Variable-Based Fault Localization via Enhanced Decision Tree

1st Jiajun Jiang
College of Intelligence and Computing
Tianjin University
Tianjin, China
jiangjiajun@tju.edu.cn

2nd Yumeng Wang
College of Intelligence and Computing
Tianjin University
Tianjin, China
jazz244008@tju.edu.cn

3rd Junjie Chen*
College of Intelligence and Computing
Tianjin University
Tianjin, China
junjiechen@tju.edu.cn

4th Delin Lv
College of Intelligence and Computing
Tianjin University
Tianjin, China
ldlmntq@tju.edu.cn

5th Mengjiao Liu
College of Intelligence and Computing
Tianjin University
Tianjin, China
mengjiaoliu@tju.edu.cn

Abstract—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, we design 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 247.8% improvements in terms of bugs located at Top-1, and the average improvements are 330.5%.

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 VARDT outperforms the state-of-the-art PATCH-SIM by filtering 26.0% more incorrect patches. The results demonstrate the effectiveness of our approach and it also provides a new way of thinking for improving automatic program repair techniques.

Index Terms—fault localization, decision tree, debugging

I. 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 [1]–[12] 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 buggy statements as the top-1 returned results [13]. Inaccurate fault localization results can be misleading and increase the risk of generating incorrect patches due to the incomplete specification [14]–[16]. Second, the granularity of existing fault localization results is still coarse-grained at statement or method levels, which provides 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 [17], [18], when providing genuine buggy 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 [19], machine learning [20], deep learning [21]–[23], etc., and incorporated diverse information like test coverage [24], program dependency [25], [26], code changes [27] and program invariants [28], 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 [29], [30] or targeting a particular type of variables [5], [31], [32], making them infeasible to further the effectiveness of downstream APR techniques.

Aiming at significantly improving the effectiveness of fault localization and thus boosting the subsequent program repair

---

*Corresponding author.

process, in this paper we propose a novel and general fault localization technique, named VARDT, targeting 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 III.

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 [33] benchmark. The results show that VARDT successfully located the fault-relevant variables at Top-1 position for 24.0% of bugs, which significantly outperformed seven state-of-the-art baseline approaches. Particularly, the improvements are at least 247.8%, and in average 330.5% regarding the bugs located at Top-1. Moreover, to investigate whether our approach can further the effectiveness of downstream APR techniques, we also adapted VARDT to the application of patch filtering, where it correctly filtered out 69.4% incorrect patches. Although not designed as a comprehensive and standalone patch filtering technique, it improves the state-of-the-art PATCH-SIM by 26.0%. 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 via an enhanced decision tree model.
- We conduct an extensive study on the widely-used Defects4J benchmark in two distinct application scenarios. 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.

II. 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 [33], where the lines leading by "+" denote code to be added while "-" to be deleted.

