This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2021.3125203, IEEE Transactions on Software Engineering

Evaluating and Improving Unified Debugging

Samuel Benton, Xia Li, Yiling Lou, Lingming Zhang

Abstract—Automated debugging techniques, including fault localization and program repair, have been studied for over a decade. However, the only existing connection between fault localization and program repair is that fault localization computes the potential buggy elements for program repair to patch. Recently, a pioneering work, ProFL, explored the idea of *unified debugging* to unify fault localization and program repair in the other direction for the first time to boost both areas. More specifically, ProFL utilizes the patch execution results from one state-of-the-art repair system, PraPR, to help improve state-of-the-art fault localization. In this way, ProFL not only improves fault localization for *manual repair*, but also extends the application scope of *automated repair* to all possible bugs (not only the small ratio of bugs that repair systems can automatically fix). However, ProFL only considers one program repair system (i.e., PraPR), and it is not clear how other existing program repair systems based on different designs contribute to unified debugging. In this work, we perform an extensive study of the unified debugging approach on 16 state-of-the-art program repair systems for the first time. Our initial experimental results on the widely studied Defects4J benchmark suite reveal various practical guidelines for unified debugging, such as (1) nearly all 16 studied repair systems positively contribute to unified debugging despite their varying repair capabilities, (2) repair systems targeting multi-edit patches can bring extraneous noise into unified debugging, (3) repair systems with more executed/plausible patches tend to perform better for unified debugging, (4) unified debugging effectiveness does not rely on the availability of correct patches from automated repair, and (5) we propose a new unified debugging technique, UniDebug++, which localizes over 20% more bugs within Top-1 than state-of-the-art unified debugging technique ProFL (evaluated against four Defects4J subjects). Furthermore, we conduct more comprehensive studies to extend the above experiments to make the following additional contributions: we (6) further perform an extensive study on 76.3% additional buggy versions from Defects4J (for Closure and Mockito) and confirm that UniDebug++ again outperforms ProFL by localizing 185 (out of 395 in total) bugs within Top-1, 14% more than ProFL, (7) investigate the impact of 33 SBFL formulae on unified debugging and observe that UniDebug++ consistently improves upon all formulae, e.g., 61% and 53% average improvement on MFR / MAR, (8) demonstrate that UniDebug++ can substantially boost state-of-the-art learning-based method-level fault localization techniques, (9) extend unified debugging to the statement level for first time and observe that UniDebug++ localizes 78 (out of 395 in total) bugs within Top-1 (22% more bugs than ProFL) and outperforms state-of-the-art learning-based fault localization techniques by 30%, and finally (10) propose a new technique, UniDebug+*, based on detailed patch statistics, to improve upon UniDebug++, e.g., further localizing up to 9% more bugs within Top-1 than UniDebug++.

Index Terms—Automated Program Repair, Fault Localization, Unified Debugging

1 Introduction

With the rapid development of information technology, software systems have been widely adopted in almost all aspects of modern society. However, software bugs (also called software faults in this paper) are inevitable because of the complexity of modern software systems. Software faults can cause software systems to crash or perform unexpected behaviors, both scenarios resulting in disaster, e.g., costing trillions of dollars in financial loss and affecting billions of people [1]. In practice, software debugging is essential for removing bugs from existing faulty software systems. Manual debugging, however, can be extremely challenging, tedious, and costly. Such impediments consume over 50% of the development time/effort [2] and cost the global economy billions of dollars [3].

To date, a huge body of research effort has been dedicated to automated debugging to relieve developer burdens, investigating both fault localization [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15] and automated program repair [16], [17], [18], [19], [20], [21], [22], [23], [24], [25], [26], [27], [28], [29], [30], [31], [32], [33], [34] techniques. *Fault localization* aims to precisely localize buggy elements within a buggy system based on dynamic and/or static program analysis, and can automatically produce a ranked list of suspicious code elements for developers, reducing their effort for manual bug checking across many forms of software systems. Classic spectrum-based fault localization (SBFL) techniques [5], [6], [7], [14] mainly analyze the statistical correlation between code coverage and test outcomes to infer potential buggy locations. For example, a code element primarily executed by failed tests is more likely to be suspicious. However, using only coverage information may not be precise enough. Therefore, researchers further propose mutation-based fault localization (MBFL) techniques [8], [9], [35], [36] by further considering the impact information between mutated code elements and tests (simulated via mutations). Recently, machine learning techniques have been used to combine various dimensions of debugging information for more powerful fault localization [12], [36],

- *Samuel Benton is with the Department of Computer Science, The University of Texas at Dallas, USA. E-mail: Samuel.Benton1@utdallas.edu*
- *Xia Li is with the Department of Software Engineering and Game Design, Kennesaw State University, USA. E-mail:xli37@kennesaw.edu*
- *Yiling Lou is with the Department of Computer Science, Purdue University, USA. E-mail:lou47@purdue.edu*
- *Lingming Zhang is with the Department of Computer Science, University of Illinois at Urbana-Champaign, USA. E-mail:lingming@illinois.edu*
- *Yiling Lou is the corresponding author.*

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2021.3125203, IEEE Transactions on Software Engineering

[37], [38].

While fault localization still requires manual repair, *automated program repair* (APR) aims to directly fix software bugs automatically with minimal human intervention. A typical test-driven APR technique takes a faulty program and its test suite as input and generates program patches with the end goal to find a patch passing all tests. Due to its promising future, various APR techniques have been proposed, including search-based, semantics-driven, and learning-based techniques [10], [21], [22], [23], [39]. For more details, please refer to recent surveys on fault localization [40] and APR [41].

Despite extensive research on automated debugging over the past decades, we still lack practical automated debugging techniques. Traditional fault localization techniques have been extensively studied in the literature [14], [15], [42], [43], but there is still no clear consensus about its effectiveness; meanwhile, although recent learning-based fault localization techniques can be more powerful, they usually require massive training data that may not always be available [12], [44]. Furthermore, it is also rather challenging for APR techniques to fix all possible bugs – even state-of-the-art APR techniques [21], [45], [46] can only fix a small ratio of real bugs (i.e., <20% for Defects4J [47]) automatically.

To enable more practical debugging, the unified debugging approach, ProFL, was recently proposed to unify fault localization and APR to boost both areas [48], [49]. While both fault localization and APR have been studied for over a decade, their only prior connection is that fault localization is leveraged as a supplier for pointing out potentially buggy locations for APR to fix. The unified debugging approach ProFL unifies the two areas in the other direction for the first time, i.e. leveraging large number of patch execution results generated during APR (even when APR fails to fix the bug) to further boost fault localization. The basic intuition is that if a patch passes some originally failing test(s), the patched location is very likely to have some close relationship with the real buggy location (e.g., sharing the same method or even same line), since otherwise the patch would not mute the bug impact and pass the originally failing test(s). Using the recent PraPR [45] APR system, ProFL is able to substantially boost/outperform state-of-the-art SBFL [6], [7], [50], MBFL [8], [9], [35], [36], and unsupervised/supervised-learning-based fault localization [11], [12], [44]. In this way, given any buggy project, ProFL not only directly returns the patches when *automated repair* works, but also provides improved fault localization hints for *manual repair* for all other cases. That is, ProFL not only significantly improves fault localization for *manual repair*, but also extends the application scope of *automated repair* to all possible bugs (not only the small portion of bugs that can be automatically fixed).

Despite this promising direction, the ProFL work only considers one APR system (i.e., PraPR), while there are many other available APR systems based on different designs and it is not clear how other APR systems contribute to unified debugging.

Therefore, to bridge this gap, we conduct the first extensive study of unified debugging on 16 state-of-the-art APR systems. These 16 systems represent recent public Java APR systems that execute without requiring specialized data or infrastructure. These selected systems utilize constraint-based [19], [20], [51], heuristic-based [21], [22], [23], and template-based [18], [52], [53] repair approaches seen in recent repair literature. Furthermore, we use real faults from Defects4J benchmark suite for our evaluation since it is the most widely used benchmark in recent fault localization and APR work (including the unified debugging work [48]). Our experimental results demonstrate that unified debugging can outperform/boost all existing fault localization techniques. We also propose two advanced unified debugging techniques and reveal various practical guidelines for further improving unified debugging and even software debugging in general.

To summarize, this paper makes the following main contributions:

- **Initial Study Contributions.** The conference version of this paper presents the first extensive study of unified debugging using 16 state-of-the-art APR systems. Our initial study [54] on 224 real bugs from Defects4J reveals various practical guidelines, including: (1) nearly all 16 studied APR tools can positively contribute to unified debugging despite their varying repair capabilities, (2) APR tools targeting multi-edit patches can bring noise and may degrade performance for unified debugging, (3) APR tools with more executed/plausible patches tend to perform better for unified debugging, and (4) unified debugging effectiveness does not exclusively rely on the availability of correct patches from APR.
- **Initial Study Technique - UniDebug++.** Based on results from our conference study (evaluated on 224 real bugs from Defects4J), we propose an advanced unified debugging technique (UniDebug++) to further rank tied code elements. From this new strategy, UniDebug++ localizes over 20% more bugs within Top-1 than ProFL.
- **Extended Study.** To enhance unified debugging, we extend our prior study [54] to 395 real bugs from Defects4J (i.e., 76.3% more bugs than the initial conference paper) and make further contributions. Specifically we: (1) confirm all our initial study findings generalize to the additional faults and UniDebug++ outperforms ProFL by 14% on all 395 Defects4J bugs and localizes 185 bugs within Top-1, (2) show UniDebug++ consistently improves upon all formulae, e.g., 61% and 53% average improvement on MFR / MAR, (3) demonstrate that UniDebug++ boosts state-of-the-art learning-based method-level fault localization techniques, and (4) observe that, at the statement level, UniDebug++ localizes 78 (out of 395 total) bugs within Top-1 (22% more bugs than ProFL) and outperforms state-of-the-art statement-level learning-based fault localization techniques by 30%.
- **Extended Study Technique - UniDebug+*.** Based on these extension results, we further propose a new technique, UniDebug+*, to even further improve upon UniDebug++, i.e., localizing up to 9% more bugs within Top-1 than UniDebug++.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2021.3125203, IEEE Transactions on Software Engineering

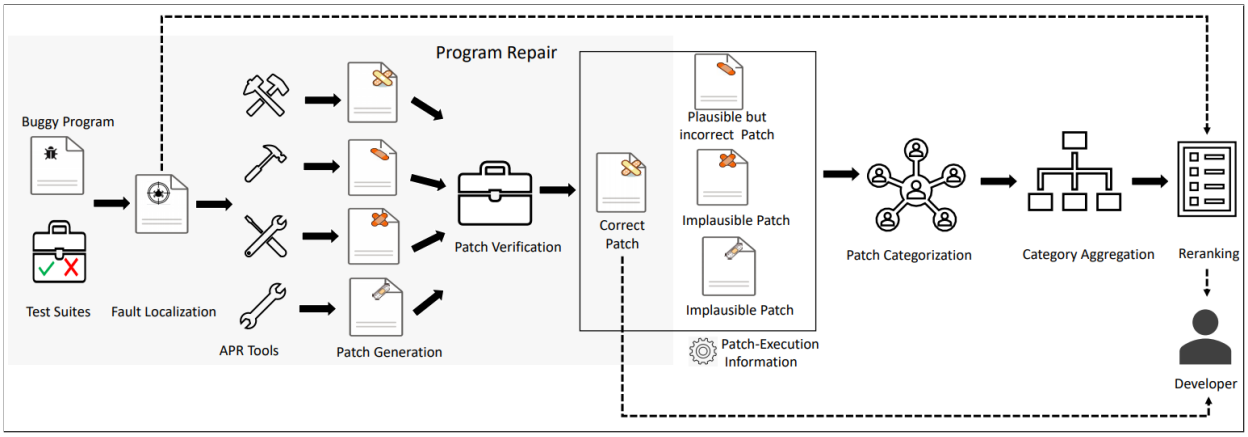

Fig. 1: The unified debugging process

2 Studied Approach

In this section, we first briefly discuss the traditional fault localization and program repair process (Section 2.1) to motivate unified debugging. Then, we present the underlying process for unified debugging (Section 2.2) and discuss variations of the unified debugging technique (Section 2.4). Lastly, we present a real-world example to further motivate our study (Section 2.3).

2.1 Fault Localization and Program Repair

Given a buggy program and its failing test suite, test-based fault localization computes each code element's probability to be buggy based on various techniques [5], [8], [12], [55], [56]. For example, the widely studied spectrum-based fault localization (SBFL) [5], [6], [7] will collect the dynamic coverage information for each failing/passing test to compute each code element's suspiciousness value. In this way, developers can choose to directly start *manual repair* with the help of such suspiciousness information.

Alternatively, developers can also choose to directly perform *automated program repair* (APR) [21], [22], [23]. Typical APR techniques leverage fault localization techniques to compute the potential buggy locations for patching, e.g., the Ochiai [5] SBFL technique has been widely used in recent APR work, such as PraPR [45], Simfix [21], and CapGen [57]. After the patch generation and validation, all the *plausible patches* (i.e., the patches that can pass all tests) are returned for manual inspection to find the final *correct patches* (i.e., the patches semantically equivalent to developer patches). In this way, the final correct patches are the only useful outcome from APR; in fact, even plausible but incorrect patches are treated as harmful in traditional APR work [51], since they require time-consuming and tedious manual inspection. However, to date, even state-of-the-art APR can only produce correct patches for a small ratio of real-world bugs, making APR a waste of resources in all other cases. For example, the current most effective APR works [17], [45], [46] cannot even fix 20% of bugs from the widely studied Defects4J [47] benchmark.

