9-2018

Break the dead end of dynamic slicing: localizing data and control omission bug

Yun LIN

Jun SUN
Singapore Management University, junsun@smu.edu.sg

Lyly TRAN

Guangdong BAI

Haijun WANG

See next page for additional authors

Break the Dead End of Dynamic Slicing: Localizing Data and Control Omission Bug

Yun Lin
National University of Singapore, Singapore

Jun Sun
Singapore University of Technology and Design, Singapore

Lyly Tran
Singapore University of Technology and Design, Singapore

Guangdong Bai
Singapore Institute of Technology, Singapore

Haijun Wang
Nanyang Technological University, Singapore

Jinsong Dong
National University of Singapore, Singapore

ABSTRACT

Dynamic slicing is a common way of identifying the root cause when a program fault is revealed. With the dynamic slicing technique, the programmers can follow data and control flow along the program execution trace to the root cause. However, the technique usually fails to work on omission bugs, i.e., the faults which are caused by missing executing some code. In many cases, dynamic slicing over-skips the root cause when an omission bug happens, leading the debugging process to a dead end. In this work, we conduct an empirical study on the omission bugs in the Defects4J bug repository. Our study shows that (1) omission bugs are prevalent (46.4%) among all the studied bugs; (2) there are repeating patterns on causes and fixes of the omission bugs; (3) the patterns of fixing omission bugs serve as a strong hint to break the slicing dead end. Based on our findings, we train a neural network model on the omission bugs in Defects4J repository to recommend where to approach when slicing can no long work. We conduct an experiment by applying our approach on 3193 mutated omission bugs which slicing fails to locate. The results show that our approach outperforms random benchmark on breaking the dead end and localizing the mutated omission bugs (63.8% over 2.8%).

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**; *Maintaining software*;

KEYWORDS

debugging, program slice, omission error, empirical study

ACM Reference Format:
Yun Lin, Jun Sun, Lyly Tran, Guangdong Bai, Haijun Wang, and Jinsong Dong. 2018. Break the Dead End of Dynamic Slicing: Localizing Data and Control Omission Bug. In *Proceedings of the 2018 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18), September 3–7, 2018, Montpellier, France.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3238147.3238163

ASE '18, September 3–7, 2018, Montpellier, France

ACM ISBN 978-1-4503-5937-5/18/09…$15.00
https://doi.org/10.1145/3238147.3238163

1 INTRODUCTION

A software fault is observed when the defect is propagated from the root cause through control and data flow. In order to track the fault back to its root cause, programmers usually need to trace through incorrect data and control flow to locate the bug. From this perspective, dynamic slicing is a useful and efficient approach for debugging. Each time a programmer finds an unexpected step or a step with variable of incorrect value, he or she can apply dynamic slicing to locate the step responsible for it.

However, dynamic slicing can only track through the executed code. It means that it cannot locate bugs caused by missing executing some code, i.e., the omission bug [35]. When an omission bug happens, dynamic slicing starts with a step with incorrect data or control flow and stops at a step where everything is correct, causing a *dead end* where slicing can no longer work.

Many research work [12, 20, 25, 28, 35] have been proposed to enhance dynamic slicing for locating omission bugs. Zhang et al. [35] first proposed a force-execution technique to include more results than reported by dynamic slicing. In their approach, they force the program to exercise the branches of unexecuted code to expose implicit dependencies among executed and unexecuted code. Similar to their work, Wang et al. [28] proposed their relevant slicing algorithm on Java byte code trace, aiming to include the relevant unexecuted code into the slicing result. In recent years, Sakuai et al. [25] enhanced Zhang and Wang's work [28, 35] with point-to-analysis technique.

However, all the above approaches have an underlying assumption that the unexecuted code exist in the project (i.e., execution omission) so that they can analyze the source code and alter the control flow towards them. Such an assumption may not always hold in practice. Our empirical study (see Section 3) shows that the miss-executed code in most of omission bugs does not exist in the project (i.e., code omission).

In this work, we aim to understand omission bugs in a more fundamental way. We would like to answer the question like how prevalent the omission bugs are, how they are caused, and whether they share some patterns guiding us to automate their localization. To this end, we conduct an empirical study on omission bugs in Defects4J bug repository. First, we confirm that omission bugs are prevalent indeed, which accounts for 46.4% of our checked bugs. Second, we find that the omission bugs caused by incorrect data and control flow are very different in terms of their causes and difficulty of being localized. Lastly, we observe that the number of

categories for causes of omission bugs are very limited, which is a good indication for automating their localization.

Based on our findings, we build a neural network model to predict where to break the dead end of an omission bug when slicing can no longer work. We first collect the training data from the omission bugs in Defects4J repository. Based on the most common cause of omission bugs, we mutate 3193 omission bugs from 5 open source projects, the result shows that our model can achieve 84.1%, 66.5%, and 50.0% predication accuracy for various omission bugs (more details at Section 5). Moreover, we conduct a simulated debugging experiment, which simulates how programmer tracking incorrect data and control flow through slicing to locate those mutated omission bugs. Our results show that, equipped with our prediction model for breaking the dead end of dynamic slicing, our approach outperforms random benchmark on localizing the mutated omission bugs (63.8% over 2.8%).

In this paper, we make the following contributions: (1) We conduct an empirical study to comprehensively study the omission bugs in Defects4J repository. We confirm their prevalence, provide a taxonomy for their causes, and show patterns to potentially guide the automation of their localization. (2) We build a tool for this empirical study which can visualize, compare, and apply step-wise slicing on the buggy and fixed trace for Defects4J bugs. (3) We build a prediction model for breaking the dead end of omission bug when slicing can no long work. (4) We conduct a simulated debugging experiment to confirm the effectiveness of our prediction model.

2 OMISSION BUG CONCEPTS AND EXAMPLES

In this section, we define omission bug and show multiple examples of them. First, we assume a ground truth version (or fixed version) of the buggy program so that we know the correctness of its executing steps. More specifically, given a step on the trace of the buggy program, we know (1) whether it should happen, and (2) whether the value of any of its used variables is correct. Such an assumption can be fulfilled in practice either by requiring programmer's feedback or by comparing the traces of buggy and fixed versions of a program. In this regard, comparing to the traditional omission error defined for execution omission (i.e., bug due to missing executing *existing* code) [35], we extend the definition from a *trace* point of view which includes both execution omission [35] and code omission (bug due to missing some code). In the following, we note the kth step on trace π as π_k ($k \in \mathbb{N}$, k starts at 1).

Definition 1. ***Data Dependency Path.*** *Let π be a trace and π_d and π_r be two steps of π such that $r > d$. If π_d defines a variable var and π_r uses var, and there does not exist a step π_k where $d < k < r$ such that π_k defines var, we call the path from π_d to π_r as a **data dependency path** with regard to var, noted as $\langle \pi_d, \pi_u \rangle [var]$. In addition, we call π_d as the **data dominator** of π_u.*

Definition 2. ***Data Omission Bug.*** *Given a data dependency path $\langle \pi_d, \pi_u \rangle [var]$, if the value of variable var is incorrect on π_u and correct on π_d, then we say that a data omission bug, b, happens. We call the path $\langle \pi_d, \pi_u \rangle [var]$ as b's **critical path** and variable var as b's **critical variable**.*

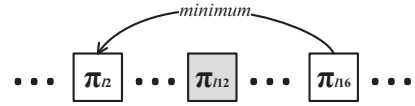


Figure 1: Critical Path of Data Omission Bug

Data omission bug indicates the critical variable should have been redefined before the end of critical path.