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.
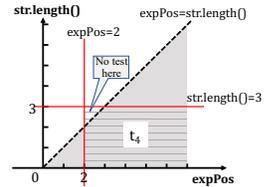
Fig. 1: Visualization of variable constraint estimation.

```
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 of Lang-27.

In this example, when providing an input string `str`, the method `createNumber(*)` 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 [3], [34]. However, existing approaches can hardly locate the accurate faulty code in this example. 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()`. However, 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 `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 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 (defined in Section IV-C). However, since there are usually many available variables that may produce a large number of primitive constraints, how to combine them and correctly 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 III by taking this bug as the running example.

III. Framework

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. Particularly, we adopt the implementation published by Jiang et al. [35]. 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 [22], [23], which can produce much better results than SBFL and potentially can further improve the performance of VaRDT.

A. 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 [25] 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 be not affected by the scale of programs under debugging but only affected by the size of a single method. 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). 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.
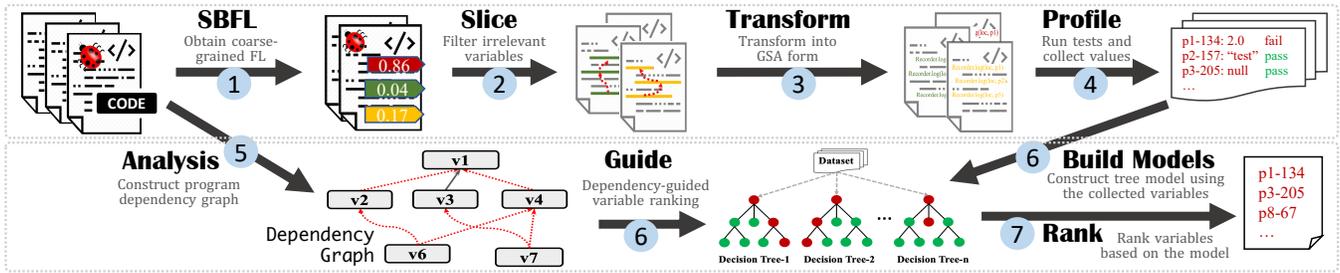
Fig. 2: Overview of our approach VARDT

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 V.

B. 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 [31], 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 error-prone in practice and many bugs are caused by incorrect sub-conditions [17], [36], [37], 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. 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. Specifically, 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, such as checking whether an object is `null`. We have summarized the details of values collected for different types of variables in Table I. Based on this definition, 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()`.

To collect the above variable values, we have implemented a simple value profiling process in VARDT, which can au-

TABLE I: A description for variables considered by VARDT

Type	Target Value	Description
Primitive	Actual Value	Primitive value or ASCII code for `char`
Object	Null Check	True if the variable is `null`, false otherwise
	Type Check	The value type of the variable. e.g., `String`
	Fields	Unfold the variable and output field values
	Size/length	Access `size()`/`length()`/`length` (if has)
	Elements	Primitive element values in collections

tomatically parse the type of fed variables and insert output statements by using the Eclipse Java Development Toolkit (https://www.eclipse.org/jdt/) for recording the corresponding values defined in Table I during the running of test cases.

C. 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 [38]–[41]. Particularly, this model is suitable for our application in two aspects. (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 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 will introduce the details of our tree model construction process in VARDT. In general, it includes two sub-processes, named *Enhanced Variable Selection* and *Tree Model Building*. 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 is built.

Algorithm 1: Variable Prioritization for Selection

> **Input:** *varList:* a list of variables to be ranked. *data:* a set of program states for each test case. *graph:* program dependency graph.
> **Output:** *varList:* an ordered list of variables

```
1  Function prioritizeVars(varList, data, graph):
2      foreach var in varList do
3          var.score ← gainRatio(var) + correlation(var, data.labels)
4          dependency ← depScore(var, graph, varList)
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)
10             if v.score > var.score then
11                 var.score ← v.score
12             end
13         end
14     end
15     return varList.sort()          // descending order by score
```

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 multiple trees with all variables   */
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
                /* prioritize different attributes            */
15         varList ← prioritizeVars(varList, data, graph)
16         var ← varList.first            // higher priority first
17         cond ← calculateCondition(data, var)
                /* divide data into groups based on cond      */
18         groups ← divide(data, var, cond)
19         tree ← internalNode(data)      // root node of subtree
20         foreach g in groups do
21             │   tree.children.add(buildTree(g, varList))
22         end
23     end
24     return tree
```

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 impacts 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 do 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 (refer to Section IV). To overcome this limitation, we propose an *enhanced variable selection strategy* depending on a novel variable prioritization algorithm 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. However, 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. In other words, the faulty variables tend to affect test cases in a small scale. Therefore, we introduce a *dependency penalty* to incorporate such an observation through static analysis. That is, a variable depended on by more other variables will rank lower, i.e., less likely to be faulty. 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.

$$depScore(v, g, l) = DEP\_FACTOR^{|S|}$$
$$s.t.\ S = \{x | x \in l \land g \vdash x \hookrightarrow v\} \quad (1)$$

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 prioritization algorithm in Algorithm 1. Specifically, for each variable var, its priority is determined by three parts (lines 3-5). The *dependency penalty* has been defined in Formula 1, while the function of *correlation(*)* returns the general *Pearson correlation coefficient* [42] between variables and the testing results. Finally, the *gainRatio(var)* is a builtin function in the C4.5 decision tree model [43] for computing how much confidence can be gained by choosing the variable *var* to distinguish the given data. In summary, a variable that has a smaller impact 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 lines 2-6 in Algorithm 1). Next, we aggregate the equivalent variables appearing at different locations (i.e., no reassignment between them) into one as the representative by removing the others according to the dependency relation (lines 8-9), and the score of the representative variable will be the maximal one of all its equivalent variables (lines 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, then two of them will be removed in the results returned by Algorithm 1. Finally, variable with larger score will have higher priority to be selected during the tree model building process.