2.2 Unified Debugging

To further boost both the fault localization and APR areas, unified debugging [48], [49] aims to unify these two areas from the other direction for the first time. The basic insight of unified debugging is that the massive patch execution information from APR (even ones that do not lead to correct patches) can further help substantially improve fault localization to facilitate manual repair. In this way, unified debugging can report correct patches when possible, and more importantly can also return refined fault localization for all cases (even the cases without correct patches). Unified debugging not only extends the application scope of APR to all possible bugs (not only the bugs that can be automatically fixed), but also provides more precise fault localization. For example, ProFL [48], the first unified debugging technique based on the recent PraPR APR system, significantly improves/outperforms various state-of-the-art fault localization techniques (e.g., SBFL [5], [6], [7], MBFL [8], [9], [35], [36], and even learning-based techniques [11], [12]).

The basic assumption of unified debugging is that if a patch can pass some originally failing tests, its patch location may be closely related to the actually buggy locations (e.g., sharing the same code element, such as method). Similarly, if a patch fails some originally passing tests, its patch location may be closely related to correct locations since otherwise the passing tests would have failed before patching [49]. In this way, all generated patches can be categorized into exactly one the following categories according to execution information automatically collected during patch validation: (1) *CleanFix Patches*: patches passing on at least one originally failed test and not failing on any originally passed test, (2) *NoisyFix Patches*: patches passing on at least one originally failed tests but also failing on some originally passed tests, (3) *NoneFix Patches*: patches not impacting the outcome for any originally failed or passed test, (4) *NegFix Patches*: patches not passing any originally failed test but failing on some originally passed tests. Note that all such patch validation information can be directly obtained from the studied test-based APR tools. Unified debugging simply leverages such existing information to classify each patch into the aforementioned categories.

The overall approach of unified debugging is presented in Figure 1. Given any buggy program and its test suite, unified debugging first applies off-the-shelf APR systems to generate and execute various possible patches. Then, while existing APR work only returns the correct patches to the developers, unified debugging further utilizes the execution

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2021.3125203, IEEE Transactions on Software Engineering

Suspicious Method	SBFL	PraPR	Kali-A	TBar	UniDebug+
PolyhedronsSet.<init>	1.0	NoneFix	Unmodified	Unmodified	NoneFix
PolygonsSet.compute...	1.0	NoneFix	CleanFix	NoneFix	CleanFix
PolygonsSet.followLoop	1.0	NoneFix	NoneFix	Unmodified	NoneFix
AVLTree.getNotSmaller	1.0	NoneFix	NoneFix	NoneFix	NoneFix

TABLE 1: Motivating example of unified debugging from Math-32

information for all patches and categorizes them into relevant categories discussed in the previous paragraph. For each code element (e.g., method), unified debugging adopts the best category from its corresponding patches according to a predefined order (i.e., CleanFix > NoisyFix > NoneFix > NegFix) [49].

Finally, all the elements are reranked first according to their patch categories, e.g., all elements with the CleanFix category are ranked higher than all elements with the NoisyFix category; after that, the elements within the same category are then further reranked in descending order by their initial suspiciousness scores computed by any existing fault localization technique (i.e., Ochiai [5] by default). In this way, the developers will obtain largely refined fault localization for all possible bugs (even including the case where no correct or plausible patch is found). We refer to the implementation of the core unified debugging approach previously outlined as **UniDebug** for the duration of this paper. Note that the basic UniDebug technique can be applied with any existing generate-and-validate automated program repair system.

2.3 Motivating Example

In this section, we use Math-32 from Defects4J (V1.0.0) [47], a widely used real-world Java bug benchmark, to motivate our study. Math-32 denotes the 32nd buggy version of Apache Commons Math project. The bug is located in method `computeGeometricalProperties` of Class `PolygonsSet`.

Table 1 shows four example suspicious methods including the actual buggy method highlighted in gray. Please note that we disregard the arguments since the class and method names can sufficiently distinguish them.

In the table, Column "SBFL" indicates the suspiciousness score of each method according to the state-of-the-art SBFL technique Ochiai [5] with aggregation strategy [44], which aggregates the maximum suspiciousness values from statements to methods and has been demonstrated to substantially outperform raw method-level SBFL. Columns "PraPR", "Kali-A", and "TBar" represent the UniDebug unified debugging approach using the patch execution information from APR systems PraPR, Kali-A, and TBar, respectively. The patch category information for each method is included in the table. Unmodified, hereby referred to as Non-Modify, represents a new patch category implying that these code elements are never patched by an APR tool. Lastly, Column "UniDebug+" presents the technique simply using all patches from the prior three APR systems. From the motivating example, we have the following interesting findings.

First, unified debugging using other APR systems can have promising fault localization results even when the PraPR system used by ProFL cannot help improve the performance. For example, Kali-A can directly rank the buggy method at the 1st location, while both SBFL and ProFL rank the bug at the 4th location. Figure 2 represents the patch generated by Kali-A (left side) and the correct patch provided by developers (right side). From the patches, we found that Kali-A generates a patch by changing the buggy conditional statement into `if (false)` which is useless for fixing the real bug; however, this patch does help pinpoint the actual bug location, demonstrating the generality of unified debugging for all possible APR systems. This finding motivates us to perform an extensive study to investigate the effectiveness of different APR systems for unified debugging. Second, different APR systems have different unified debugging performances and combining them may potentially result in even more powerful unified debugging. Shown in the last column of Table 1, simply combining all patches generated by different APR systems can also localize the bug within Top-1.

2.4 Unified Debugging Variants

Shown in our motivating example (Section 2.3), leveraging UniDebug on one particular program repair tool may not always be able to sufficiently distinguish between elements with somewhat similar repair execution information, and it is promising to consider multiple repair tools together. Therefore, we further consider a few extension points for the core unified debugging technique and describe such extension points in the following subsections.

2.4.1 The Use of Multiple APR Tools

The foundation of such extension points (hereby referred to as variants) relies on using the repair execution information from multiple APR tools. UniDebug, as outlined in Section 2.2, employs a single repair tool, only using the given tool's patch repair information. Rather than using the patch repair information generated from a single APR tool, the unified debugging technique can be adjusted to adopt the overall best category from **multiple employed tools** (see Section 2.3 and Table 1). In this way, (1) we can leverage inherent advantages contained within a wide variety of APR tools and (2) we mitigate any potential biases / overfitting issues inherent to individual tools. Beyond the usage of repair information from multiple APR tools, the unified debugging approach continues unchanged. We refer to this variant as **UniDebug+** and further investigate its effectiveness in Section 4.4.

2.4.2 Including Additional Refined Ranking Features

The final phase of the unified debugging approach reranks all elements according to each element's suspiciousness value and patch execution information. For UniDebug, this reranking utilizes (1) the element's aggregated patch category and (2) the element's suspiciousness value. This simple strategy however frequently results in elements earning the same rank (e.g., a tie) without any further way to distinguish buggy elements from correct elements, even if they have wildly different detailed repair execution results (e.g., with different number of tools and patches being able to produce the best patch category). As these elements cannot be further distinguished, they remained tied which degrades overall effectiveness due to the ranking scheme employed by unified debugging and similar fault localization studies.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2021.3125203, IEEE Transactions on Software Engineering


```
/** File = PolygonsSet.java */
protected void computeGeometricalProperties() {
    ...
    if (v.length == 0) {
        final BSPTree<Euclidean2D> tree = getTree(false);
-       if ((Boolean) tree.getAttribute()) {
+       if (false) {
            // the instance covers the whole space
            setSize(Double.POSITIVE_INFINITY);
            setBarycenter(Vector2D.NaN);
    ...
}
```

(a) Kali-A Math-32 CleanFix patch

```
/** File = PolygonsSet.java */
protected void computeGeometricalProperties() {
    ...
    if (v.length == 0) {
        final BSPTree<Euclidean2D> tree = getTree(false);
-       if ((Boolean) tree.getAttribute()) {
+       if (tree.getCut() == null && (Boolean) tree.getAttribute()) {
            // the instance covers the whole space
            setSize(Double.POSITIVE_INFINITY);
            setBarycenter(Vector2D.NaN);
    ...
}
```

(b) Math-32 developer patch

Fig. 2: Generated patch of Kali-A and developer patch for Math-32

Thus, the following variants use even more patch repair information to break these ties and more precisely distinguish buggy elements from correct elements.

One variant involves utilizing **the number of tools which produce an element's best patch category** (Column "Tool Freq." in Table 2). Our intuition is that, for quality patches, the likelihood of an element actually being faulty increases as more distinct tools also generate patches of equal quality. Thus, the ranking scheme is adjusted to the following: sorted by (1) aggregated element patch category, (2) the element's suspiciousness (descending), and (3) **the frequency of the aggregated element patch category generated by all tools** (descending). Note that this variant is the same as UniDebug+ except that it further uses (3) to break the ties. In this paper, we refer to this variant as **UniDebug++**. An example of how UniDebug++ further distinguishes buggy elements from correct elements is described in Table 6 and further variant details are discussed in Section 4.4.

Again, the idea behind unified debugging is code elements involved in high quality repair patches are likely to be incorrect. Consequently, the confidence in an element's incorrectness increases as more high quality patches are discovered for the element. To this end, we further explore the use of *patch frequency* as a ranking feature for unified debugging. Specifically, *the number of patches categorized as the element's aggregated patch category* is used as a ranking feature to further break ties (see Column "Patch Freq." in Table 2). The ranking scheme is further modified as follows: sorted by (1) aggregated element patch category, (2) the element's suspiciousness (descending), and (3) **the number of generated patches categorized as the aggregated element patch category** (descending). Note that this variant is the same as UniDebug+ except that it further uses (3) to break the ties. We call this variant **UniDebug+*** and further discuss details in Section 4.8.

Table 2 delineates the exact differences between all unified debugging variants considered in this paper.

	Number of Employed Tools	Ranking Features			
		Category	Susp.	Tool Freq.	Patch Freq.
UniDebug[1]	**1**	Yes	Yes	**No**	**No**
UniDebug+	≥ 2	Yes	Yes	**No**	**No**
UniDebug++	≥ 2	Yes	Yes	Yes	**No**
UniDebug+*	≥ 2	Yes	Yes	**No**	Yes

[1] Note that we consider as ProFL an instance of UniDebug which specifically utilizes PraPR. Thus, in the context of UniDebug, PraPR and ProFL are the same and interchangeable.

TABLE 2: Differences between unified debugging variants

3 Study Design

3.1 Research Questions

As part of this paper's original publication, we investigated the following research questions *(but now with 76.3% more studied bugs):*

- **RQ1:** How does unified debugging perform with all studied APR systems?
- **RQ2:** How do unmodified code elements during APR impact unified debugging?
- **RQ3:** How does unified debugging correlate with program repair effectiveness?
- **RQ4:** How do we further advance state-of-the-art unified debugging with all studied APR systems via UniDebug++?

As an extension of the original paper, we aim to investigate these additional research questions:

- **RQ5:** How does unified debugging perform with different SBFL formulae?
- **RQ6:** How does unified debugging boost learning-based fault localization?
- **RQ7:** How does unified debugging perform at the statement level?
- **RQ8:** How does a new advanced ranking strategy UniDebug+* improve unified debugging?

3.2 Research Question Motivations

In short, the following are the motivations for our research questions (**RQ**): RQ1 assesses the effectiveness of the unified debugging with different state-of-the-art APR tools on real-world systems. Since many of the studied APR tools may not generate any patch for certain code elements, RQ2 empirically (1) examines the impact of code elements unmodified/unpatched by APR tools and (2) investigates what patch category is most optimal for such elements. As different APR tools may have totally different capabilities for fixing bugs, RQ3 further examines the impact of APR effectiveness (e.g., generating correct patches) on unified debugging effectiveness. Then, RQ4 examines how to further improve the unified debugging approach for enhanced fault localization via considering APR tool frequency information.

As the extension of the original conference paper, we also additionally consider the following RQs besides augmenting all our first four RQs with 76.3% more subjects. More specifically, RQ5 examines the fragility of unified debugging

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2021.3125203, IEEE Transactions on Software Engineering

Tool Category	Tools
Constraint-based	ACS, Cardumen, Dynamoth
Heuristic-based	Arja, GenProg-A, jGenProg, jKali, jMutRepair, Kali-A, RSRepair-A, Simfix
Template-based	AVATAR, FixMiner, kPar, PraPR, TBar

TABLE 3: Repair systems studied

Project	Name	# Bugs	# Tests	LOC
Chart	JFreeChart	26	2,205	96K
Lang	Apache Lang	65	2,245	22K
Math	Apache Math	106	3,602	85K
Time	Joda-Time	27	4,130	28K
Mockito	Mockito framework	38	1,366	23K
Closure	Google Closure compiler	133	7,927	90K
	Total	**395**	**21,475**	**344K**

TABLE 4: Studied bugs from Defects4J v1.2.0

effectiveness with respect to state-of-the-art SBFL formulae. Given the inclusion of repair information in the fault localization process, RQ6 examines how such information may also boost existing state-of-the-art learning-based fault localization techniques (including both supervised-learning-based and unsupervised-learning-based ones). In addition to the method-level fault localization, RQ7 also assesses the feasibility of the unified debugging approach for statement-level fault localization, which is also quite important for automated debugging. Lastly, RQ8 examines how to further improve the unified debugging approach via considering the detailed patch frequency information. These research questions are designed to evaluate unified debugging in different usage scenarios and in a more extensive way.