Example. Listing 1 shows an example of a data omission bug, which comes from the 2nd bug of Chart project in Defects4J repository. The bug is caused by missing assigning values to `minimum` and `maximum` variables in a corner case (line 8–11). From a debugging point of view, the programmer may observe that the value of `minimum` variable is unexpected to be `Infinity` at line 16 and apply dynamic data slicing to reach line 2 where `minimum` is initialized. Figure 1 shows the critical path of such a data omission bug. We note π_{ln} as the step executing line n. In this critical path, the critical variable is `minimum`. Moreover, it is a data dependency path starts from π_{l2} where `minimum` is defined and ends at π_{l16} where `minimum` is used, and no step in between π_{l2} and π_{l16} redefines `minimum`.

```
1  public static Range iterateDomainBounds(...){
2    double minimum = Double.POSITIVE_INFINITY;
3    double maximum = Double.NEGATIVE_INFINITY;
4    ...
5    if (..) {
6      for (...) {
7        for (...) {
8          + if (!Double.isNaN(value)) {
9          +     minimum = Math.min(minimum, value);
10         +     maximum = Math.max(maximum, value);
11         + }
12         uvalue = getEndXValue(series, item);
13       }
14     }
15   }
16   if (minimum > maximum) {
17     return null;
18   }
19   ...
20 }
```

Listing 1: Example of Data Omission Bug

Definition 3. ***Control Dependency Path.*** *Given two steps π_c, π_f such that $c < f$, π_c executing n_c and π_f executing n_f, we call a path from π_c to π_f as a **control dependency path** if n_f control-depended[1] on n_c, and, there does not exist a step π_k ($c < k < f$) executing n_k and n_f control-depends on n_k. We denote the condition of π_c as con and the control dependency path as $< \pi_c, \pi_f > (con)$. In addition, we call π_c as the **control dominator** of π_k.*

Definition 4. ***Control Omission Bug.*** *Given a control dependency path $\langle \pi_c, \pi_k \rangle (con)$, if π_k should not happen and π_c is correct, then we say that a control omission bug happens. We call such a control dependency path $\langle \pi_c, \pi_k \rangle (con)$ as bug's **critical path**.*

Control omission bug indicates the control flow in between the critical path should have been altered to avoid the step at its end.

Example. Listing 2 shows an example of a control omission bug, which comes from the 3rd bug of Math project in Defects4J repository. The bug is caused by missing the code to return in a corner case (line 6–7). From a debugging point of view, the programmer may observe that line 9 is unexpectedly executed and he can apply dynamic control slicing to reach line 3 where the condition `len !=`

[1] A more detailed definition on static control dependency can be referenced in [21]

b.len is correctly evaluated to false. The critical path of such a control omission bug is showed in Figure 2.

```
1  public double linearCombination(){
2     ...
3     if (len != b.length)
4        throw new DimensionMismatchException(len, b.length);
5     ...
6     +if (len == 1)
7     +  return a[0] * b[0];
8     final double[] prodHigh = new double[len];
9     double prodLowSum = 0;
10    ...
11 }
```

Listing 2: Example of Control Omission Bug

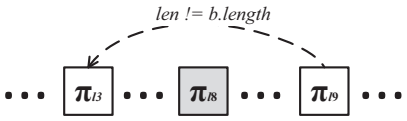


Figure 2: Critical Path of Control Omission Bug

Definition 5. ***Occur Step and Dead End Step.*** *Let p be the critical path of a control or data omission bug, b, which starts at π_s and ends at π_e, we call π_s and π_e as the **dead end step** and **occur step** of b respectively.*

Definition 6. ***Break Step.*** *Let b be an omission bug on trace π, and the trace after b is fixed be π'. The **break step** of b is either (1) a step before which the fix is applied [2] or (2) an incorrect step in between the occur step and the dead end step which allows programmers to apply dynamic slicing for approaching the root cause.*

Example. Figure 3 shows an example for the first and second condition in Definition 6 for Listing 1. In Figure 3, the upper trace π is buggy, and the lower trace π' is obtained from applying the fix. Each rectangle indicates a trace step and its name π_{ln} means the line number n it visits. The dotted lines represent the matching relation between two traces, the solid lines represent the data flow, and the dash lines represent the control flow. In such case, π_{l12} (the grey rectangle with solid border) is regarded as the break step according to the first condition, as the fix is applied before this step.

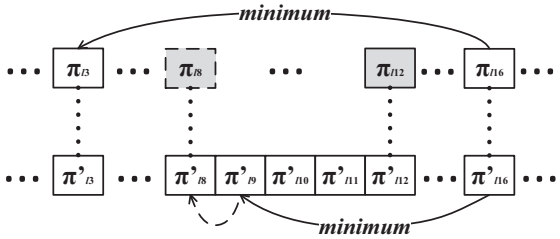


Figure 3: Example of Break Step

Now, let us assume that code in line 8–11 of Listing 1 has already existed in the buggy version (i.e., they are not the fix) and π_{l3} is still the data dominator of π_{l16} with regard to variable minimum because line 9 is not executed. As showed in Figure 3, line 8 will be executed at π_{l8}, which indicates that the condition !Double.isNaN(value) is evaluated to be false. In such case, π_{l8} is regarded as the break step according to the second condition as it allows us to apply dynamic slicing to further approaching the root cause.

[2] Note that it is possible to have more than one fix options for an omission bug, in such case, we regard the step before which any possible fix option is applied as the break step.

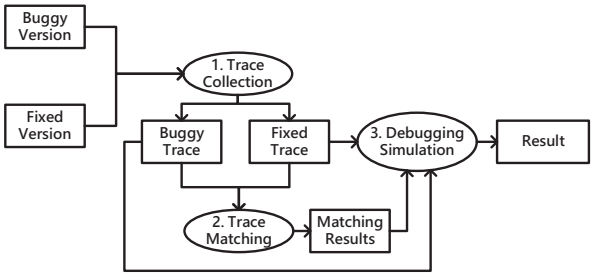


Figure 4: Study Setup

3 EMPIRICAL STUDY

In order to have a thorough understanding of data and control omission bugs, we conduct an empirical study on Defects4J repository [11] with the following research questions:

- **RQ1:** How prevalent are the omission bugs with regard to all the bugs in repository.
- **RQ2:** What are the main reasons for omission bugs in repository?

3.1 Study Setup – Identifying Omission Bugs

Figure 4 shows how we process each Defects4J bug in our empirical study. Defects4J repository describes each bug by its buggy version, its fixed version, and the test cases which fail in the buggy version and pass in the fixed version. Given a Defects4J bug and its failing test case, we first collect the trace of its buggy version and that of its fixed version (Step 1). Then, we leverage a trace matching technique to align the steps between the buggy trace and the fixed trace (Step 2). With the reference to the fixed trace, as indicated by the matching results, we can know each step on the buggy trace is either correct or incorrect from data (reading wrong variables) or control (should not happen) point of view. Then, we simulate programmers' debugging process on the buggy trace (Step 3). More specifically, starting from the end of the buggy trace where the fault is revealed, we continuously conduct dynamic data slicing (if a step reads incorrect variable) and dynamic control slicing (if a step should not happen). After the debugging process stops and the root cause cannot be located in such a manner, we know the dynamic slicing comes to a dead end and it is an omission bug.