2) Tree Model Building: According to the above variable selection strategy, we present the details of our model building process, which is shown in Algorithm 2. When providing the values of a set of variables per 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 variables (line 4). As shown in line 9, each variable can be used in no more than one decision tree to avoid duplication and guarantee this process always terminates. In other words, 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. In this way, all variables will have the possibility to be located since the fault-relevant variables can be multiple. 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., *varList.first*) will be selected to construct a predicate for dividing the given *data* into different groups (lines 15-18). 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. According to the predicate, *data* will be divided into different groups. VARDT recursively performs the above construction process (lines 20-22) for each group until the input data do not require further discrimination (line 14).

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[1], 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 III-D).

D. 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

---

[1]Please note that the constant value "4" is automatically computed by the default builtin function of decision tree model in Weka (https://www.weka.io).
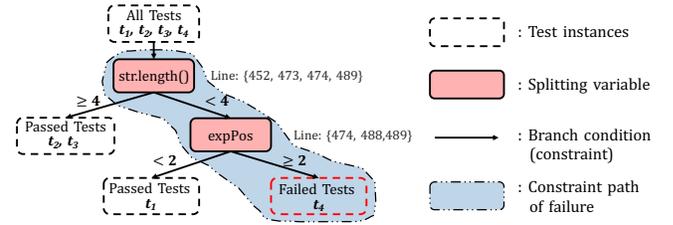

Fig. 3: A sketch of tree construction for Listing 1

variables that are fault-relevant. Please note that this ranking strategy is different from the variable prioritization 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 between 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.

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 2.

$$DS(v) = \frac{(1 - Gini(p)) \times \sqrt{|D|}}{failNodeDist} + depScore(v, g, l) \quad (2)$$

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. $Gini(p)$ represents the general *Gini index* [44] of predicate p, denoting the *impurity* of the tree rooted t. *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. 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 3, 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.

$$FS(v) = DS(v) \times methodScore(m_v)^2 \quad (3)$$

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.

TABLE II: Subjects for fault localization

Project	#Bugs	Project	#Bugs	Project	#Bugs
Math	23	JxPath	8	Compress	9
Chart	12	Cli	8	JacksonXml (JXml)	4
Lang	23	Gson	7	JacksonCore (JCore)	16
Time	12	Csv	5	Mockito	21
Codec	10	Jsop	18	JacksonDatabind (JDatabind)	28
TOTAL				204	

IV. 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?

A. Subjects

In the evaluation of fault localization (RQ1&RQ2), we adopt the Defects4J (version 2.0) benchmark [33], which is widely-used in previous studies [17], [24], [34]–[37]. 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 [23]. Therefore, targeting this portion of bugs can be significant for practical use and also reduce the overhead of variable profiling. Second, the faulty method has to be covered by at least three (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 II. Regarding the patch filtering application (RQ3), we adopt the dataset constructed by Xiong et al. [15] and use all the patches for bugs included in Table II.

B. Baseline and Configuration

In the experiment of fault localization, following the latest research [31], we compare the effectiveness of our approach with five state-of-the-art variable-level fault localization techniques: **UniVal** [31], 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 [45] (specifically the two variants **NUMFL-QRM** and **NUMFL-DLRM**), locating variables by combining causal and statistical analyses to characterize the causal effects of individual numerical expressions on output errors; **ESP** [46], locating variables via measuring the difference between an assigned variable in the failed run and its average value in all test runs; and **Baah2010** [47], locating variables by using a linear regression model to measure the *confounding bia* among variables. Specifically, we adopt their open-source implementation published by Küçük et al. [31] to perform the experiment. Besides, we also adapt two representative and most widely-used spectrum-based fault localization techniques to work at variable level, i.e., **Ochiai** [48] and **DStar** [49], which were proved to perform well [24], [50].

Regarding the configurations of VARDT, we set the Top-10 most suspicious methods as the interested ones as explained in Section IV-A, and 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.

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 in the final variable ranking process of VARDT, i.e., $FS(v) = DS(v)$.