3.3 Experimental Setup

For this study, we considered all 16 program repair systems accessible from a recent study [58]. Furthermore, we also considered the recent PraPR repair system [45] which the initial unified debugging work is based on. Table 3 shows the breakdown of all the APR systems studied, including: heuristic-based - Arja [23], GenProg-A [23], jGenProg [22], jKali [22], jMutRepair [22], Kali-A [23], RSRepair-A [23], and Simfix [21]; constraint-based - ACS [51], Cardumen [20], and Dynamoth [19]; and template-based - AVATAR [18], FixMiner [52], kPar [53], TBar [17], and PraPR [45]. We manually modified all studied APR systems to collect the detailed patch execution information required by unified debugging and ensured our modified versions did not impact underlying tool functionality. Each system used original time settings suggested by the original papers.

We perform the study on the widely-used benchmark Defects4J 1.2.0 [47], which consists of 395 real-world bugs from six software systems. Detailed statistics are shown in Table 4. However, since many studied APR systems have been implemented to target version 1.0.0 (and older) of Defects4J, we found that most of them (e.g., CapGen, ACS, Arja, and GenProg-A) do not natively support or cannot successfully execute Closure or Mockito. Therefore, all studied tools are evaluated against the four core subjects (i.e., Chart, Time, Lang, and Math); in addition, TBar, kPar, AVATAR, FixMiner, and PraPR are evaluated on subjects Closure and Mockito.

Note that the paper's original publication (RQs 1-4) considered only **core subjects** (Chart, Time, Lang, and Math) while this paper's new material (e.g., RQs 5-8 and Section 4.1.4) additionally consider **extra subjects** (Closure and Mockito).

Each tool was executed using the same JDK version found in the tool's original publication, allowing us to obtain repair execution results as close as possible to the tool's original results. Thus, in our experiments, we ultimately utilized two JDK versions, again as dictated by each repair system's original publication, JDK 1.8.0.242 (hereby referred to as JDK 1.8) and JDK 1.7.0.80 (hereby referred to as JDK 1.7). Systems Simfix and Dynamoth executed using JDK 1.8 exclusively. Systems Cardumen, jGenProg, jKali, and jGenProg executed using JDK 1.8 and validated system test suites with JDK 1.7. All other systems executed using JDK 1.7 exclusively.

All our experiments (except for Section 4.7) utilized the following environment: 36 3.0GHz Intel Xeon Platinum Processors, 60GBs of memory, and Ubuntu 18.04.4 LTS operating system. As part of our original conference paper's extension, we recollect the data for Section 4.7. This recollection used the following environment: 56 2.0GHz Intel XeonCPU E5-2660 v4 processors, 225GBs of memory, and Ubuntu 14.04.6 LS operating system.

3.4 Implementation Details

3.4.1 Configurations

Although unified debugging can be used to refine any existing fault localization technique, by default, the original unified debugging work, ProFL, utilizes APR to refine state-of-the-art SBFL technique, Ochiai [50] with aggregation strategy [44]. Actually, the original ProFL work demonstrates that the basic unified debugging approach has consistent performance for refining different state-of-the-art fault localization techniques. Therefore, in this paper, we also focus on using Ochiai (with aggregation) to investigate the impact of different repair systems. Meanwhile, we investigate the impact of 33 total SBFL formulae on our improved unified debugging in RQ5.

Following the original ProFL work, this study primarily focuses on method-level fault localization (i.e., localizing potential buggy methods), as researchers have demonstrated that class-level fault localization can be too coarse-grained [43] while statement-level fault localization can be too fine-grained and miss necessary contextual information helpful with later *manual* program repair [42]. Note that in Section 4.7, we further investigate unified debugging for statement-level fault localization, which has been widely leveraged for *automated* program repair [10], [21], [22], [23], [39], [45], [59].

3.4.2 Non-Modify Category

For the original ProFL work, the used PraPR repair system [45] is extremely fast due to the bytecode-level manipulation and can generate patches for almost all possible *suspicious* methods, i.e., methods executed by failed tests (since methods not executed by failed tests should not be responsible for the current test failures). However, for all other APR tools, there may exist many suspicious methods without any patch, since it is expensive for APR tools to generate patches for every method. Therefore, besides the

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2021.3125203, IEEE Transactions on Software Engineering

four categories of methods mentioned in Section 2.2, we create a new category, Non-Modify, to represent the methods that do not receive any patch for a specific APR system. It is unclear how this new category compares with the other four categories studied in the original unified debugging work. Therefore, we explore the impact of ranking this new Non-Modify category within the existing four ProFL categories in Section 4.2. Note that as the default setting, we put Non-Modify alongside the NegFix category since the plurality of *all patches* fall into the NegFix category (thus the majority of Non-Modify methods may also fall into the NegFix category if they had been generated with patches).

3.4.3 Repair Tool Integration with ProFL

The original ProFL tool has been implemented as a publicly available Maven plugin. We obtained the original ProFL source code from the authors and analyzed the interface between ProFL and its underlying APR system. Then, we modified all 16 studied APR system to produce detailed patch execution information consistent with the original ProFL interface (e.g., regarding the patch location, failing and passing tests for each patch). In this way, we can safely replace the original PraPR system with any other studied APR systems for our study. Please note that we also augment the original ProFL code to handle the new Non-Modify method category.

3.5 Evaluation Metrics

Following prior work [11], [12], [44], we measure the number of bugs localized within Top-1, Top-3, and Top-5 positions as the primary metrics for this study. The reason is that researchers have observed that most developers will abort automated debugging tools if they cannot return the actual buggy elements within the Top-5 positions [43]. Specifically, given a set of elements (e.g., method or statement) which tie for the same rank, each element is assigned the **worst** rank of the tied elements, following prior work [12], [36], [48]. Furthermore, we also present the mean first rank (MFR) and mean average rank (MAR) results widely used in prior fault localization [12], [36] and unified debugging [48] work. More specifically, for precise localization of all buggy elements of each bug, we compute the average ranking of all the buggy elements for each bug; MAR is simply the mean of the average ranking of all bugs. Similarly, for bugs with multiple buggy elements, the localization of the first buggy element is critical since the remaining buggy elements may be directly localized after that; therefore, we use MFR to compute the mean of the first buggy element's rank for each bug.

4 Result Analysis

4.1 RQ1 - Performance of Unified Debugging with Different APR Systems

In this subsection, we first investigate the effectiveness of unified debugging on all 16 studied APR systems against four of studied subjects (i.e., Lang, Chart, Time and Math from the Defects4J benchmark). All 16 considered APR tools have been explicitly designed to execute or are natively compatible with these four core subjects (i.e. from each tool's original paper). Thus we explicitly focus on these four Defects4J subjects (totaling 224 bugs) which represent real-world scenarios where many APR tools can successfully execute on a system. Then, we further study the subset of the studied APR systems compatible with all six Defects4J subjects on all the 395 bugs.

4.1.1 Experimental Results on Core Subjects

Figure 3 shows the fault localization results on these four subjects in terms of the Top-1, Top-3, Top-5, MFR, and MAR metrics. The upper sub-figure represents the Top-N results and bottom sub-figure indicates the MFR/MAR results. Each bar in both sub-figures represents different APR systems. Note that we use the default treatment for the Non-Modify category which inserts such methods within the NegFix category, i.e., CleanFix $>$ NoisyFix $>$ NoneFix $>$ NegFix $=$ Non-Modify (discussed in Section 3.4.2). Also note that the 16 repair systems in this figure are ordered chronologically with respect to the date for each publication following an existing APR study [58]. We also include the results of state-of-the-art SBFL (i.e., Ochiai with aggregation) and the first unified debugging technique (i.e., ProFL which uses the program repair tool PraPR) for comparison (Note that ProFL has been demonstrated to outperform/improve all state-of-the-art fault localization [48], [49]). From Figure 3, we have the following observations. First, with most APR systems unified debugging performs better than state-of-the-art SBFL! For example, in terms of Top-1, 15 out of 16 tools help improve SBFL and only Arja fails to meet the initial SBFL results. That said, Arja still localizes 74 faults within Top-1 which is fairly close to the SBFL result. This finding indicates the broad applicability of the unified debugging approach. Second, even though existing APR study [58] has observed that more recent APR systems can fix more bugs than earlier systems, there is no obvious trend showing that unified debugging with more recent (i.e., chronologically later) APR systems can help localize more bugs than earlier ones. This finding demonstrates that APR systems' capability to produce correct patches is not highly correlated to the unified debugging effectiveness in fault localization. Lastly, different results of the 16 APR systems indicate that each APR system has its own advantages and disadvantages for unified debugging. The potential reason is that some tools have exclusive abilities to repair various classes of bugs by leveraging different algorithms to generate patches, incurring various levels of effectiveness for fault localization. This finding further motivates us to combine multiple APR systems to advance state-of-the-art unified debugging (studied in Sections 4.4 and 4.8).

> **Finding 1:** Despite their varying repair capabilities, almost all the studied 16 APR systems individually boost state-of-the-art SBFL and contribute to unified debugging.

4.1.2 Qualitative Analysis

Now we perform a detailed qualitative analysis to investigate the different performance of different APR systems. Table 5 shows a subset of the unified debugging

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2021.3125203, IEEE Transactions on Software Engineering

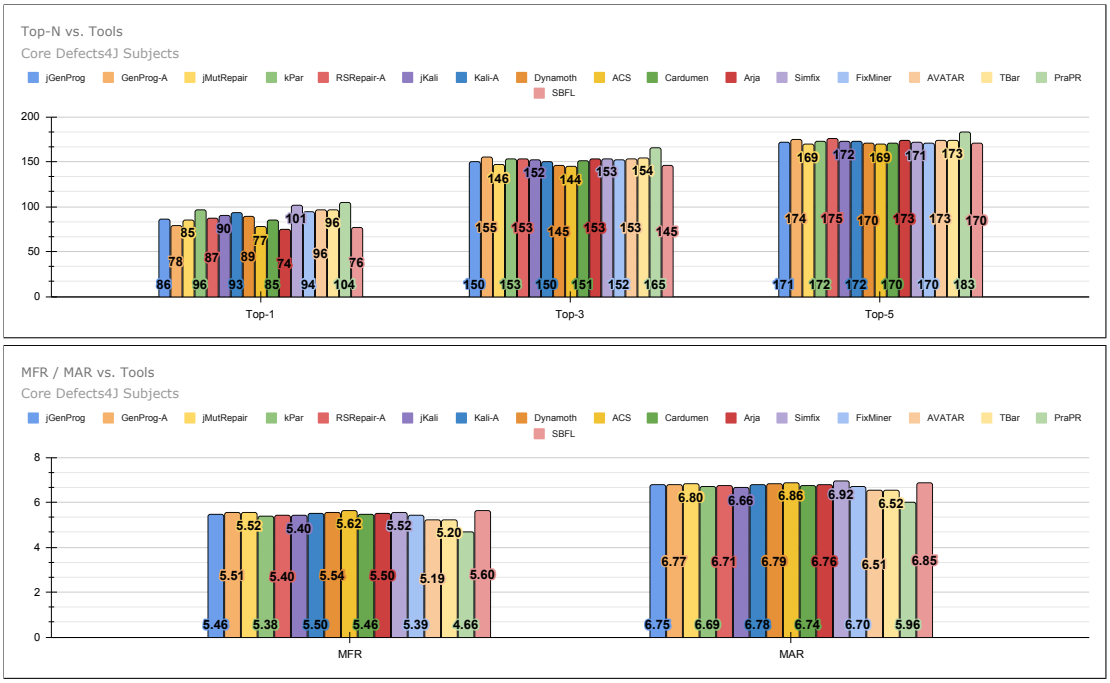

Fig. 3: UniDebug results with all studied APR systems on core Defects4J subjects

results with different APR systems from Math-77. Note that we also include the results for UniDebug, which simply uses all patches from different APR systems. Column "EID" describes the ID for each method. Column "Category" represents the category computed for each method based on each tool or technique. Column "Rank" describes where each method ranks in the final results for the given tool or technique. In this table, actual buggy elements are highlighted in gray. Math-77 fails on two tests, `testBasicFunctions` within class `ArrayRealVectorTest` and `testBasicFunctions` within class `SparseRealVectorTest`, both from the `org.apache.commons.math.linear` package. The buggy methods for Math-77 involve modifications of methods `e4` and `e5`, according to developer patch in Figure 4. According to traditional SBFL, all five methods tie and consequently rank 5th (according to the *worst* ranking). We next discuss the performance of three example APR systems for unified debugging on Math-77:

TBar is able to generate a CleanFix patch by exclusively modifying method `e4`, shown in Figure 5. This CleanFix patch successfully passes one of the originally failed tests and passes every other test. Even though this patch is not a correct patch, it still helps boost the rank of one buggy method to Top-1 since the patch shares the same location with the bug and thus is able to positively mute the bug impact via modifying the return value. This further demonstrates the effectiveness of unified debugging, boosting an actual buggy method to Top-1 with an incorrect patch.