3.1.1 Trace Collection and Matching. When collecting the program trace, we not only sequentially collect the executed statements, we also build the data and control dependencies for the trace steps. We improved Zhang's control-flow based trace matching algorithm [30, 35] to make it work on two traces with different source code. Our improvement is to match trace steps with regard to the code changes. For example, if a if keyword is modified to a while, the steps influenced by the if-condition should be matched to steps in the first iteration of the while-loop. Based on the matching result, each step on the buggy trace, whose source code is not modified, can fall into either of the following categories:

Algorithm 1: Debugging Simulation

```
Input  : a fault-revealing step step_f on the buggy trace
Output : a critical path path_c
1  step_d, step_prev ← step_f;
2  while step_d is not correct and step_d.source is not fixed do
3      step_prev ← step_d;
4      if step_d is data incorrect then
5          var ← incorrect variable on step_d;
6          step_d ← data_dom(step_d, var);
7      else if step_d is control incorrect then
8          step_d ← control_dom(step_d);
9  end
10 if step_d is correct then
11     if step_prev is data incorrect then
12         var ← incorrect variable on step_prev;
13         return < step_d, step_prev >[var];
14     else if step_prev is control incorrect then
15         con ← conditional expression on step_d;
16         return < step_d, step_prev >(con);
17 end
18 else
19     return null ;                    // step_d.source is fixed
20 end
```

- **correct:** the step can be matched with a step on the fixed trace and all its read variables has the same value as its matched step.
- **data incorrect:** the step can be matched with a step on the fixed trace and some of its read variable has different value from that of its matched step.
- **control incorrect:** the step cannot be matched with any step on the fixed trace.

3.1.2 Debugging Simulation. Algorithm 1 depicts how our debugging simulation process identify omission bug. In the debugging process, we continuously apply dynamic data or control slicing on the buggy trace until the process goes into a dead end or finds the root cause. We consider the bug is an omission bug in the former case.

In Algorithm 1, we start the simulated debugging at the end of the buggy trace, which is where the fault is revealed. If the working step ($step_d$) is data incorrect, we find its data dominator (see Definition 2) by dynamic data slicing (line 4–6). If the working step is control incorrect, we find its control dominator (see Definition 3) by dynamic control slicing (line 7–8). The process continues until the working step is correct (i.e., dead end) or the source code is modified as a fix (line 2). In the former case, we consider an omission bug happens. We record the critical path (see Definition 2 and 4) as either $\langle step_d, step_{prev}\rangle[var]$ or $\langle step_d, step_{prev}\rangle(con)$ for the omission bug (line 10–17).

3.2 Implementation

We build an Eclipse plugin called Tregression to visualize the bugs in Defects4J repository. A snapshot and demo video of our tool is available on its Github website [1]. In our tool, we visualize the buggy trace and its fixed trace on left and right panel. Users can click the step on either trace to (1) check its read and written variables, (2) compare its corresponding step on the other trace, and (3) compare the code before and after the fix. Moreover, users can apply dynamic data and control slicing operation on each step.

Table 1: Overview of Omission Bug Prevalence

Simulation Result		Project						Total
		Chart	Closure	Lang	Math	Mockito	Time	
Omission bugs	**Control**	7	5	24	16	8	6	67
	Data	5	12	9	12	2	4	43
Localizable bugs		14	7	27	44	20	15	127
Discarded bugs	**Non-deterministic**	0	13	1	0	1	1	16
	Not applicable for slicing	0	17	3	2	7	1	30
	Over-long trace	0	79	1	32	0	0	112
Total		26	133	65	106	38	27	395

3.3 Study Result

3.3.1 RQ1: Omission Bug Prevalence. Table 1 shows an overview of omission bugs in Defects4J repository.

First, 237 out of 395 bugs are applicable for this study. We discard 158 bugs because of the following three reasons: (1) the bug is a non-deterministic program (our trace matching algorithm requires stable trace to replay the bug), (2) the buggy trace only contain correct step, or (3) either the buggy trace or the fixed trace is over-long (i.e., over 100K steps). We justify discarding the bugs due to the second and third reason as follows.

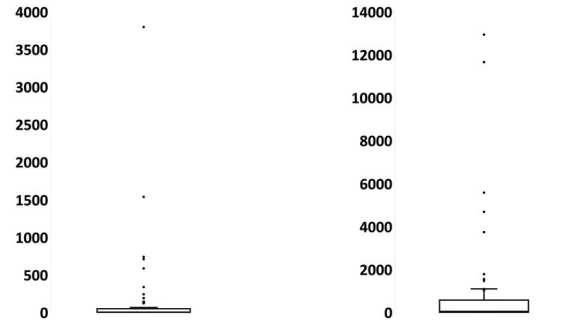
As for the second reason, Listing 3 shows an example, which is the simplified version of the 18th bug of Time project in Defects4J repository. In this bug, the buggy version miss enclosing the method invocation `iGregorianChronology.getDateTimeMillis()` with try-catch block. It results in the termination of buggy version with an exception at line 4. In contrast, the fixed version does not terminate at line 4 and continue its execution to line 6. As a result, every step in buggy trace is correct except that the trace misses some steps (for catching the exception). In this case, our simulated debugging approach is not applicable as the slicing technique cannot work. Moreover, such bugs are also not our interested omission bugs.

```
1   public long getDateTimeMillis(...) {
2       ...
3     + try{
4         instant = iGregorianChronology.getDateTimeMillis
5                 (year, monthOfYear, ...)
6     + }catch (IllegalFieldValueException ex) {
7     +  ...
8     + }
9       ...
10  }
```

Listing 3: Example of the Bug Unapplicable for Slicing

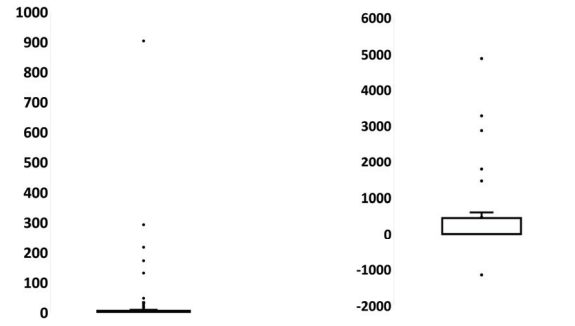
As for the third reason, we discard the bug with trace length over 100K for the limited scalability of existing Java trace collection tool [16]. We will discuss more about this issue in Section 3.4.

Of the 237 bugs we studied, 110 bugs (46.4%) are reported as omission bugs by Algorithm 1, i.e., tracking incorrect data and control cannot lead to the root cause. Of all the omission bugs we detected, the control omission bugs account for 60.9% (67/110) and the data



(a) Control Omission Bug (b) Data Omission Bug

Figure 5: Critical Path Length



(a) Control Omission Bug (b) Data Omission Bug

Figure 6: Over-skipped Step Number

omission bugs account for 39.1% (43/110). Thus, we conclude that:

> Omission bug is prevalent among all the Defects4J repository, and control omission bugs are of comparable proportion as data omission bugs.