Particularly, since our approach is not designed as a standalone patch filtering tool, we further adapt it to this scenario. Specifically, we perform this process by simply using the located fault-relevant variables to filter patches directly. If no fault-relevant variable is involved in the patch, i.e., not modified or inserted, the patch will be filtered, otherwise regarded as correct. In this study, we compare the results of our approach with PATCH-SIM [15], the state-of-the-art patch filtering technique based on generating new test cases.

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**.

C. 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. [31] only focus on numerical assignments and predicates, while Liblit et al. [5] 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. 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, there are in average 4 fault-relevant variables per each bug in our studied dataset. As it will be presented in Section V-C that correctly locating these fault-relevant variables indeed can boost existing automatic program repair techniques, further demonstrating the reliability of the ground truth.

1) Metrics: Following previous studies [4], [22]–[24], [34], [48], 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 [34], [35]. **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 [34], [35], 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 of different predicates. Besides, we do not use the metric of *Exam Score* used in statement-level fault localization [24], [34], [35]. 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.

In the application of patch filtering, we adopt two metrics following previous studies [15], [51], 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.

V. RESULT ANALYSIS

A. RQ1: Overall Effectiveness of VARDT

As introduced, we conducted our experiment over 204 real-world bugs from Defects4J benchmark and compared the results with seven baseline approaches. Table III presents the experimental results of different approaches. From the table, we can see that our approach significantly outperforms the baselines. Specifically, VARDT successfully located the desired fault-relevant variables for 24.0%, 39.7%, 49.0% and 65.7% of bugs within the Top-1/3/5/10 positions, respectively. The improvements over the baseline approaches range from 247.8% to 515.4% regarding the Top-1 recall, and the average improvements are 330.5%. Particularly, VARDT significantly outperforms the latest state-of-the-art UniVal by 247.8% with respect to the Top-1 recall. The results demonstrate that our approach is much more effective. Though effective, the absolute number of bugs located at Top-1 is still small (i.e., 24.0%) for VARDT. A major reason is the inaccuracy of the coarse-grained fault localization results used by VARDT. As will be presented later (see Figure 4), when providing 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 [31]. 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. We guess the decline may be caused by the different definitions of ground-truth variables, but they were not published by the authors, and thus we cannot reproduce their results.

Furthermore, we also reported the detailed Top-1 results of different approaches on each project in Table IV. According to the results, VARDT performs consistently well over different projects, and always outperforms the baselines, indicating the generalizability of our approach. Regarding the metrics of MAR and MFR shown in Table III, our approach also outperforms the competitors with at least 77.3% and 71.5% improvements, and the average improvements are 78.8% and 73.6%, respectively. The results further demonstrate the superiority of our approach. Please note that though VARDT depends on the decision tree building process compared with baselines, **the cost of it is relatively small, i.e., 1.9s in average**.

TABLE III: Experimental result summary of all approaches

Metric	VARDT	UniVal	Baah-2010	ESP	NUMFL-DLRM	NUMFL-QRM	Ochiai	Dstar (Star=2)
Top-1	**24.0%**	6.9%	5.9%	6.4%	4.4%	3.9%	6.4%	6.9%
Top-3	**39.7%**	12.3%	10.3%	9.8%	8.8%	11.8%	14.2%	14.2%
Top-5	**49.0%**	15.2%	15.2%	12.3%	12.8%	16.2%	18.6%	18.1%
Top-10	**65.7%**	19.6%	18.1%	15.7%	17.2%	20.6%	21.1%	20.6%
MFR	**8.0**	28.2	29.9	44.0	29.5	28.6	28.5	28.1
MAR	**11.2**	49.3	53.2	65.6	52.0	49.5	52.5	51.4

TABLE IV: Top-1 results of all approaches

Project	VARDT	UniVal	Baah-2010	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%
TOTAL	**24.0%**	6.9%	5.9%	6.4%	4.4%	3.9%	6.4%	6.9%

