–543.

[62] Jeongju Sohn and Shin Yoo. 2017. FLUCCS: Using Code and Change Metrics to Improve Fault Localization *(International Symposium on Software Testing and Analysis)*. 273–283. https://doi.org/10.1145/3092703.3092717

[63] Mauricio Soto, Ferdian Thung, Chu-Pan Wong, Claire Le Goues, and David Lo. 2016. A Deeper Look into Bug Fixes: Patterns, Replacements, Deletions, and Additions. In *Working Conference on Mining Software Repositories*. 512–515. https://doi.org/10.1145/2901739.2903495

[64] David A Spuler and A Sayed Muhammed Sajeev. 1994. Compiler detection of function call side effects. *Informatica* 18, 2 (1994), 219–227.
[65] Shin Hwei Tan, Hiroaki Yoshida, Mukul R Prasad, and Abhik Roychoudhury. 2016. Anti-patterns in Search-Based Program Repair. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering*. https://doi.org/10.1145/2950290.2950295
[66] Haoye Tian, Yinghua Li, Weiguo Pian, Abdoul Kader Kaboré, Kui Liu, Andrew Habib, Jacques Klein, and Tegawendé F. Bissyandé. 2022. Predicting Patch Correctness Based on the Similarity of Failing Test Cases. *ACM Transactions on Software Engineering and Methodology* 31, 4, Article 77 (aug 2022), 30 pages. https://doi.org/10.1145/3511096
[67] Saeid Tizpaz-Niari, Pavol Cerny, Bor-Yuh Evan Chang, and Ashutosh Trivedi. 2018. Differential performance debugging with discriminant regression trees. In *AAAI Conference on Artificial Intelligence*, Vol. 32.
[68] Susana M Vieira, Uzay Kaymak, and João MC Sousa. 2010. Cohen's kappa coefficient as a performance measure for feature selection. In *International Conference on Fuzzy Systems*. 1–8.
[69] Bo Wang, Sirui Lu, Jiajun Jiang, and Yingfei Xiong. 2020. Survey of Dynamic Analysis Based Program Invariant Synthesis Techniques. *Journal of Software* 31, 6 (2020), 1681–1702.
[70] Shangwen Wang, Ming Wen, Bo Lin, Hongjun Wu, Yihao Qin, Deqing Zou, Xiaoguang Mao, and Hai Jin. 2020. Automated Patch Correctness Assessment: How Far Are We?. In *International Conference on Automated Software Engineering*. 13 pages. https://doi.org/10.1145/3324884.3416590
[71] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. 2018. Context-Aware Patch Generation for Better Automated Program Repair. In *International Conference on Software Engineering*.
[72] Ming Wen, Zifan Xie, Kaixuan Luo, Xiao Chen, Yibiao Yang, and Hai Jin. 2022. Effective Isolation of Fault-Correlated Variables via Statistical and Mutation Analysis. *IEEE Transactions on Software Engineering* 49, 4 (2022), 1–16. https://doi.org/10.1109/TSE.2022.3209590
[73] Ratnadira Widyasari, Gede Artha Azriadi Prana, Stefanus A. Haryono, Yuan Tian, Hafil Noer Zachiary, and David Lo. 2022. XAI4FL: Enhancing Spectrum-Based Fault Localization with Explainable Artificial Intelligence. In *International Conference on Program Comprehension*. 499–510. https://doi.org/10.1145/3524610.3527902
[74] Ian H. Witten, Eibe Frank, Mark A. Hall, and Christopher J. Pal. 2016. *Data Mining, Fourth Edition: Practical Machine Learning Tools and Techniques* (4th ed.).
[75] W. E. Wong, V. Debroy, R. Gao, and Y. Li. 2014. The DStar Method for Effective Software Fault Localization. *IEEE Transactions on Reliability* 1 (March 2014), 290–308.
[76] Robert F Woolson. 2007. Wilcoxon signed-rank test. *Wiley encyclopedia of clinical trials* (2007), 1–3.
[77] Qi Xin and Steven P. Reiss. 2017. Identifying Test-Suite-Overfitted Patches through Test Case Generation. In *International Symposium on Software Testing and Analysis*. 11 pages. https://doi.org/10.1145/3092703.3092718
[78] Yingfei Xiong, Xinyuan Liu, Muhan Zeng, Lu Zhang, and Gang Huang. 2018. Identifying Patch Correctness in Test-Based Program Repair. In *International Conference on Software Engineering*.
[79] Yingfei Xiong and Bo Wang. 2022. L2S: A Framework for Synthesizing the Most Probable Program under a Specification. *ACM Transactions on Software Engineering and Methodology* 31, 3, Article 34 (2022), 45 pages. https://doi.org/10.1145/3487570
[80] Jifeng Xuan and Martin Monperrus. 2014. Learning to Combine Multiple Ranking Metrics for Fault Localization. In *IEEE International Conference on Software Maintenance and Evolution*. 191–200.
[81] Jinqiu Yang, Alexey Zhikhartsev, Yuefei Liu, and Lin Tan. 2017. Better Test Cases for Better Automated Program Repair. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering*. 831–841. https://doi.org/10.1145/3106237.3106274
[82] He Ye, Jian Gu, Matias Martinez, Thomas Durieux, and Martin Monperrus. 2021. Automated Classification of Overfitting Patches with Statically Extracted Code Features. *IEEE Transactions on Software Engineering* 48, 8 (2021). https://doi.org/10.1109/tse.2021.3071750
[83] Andreas Zeller. 2002. Isolating Cause-effect Chains from Computer Programs *(ACM SIGSOFT Symposium on the Foundations of Software Engineering)*. 1–10. https://doi.org/10.1145/587051.587053
[84] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and Isolating Failure-Inducing Input. *IEEE Transactions on Software Engineering* 28, 2 (2002), 183–200. https://doi.org/10.1109/32.988498
[85] Muhan Zeng, Yiqian Wu, Zhentao Ye, Yingfei Xiong, Xin Zhang, and Lu Zhang. 2022. Fault Localization via Efficient Probabilistic Modeling of Program Semantics. In *International Conference on Software Engineering*. 12 pages. https://doi.org/10.1145/3510003.3510073
[86] Haotian Zhang, Weiyu Dong, and Jian Lin. 2021. A Partial-Lifting-Based Compiling Concolic Execution Approach. In *International Conference on Cryptography, Security and Privacy*. 123–128. https://doi.org/10.1109/CSP51677.2021.9357495
[87] Xiangyu Zhang, Neelam Gupta, and Rajiv Gupta. 2006. Locating faults through automated predicate switching. In *International Conference on Software Engineering*. 272–281.