Moreover, we further investigate the critical path length and over-skipped step number (i.e., distance between the break step and the dead end step, see Definition 6) for each omission bug. Critical path length indicates the worst effort for a programmer to manually break the dead end of an omission bug. In contrast, over-skipped step number indicates the effort to break the dead end of an omission bug if a programmer sequentially check through the trace from the dead end step. Figure 5 and Figure 6 show the distribution of critical path length and over-skipped step number of control and data omission bugs respectively.

In both figures, we can observe that control omission bugs show different characteristics from data omission bugs. First, both the average critical path length and the average over-skipped step number in control omission bugs are smaller than that in data omission bug. More specifically, average critical path length of control omission bugs is 138.9 and that of data omission bugs is 1095.5, in contrast, average over-skipped step number of control omission bugs is 32.2 and that of data omission bugs is 432.2. Second, as showed in Figure 6b, one data omission bug has the phenomenon of "under-skip", that is, the root cause happens before the dead end step.

In summary, we conclude that:

> In general, data omission bugs are harder to be localized than control omission bugs. Moreover, seldom as it is, the root cause of the bug may happen before the dead end step in data omission bugs.

3.3.2 RQ2: Omission Bugs Taxonomy. For those 110 bugs, we further studied how those omission is fixed in Defects4J repository.

Control Omission Bugs Taxonomy. From a syntactic point of view, we summarize 5 categories of control omission bugs, i.e., missing if-block, missing if-throw, invoking different method, missing try-catch block, and passing wrong parameter.

Missing if-block. A condition is missed in the code. This category includes missing `if`, `for`, `while` condition. Note that their block bodies do not include `throw` statement.

Missing if-throw. A condition is missed in the code. This category includes missing `if`, `for`, `while` condition and their block bodies include `throw` statement.

Invoking incorrect method. The control flow is unexpectedly altered by calling an incorrect method. Thus, all the follow-up steps incurred by that method invocation are control incorrect. A typical scenario is the misuse of polymorphism, for example, the programmer forget to override a method in the superclass.

Missing try-catch block. Try-catch block can be considered as an alternative of if-else-block. Missing a try-catch block can lead program to crash out of unexpected exception. As an example in Listing 4 (from the 83th bug of Mockito project), the buggy program is unexpectedly terminated in line 3 out of a `CmdLineException`.

```
1  public int parseArguments(Parameters params){
2     ...
3     - String param = params.getParameter(0);
4     + String param = null;
5     + try {
6     +  param = params.getParameter(0);
7     + } catch (CmdLineException e) {}
8  }
```

Listing 4: Example of Missing Try-Catch Block

Passing wrong parameter. For such a category, an expected exception does not happen because of passing wrong parameter during a method invocation. Listing 5 shows an example (from 4th bug of Time project), the program is not supposed to run into line 10 as an expected exception is supposed to happen at line 7. Tracing through dynamic control flow leads to line 3, which causes an omission bug.

```
1  public Partial with(...){
2     ...
3     if(...){
4        return;
5     }
6     ...
7     - Partial newPartial = new Partial(iChronology, newTypes, newValues);
8     + Partial newPartial = new Partial(newTypes, newValues, iChronology);
9     ...
10    return newPartial;
11 }
```

Listing 5: Example of Passing Wrong Parameter

Strictly, missing if-throw is a special case of missing if-block, we make them exclusive as the follow reasons. From the perspective of occur step (see Definition 5, i.e., where the fault is revealed) and where is the root cause, we divide the influence of a control

omission bug into intra-method and inter-method. If both occur step and root cause are within a method, we regard the influence as intra-method, otherwise, we regard it as inter-method. The influence of missing if-block is intra-method, while that of missing if-throw is sometimes inter-method. For example, line 8 in Listing 6 is unexpectedly executed because `guess()` method (line 7) misses a if-throw inside its method body. In such case, where the fault is revealed (i.e., occur step) is not within the same method of the root cause, we regard its influence is inter-method.

```
1  public void testMath844(...){
2     ...
3     if(...){
4        return;
5     }
6     ...
7     guesser.guess();
8  }
```

Listing 6: Example of If-Throw

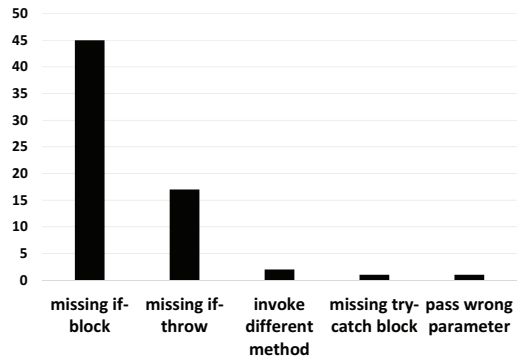


Figure 7: Solution for Control Omission Bug

Figure 7 shows the distribution of all these categories. We can see that the majority of control omission bug is caused by missing if-block and missing if-throw while the rest ones are the minority.

Data Omission Bugs Taxonomy. Still from a syntactic point of view, we summarize 4 categories of solution for data omission bugs, i.e., missing assignment, incorrect evaluated condition, incorrect condition, invoking new method, and missing if-block. It is straightforward to see how missing an assignment leads to a data omission bug. Thus, we explains other three categories.

```
1  private Cluster<T> getNearestCluster(...) {
2     ...
3     Cluster<T> minCluster = null;
4     for (Cluster<T> c : clusters) {
5        double distance = point.distanceFrom(c.getCenter());
6        if (distance < minDistance) {
7           minDistance = distance;
8           minCluster = c;
9        }
10    }
11    return minCluster;
12 }
```

Listing 7: Example of Incorrect Evaluated Condition

Incorrect evaluated condition. Incorrect evaluated condition alters the control flow to avoid the redefinition of the critical variable (see Definition 2). Listing 7 shows an example (from the 79th bug of Math project). The value of variable `minCluster` at line 11 is `null`, which is caused by the fact that the condition `distance < minDistance` (line 6) never been true and line 8 is never executed.

Incorrect condition. Different from *incorrect evaluated condition*, incorrect condition avoids the redefinition of the critical variable because of incorrect boolean expression in code. Listing 8 shows an example (from the 43th bug of Math project). The `value` variable (line 5) is not incremented due to incorrect comparison for `varianceImpl` and `variance` in code.

```
1  public void addValue(...){
2     ...
3     -if (!(varianceImpl instanceof Variance))
4     +if (varianceImpl != variance)
5        varianceImpl.increment(value);
6     ...
7  }
```

Listing 8: Example of Incorrect Condition

Miss invoking method. Miss invoking method can be regarded as a special case of missing assignment. The difference lies in that the new method is usually the library method which assigns a field inside a library class. For example, the invocation of `java.util.Calendar.getTime()` sets its `fields[0]` field.