Fig. 4: Results of VARDT and its variants

Fig. 5: Effects of different values of dependency factor

B. 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 IV-B. Figure 4 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 25.7% higher effectiveness (vs $VARDT_{slice}$), the tree model contributed 144.9% (vs $VARDT_{tree}$), the *dependency penalty* contributed 32.6% (vs $VARDT_{dep}$), and the use of method score for variable ranking contributed 103.4% (vs $VARDT_{ms}$), respectively. However, all of them always 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 4. $VARDT_{mk}$ successfully located the desired variables for 37.8% of bugs at Top-1, the improvements over VARDT are about 57.5%. Moreover, the Top-10 recall is as high as 80.4%, indicating the promise of incorporating a more effective method-level fault localization technique into VARDT.
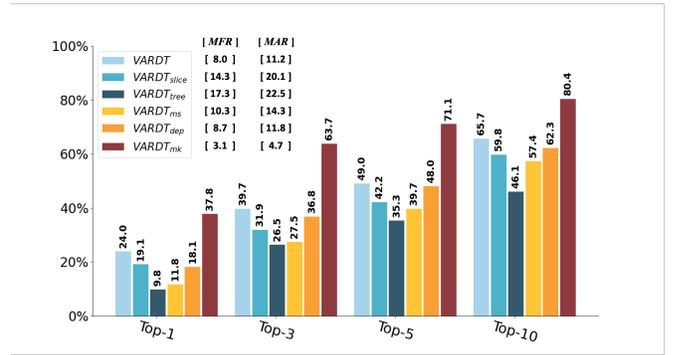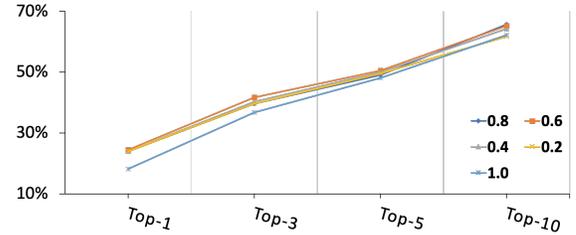
Then, we further investigated the impact of the configuration for *DEP_FACTOR*, which represents the strength of *dependency penalty*. According to Formula 1, the smaller the value is, the larger the penalty will be (i.e., the variable will be less likely to be selected). Figure 5 presents the results 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 0.6, VARDT achieved the highest Top-1 recall as 24.5%, whereas it achieved the lowest Top-1 recall as 18.1% 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 in average is 31.9%, denoting the necessity of it for improving efficiency.

C. RQ3: Performance in Patch Filtering

To evaluate whether our finer-grained variable-level fault localization results can further 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. The details have been introduced in Section IV-B.

Table V presents the experimental results. 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, DT_{Top-N} represents that we use the Top-N variables located by VARDT to filter patches.

TABLE V: Experimental results in patch filtering

Project	#All(Correct)	DT_{Top-1}	DT_{Top-3}	DT_{Top-5}	DT_{Top-10}	PATCH-SIM
Math	24(3)	22(2)	22(2)	20(1)	14(0)	15(0)
Lang	10(2)	10(2)	10(2)	10(2)	9(1)	2(0)
Chart	14(2)	14(2)	8(1)	7(1)	6(1)	4(0)
Time	8(1)	7(1)	6(0)	6(0)	6(0)	6(0)
Mockito	2(1)	1(0)	1(0)	1(0)	1(0)	-
TOTAL	58(9)	**54(7)**	**47(5)**	**44(4)**	**36(2)**	**27(0)**
Precision		87.0%	89.4%	90.9%	94.4%	100.0%
Recall		95.9%	85.7%	81.6%	69.4%	55.1%

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 still could filter out about 69.4% incorrect patches using the Top-10 results of VARDT, while PATCH-SIM only filtered 55.1%. That is, VARDT outperforms PATCH-SIM by 26.0% in terms of incorrect patches filtered. Particularly, the patch precision (the percentage of correct patches over all patches) increased to 31.8% and 29.0% from 15.5% by DT_{Top-10} and PATCH-SIM respectively 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, two correct patches were filtered out by DT_{Top-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), 7 out of 9 correct patches will be filtered due to the inaccuracy of VARDT. Particularly, after further analyzing the results of the two approaches, we found that there were only 19 incorrect patches that were commonly filtered out by both DT_{Top-10} and PATCH-SIM. In other words, (34+27)-19=42 incorrect patches and 2 correct patches could be filtered by combining these two, leaving the *precision* and *recall* of filtering as 95.5% and 85.7%, respectively, and the patch precision will also increase to 50%. The result reflects that they complement each other.