RSRepair-A generates 43 NegFix and 355 NoneFix patches across 14 unique methods for this bug. RSRepair-A produces NoneFix patches for all five methods in Table 5. From this categorization, `e1` - `e5` are ranked the same as the SBFL results. Note that, in this case, although RSRepair-A was not able to improve SBFL, it will not deteriorate the fault localization results when combining with the more effective TBar. This is because when putting all patches together, methods with higher patch categories have precedence over lower patch categories (i.e. `e4`'s category will remain CleanFix).

Arja is a rather interesting case. It actually produces many incorrect CleanFix patches for Math-77, including all five suspicious methods shown in Table 5. We were surprised by the fact that Arja can produce so many CleanFix patches since they are usually hard to generate. Digging into various such patches, we found that Arja specifically targets multi-edit patches (i.e., each patch modifies multiple program locations). For example, one such CleanFix patch is shown in Figure 6. In this way, if any part of the multiple edits within a multi-edit patch makes some failing tests to pass, the patch can potentially be CleanFix even if other edits do not otherwise affect functionality. This leads unified debugging to associate all modified methods of the patch with the CleanFix category. Furthermore, such noise incurred by multi-edit APR systems can also be rather harmful when combining different APR systems for unified debugging. For example, shown in the last column of Table 5, UniDebug+ also cannot distinguish the five suspicious methods as they all fall into the CleanFix category due to Arja's inclusion. Therefore, we exclude all such multi-edit APR tools (i.e., Arja, GenProg-A, and RSRepair-A) when combining different APR systems for all unified debugging variants to remove unnecessary noise.

> **Finding 2:** APR systems specifically targeting multi-edit patches can bring noise into unified debugging, as each multi-edit patch involves multiple modifications and many modifications are not helpful in muting the bug impacts even if the patch can pass some originally failed test(s).

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2021.3125203, IEEE Transactions on Software Engineering

EID	Suspicious Method	SBFL		TBar		RSRepair-A		Arja		UniDebug+	
		Susp.	Rank	Category	Rank	Category	Rank	Category	Rank	Category	Rank
e1	AbstractRealVector:getL1Norm()D	0.707	5	Non-Modify	5	NoneFix	5	CleanFix	5	CleanFix	5
e2	AbstractRealVector:getNorm()D	0.707	5	Non-Modify	5	NoneFix	5	CleanFix	5	CleanFix	5
e3	ArrayRealVector:getL1Norm()D	0.707	5	Non-Modify	5	NoneFix	5	CleanFix	5	CleanFix	5
e4	ArrayRealVector:getLInfNorm()D	0.707	5	**CleanFix**	**1**	NoneFix	5	CleanFix	5	CleanFix	5
e5	OpenMapRealVector:getLInfNorm()D	0.707	5	Non-Modify	5	NoneFix	5	CleanFix	5	CleanFix	5

TABLE 5: UniDebug+ with different APR systems for Math-77

```
/** File = ArrayRealVector.java */
public double getLInfNorm() {
    double max = 0;
    for (double a : data) {
-       max += Math.max(max, Math.abs(a));
+       max = Math.max(max, Math.abs(a));
    }
    return max;
}
```

(a) Modifications for ArrayRealVector.java

```
/** File = OpenMapRealVector.java */
- public double getLInfNorm() {
-     double max = 0;
-     Iterator iter = entries.iterator();
-     while (iter.hasNext()) {
-         iter.advance();
-         max += iter.value();
-     }
-         return max;
- }
```

(b) Modifications for OpenMapRealVector.java

Fig. 4: Correct developer patch for Math-77

```
/** File = ArrayRealVector.java */
public double getLInfNorm() {
    double max = 0;
    for (double a : data) {
        max += Math.max(max, Math.abs(a));
    }
-       return max;
+       return getDimension();
}
```

Fig. 5: TBar's incorrect CleanFix patch for Math-77

4.1.3 Quantitative Analysis

Since the capability to produce correct patches is not highly correlated with unified debugging effectiveness, we further perform detailed quantitative analysis to explore what factors of APR systems are highly correlated to the effectiveness of unified debugging. Figure 7 represents the correlation analysis between different factors of APR systems and representative fault localization metrics (i.e., Top-1 and MFR). Note that we excluded APR systems targeting multi-edit patches. In this figure, "TotalPatch" represents the number of all executed compilable patches generated from each APR tool, "MethodByTotal" represents the number of unique methods modified by all executed patches, "PlausiblePatch" represents the number of plausible patches generated by each tool, and "MethodsByPlausible" represents the number of unique methods covered by the plausible patches. Within each sub-figure, each data point represents one APR system, and we perform Pearson Correlation Coefficient analysis [60] at a 0.05 significance level. From this figure, we observe that APR systems tend to perform significantly better for unified debugging when executing more patches for more methods and/or producing more plausible patches for more methods. The finding is statistically significant for all sub-figures at the significance level of 0.05. Although the finding is surprisingly uniform, this makes intuitive sense since APR systems patching more code elements tend to accumulate more information for debugging. This finding suggests future APR systems to explore more diverse patches for more powerful unified debugging, and also calls for research for faster patch execution (otherwise APR systems cannot afford massive patch executions).

> **Finding 3:** APR systems executing more patches across code elements and/or producing more plausible patches tend to perform better for unified debugging, calling for future research on fast & exhaustive patch exploration.

4.1.4 Experimental Results on All Subjects

In this section, we investigate the performance of unified debugging on two additional subjects, i.e., Closure and Mockito. Note that this section's material reflects material *not present* in the paper's original publication. As only a minor subset of considered APR tools successfully run these subjects (PraPR, kPar, TBar, AVATAR, and FixMiner), this subsection assesses the effectiveness of unified debugging where a limited selection of APR tools are applicable. Thus the investigation into these subjects warrants its own subsection.

We now present the results of UniDebug on Closure and Mockito in Figure 8. From this figure, we observe mixed results compared to core subject effectiveness; some tools improve SBFL effectiveness while others degrade SBFL effectiveness. For example, PraPR localizes 15 more faults within Top-1 than SBFL while TBar localizes 2 less faults within Top-1. We further inspect cases where SBFL effectiveness worsens and find that some tools generate more higher-quality patches for non-buggy elements than buggy code elements on Closure and Mockito, resulting in unstable performance on these two subjects. For example, in Mockito-2, the suspiciousness value of the buggy method `org.mockito.internal.util.Timer:<init>(J)V` is 0.42 as computed by SBFL Ochiai which is greater than any other method. However, when TBar attempts to fix this bug, it generates CleanFix patches for one non-buggy method `org.mockito.Mockito:after(J)Lorg/mockito/verification/VerificationAfterDelay;`, but **does not generate** any patches for the buggy method. Therefore, based on such misleading patch execution information,

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2021.3125203, IEEE Transactions on Software Engineering


```
/** File = ArrayRealVector.java */
public double getLInfNorm() {
    double max = 0;
    for (double a : data) {
        max+=Math.max(max,Math.abs(a));
    }
-   return max;
+   return data.length;
}
```

(a) Modifications for ArrayRealVector.java

```
/** File = OpenMapRealVector.java */
public double getLInfNorm() {
    double max = 0;
    Iterator iter=entries.iterator();
    while (iter.hasNext()) {
        iter.advance();
        max += iter.value();
    }
-   return max;
+   return virtualSize;
}
```

(b) Modifications for OpenMapRealVector.java

```
/** File = AbstractRealVector.java */
public double getL1Norm() {
    double norm = 0;
    Iterator<Entry> it = sparseIterator();
    Entry e;
+   while (it.hasNext()&&(e=it.next())!=null){
+       norm += Math.abs(e.getValue());
+   }
    while (it.hasNext()&&(e=it.next())!=null){

        norm += Math.abs(e.getValue());
    }
    return norm;
}
```

(c) Modifications for AbstractRealVector.java

Fig. 6: An incorrect Arja CleanFix patch (with three modified methods) for Math-77

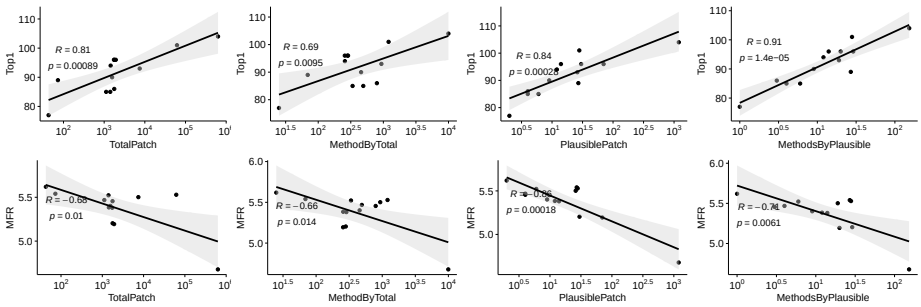

Fig. 7: Correlation analysis

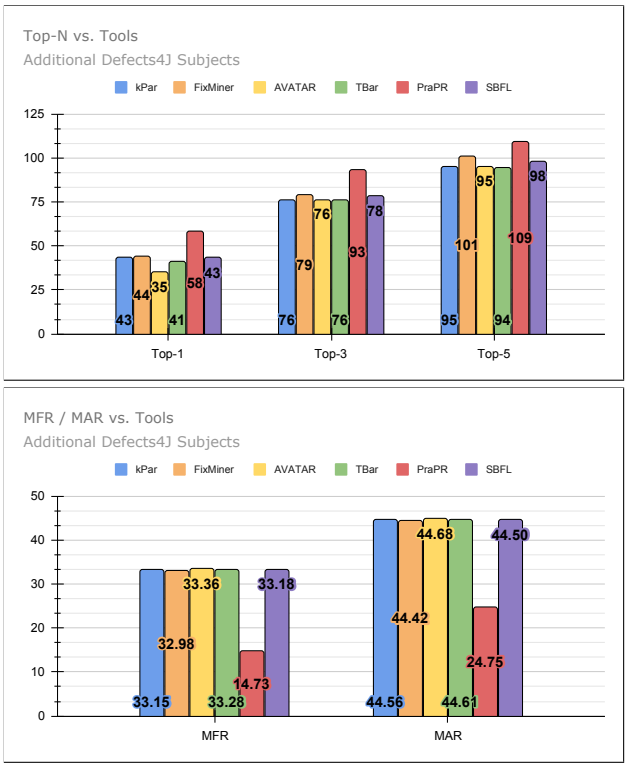

Fig. 8: UniDebug results with all studied APR systems on additional Defects4J subjects

unified debugging ranks this non-buggy method higher than the buggy method.

The results of the effectiveness of UniDebug on all six projects are provided in Figure 9. Note that we are able to present the results of all APR tools on all subjects since unified debugging simply degrades to SBFL if APR tools cannot produce any patch for a specific bug. Rather similar to the results on four core subjects shown in Section 4.1.1, we can observe that 15 out of 16 APR tools (excluding Arja) outperform the traditional SBFL for UniDebug according to Figure 9. For Top-1, Arja is the sole tool underperforming SBFL but it still localizes 117 bugs in Top-1, only 2 less than SBFL. Similarly, ACS slightly underperforms SBFL in Top-3 (222 vs 223) while no tool underperforms SBFL in Top-5. These results on the six Defects4J subjects leads us to the following finding.

> **Finding 4:** The experimental results on all 395 bugs from Defects4J 1.2.0 are overall consistent with the experimental results on 224 bugs from the four core Defects4J subjects.

Note that we use all the 395 bugs from Defects4J in the experiments of all our following RQs unless otherwise specified.

4.2 RQ2 - Impacts of Non-Modify Code Elements on Unified Debugging

As discussed in Section 3.4.2, we add one new patch category, Non-Modify, which represents suspicious methods that are not associated with any repair patch for a given tool. The first research question has demonstrated the effectiveness of default unified debugging setting on 16 APR systems by treating Non-Modify equivalently as the fourth patch category, NegFix. In this research question, we further evaluate the unified debugging effectiveness when casting the Non-Modify category into each of the four different ProFL categories. Figure 10 represents the Top-1 and MFR results of all 16 APR systems on all the 395 studied bugs from Defects4J with such four settings represented with lines in different colors. From the figure we observe that for most systems, casting Non-Modify code elements into the NegFix category leads to better Top-1 and MFR than casting them into any other category. For example, Simfix can localize 144 bugs within Top-1 when casting Non-Modify into NegFix category, and can only localize 139/116/98 bugs within Top-1 when casting Non-Modify into NoneFix/NoisyFix/Clean-

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2021.3125203, IEEE Transactions on Software Engineering

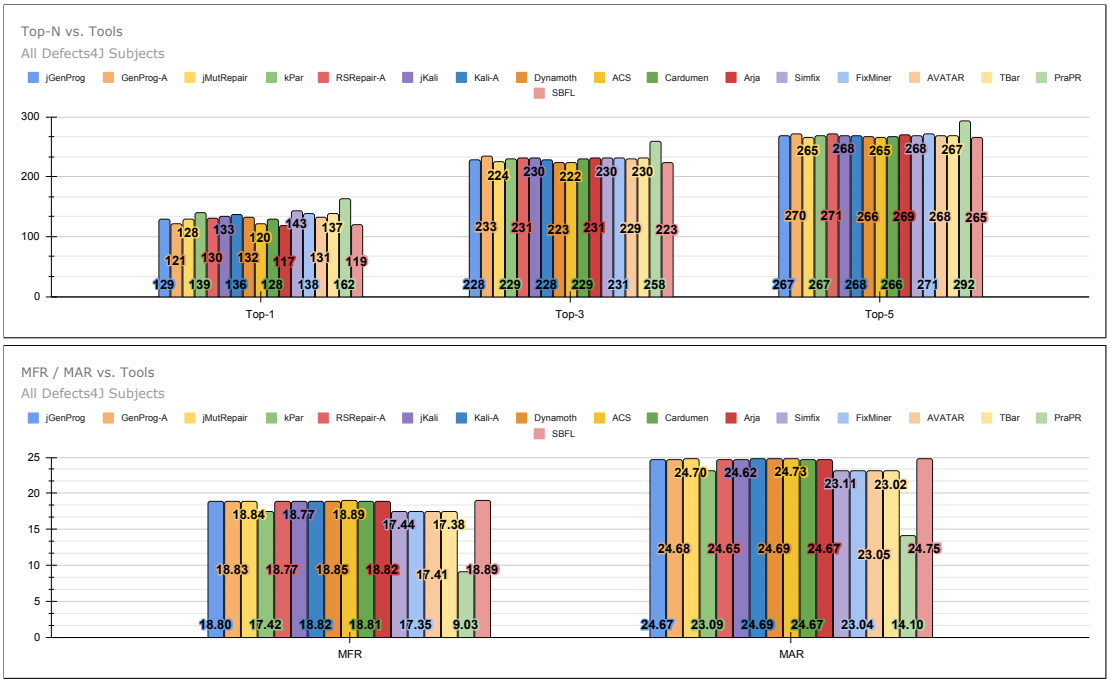

Fig. 9: UniDebug results with all studied APR systems on all Defects4J subjects

Fix category. Also, in terms of MFR, Kali-A achieves 18.82 with NegFix, which is also increasingly better than Kali-A with the other three categories (19.88/22.74/24.27). We find the reason is to be that NegFix patches are significantly more prevalent for most code elements (including Non-Modify ones), while other patch categories can be harder to generate.