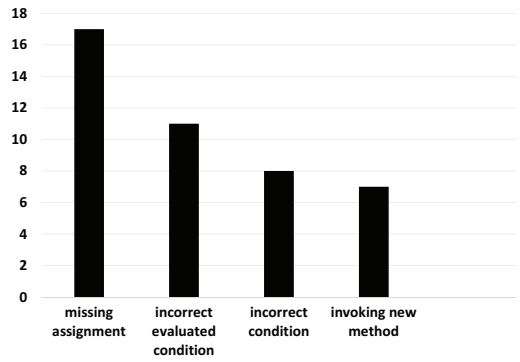


Figure 8: Data Omission Bug Cateogry

Figure 8 shows the distribution of all these categories. Compared to Figure 7, the distribution of categories of data omission bug is more even in the four categories.

In summary, we conclude that:

> In general, there are lots of syntactic means to alter data and control flow to create omission bugs. However, from an empirical point of view, the omission bugs are caused by only a limited number of syntactic reasons.

3.4 Threats to Validity

The major threat to validity in this empirical study is that we miss the bugs with trace length over 100K. The reason is that the state-of-the-art trace recoding techniques [16] do not scale well for building very large Java trace model, including the read/written (library) variables of each trace step and the data and control dominance relationships among trace steps. From this point, we may miss some omission bug categories in those large traces. In the future, we aim to build a more scalable trace recording technique to generalize our findings. The other threat is that the "fix version" (i.e., ground truth) of each omission bug is unique in this study. In the future,

we need to conduct more study to compare the break steps when multiple fix options exist for an omission bug.

4 APPROACH

As our empirical study indicates strong patterns to escape the dead end of omission bugs, we propose to localize an omission bug in a data-driven manner. In this section, we design a technique for assisting slicing-based debugging. More specifically, we assume that programmers can debug a software fault by providing their feedback (e.g., data or control incorrect) on trace steps and using slicing to gradually approach the root cause. When the dead end caused by omission bug happens, i.e., given a data or control dependency path starting from step π_s and ending by step π_e, and π_e is incorrect while π_s is correct, we aims to recommend a trace step in between π_s and π_e,

- where the critical variable should have been assigned for the data omission bug, or
- where control flow should be altered to avoid the unexpected step for the control omission bug.

By observing the omission bugs in Defects4J repository, we first manually feature engineer these bugs for predicting break steps of omission bugs. More specifically, the prediction model takes input as the features of the critical path of an omission bug and an arbitrary trace step and, outputs the probability of the step to serve as the break step. To this end, we choose neural network to conduct the predication for its rich expressiveness over other classification models such as Naive Bayes [7] and SVM [10]. We train the neural network model with the omission bugs in Defects4J repository, and test the model with our mutated omission bugs. We mutate the omission bugs with regard to the omission bug taxonomy (see Section 3.3.2) so that the mutated omission bugs are close to real ones. Our aim is to build a model which can fit well in real omission bugs and generalize well in mutated the omission bugs.

4.1 Feature Engineering

In this section, we introduce the common features shared by control and data omission bug as well as their specific features.

4.1.1 Common Features. Given a trace step s, and an omission bug b, the common features include the length of critical path and the syntactic features of the occur step, dead end step, the trace step s, and their contexts. The syntactic feature of a step s describes the encoding of the minimum AST node containing the source line of s. The encoding is represented by a vector of boolean variables.

Encoding AST Node. We first categorizing AST node types into a taxonomy tree with regard to their similarity with each other. In our implementation for Java programming language, we refer to Eclipse Java AST document [27] for building the taxonomy tree for all the Java AST node types. A simplified tree is shown in Figure 9. In Figure 9, under the top virtual nodes, there are totally 11 nodes in the second layer, representing abstract AST node such as variable declaration and expression. The third layer has 92 nodes each of which represents a concrete AST node such as single variable declaration and field access. Thus, we encode a AST node with a vector with length of 103 (11+92), each dimension represents a concrete node in the third layer or an abstract node in the second layer. Any

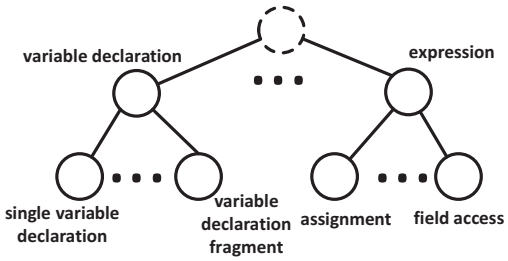


Figure 9: Simplified Java Taxonomy Tree Example

AST node must fall in either of the nodes in the third layer (concrete nodes) along with its parent in the taxonomy tree. Therefore, the corresponding two dimensions are set to 1 and all the other dimensions are set to 0. The idea of using hierarchical structure allows similar AST nodes (e.g., assignment and field access) to share training results.

Encoding Syntax of Trace Step. For a trace step s, we concern 6 AST nodes. They are the AST nodes for s, occur step, dead end step, as well as each of their AST parents in the code (readers may refer to Figure 10 as the parent-child relationship in AST). The AST parent indicates the context information. For example the occur step (line 11) in Listing 7, we care about both the AST node type of line 11 (i.e., return statement) and the AST node type of its context (i.e., method declaration).

In summary, the common features concatenate all the above features and vectors, and have 1 + 6×(11+92) = 619 dimensions.

4.1.2 Specific Features for Control Omission Bugs. For control omission bug, we use AST walk and split data/control dependency to embed the specific features of a trace step.

AST Walk. Given an omission bug b whose occur step π_o and dead end step π_d, and let π_k be a step between π_o and π_d, AST walk indicates the syntactic proximity from the source code of π_k to that of π_o. From the perspective of AST traverse, the *walk* consists of three directions, i.e., up, right, and down, in terms of AST of the code. Taking the code in in Listing 7 as example, the occur step happens at line 11 and the break step is at line 6. The AST walk from line 11 to line 6 is showed in Figure 10. In Figure 10, each node represents an AST node, its number indicates the line number in Listing 7, and arrows represents the walk direction. We can see that the walk from line 11 to line 6 takes 0 ups, 1 rights, and 2 downs.

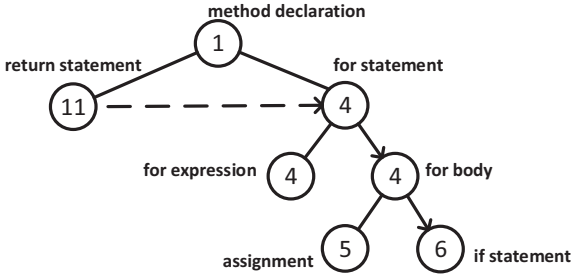


Figure 10: AST Walk Example

Split Data and Control Dependencies. The break step of a control omission bug must lie in between the dead end step π_c and occur step π_k. Thus, looking for such a break step is similar to looking for a step s which can split the code so that $\langle \pi_c, s \rangle$ is allowed while $\langle s, \pi_k \rangle$ is to be avoided. Thus, we take the number of data and control dependencies between $\langle \pi_c, s \rangle$ and $\langle s, \pi_k \rangle$ as two features.

4.1.3 Specific Features for Data Omission Bugs. For data omission bug (let its critical path be $\langle \pi_d, \pi_u \rangle[var]$), we introduce critical conditional step and variable similarities to embed the features of a trace step.