VI. DISCUSSION

Limitation: As explained in Section IV-A, 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 [14], [15]. In such cases, the state-of-

the-art test generation approaches [52]–[54] may be potentially combined to overcome this limitation.

Internal threats: 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. Regarding the ground truth, we have provided a clear definition of fault-relevant variables, based on which we manually analyzed the source code. Therefore, we believe it is reliable. Additionally, the evaluation result of VARDT in the patch filtering application also improves our confidence. Finally, we have published all our data and implementation to ease the replication.

External threats: 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 IV-A. 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.

Future Work: As can be observed, the fault-relevant variables and their constraints indeed can provide possible hints for program repair according to the example shown in Figure 3 and the results in patch filtering. In the future, we plan to further investigate the performance of VARDT for assisting human developers in the manual repair process.

VII. RELATED WORK

A. Variable-based Fault Localization

Our approach targets the variable-based fault localization, the most related techniques are UniVal [31], NUMFL [45], ESP [46], and Baah2010 [47], which have been introduced in Section IV-B. 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. Besides, the statistical debugging [32] and its following work [7]–[9], [11], [12], [46] 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 [55] or random forest models [56] in fault localization. However, they were designed for improving the statement-level fault localization, whereas our approach targets the variable level and additionally incorporates the dependency factor for model building.

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 [57]–[59]. Recent studies [4], [19] incorporated mutation analysis to improve fault localization. Shen et al. [60]

combined statistical localization with directed fuzzing to over-come 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 [29], [30], Com-pared with them, our approach is fully automatic. The latest studies also employed different models to combine the strength of multiple techniques [20], [22], [34], [35]. 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.

B. Automatic Patch Filtering

In order to improve the patch quality (i.e., precision) in au-tomatic 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. [61] proposed Opad, which filters patches that cause program crashes or pro-duce memory errors. Xin and Reiss [51] proposed DiffTGen which depends on human experts to provide the test oracle. While Xiong et al. [15] 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. [62] pre-defined a set of anti-patterns that easily produce incorrect patches for patch filtering. Recently, Ye et al. [63] proposed to employ a machine learning method to classify the correctness of patches, while Wang et al. [64] proposed a deep-learning-based approach. 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.

VIII. Conclusion

In this paper, we have 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 approaches, where the improvements are at least 247.8% and 26.0% regarding bugs located within Top-1 and the number of incorrect patches fil-tered, respectively in the aforementioned applications, demon-strating the effectiveness of our approach.

References

[1] J. A. Jones, M. J. Harrold, and J. Stasko, "Visualization of test infor-mation to assist fault localization," in *ICSE*, 2002.

[2] J. A. Jones and M. J. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in *ASE*, 2005, pp. 273–282.

[3] R. Abreu, P. Zoeteweij, and A. J. Van Gemund, "On the accuracy of spectrum-based fault localization," in *TAICPART-MUTATION*, 2007, pp. 89–98.

[4] S. Moon, Y. Kim, M. Kim, and S. Yoo, "Ask the mutants: Mutating faulty programs for fault localization," in *ICST*, 2014, pp. 153–162.

[5] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, "Scalable statistical bug isolation," in *PLDI*, 2005, pp. 15–26.

[6] P. Arumuga Nainar and B. Liblit, "Adaptive bug isolation," ser. ICSE, 2010, pp. 255–264.

[7] P. Arumuga Nainar, T. Chen, J. Rosin, and B. Liblit, "Statistical debugging using compound boolean predicates," ser. ISSTA, 2007, pp. 5–15.

[8] T. M. Chilimbi, B. Liblit, K. Mehra, A. V. Nori, and K. Vaswani, "Holmes: Effective statistical debugging via efficient path profiling," ser. ICSE, 2009, pp. 34–44.

[9] A. X. Zheng, M. I. Jordan, B. Liblit, M. Naik, and A. Aiken, "Statistical debugging: Simultaneous identification of multiple bugs," ser. ICML, 2006, pp. 1105–1112.

[10] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff, "Sober: Statistical model-based bug localization," ser. ESEC/FSE-13, 2005, pp. 286–295.

[11] C. Liu, L. Fei, X. Yan, J. Han, and S. P. Midkiff, "Statistical debugging: A hypothesis testing-based approach," *TSE*, vol. 32, no. 10, pp. 831–848, 2006.