[88] Xiangyu Zhang, Neelam Gupta, and Rajiv Gupta. 2006. Pruning Dynamic Slices with Confidence *(Programming Language Design and Implementation)*. 169–180. https://doi.org/10.1145/1133981.1134002

[89] Yuntong Zhang, Xiang Gao, Gregory J. Duck, and Abhik Roychoudhury. 2022. Program Vulnerability Repair via Inductive Inference *(International Symposium on Software Testing and Analysis)*. 691–702. https://doi.org/10.1145/3533767.3534387

[90] Alice X. Zheng, Michael I. Jordan, Ben Liblit, Mayur Naik, and Alex Aiken. 2006. Statistical Debugging: Simultaneous Identification of Multiple Bugs *(International Conference on Machine Learning)*. 1105–1112. https://doi.org/10.1145/1143844.1143983

[91] Qihao Zhu, Zeyu Sun, Yuan-an Xiao, Wenjie Zhang, Kang Yuan, Yingfei Xiong, and Lu Zhang. 2021. A Syntax-Guided Edit Decoder for Neural Program Repair. In *European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*. 13 pages.

[92] Daming Zou, Jingjing Liang, Yingfei Xiong, Michael D. Ernst, and Lu Zhang. 2019. An Empirical Study of Fault Localization Families and Their Combinations. *IEEE Transactions on Software Engineering* 47, 2 (2019).

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009