Critical Conditional Step. The critical conditional steps are the trace steps on π where a conditional expression is evaluated and the negation of its value can lead to the execution of redefining var. Such a feature is a boolean value, i.e., a trace step is either a critical conditional step or not.

Read and Written Variable Similarity. Given a trace step s, we define features to measure how similar its read and written variables to the critical variable var. Based on the variable type (i.e., field or local variable), we use different number of dimensions to describe similarity features. If a variable var_c is of different type with var, all its similarity dimensions are 0, otherwise, we use the following rules to create a vector for var_c:

- **Field:** If the critical variable is a field, we have a vector of four boolean dimensions, i.e., (1) whether var_c shares the same parent (object using var_c as its field) with var, (2) whether the parent of var_c has the same data type with var (e.g., both of them are fields in `Calendar` data type), (3) whether var_c is of the same data type with var_c, and (4) whether var_c is of the same name with var.
- **Local Variable:** If the critical variable is a local variable, we have have a vector of two boolean dimensions, i.e., (1) whether var_c is of the same data type with var_c, and (2) whether var_c is of the same name with var.

Note that a step may read or write multiple variables, each of which can be represented by a vector of k dimensions (e.g., k is 4 if the variable is a field and 2 if the variable is a local variable). In such case, we select the most similar read (written) variable vector as the read (written) variable similarity vector for s.

4.2 Neural Network Structure

Figure 11 shows our neural network structure, which has 1 input layer, 1 hidden layer, and 1 output layer. Different from traditional fully connected network, the number of neurons in the hidden layer corresponds to the number of "groups" in input features. For example, the three dimensions in AST walk corresponds to a neuron in the hidden layer (Group1 in Figure 11), and the last 103 dimensions of AST node encoding (Group9 in Figure 11) corresponds to another. Therefore, the network structures for control, field, and local variable omission bugs are different from each other.

We design such a network not only for predicting, but for interpretation as well. With such a network, the weight on each edge from hidden layer to output layer indicates the significance of each group to the final probability and the weight on each edge from input layer to hidden layer indicates the significance of each dimension to its group.

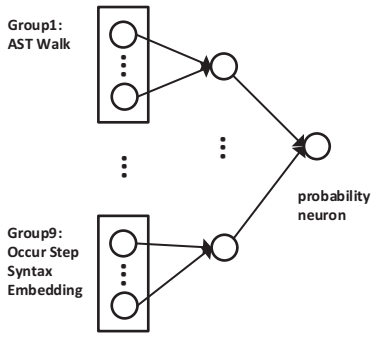


Figure 11: Neural Network Structure

In this work, we use cross-entropy loss function to evaluate our model during the model training. We use ReLu activation function for hidden layer and Sigmoid activation function for output layer.

5 EVALUATION

We conduct our evaluation to answer the following two research questions:

- **RQ1:** Whether our model can predict the break steps for omission bugs accurately?
- **RQ2:** Enhanced dynamic slicing with our model, can we localize the omission bugs efficiently?

5.1 Training Evaluation

With the findings of our empirical study, we define five types of mutations, i.e., remove an assignment, remove a if-condition (i.e., make the if-body always be executed), negate a if-condition, remove a if-throw, and remove the whole if-block. These five mutation types cover the majority of causes described in Section 3.3.2, and they are effective to cause omission bugs. We conduct the mutation on 5 Java open source projects, as showed in Table 2. The valid mutation means the mutations successfully kill the test case and cause omission bugs.

Table 2: Mutated Project Overview

Project	Version	LOC	#Valid Mutation
Aapache-math	2.2	97449	186
Apache-lang	3.5	73423	1099
Jfreechart	1.2.0	148852	1457
Apache-collections	3.2.2	56134	373
Apache-cli	1.3.1	6552	78

Learning Settings. Given the inputs and network structure are different for learning control omission bug, field omission bug, and local variable omission bug, we tune their model with different parameters, as showed in Table 3. We decide the parameters by empirical trials. Note that the training process of the neural network is a process to iteratively decrease the loss value of cross entropy loss function. In this experiment, we use loss threshold instead of iteration number to decide when the learning process stops. That is, once the loss value is below the threshold, the learning process stops. We also attach random seed for repeating our approach.

Learning Effect. Table 4 shows our learning effect on control, field, and local variable omission bug (i.e., COB, FOB, and LVOB

Table 3: Parameter Setting

omission bug\parameter	Learning Rate	Threshold	Random Seed
Control Omission Bug	0.05	0.4	18
Field Omission Bug	0.1	0.65	20
Local Variable Omission Bug	0.05	0.5	0

in Table 4). We compare learning effect in terms of true positive rate (TP Rate), true negative rate (TN Rate), and total accuracy. In general, our model achieve acceptable performance on control and local variable omission bugs. However, the model does not perform well on field omission bugs. Our observation indicates that the difference between field omission bugs is much larger than control and local variable omission bugs. In Table 4, the model achieves poor true positive ratio (27.3%) and good true negative ratio (76.5%) on training set but the situation reverses on testing set, or mutation set. It indicates that the data distribution of field omission fluctuates more than the other two omission bugs. Furthermore, we investigate into these omission bugs and find that, the influence of field omission bugs is usually inter-method while that of control omission bugs and local variable omission bugs is usually intra-method. As a result, field assignment can happen almost anywhere along the trace, which is more random than local variable assignment.

Table 4: Learning Effect

Omission Bug		COB	FOB	LVOB
Training	TP Rate	74.2%	27.3%	77.8%
	TN Rate	88.5%	76.5%	77.1%
	Total	84.3%	51.9%	76.0%
Testing	TP Rate	87.1%	82.4%	57.5%
	TN Rate	81.0%	49.4%	68.0%
	Total	84.1%	50.0%	66.5%

Feature Significance. As mentioned in Section 4.2, our neural network structure is also designed for interpretation to understand how significant a feature is. The larger the absolute value a weight has, the more influence it has on the predication result. The sign indicates its positive or negative impact on the result. Table 5 shows the weights for various omission bugs. For example, control omission is more influenced by the feature of AST walk as well as syntactic feature of occur step and dead end step. For the weight of AST walk, i.e, -0.68, it indicates that the break step usually appears in small walk from the occur step. Similarly, the weight 1.17 indicates that being a critical conditional step is a strong indicator for being a break step for a local variable omission bug. For syntactic features, they indicate that the AST node type with larger (e.g., 0.62) or smaller (-0.62) index usually has stronger influence on the result. Given the limit of paper space, readers can refer to our website [2] for our indexing for AST node.

5.2 Simulated Debugging Experiment

Based on our prediction model, we enhance the our simulated debugging algorithm (Algorithm 1) by suggesting breaker steps with our model. Algorithm 2 describes our enhanced debugging simulation algorithm. The algorithm takes three inputs, a fault-revealing step, the number of break steps we can recommend for an omission bug, and a work list that we keep other suggested breakers.