[12] L. Jiang and Z. Su, "Context-aware statistical debugging: From bug predictors to faulty control flow paths," ser. ASE, 2007, pp. 184–193.

[13] M. Zeng, Y. Wu, Y. Ye, Y. Xiong, X. Zhang, and L. Zhang, "Fault localization via efficient probabilistic modeling of program semantics," in *ICSE*, 2022.

[14] Z. Qi, F. Long, S. Achour, and M. Rinard, "An analysis of patch plausibility and correctness for generate-and-validate patch generation systems," in *ISSTA*, 2015, pp. 257–269.

[15] Y. Xiong, X. Liu, M. Zeng, L. Zhang, and G. Huang, "Identifying patch correctness in test-based program repair," in *ICSE*, 2018.

[16] E. K. Smith, E. T. Barr, C. Le Goues, and Y. Brun, "Is the cure worse than the disease? overfitting in automated program repair," in *FSE*, 2015, pp. 532–543.

[17] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé, "Tbar: Revisiting template-based automated program repair," in *ISSTA*, 2019.

[18] Q. Zhu, Z. Sun, Y.-a. Xiao, W. Zhang, K. Yuan, Y. Xiong, and L. Zhang, "A syntax-guided edit decoder for neural program repair," in *ESEC/FSE*, 2021.

[19] M. Papadakis and Y. Le Traon, "Metallaxis-fl: Mutation-based fault localization," *Softw. Test. Verif. Reliab.*, pp. 605–628, Aug. 2015.

[20] J. Xuan and M. Monperrus, "Learning to combine multiple ranking metrics for fault localization," in *ICSME*, 2014, pp. 191–200.

[21] X. Li and L. Zhang, "Transforming programs and tests in tandem for fault localization," no. OOPSLA, pp. 92:1–92:30, 2017.

[22] Y. Li, S. Wang, and T. N. Nguyen, "Fault localization with code coverage representation learning," in *ICSE*, 2021, pp. 661–673.

[23] Y. Lou, Q. Zhu, J. Dong, X. Li, Z. Sun, D. Hao, L. Zhang, and L. Zhang, "Boosting coverage-based fault localization via graph-based representation learning," ser. ESEC/FSE, 2021.

[24] S. Pearson, J. Campos, R. Just, G. Fraser, R. Abreu, M. D. Ernst, D. Pang, and B. Keller, "Evaluating and improving fault localization," ser. ICSE, 2017, pp. 609–620.

[25] H. Agrawal and J. R. Horgan, "Dynamic program slicing," ser. PLDI, 1990, pp. 246–256.

[26] X. Zhang, N. Gupta, and R. Gupta, "Pruning dynamic slices with confidence," ser. PLDI, 2006, pp. 169–180.

[27] J. Sohn and S. Yoo, "Fluccs: Using code and change metrics to improve fault localization," ser. ISSTA, 2017, pp. 273–283.

[28] T.-D. B. Le, D. Lo, C. Le Goues, and L. Grunske, "A learning-to-rank based fault localization approach using likely invariants," in *ISSTA*, 2016, pp. 177–188.

[29] A. Zeller, "Isolating cause-effect chains from computer programs," ser. SIGSOFT '02/FSE-10, 2002, pp. 1–10.

[30] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *TSE*, pp. 183–200, 2002.

[31] Y. Küçük, T. A. D. Henderson, and A. Podgurski, "Improving fault localization by integrating value and predicate based causal inference techniques," in *ICSE*, 2021.

[32] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan, "Bug isolation via remote program sampling," in *PLDI*, 2003, pp. 141–154.

[33] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for java programs," in *ISSTA*, 2014, pp. 437–440.

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

[35] J. Jiang, R. Wang, Y. Xiong, X. Chen, and L. Zhang, "Combining spectrum-based fault localization and statistical debugging: An empirical study," in *ASE*, 2019, pp. 502–514.

[36] J. Jiang, Y. Xiong, H. Zhang, Q. Gao, and X. Chen, "Shaping program repair space with existing patches and similar code," in *ISSTA*, 2018.

[37] M. Wen, J. Chen, R. Wu, D. Hao, and S.-C. Cheung, "Context-aware patch generation for better automated program repair," in *ICSE*, 2018.