> **Finding 5:** Non-Modify code elements (i.e., elements with no patches) can be treated in the same way as elements with only NegFix patches (i.e., the patches that cannot fix any failing test but can cause originally passing tests to fail) for precise unified debugging.

As a consequence from this finding, we assign any unmodified element the NegFix category for all future RQs unless otherwise specified.

4.3 RQ3 - How Does Unified Debugging Correlate with APR Effectiveness?

APR systems all aim to produce correct patches for as many bugs as possible. However, this is a rather challenging goal, and even state-of-the-art APR tools cannot even fix 20% of the studied bugs [45]. Therefore, in this research question, we empirically study whether unified debugging is also limited by APR effectiveness (i.e., in producing correct patches). The three sub-figures in Figure 11 show the representative Top-1 metric for SBFL and UniDebug using each APR system on (1) buggy versions where the corresponding APR system has correct patches, (2) buggy versions with incorrect but plausible patches, and (3) buggy versions without even plausible patches, respectively.

Please note that we omit APR systems that did not present detailed correct patch IDs in their original publications from these figures. Also, this research question only considers the four core subjects studied from Defects4J because most of the studied APR tools cannot be applied to the remaining Mockito and Closure subjects and thus cannot generate any patch for our analysis.

From the figures, we observe that different APR systems perform differently in all three different bug sets, and almost all APR systems contribute to unified debugging to outperform SBFL. One potential reason is that as long as a patch can pass some originally failing test(s), its patch location may be closely related to the actual buggy location, since otherwise it cannot mute the bug impact to pass failing tests. In this way, patches do not need to be correct or even plausible to contribute to unified debugging. Furthermore, even the patches that only make originally passing tests fail can help eliminate the potentially correct/benign locations to also boost unified debugging. This further demonstrates the general applicability and promising future for unified debugging.

> **Finding 6:** Unified debugging effectiveness does not rely on the availability of correct or even plausible patches from APR. Similar as when conducting manual program repair, APR patch execution results from even incorrect/implausible patches can still reveal actual buggy locations (when they pass some failing test(s)) or eliminate correct locations (even when they only fail on originally passing tests).

4.4 RQ4 - Advanced Unified Debugging via APR System Statistics

To combine the strengths of different APR systems, one naive way is to simply combine the patches of different APR systems for unified debugging, i.e., the UniDebug+ technique talked about in Section 2.4. Again, the final cat-

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2021.3125203, IEEE Transactions on Software Engineering

Fig. 10: Impact of casting Non-Modify code elements into different categories

EID	SBFL	Tool1	Tool2	Tool3	UniDebug+	UniDebug++
e1	0.8	CleanFix	CleanFix	CleanFix	CleanFix	CleanFix(3)
e2	0.8	CleanFix	NegFix	CleanFix	CleanFix	CleanFix(2)
e3	0.8	CleanFix	NoneFix	NoneFix	CleanFix	CleanFix(1)

TABLE 6: Example of UniDebug++

Tech Name	Top-1	Top-3	Top-5	MFR	MAR
SBFL	119	223	268	17.55	23.16
PraPR / ProFL	162	258	292	9.03	14.10
UniDebug+$_{all}$	154	253	287	8.84	13.84
UniDebug++$_{all}$	179	261	290	8.64	13.67
UniDebug+	169	261	292	8.64	13.68
UniDebug++	**185**	**265**	**294**	**8.52**	**13.59**

TABLE 7: Effectiveness of unified debugging variants on all Defects4J subjects

egory information for a code element can be determined by the code element's best category information from all patches across multiple APR systems. In this section, we further propose a more advanced technique, UniDebug++, which further distinguishes code elements with the same suspiciousness values in the same category. More specifically, after assigning the patch group category (for patches generated by all combined APR systems) to a code element in the category aggregation step (shown in Figure 1), we further count the total number of APR systems that generate patches in the same category as this code element. The intuition is that if more APR systems assign the best category information to a code element, this element should have higher priority in the ranked list compared to its tied peers. Table 6 shows a simple example to illustrate UniDebug++. In this example, elements `e1`, `e2`, and `e3` have the same SBFL 0.8 suspiciousness value and are all in the CleanFix category according to UniDebug+; therefore they cannot be distinguished when using UniDebug+. In contrast, UniDebug++ further considers the number of APR systems producing CleanFix patches for each element. For example, `e1` has CleanFix patches when using all three APR systems and should be ranked higher than other elements. In this way, we leverage more precise APR information for more powerful unified debugging.

Table 7 shows the results of original SBFL, ProFL, UniDebug+, and UniDebug++ on all the studied Defects4J subjects in terms of Top-1, Top-3, Top-5, MFR, and MAR. Note that as discussed in Section 4.1.2, APR systems specifically targeting multi-edit patches can introduce extra noise for unified debugging and have been excluded for UniDebug+ and UniDebug++. Meanwhile, we also include, as references, their variants which consider all studied APR systems, denoted as UniDebug+$_{all}$ and UniDebug++$_{all}$ in the table. From the results, we have the following observations. First, UniDebug+$_{all}$ and UniDebug++$_{all}$ perform worse than UniDebug+ and UniDebug++, respectively. In fact, UniDebug+$_{all}$ even performs worse than ProFL which only uses the PraPR APR system. This finding further confirms our earlier qualitative analysis and *Finding 2* that APR systems targeting multi-edit patches are ill-suited for unified debugging. Second, both UniDebug+ and UniDebug++ can significantly outperform SBFL and ProFL in all metrics. For example, UniDebug+ localizes 169 bugs within Top-1, i.e., 50/7 more than SBFL/ProFL. Third, UniDebug++

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2021.3125203, IEEE Transactions on Software Engineering

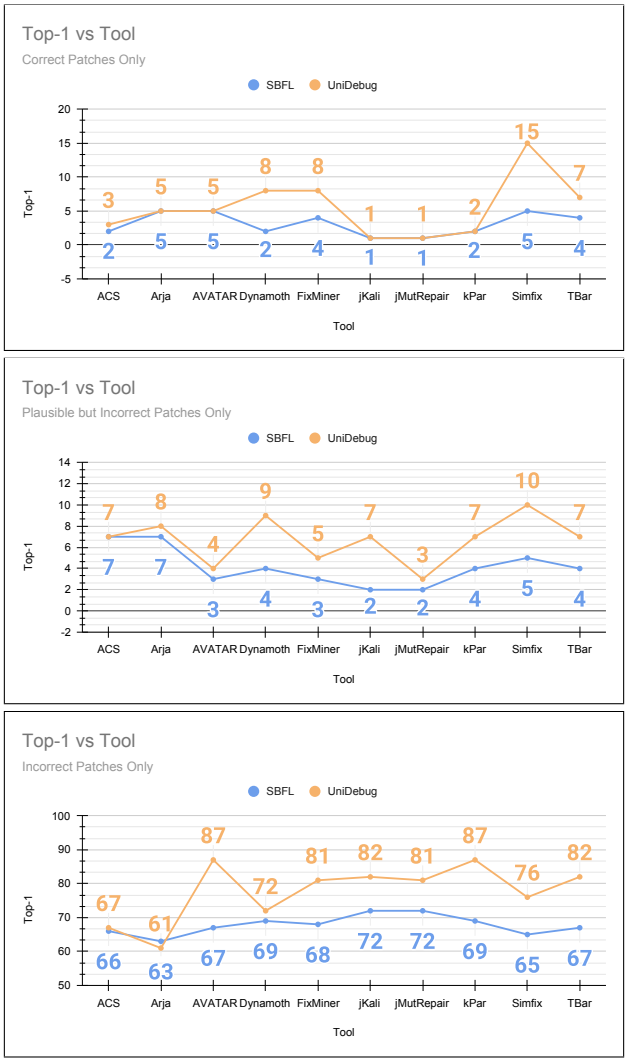

Fig. 11: Unified debugging on buggy versions with (1) correct, (2) incorrect but plausible, and (3) implausible patches

achieves the best result (even comparing against our own UniDebug+), localizing 185 bugs within Top-1, i.e., 66/23 more than state-of-the-art SBFL/ProFL.

4.4.1 *Incremental Unified Debugging*

Shown in Section 4.1, APR systems with more plausible patches tend to perform better in unified debugging. Therefore, we further study the impact of having different subsets of APR systems for UniDebug+ and UniDebug++ on all the 395 studied bugs. To that end, we rank all APR systems in descending order of the number of *additional* bugs that each APR system can come up with plausible patches. In this way, we can observe the effectiveness trend of UniDebug+ and UniDebug++ with more and more APR systems. Figure 12 presents Top-1 results when including more and more APR systems under UniDebug+ and UniDebug++. Note that each bar is cumulative, so bar Simfix represents adding Simfix to the set of tools (PraPR, ACS, Simfix). From the figure, we observe several interesting findings. First, both UniDebug+ and UniDebug++ overall have increasing effectiveness when including more and more APR systems. Second, we also observe that both techniques tend to saturate when new systems cannot provide additional plausible patches. This actually indicates that a small subset of the studied APR systems (e.g., 6 of them) can be combined to achieve same effectiveness as the whole set (later discussed in Section 4.7.3). Third, UniDebug++ has much better effectiveness compared to UniDebug+, which simply combines all patches from different APR systems together. The reason is that most APR systems are helpful for fault localization and the code elements ranked high by multiple APR systems are indeed more likely to be buggy.

Please note that running multiple APR systems for UniDebug+ or UniDebug++ can be costly. Although the efficiency issue of these program repair systems is out of scope for this paper, our above findings demonstrate that a small subset (e.g., 6) of repair systems can already be sufficient for effective unified debugging, meaning some systems are not required to execute, substantially saving the repair time (e.g., the 6 program repair systems together take around 8 hours in total for each studied buggy version on average using one thread). Also, as nowadays companies have an abundance of computation resources (e.g., multi-core servers, clusters, and clouds), such APR systems can be easily executed in parallel to take full advantage of modern computation resources and further increase the potential for improved unified debugging with minimal delay; this not only (1) improves the probability for the buggy project to be directly automatically fixed via multiple APR systems, but also (2) further boosts fault localization for manual repair even when none of the APR systems can directly fix the bug. Lastly, more and more techniques have recently also been proposed to further speed up individual program repair techniques, e.g., Chen et al. [61] have recently proposed a general on-the-fly patch-validation framework to substantially speed up existing APR systems. They can also be easily applied to speed up the unified debugging process.

> **Finding 7:** Our new unified debugging technique considering common behaviors between multiple APR systems, UniDebug++, can localize 185 bugs within Top-1 on all studied Defects4J subjects, i.e., 66/23 more than state-of-the-art SBFL/ProFL.

4.5 RQ5 - Impact of Different SBFL Formulae

In our previous research questions, we execute unified debugging using one of the state-of-the-art SBFL formulae (i.e., Ochiai). Besides Ochiai, unified debugging can also be applicable alongside any other SBFL formula, but it is unknown (1) if the effectiveness of unified debugging is reliant on the underlying SBFL formula and (2) the volatility of unified debugging effectiveness with respect to different SBFL formulae. Therefore, in this research question, we implement all 33 SBFL formulae considered in prior work [36], [44], [49] and investigate their impact on the effectiveness of unified debugging. Note that, similar to how we use Ochiai with aggregation, we employ the SBFL aggregation strategy for all SBFL formulae as described in Section 3.4.1.