Algorithm 2: Enhanced Debugging Simulation

Input : a fault-revealing step $step_f$ on the buggy trace, breaker number $limit$, a stack for breaker steps $worklist$
Output: whether the find is found

```
1  step_stop ← original_simulated_debugging(step_f);
2  if step_d is the root cause then
3  |  return true;
4  end
5  omission_bug ← get_omission_bug(step_stop);
6  breakers ← recommend_breakers(omission_bug, limit);
7  for breaker ∈ breakers do
8  |  worklist.push(breaker)
9  end
10 while worklist is not empty do
11 |  breaker ← worklist.pop();
12 |  enhanced_debugging_simulation(breaker, worklist)
13 end
14 if stack is empty then
15 |  return false
16 end
```

In Algorithm 2, when we detect an omission bug (line 5) by the process described in Algorithm 1 (line 1, i.e., dynamic slicing), we get recommended break steps and keep them in the work list (line 5–6). Then, we retrieve a break step from the work list and restart debugging from the step in the same way (the recursive call in line 12). The algorithm stops either because we localize the root cause or the work list is empty.

In this experiment, we take a random recommender as our benchmark. The random recommender indicates the performance of omission bug localization if the simulated programmer choose a random step as the break step. We deem our approach as ineffective if its performance is comparable to that of the random recommender. We compare our approach with the benchmark by setting the number of recommended break steps to 1, 3, and 5. That is, we recommend top-1, top-3, and top-5 breakers in our prediction model while 1, 3, and 5 random steps as breakers for benchmark (line 6). We run our approach and random recommender on all the mutated omission bugs from open source projects, Table 6 shows the results. Table 6 shows that our approach outperforms benchmark in all options on all types of omission bugs. With regard to Table 4, the performance of bug localization is highly relevant to model prediction accuracy. Based on our prediction model, we can accurately localize the control omission bugs even with top-1 option. Moreover, Table 6 also indicates that field omission bugs is the most difficult omission bug to localize. Noteworthy, though our model does not perform well for field omission bugs, we still improve the performance of the benchmark significantly.

In summary, our evaluation shows that (1) our approach can work well to localize certain specific omission bug such as control and local variable omission bugs, (2) combined with our prediction model, we can break the dead end of slicing and localize the omission bug more efficiently.

5.3 Threats to Validity

One major threat is that we use simulated debugging experiment to imitate how human programmer debug their code. It is essential to

Table 5: Feature Significance

Omission Bug	AST Walk	Split Dep	Critical Conditional Step	Read Var Sim	Written Var Sim	Critical Path Length	Step Syntax	Step Context	Occur Step Syntax	Occur Step Context	Dead End Step Syntax	Dead End Step Context
Control Omission Bug	-0.68	-0.04	/	/	/	-0.01	0.4	-0.67	-0.08	0.34	0.62	-0.62
Field Omission Bug	/	/	0.038	0.24	0.69	-0.018	0.94	0.46	-0.32	0.56	-2.2	-0.03
Local Variable Omission Bug	/	/	1.17	-0.87	0.42	0.93	-0.68	0.71	-0.29	-1.0	0.44	-0.1

Table 6: Simulation Result

Omission Bug	top-1		top-3		top-5	
	Model Breaker	Random Breaker	Model Breaker	Random Breaker	Model Breaker	Random Breaker
Control Omission Bug	94.6%	10.2%	98.4%	19.2%	99.5%	23.7%
Field Omission Bug	34.8%	2.5%	53.3%	7.3%	59.1%	10.2%
Local Variable Omission Bug	50.9%	14.2%	72.5%	30.0%	82.2%	38.0%
Total	63.8%	8.2%	76.6%	16.9%	81.1%	21.5%

deliver a tool and collect the feedback of human programmer. In our future work, we will conduct a controlled user study on how our tool integrated with learned model can help human programmer in the process of debugging. The other threat is the limited number of mutation types for generating mutated bugs. We choose those mutation type based on the finding of our empirical study. The experiment shows that these mutations can generate omission bugs in a much more efficient way than traditional mutation such as changing operator and numbers. Nevertheless, more experiments are necessary to generalize our findings with more dynamic types of mutation.

6 RELATED WORK

Omission Error Research. Many research work [12, 20, 25, 28, 35] have pointed out that dynamic slicing cannot be used to localize the code "should have been" executed. To overcome the shortcoming of the slicing technique, Zhang et al. [35] proposed a force-execution technique and Wang et al. [28] proposed their relevant slicing algorithm to this end. Moreover, Qi et al. [20] proposed a solver-based approach to localize the regression bug. By comparing the correct version of the program, their approach can infer code omission error by encoding the buggy program, the correct program, and the test case into a satisfiability problem. In recent years, Sakuai et al. [25] enhanced Zhang and Wang's work [28, 35] with point-to-analysis.

All the above techniques have the assumption that the omitted code exists in the project so that they can analyze the program to lead the control flow to code that should have been executed. However, our empirical study shows that their techniques only solve a small portion of the whole omission bugs, i.e., data omission bugs under the category of incorrect evaluated condition and incorrect condition. Based on our study findings, we proposed a data-driven approach to train the model to handle omission bugs in a more comprehensive way.

Record and Replay for Debugging. Our approach is an enhancement for record and replay debugging, or time-travelling debugging [5, 13–15, 18, 19, 24, 31]. Such a technique allows the programmers to build a data causality chain to localize the root cause [18, 19, 31]. Moreover, based on the recorded trace, different queries can be used to localize trace steps. For example, Ressia et al. [24] proposed an approach which can track steps by specific object instance.

The most relevant techniques are *Whyline* proposed by Ko et al. [13–15] and *Microbat* proposed by Lin et al. [16]. *Whyline* allows user to select auto-generated questions, including why or why not question on program output as well as the recorded program trace and suggest the trace step based on slicing. *Microbat* asks four types of user feedback to suggest suspicious trace step and it can further learn these feedbacks to speed up reasoning the root cause. Our approach is complementary to both techniques to improve the localization of omission bugs.

Delta Debugging and Statistical Fault Localization. An important branch of debugging research is delta debugging [6, 9, 17, 20, 32–34] and statistical fault localization [3, 4, 8, 23, 26, 29]. Zeller et al. [32] first proposed the concept of delta debugging and apply it on regression testing. Then, the technique is soon to be refined by Misherghi et al. [17] to improve the result and the concept is soon applied to simplify test cases [34], isolate bug-causing variable [6, 33], and etc. We regard statistical fault localization as a special case of delta debugging, which are used to locate bugs by comparing a set of passed and failed test cases. The more time executed by failed test cases, a more suspicious a line of code is, and vice versa. Many metrics haven proposed to refine the statistical fault localization [3, 22, 29] An overview of spectrum-based techniques can be checked in [4].

All the above approaches need a reference to infer where the bug lies. Similar to existing debugging technique, we first use the correct version of program as a reference to look for omission bugs. However, by learning through a large number of bug corpus, we can localize the omission bug without any reference.

7 CONCLUSION

In this work, we comprehensively studied omission bugs in Defects4J repository and provide a taxonomy on how they can be caused from a syntactic point of view. Based on the findings in our empirical study, we build a deep learning model to predict the break steps of both control and data omission bugs. In the future, we will generalize our study on more bugs with longer traces and further improve the precision our prediction on break steps.

ACKNOWLEDGEMENT

We thank the anonymous reviewers for their valuable comments and suggestions. This research is supported by the National Research Foundation, Singapore (No. NRF2015NCR-NCR003-003).

REFERENCES

[1] [n. d.]. Tregression Github Website. https://github.com/llmhyy/tregression. ([n. d.]). Accessed Feb 2, 2018.