[38] X. Deng, Y. Li, J. Weng, and J. Zhang, "Feature selection for text classification: A review," *Multimedia Tools and Applications*, vol. 78, no. 3, pp. 3797–3816, 2019.

[39] A. Gupta, S. Sharma, S. Goyal, and M. Rashid, "Novel xgboost tuned machine learning model for software bug prediction," in *ICIEM*, 2020, pp. 376–380.

[40] Y. Xiong and B. Wang, "L2s: A framework for synthesizing the most probable program under a specification," *ACM Trans. Softw. Eng. Methodol.*, vol. 31, no. 3, 2022.

[41] S. Tizpaz-Niari, P. Cerny, B.-Y. E. Chang, and A. Trivedi, "Differential performance debugging with discriminant regression trees," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 32, no. 1, 2018.

[42] J. Benesty, J. Chen, Y. Huang, and I. Cohen, "Pearson correlation coefficient," in *Noise reduction in speech processing*. Springer, 2009, pp. 1–4.

[43] J. R. Quinlan, *C4. 5: programs for machine learning*. Elsevier, 2014.

[44] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone, *Classification And Regression Trees*, 1984.

[45] Z. Bai, G. Shu, and A. Podgurski, "Numfl: Localizing faults in numerical software using a value-based causal model," in *ICST*, 2015, pp. 1–10.

[46] R. Gore, P. F. Reynolds, and D. Kamensky, "Statistical debugging with elastic predicates," in *ASE*, 2011, pp. 492–495.

[47] G. K. Baah, A. Podgurski, and M. J. Harrold, "Causal inference for statistical fault localization," in *ISSTA*, 2010.

[48] R. Abreu, P. Zoeteweij, and A. J. C. v. Gemund, "An evaluation of similarity coefficients for software fault localization," ser. PRDC, 2006, pp. 39–46.

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

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

[51] Q. Xin and S. P. Reiss, "Identifying test-suite-overfitted patches through test case generation," in *ISSTA*, 2017.

[52] G. Fraser and A. Arcuri, "Evosuite: automatic test suite generation for object-oriented software," in *SEC/FSE*, 2011, pp. 416–419.

[53] H. Zhang, W. Dong, and J. Lin, "A partial-lifting-based compiling concolic execution approach," in *CSP*, 2021, pp. 123–128.

[54] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++ : Combining incremental steps of fuzzing research," in *14th USENIX Workshop on Offensive Technologies*, 2020.

[55] L. C. Briand, Y. Labiche, and X. Liu, "Using machine learning to support debugging with tarantula," in *ISSRE*, 2007.

[56] R. Widyasari, G. A. A. Prana, S. A. Haryono, Y. Tian, H. N. Zachiary, and D. Lo, "XAI4FL: Enhancing spectrum-based fault localization with explainable artificial intelligence," in *ICPC*, 2022, pp. 499–510.

[57] X. Zhang, N. Gupta, and R. Gupta, "Locating faults through automated predicate switching," in *ICSE*, 2006, pp. 272–281.

[58] D. Jeffrey, N. Gupta, and R. Gupta, "Fault localization using value replacement," in *ISSTA*, 2008.

[59] S. Chandra, E. Torlak, S. Barman, and R. Bodik, "Angelic debugging," in *ICSE*, 2011.

[60] S. Shen, A. Kolluri, Z. Dong, P. Saxena, and A. Roychoudhury, "Localizing vulnerabilities statistically from one exploit," in *ASIA CCS*, 2021, p. 537–549.

[61] J. Yang, A. Zhikhartsev, Y. Liu, and L. Tan, "Better test cases for better automated program repair," in *FSE*, 2017, pp. 831–841.

[62] S. H. Tan, H. Yoshida, M. R. Prasad, and A. Roychoudhury, "Anti-patterns in search-based program repair," in *FSE*, 2016.

[63] H. Ye, J. Gu, M. Martinez, T. Durieux, and M. Monperrus, "Automated classification of overfitting patches with statically extracted code features," *IEEE Transactions on Software Engineering*, 2021.

[64] S. Wang, M. Wen, B. Lin, H. Wu, Y. Qin, D. Zou, X. Mao, and H. Jin, "Automated patch correctness assessment: How far are we?" in *ASE*, 2020.