Given that UniDebug++ is the most effective technique of unified debugging, we evaluate the effectiveness of UniDebug++ based on different SBFL formulae and present

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2021.3125203, IEEE Transactions on Software Engineering

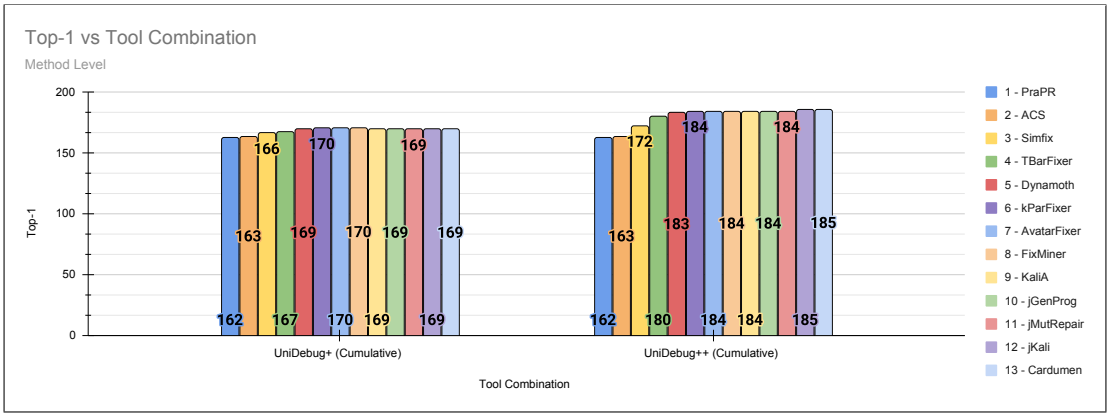

Fig. 12: UniDebug+ and UniDebug++ with increasing number of APR systems

the results in Figure 13 and Figure 14. In the figure, the x axis represents all 33 spectrum-based fault localization formulae while the y axis represents the metrics in terms of Top-1, Top-3, Top-5, MFR, and MAR respectively. In particular, the blue and orange columns refer to the original SBFL formulae and UniDebug++ respectively. Based on the figure, we observe that UniDebug++ achieves a consistent improvement over all SBFL formulae. In particular, for Top-1, the improvement of UniDebug++ on all original SBFL formulae ranges from 61 (e.g., ER1a) to 129 (e.g., Wong); for Top-3, the improvement ranges from 40 (e.g., Ochiai2) to 173 (e.g., ER5c); and for Top-5, the improvement ranges from 32 (e.g., Kulczynski2) to 178 (e.g., Anderberg). Besides the Top-N metrics, MFR and MAR are both significantly improved as well. For example, compared to all the original SBFL formulae, MFR is improved by 61% on average and MAR is improved by 53% on average.

In addition, we also find that even for those very ineffective SBFL formulae with extremely poor performance in Top-1 (e.g., Overlap with only 13 Top-1 and Wong with only 14 Top-1), UniDebug++ can still boost them significantly (e.g., 137 Top-1 and 143 Top-1 respectively). These results imply that the performance of unified debugging is not restricted by the initial fault localization techniques adopted in the beginning phase of APR, and the patch execution information from APR tools is helpful for fault localization refinement in every circumstance.

> **Finding 8:** Even the poorest SBFL formulae massively benefit from the unified debugging technique. UniDebug++'s effectiveness is largely independent of underlying SBFL formulae and consistently improves upon all considered SBFL formulae, e.g., 61% and 53% average percent improvement on MFR and MAR respectively.

4.6 RQ6 - Boosting Learning-Based Fault Localization

4.6.1 Effectiveness

In this section, we further investigate how UniDebug++ may boost state-of-the-art learning-based fault localization techniques, including unsupervised-learning-based (i.e., PRFL [11] and PRFL_{MA} [62]) and supervised-learning-based (i.e., DeepFL [12]) fault localization. First, the suspiciousness values computed by unsupervised-learning-based techniques can be directly used as the initial values in the first step of unified debugging. Therefore, we directly apply the suspiciousness values computed by PRFL and PRFL_{MA} into UniDebug++, denoting them as $\text{PRFL}_{\text{UniDebug++}}$ and $\text{PRFL}_{\text{MA UniDebug++}}$, respectively. Second, UniDebug++ with 33 SBFL formulae can be served as one additional dynamic feature dimension for supervised-learning-based techniques. Therefore, we extend DeepFL by injecting 33 new UniDebug++ features between the original two dynamic feature dimensions, i.e., mutation features and spectrum features, denoting it as $\text{DeepFL}_{\text{UniDebug++}}$. Table 8 shows the experimental results of $\text{PRFL}_{\text{UniDebug++}}$, $\text{PRFL}_{\text{MA UniDebug++}}$ and $\text{DeepFL}_{\text{UniDebug++}}$. Due to the randomness of DNN [12], we run DeepFL and $\text{DeepFL}_{\text{UniDebug++}}$ 10 times to calculate their average results. From the table, we have the following observations. First, the basic UniDebug++ using default Ochiai formula already significantly outperforms state-of-the-art unsupervised-learning-based fault localization. For example, as the best unsupervised technique, PRFL_{MA} can localize 139 bugs within Top-1 while the basic UniDebug++ localizes 185 bugs within Top-1. In addition, UniDebug++ can significantly boost both unsupervised-learning-based and supervised-learning-based fault localization. For example, $\text{PRFL}_{\text{MA UniDebug++}}$ localizes 199 bugs within Top-1. To the best of our knowledge, this is the best unsupervised-learning-based fault localization results on Defects4J to date. $\text{PRFL}_{\text{MA UniDebug++}}$ can even significantly outperform many state-of-the-art supervised-learning-based techniques, e.g., TraPT [36], FLUCCS [44], and CombineFL [63] report localizing only 156, 160, and 168 bugs respectively from the same dataset within Top-1. We also observe that UniDebug++ can further boost state-of-the-art supervised-learning-based technique DeepFL (e.g., localizing 213.80 bugs within Top-1, outperforming DeepFL with 206.30 bugs).

4.6.2 Overhead Analysis

The trade-off for boosting the effectiveness of learning-based techniques often comes at a cost to performance. Sometimes this compromise in performance makes the boost infeasible in practice. Thus, we will also briefly discuss the overhead of the considered learning-based approaches when additionally considering the unified debugging features (while the cost of running the APR tools for unified

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2021.3125203, IEEE Transactions on Software Engineering

Fig. 13: Top-N comparison of SBFL and UniDebug++ over all formulae

Tech Name	Top-1	Top-3	Top-5	MFR	MAR
UniDebug++	185	265	294	8.52	13.59
PRFL	114	202	246	23.37	27.85
$PRFL_{UniDebug++}$	187	257	293	9.92	14.71
$PRFL_{MA}$	139	247	275	17.81	22.71
$PRFL_{MA\,UniDebug++}$	199	273	301	8.53	13.54
DeepFL	206.3	**281.9**	**310.6**	**6.67**	**8.28**
$DeepFL_{UniDebug++}$	**213.8**	288.1	316.3	6.90	8.72

TABLE 8: Effectiveness of unified debugging on boosting learning-based fault localization

debugging has already been discussed in Section 4.4). Note that we only present the cost analysis for boosting the state-of-the-art DeepFL work as the overheads incurred by unified debugging features are similar for all studied learning-based techniques.

We have collected detailed subject execution timings for $DeepFL_{UniDebug++}$ and DeepFL to investigate subject-by-subject time costs, shown in Table 9. Column "Stage" represents the training or testing phase of the approach. Column "DeepFL" represents the time cost for DeepFL. Column "$DeepFL_{UniDebug++}$" represents the time cost for $DeepFL_{UniDebug++}$ while the smaller parenthesised number represents the percent increase from DeepFL. Note that since the testing cost is always far less than 1s and the training cost dominates the overall execution cost, we only discuss the training execution costs.

Each time cost represents the **average cost** to run the leave-one-out training process for each subject (e.g., DeepFL takes 2.56s on average to perform training for each Chart subject). We see that DeepFL takes ∼39 seconds on average while $DeepFL_{UniDebug++}$ takes ∼41 seconds to execute against Defects4J. These detailed time costs show that

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2021.3125203, IEEE Transactions on Software Engineering

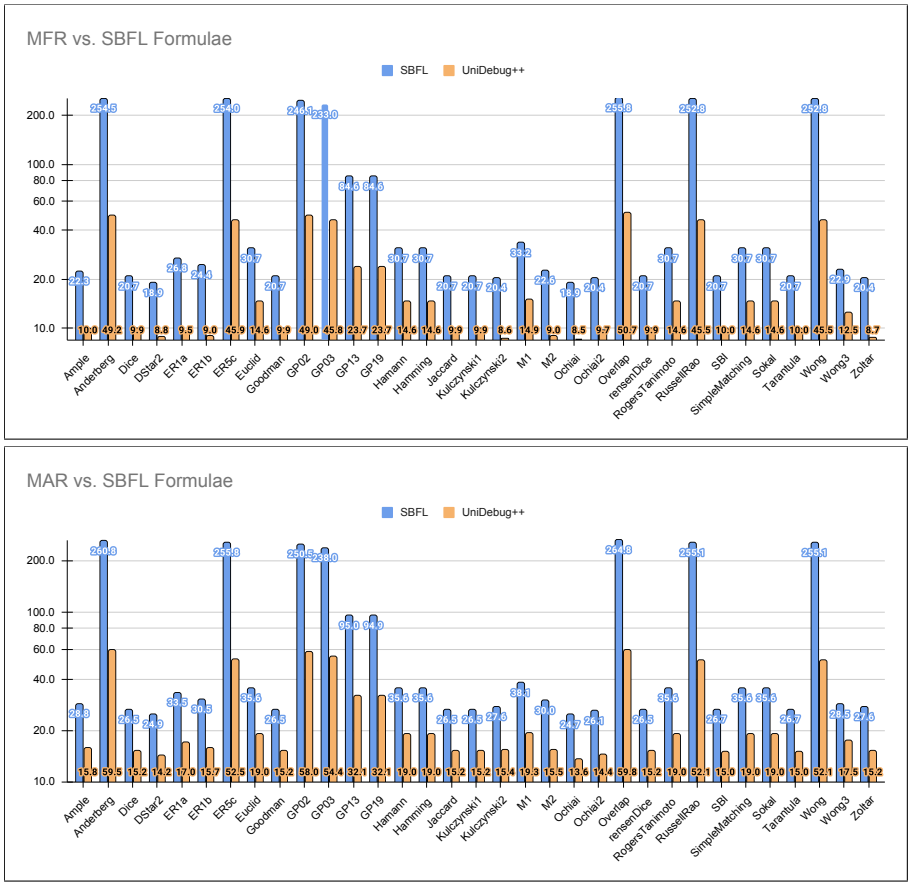

Fig. 14: MFR and MAR comparison of SBFL and UniDebug++ over all formulae

$\text{DeepFL}_{\text{UniDebug++}}$ consistently has identical execution times as DeepFL on all subjects. In fact, the worst observed difference is an increased execution cost of 6.17% coming from the Math subject. More so, even among the largest subject in Defects4J, Closure, we see an increased training execution cost of only **4.98%**! Based on these observations, it is evident that utilizing UniDebug++ features within various learning-based technique improves the effectiveness of the technique while having **minimal impact** on the technique's performance.

> **Finding 9:** UniDebug++ further boosts both state-of-the-art unsupervised-learning-based and supervised-learning-based fault localization techniques. Utilizing UniDebug++ to boost state-of-the-art learning-based fault localization techniques also incurs negligible overhead for both training and testing.

Subject	Stage	Time (sec) (Percent Increase)	
		DeepFL	$\text{DeepFL}_{\text{UniDebug++}}$
Chart	Train	2.56	2.60 (1.56%)
	Test	0.069	0.091 (31.88%)
Closure	Train	109.46	114.91 (4.98%)
	Test	0.076	0.086 (13.16%)
Lang	Train	0.58	0.60 (3.45%)
	Test	0.064	0.070 (9.38%)
Math	Train	4.70	4.99 (6.17%)
	Test	0.069	0.073 (5.80%)
Mockito	Train	6.36	6.65 (4.56%)
	Test	0.069	0.074 (7.25%)
Time	Train	3.07	3.24 (5.54%)
	Test	0.062	0.073 (17.74%)
Average	**Train**	**39.21**	**41.16 (4.97%)**
	Test	**0.070**	**0.078 (11.43%)**

TABLE 9: Detailed time costs for DeepFL and $\text{DeepFL}_{\text{UniDebug++}}$

4.7 RQ7 - Unified Debugging on Statement-Level Fault Localization

Previous and current work demonstrate unified debugging's usefulness towards method-level fault localization for *manual* program repair, but assessing the exact faulty statement(s) can prove substantially more useful for *automated* program repair. In this RQ, therefore, we apply unified debugging at the statement level for the first time. Each statement is identified by its enclosing class, method signature, and line number as reported by each tool per repair patch. Then we calculate each statement's best category and rank all statements using the technique's ranking features previously discussed.