[2] 2018. Slice Breaker Website for ASE submission. https://sites.google.com/view/slicebreaker/home. (2018).

[3] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. 2009. Spectrum-Based Multiple Fault Localization. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*. 88–99.

[4] Rui Abreu, Peter Zoeteweij, Rob Golsteijn, and Arjan J.C. van Gemund. 2009. A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software* 82, 11 (2009), 1780 – 1792.

[5] Earl T. Barr and Mark Marron. 2014. *Tardis: Affordable Time-Travel Debugging in Managed Runtimes*. Technical Report. http://research.microsoft.com/apps/pubs/default.aspx?id=212395

[6] Holger Cleve and Andreas Zeller. 2005. Locating Causes of Program Failures. In *Proceedings of the 27th International Conference on Software Engineering*. 342–351.

[7] Nir Friedman, Dan Geiger, and Moises Goldszmidt. 1997. Bayesian Network Classifiers. *Mach. Learn.* 29, 2-3 (Nov. 1997), 131–163. https://doi.org/10.1023/A:1007465528199

[8] L. Gong, D. Lo, L. Jiang, and H. Zhang. 2012. Interactive fault localization leveraging simple user feedback. In *IEEE International Conference on Software Maintenance*. 67–76.

[9] Neelam Gupta, Haifeng He, Xiangyu Zhang, and Rajiv Gupta. 2005. Locating Faulty Code Using Failure-inducing Chops. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*. 263–272.

[10] Marti A. Hearst. 1998. Support Vector Machines. *IEEE Intelligent Systems* 13, 4 (July 1998), 18–28. https://doi.org/10.1109/5254.708428

[11] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA 2014)*. ACM, New York, NY, USA, 437–440. https://doi.org/10.1145/2610384.2628055

[12] Andrew J. Ko and Brad A. Myers. 2004. Designing the Whyline: A Debugging Interface for Asking Questions About Program Behavior. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '04)*. ACM, New York, NY, USA, 151–158. https://doi.org/10.1145/985692.985712

[13] Andrew J. Ko and Brad A. Myers. 2004. Designing the Whyline: A Debugging Interface for Asking Questions About Program Behavior. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 151–158.

[14] Andrew J. Ko and Brad A. Myers. 2008. Debugging Reinvented: Asking and Answering Why and Why Not Questions About Program Behavior. In *Proceedings of the 30th International Conference on Software Engineering*. 301–310.

[15] Andrew J. Ko and Brad A. Myers. 2009. Finding Causes of Program Output with the Java Whyline. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 1569–1578.

[16] Yun Lin, Jun Sun, Yinxing Xue, Yang Liu, and Jinsong Dong. 2017. Feedback-based Debugging. In *Proceedings of the 39th International Conference on Software Engineering (ICSE '17)*. IEEE Press, Piscataway, NJ, USA, 393–403. https://doi.org/10.1109/ICSE.2017.43

[17] Ghassan Misherghi and Zhendong Su. 2006. HDD: Hierarchical Delta Debugging. In *Proceedings of the 28th International Conference on Software Engineering*. 142–151.

[18] G. Pothier and ÃĽ Tanter. 2009. Back to the Future: Omniscient Debugging. *IEEE Software* 26, 6 (2009), 78–85.

[19] Guillaume Pothier and Éric Tanter. 2011. Summarized Trace Indexing and Querying for Scalable Back-in-time Debugging. In *Proceedings of the 25th European Conference on Object-oriented Programming*. 558–582.

[20] Dawei Qi, Abhik Roychoudhury, Zhenkai Liang, and Kapil Vaswani. 2009. Darwin: An Approach for Debugging Evolving Programs. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*. 33–42.

[21] Venkatesh Prasad Ranganath, Torben Amtoft, Anindya Banerjee, John Hatcliff, and Matthew B. Dwyer. 2007. A New Foundation for Control Dependence and Slicing for Modern Program Structures. *ACM Trans. Program. Lang. Syst.* 29, 5, Article 27 (Aug. 2007). https://doi.org/10.1145/1275497.1275502

[22] Manos Renieris and Steven P. Reiss. 2003. Fault Localization With Nearest Neighbor Queries.. In *Proceedings of International Conference on Automated Software Engineering*. 30–39.

[23] Thomas Reps, Thomas Ball, Manuvir Das, and James Larus. 1997. The Use of Program Profiling for Software Maintenance with Applications to the Year 2000 Problem. In *Proceedings of the 6th European SOFTWARE ENGINEERING Conference Held Jointly with the 5th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 432–449.

[24] Jorge Ressia, Alexandre Bergel, and Oscar Nierstrasz. 2012. Object-centric Debugging. In *Proceedings of the 34th International Conference on Software Engineering*. 485–495.

[25] Kouhei Sakurai and Hidehiko Masuhara. 2015. The Omission Finder for Debugging What-should-have-happened Bugs in Object-oriented Programs. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing (SAC '15)*. ACM, New York, NY, USA, 1962–1969. https://doi.org/10.1145/2695664.2695735

[26] R. Santelices, J. A. Jones, Yanbing Yu, and M. J. Harrold. 2009. Lightweight fault-localization using multiple coverage types. In *Proceedings of 31st International Conference on Software Engineering*. 56–66.

[27] Olivier Thomann Thomas Kuhn, Eye Media GmbH. [n. d.]. Abstract Syntax Tree. http://www.eclipse.org/articles/article.php?file=Article-JavaCodeManipulation_AST/index.html. ([n. d.]).

[28] Tao Wang and Abhik Roychoudhury. 2008. Dynamic Slicing on Java Bytecode Traces. *ACM Trans. Program. Lang. Syst.* 30, 2, Article 10 (March 2008), 49 pages. https://doi.org/10.1145/1330017.1330021

[29] Xinming Wang, S. C. Cheung, W. K. Chan, and Zhenyu Zhang. 2009. Taming Coincidental Correctness: Coverage Refinement with Context Patterns to Improve Fault Localization. In *Proceedings of the 31st International Conference on Software Engineering*. 45–55.

[30] Bin Xin, William N. Sumner, and Xiangyu Zhang. 2008. Efficient Program Execution Indexing. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '08)*. ACM, New York, NY, USA, 238–248. https://doi.org/10.1145/1375581.1375611

[31] Ding Yuan, Haohui Mai, Weiwei Xiong, Lin Tan, Yuanyuan Zhou, and Shankar Pasupathy. 2010. SherLog: Error Diagnosis by Connecting Clues from Run-time Logs. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*. 143–154.

[32] Andreas Zeller. 1999. Yesterday, My Program Worked. Today, It Does Not. Why?. In *Proceedings of the 7th European Software Engineering Conference Held Jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 253–267.

[33] Andreas Zeller. 2002. Isolating Cause-effect Chains from Computer Programs. In *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering*. 1–10.

[34] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and Isolating Failure-Inducing Input. *IEEE Transaction on Software Engineering* 28, 2 (2002), 183–200.

[35] Xiangyu Zhang, Sriraman Tallam, Neelam Gupta, and Rajiv Gupta. 2007. Towards Locating Execution Omission Errors. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*. ACM, New York, NY, USA, 415–424. https://doi.org/10.1145/1250734.1250782