4.7.1 Individual Tools

Figure 15 describes the results of unified debugging for each of the 16 APR tools executed on the core Defects4J subjects. Since not every tool executes Closure or Mockito, we exclude these two subjects from Figure 15 to show a clear trend. Meanwhile, Table 10 also presents the unified debugging results per tool on the entire Defects4J benchmark (including Closure and Mockito) to demonstrate the

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2021.3125203, IEEE Transactions on Software Engineering

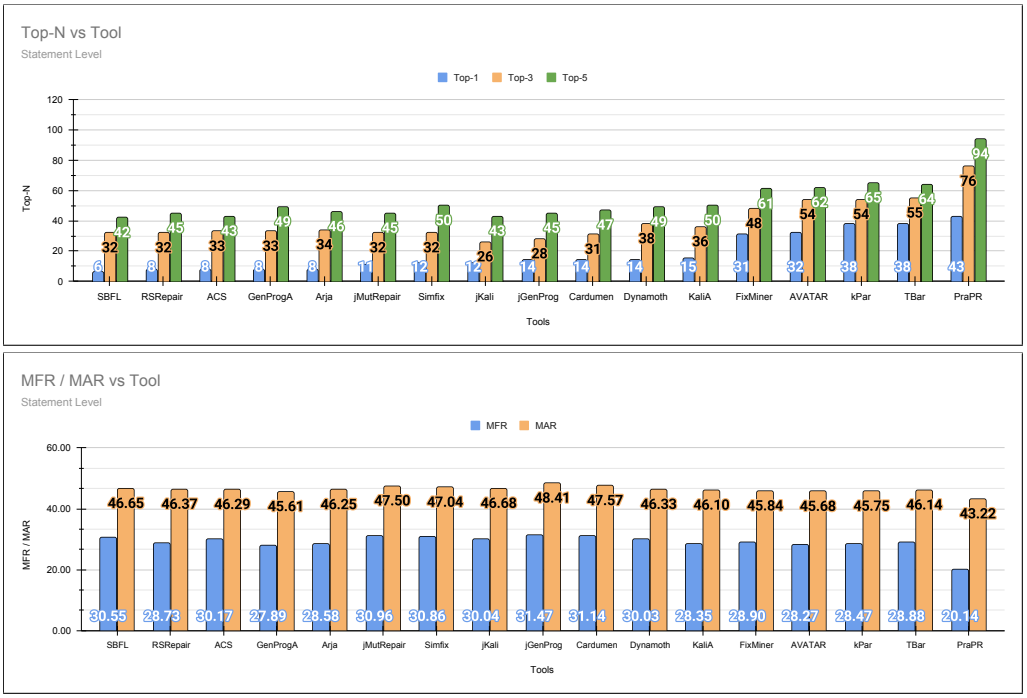

Fig. 15: Statement-level UniDebug effectiveness for individual tools on the four core Defects4J subjects

Tool	Top-1	Top-3	Top-5	MFR	MAR
SBFL	15	54	72	114.96	133.06
ACS	16	55	73	114.73	132.83
Arja	17	56	75	113.64	132.69
RSRepair-A	17	55	78	113.50	132.72
GenProg-A	18	59	82	113.24	132.50
Cardumen	19	56	77	115.14	133.35
jKali	19	56	78	114.89	133.12
jMutRepair	19	55	75	115.00	133.37
jGenProg	20	54	78	115.19	133.67
Kali-A	20	62	83	113.12	132.35
Simfix	20	55	84	114.72	133.00
Dynamoth	23	60	79	114.63	132.85
FixMiner	42	74	92	113.78	132.55
AVATAR	58	85	102	113.40	132.47
TBar	58	80	97	114.03	132.94
kPar	61	83	99	114.01	133.13
PraPR	**64**	**116**	**139**	**70.95**	**115.36**

TABLE 10: Statement-level UniDebug tool effectiveness on all Defects4J subjects

applicability of statement-level unified debugging where some tools cannot execute on all systems.

From data described in Table 10 and Figure 15, we can observe that unified debugging outperforms SBFL for most tools across most metrics. Particularly, with respect to the most crucial metric, Top-1, every tool at least performs on par with SBFL and most tools actually outperform SBFL. Looking at Figure 15, PraPR is the most effective by having a Top-1 of 43 while Arja, RSRepair-A, ACS, and GenProg-A have the worst effectiveness with a Top-1 of 8. All tools still outperform SBFL's 6 Top-1, showing our technique is still useful even at the statement level. This leads us to our first statement-level finding.

> **Finding 10:** Unified debugging is still applicable and useful at the statement-level granularity. At the statement level, unified debugging still outperforms SBFL across most studied APR tools.

With respect to PraPR, we again see the tool performing significantly better than any other individual tool across all metrics at the statement level as seen by Figure 15 and Table 10. We believe this, again, to be related due to a combination of the mass number of patches produced by PraPR and the diverse range of bug classes repairable by the tool. Focusing on the set of multi-edit tools (Arja, GenProg-A, RSRepair-A), we also see poor performance relative to other tools; they barely outperform SBFL and still have among the worst metrics compared to all tools.

> **Finding 11:** PraPR performs the best out of all analyzed tools across all metrics at statement-level granularity. Likewise, multi-edit tools Arja, GenProg-A, and RSRepair-A again perform the worst in Top-1.

4.7.2 *Qualitative Analysis*

We now briefly give a qualitative analysis on some examples at the statement-level granularity, one from AVATAR/Math-69 and the other from TBar/Math-46.

AVATAR: Figure 16 shows (1) the correct developer patch and (2) one of several NoisyFix patches produced by AVATAR on Math-69, while Table 11 describes the subject's detailed localization results (column descriptions the same as in Table 5 with the gray row denoting the buggy statement). This buggy statement, **`e13`**, is ranked 13th according to its (largely tied) SBFL suspiciousness value. Upon execution of Math-69, AVATAR finds multiple NoisyFix patches (Figure 16) for `e13`, a NoneFix patch for `e3`, and NegFix

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2021.3125203, IEEE Transactions on Software Engineering


```
/** File = FastDateFormat.java */
/** @@ -169,3 +169,3 @@ public RealMatrix getCorrelationPValues() */
    double r = correlationMatrix.getEntry(i, j);
    double t = Math.abs(r * Math.sqrt((nObs - 2)/(1 - r * r)));
-   out[i][j] = 2 * (1 - tDistribution.cumulativeProbability(t));
+   out[i][j] = 2 * tDistribution.cumulativeProbability(-t);
```

(a) Developer patch

```
/** File = FastDateFormat.java */
/** @@ -169,3 +169,3 @@ public RealMatrix getCorrelationPValues() */
    double r = correlationMatrix.getEntry(i, j);
    double t = Math.abs(r * Math.sqrt((nObs - 2)/(1 - r * r)));
-   out[i][j] = 2 * (1 - tDistribution.cumulativeProbability(t));
+   out[i][j] = 2 * (1 - tDistribution.cumulativeProbability(nVars));
```

(b) AVATAR NoisyFix patch

Fig. 16: Math-69 Patches

```
/** File = Complex.java */
/** @@ -258,3 +258,3 @@ public Complex divide(Complex divisor) */
if (divisor.isZero) {
    // return isZero ? NaN : INF; // See MATH-657
-   return isZero ? NaN : INF;
+   return NaN;
}
...
/** @@ -295,3 +295,3 @@ public Complex divide(double divisor) */
if (divisor == 0d) {
    // return isZero ? NaN : INF; // See MATH-657
-   return isZero ? NaN : INF;
+   return NaN;
}
```

(a) Developer patch

```
/** File = Complex.java */
/** @@ -258,3 +258,3 @@ public Complex divide(Complex divisor) */
if (divisor.isZero) {
    // return isZero ? NaN : INF; // See MATH-657
-   return isZero ? NaN : INF;
+   return isZero ? NaN : Complex.NaN;
}
```

(b) TBar NoisyFix patch

Fig. 17: Math-46 Patches

EID	Suspicious Line	SBFL		AVATAR	
		Susp.	Rank	Category	Rank
e1	org.apache.[...].PearsonsCorrelation#**68**	1.00	2	NegFix	4
e2	org.apache.[...].PearsonsCorrelation#**69**	1.00	2	NegFix	4
e3	org.apache.[...].PearsonsCorrelation#**175**	0.53	13	NoneFix	2
e4	org.apache.[...].PearsonsCorrelation#**161**	0.53	13	NegFix	13
e5	org.apache.[...].PearsonsCorrelation#**162**	0.53	13	NegFix	13
e6	org.apache.[...].PearsonsCorrelation#**163**	0.53	13	NegFix	13
e7	org.apache.[...].PearsonsCorrelation#**164**	0.53	13	NegFix	13
e8	org.apache.[...].PearsonsCorrelation#**165**	0.53	13	NegFix	13
e9	org.apache.[...].PearsonsCorrelation#**166**	0.53	13	NegFix	13
e10	org.apache.[...].PearsonsCorrelation#**167**	0.53	13	NegFix	13
e11	org.apache.[...].PearsonsCorrelation#**169**	0.53	13	NegFix	13
e12	org.apache.[...].PearsonsCorrelation#**170**	0.53	13	NegFix	13
e13	org.apache.[...].PearsonsCorrelation#**171**	0.53	13	NoisyFix	1

TABLE 11: AVATAR localization details for Math-69

EID	Suspicious Line	SBFL		TBar	
		Susp.	Rank	Category	Rank
e1	org.apache.[...].Complex#**260**	0.71	2	NoisyFix	1
e2	org.apache.[...].Complex#**1183**	0.71	2	NoneFix	2
e3	org.apache.[...].Complex#**587**	0.41	3	NegFix	3

TABLE 12: TBar localization details for Math-46

patches for all other statements. When it is time to rerank all elements, `e13` is the only statement in the NoisyFix category, allowing the statement to be ranked 1st above all other statements. Thus, in spite of the limited effectiveness of APR tools at the statement level, we still see erroneous statements nearly exclusively receive high-quality patches which in turn yields higher localization results, further strengthening the potential of unified debugging, especially at the statement level.

TBar: Figure 17 shows (1) the correct developer patch for Math-46 and (2) an incorrect NoisyFix patch produced by TBar on Math-46 while Table 12 describes the subject's detailed localization results (column descriptions the same as in Table 5 with the gray row denoting the buggy statement). We see that, from only SBFL, one of the buggy elements, **`e1`**, is tied with another element (i.e. they have the same suspiciousness value) and are thus both ranked 2nd (according to the worst-case ranking). From Figure 17, we see that the correct patch modifies two return statements. Since this is at the statement-level granularity, TBar is unable to modify **both** buggy lines and instead only modifies one of the buggy lines resulting in a NoisyFix patch. After acquiring the NoisyFix repair information from TBar (Figure 17), one of the buggy lines, `e1`, is now categorized as NoisyFix which is higher than all other statements in the program. Upon TBar's completion of the repair process, only `e2` receives a NoneFix patch and all other statements receive NegFix patches. Thus according to the patch category hierarchy (Section 2.2), `e1` can still be differentiated from `e2`, allowing the approach to correctly rerank `e1` as the most buggy statement in the program. In summary, at the statement granularity, we observe the same phenomenon as the method granularity that incorrect patches can still yield more precise fault localization within unified debugging.

4.7.3 Unified Debugging Variants

Table 13 shows the Top-N metrics for all considered APR tools. Rows UniDebug+ and UniDebug++ represent UniDebug+ and UniDebug++ using all 13 single-edit tools. Rows UniDebug+$_{refined}$ and UniDebug++$_{refined}$ respectively represent UniDebug+ and UniDebug++ using the refined selection of tools, namely PraPR, kPar, and AVATAR.

Similar to method-level fault localization, we see substantial uniform improvements from SBFL across all metrics for statement-level fault localization on all unified debugging techniques. Further results are shown in Figure 18 which describe the Top-N, MFR, and MAR across unified debugging techniques. Despite these improvements, we observe that using all single-edit tools from method-level unified debugging failed to outperform using just ProFL in terms of Top-1, results detailed in Table 13 via UniDebug+ and UniDebug++. Thus while unified debugging can still improve upon SBFL, the usage of all single-edit APR tools actually degrades unified debugging results.

> **Finding 12:** All unified debugging techniques substantially improve upon SBFL for all metrics, but tool combinations optimal for method-level fault localization may conflict at the statement level.

This motivates us to investigate to further improve unified debugging at the statement level. Since using all single-edit tools actually degrades ProFL's effectiveness, we believe it is possible to find a subset of tools which outperforms ProFL. From Table 10, we observe five tools outperform other tools by large margins in Top-N; PraPR, AVATAR, kPar, TBar, and FixMiner. In fact, each of these

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2021.3125203, IEEE Transactions on Software Engineering

tools perform template-based repairs, indicating template-based APR tools may be better-suited for statement-level fault localization than other classes of repair tools. Even furthermore, this pattern also appears at the method level, further suggesting the suitability of template-based repair tools.

> **Finding 13:** Template-based APR tools consistently perform the among the best of all APR tools, indicating template-based tools are best suited for unified debugging fault localization.

Due to their substantial improvements, we explore if a subset of these tools may yield unified debugging results better than ProFL's results. Executing multiple tools in a real-world setting can be costly, so we examine unified debugging with top three tools instead of all tools to more closely resemble real-world repair. Specifically, we decide to employ the top three tools which individually exhibit the best Top-5 results, as determined by Table 10. We additionally prioritize based on Top-5 instead of Top-1 because significantly more elements are being represented at the statement level (statements vs methods). These extra elements further dilute results to the point where unified debugging localizes more correct elements above buggy elements, seen from higher MFR / MAR in Table 13. Prioritizing tools with Top-5 (and higher Top-N) should promote only APR tools which exclusively localize buggy methods whereas prioritizing tools with Top-1 (and lower Top-N) are less likely to yield consistent improvements, resulting in degradation with more tools.

The results of this refined tool combination are shown in Table 13. We see improvements of 420% (Top-1), 138% (Top-3), 110% (Top-5), 40% (MFR), and 15% (MAR) from SBFL by using this new tool combination. More impressively, our technique generates 78 Top-1 which even outperforms CombineFL, a recent state-of-the-art statement-level supervised-learning-based fault localization technique [63]. Note that the CombineFL work reports a Top-1 value of 72 but uses **a mixture of average-case ranking and best-case ranking** instead of **worst-case ranking** [63]. We discover CombineFL localizes only 60 Top-1 when worst-case ranking is applied. Surprisingly, with this worst-case ranking, CombineFL also has vastly higher MAR than even SBFL (192 vs 133). By comparison, UniDebug++$_{refined}$ is strictly better than CombineFL. In addition to avoiding the high-quality training set (which is often unavailable) and training time as required by CombineFL, these facts show our technique leads the state-of-the-art in statement-level fault localization.

> **Finding 14:** Unified debugging performs on par with state-of-the-art supervised-learning-based statement-level fault localization techniques. UniDebug++ in particular generates **78 Top-1** compared to 60 Top-1 from CombineFL (i.e., 30% improvement), the current state-of-the-art supervised-learning-based statement-level fault localization.

Tech Name	Top-1	Top-3	Top-5	MFR	MAR
SBFL	15	54	72	114.96	133.06
CombineFL [1]	60	102	125	116.22	192.33
PraPR / ProFL	64	116	139	70.95	115.36
UniDebug+ [2]	41	97	130	69.42	113.1
UniDebug++ [2]	64	120	142	**68.31**	**112.14**
UniDebug+$_{refined}$ [3]	59	118	144	70.20	114.41
UniDebug++$_{refined}$ [3]	**78**	**129**	**151**	69.27	113.62

[1] Represents original CombineFL [63] data converted to worst-case ranking
[2] Uses all 13 single-edit tools
[3] Uses refined set of tools; PraPR, AVATAR, and kPar

TABLE 13: Statement-level effectiveness across all six Defects4J subjects

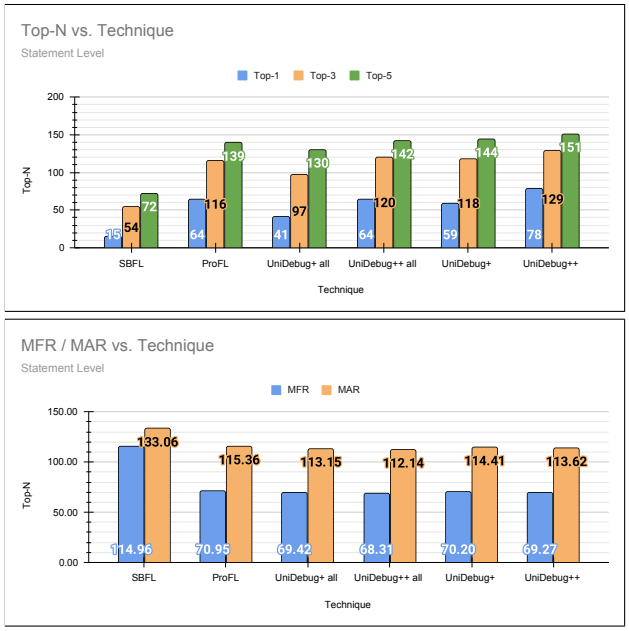

Fig. 18: Statement-level unified debugging effectiveness on six subjects

4.8 RQ8 - Advanced Unified Debugging via Patch Statistics

In previous sections, UniDebug++ uses the number of APR systems to rank the tied code elements with the same suspiciousness values in the same category. However, this strategy may perform worse when there are not enough APR systems. In this section, we propose an advanced technique, UniDebug+⋆, to further improve unified debugging by utilizing the number of patches to rank the tied code elements. In detail, after one code element is assigned a best patch group category in the category aggregation step (shown in Figure 1), we count the total number of patches with the same category from all APR systems for this code element. We then use such number to further rank the tied code elements within the same category. In summary, the major difference between UniDebug++ and UniDebug+⋆ is that the former utilizes the number of APR tools to further rank the tied code elements while the latter utilizes the number of patches.

The intuition is that if APR systems can generate more patches with the best category information for a code element, this element should be ranked higher in the ranked list compared to other tied peers. In fact, such a strategy using patch statistics can also be applied to individual APR systems to extend the original idea of unified debugging

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2021.3125203, IEEE Transactions on Software Engineering

Tech Name	Granularity	Top-1	Top-3	Top-5	MFR	MAR
UniDebug++	Method	185	265	294	8.52	13.59
UniDebug+⋆	Method	**195**	**271**	**298**	**8.06**	**12.98**
UniDebug++$_{refined}$	Statement	78	129	151	69.27	113.62
UniDebug+⋆$_{refined}$	Statement	**85**	**142**	**160**	**65.75**	**109.81**

TABLE 14: Effectiveness of UniDebug+⋆ at method level and statement level

Tech Name	Top-1	Top-3	Top-5	MFR	MAR
PRFL$_{\text{MA UniDebug++}}$	199	273	301	8.53	13.54
PRFL$_{\text{MA UniDebug+⋆}}$	201	276	304	8.30	13.28
DeepFL$_{\text{UniDebug++}}$	**213.8**	288.1	316.3	6.90	8.72
DeepFL$_{\text{UniDebug+⋆}}$	**213.8**	**292.0**	**317.7**	**6.13**	**8.02**

TABLE 15: Effectiveness of UniDebug+⋆ to boost learning-based fault localization

(e.g. ProFL⋆).

We now compare the effectiveness of UniDebug++ and UniDebug+⋆ at both method level and statement level (shown in Table 14). In this table, we observe that UniDebug+⋆ localizes 195 faults within Top-1 at the method level and localizes 89 faults within Top-1 at the statement level, further outperforming the corresponding original UniDebug++ and showing the advantage of considering patch statistics in ranking tied code elements.

With respect to learning-based approaches, we observe that UniDebug+⋆ can also slightly help improve learning-based fault localization (shown in Table 15). For example, PRFL$_{\text{MA UniDebug+⋆}}$ (localizing 201 faults within Top-1) outperforms corresponding PRFL$_{\text{MA UniDebug++}}$ while DeepFL$_{\text{UniDebug+⋆}}$ achieves better MFR and MAR values (i.e., 6.13 and 8.02 respectively) than corresponding DeepFL$_{\text{UniDebug++}}$. For DeepFL, the usage of UniDebug+⋆ within DeepFL improves Top-3, Top-5, MFR, and MAR compared to DeepFL$_{\text{UniDebug++}}$ (Table 15). This leads us to our final finding.

> **Finding 15:** Using the number of patches associated to an element as an advanced ranking strategy, UniDebug+⋆, can further improve the effectiveness of UniDebug++. Even for learning-based approaches, using the UniDebug+⋆ feature further improves upon using the UniDebug++ feature for all considered approaches.

5 Discussion

With all the information presented in Section 4, we now (1) present a brief summarizing comparison of all unified debugging variants at method-level and statement-level granularities by breaking down the patch category composition of each unified debugging variant and (2) discuss the potential application of mutation testing tools (e.g PIT [64]) used alongside/in lieu of APR tools in unified debugging.

5.1 Distribution of Patch Categories

Figure 19 shows the composition of patch categories for all Top-N for each considered unified debugging variant on all the 395 studied bugs for both method and statement granularities. The total number of Top-N patches for each variant is described in red at the top of each column.

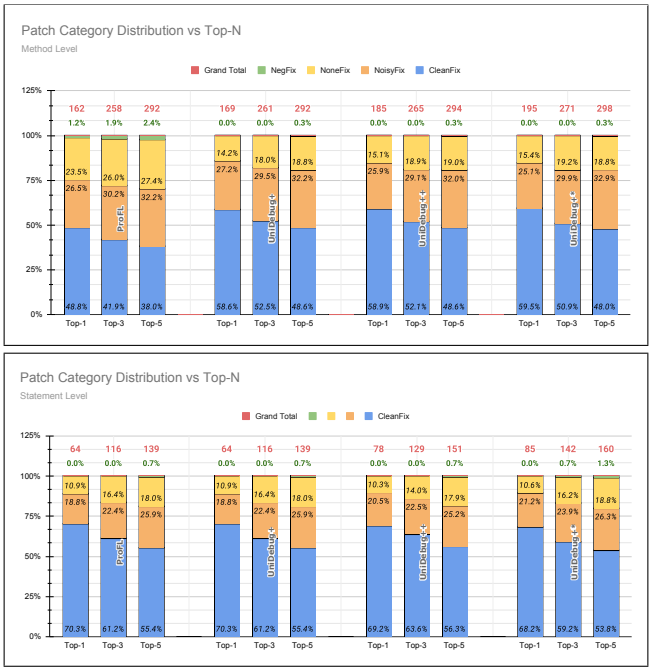

Fig. 19: Distribution of patch categories for Top-N for unified debugging variants

Method Granularity. Following our primary evaluation metrics, we find that the CleanFix category comprises the largest percentage of Top-N across all considered unified debugging variants. For Top-1 in particular, we see the CleanFix category comprising 49% of Top-1 in ProFL. As unified debugging effectiveness increases, the number and percentage of CleanFix patches monotonically increases, up to an impressive ∼60% of Top-1 from UniDebug+⋆. In fact, Top-N is dominated by the CleanFix category across all variants, respectively followed by the NoisyFix, NoneFix, and NegFix categories. As Top-N increases, CleanFix patches become less important while lower quality patch categories become increasingly more prevalent, further showing the critical importance of patch repair information from all types of patches, not just high-quality patches.

Statement Granularity. At the statement granularity, we observe similar results. For example, we still observe the dominance of CleanFix over all other patch categories for all Top-N across all unified debugging variants and the prevalence of lower quality patch categories as Top-N increases. Meanwhile, it is interesting to observe that the percentage of CleanFix patches is rather stable across different unified debugging variants, which differs from the method granularity. This may result from more CleanFix patches sharing the same method(s) (rather than the same statement(s)) with the actual buggy elements as unified debugging includes more APR tools.

From these results at the method and statement granularities, we found that: (1) Top-1 heavily relies on CleanFix patches for precise fault localization and increasingly relies on lower quality patches as Top-N increases, (2) both tie-breaking unified debugging variants, UniDebug++ and UniDebug+⋆, boost UniDebug while maintaining pre-existing high-quality patch repair information, and (3) compared to UniDebug, utilizing repair information from multiple APR tools yields significantly more information crucial

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2021.3125203, IEEE Transactions on Software Engineering

Tech Name	Top-1	Top-3	Top-5	MFR	MAR
SBFL	119	223	268	17.55	23.16
PIT	131	229	272	10.95	16.68
PraPR / ProFL	162	258	292	9.03	14.10
UniDebug+	169	261	292	8.64	13.68
UniDebug+$_{PIT}$ [1]	169	261	292	8.64	13.68
UniDebug++	185	265	294	8.52	13.59
UniDebug++$_{PIT}$ [1]	185	265	294	8.52	13.59
UniDebug+*	195	271	298	8.06	12.98
UniDebug+$^\star{}_{PIT}$ [1]	195	271	298	8.06	12.98

[1] Includes all single-edit APR tools and the PIT mutation testing tool.

TABLE 16: Effectiveness of using PIT alongside unified debugging variants

for fault localization. Furthermore, we find the following implications for pre-existing fault localization techniques: (1) the coarse-grained information provided by traditional SBFL techniques is grossly insufficient to break tied elements, and (2) aggregating patch repair information or other dynamic information alongside existing fault localization may yield substantial improvements. Lastly, the high prevalence of CleanFix categories in all Top-N also indicates synergistic behaviour of unified debugging with techniques whose ultimate objective is to also generate high quality patches. Meanwhile, the increasing importance of lower-quality patches as Top-N increases also indicates the importance to generate a large number of diverse patches for more powerful unified debugging.

5.2 Program Repair Tools vs. Mutation Testing Tools

Program repair is heavily related to mutation testing, the former aims to find/generate bug-fixing patches while the latter aims to find/generate bug-inducing mutants to simulate real bugs [35], [65], [66], [67], [68]. As mutation testing generates mutants that will impact program tests, mutation testing tools are also applicable to unified debugging. Thus we examine how the inclusion of a leading mutation testing tool called PIT [64] impacts unified debugging. Table 16 describes the effectiveness of various unified debugging variants and PIT on all Defects4J subjects. Note that (1) we execute PIT with all its available mutators and (2) we compare UniDebug+, UniDebug++, and UniDebug+* *with and without PIT*.

As we can see from Table 16, PIT by itself provides adequate Top-1 gains over SBFL (131 vs 119), suggesting the feasibility of mutations tools alongside or in lieu of APR tools for our general unified debugging approach. Meanwhile, closer inspection reveals not a single metric is changed with the inclusion of PIT in UniDebug+, UniDebug++, or UniDebug+*, indicating the inclusion of PIT in these variants likely has no impact on effectiveness. This makes sense as the inclusion of PIT is unlikely to yield unique repair information (e.g., the studied PraPR APR tool was built on PIT and can strictly generate more patches than PIT). Thus unified debugging is indifferent and does not necessarily favor APR tools or mutation testing tools, but rather synergizes with tools and techniques that generate diverse and high quality patch repair information.

5.3 Threats to Validity

5.3.1 Internal Validity

All of our study results are directly dependent on the correctness of our implementation of all the studied techniques. Faulty implementations in any aspect will yield misleading/inaccurate results. To mitigate this threat, we reuse the implementation of the SBFL and ProFL techniques obtained from the ProFL authors. We also obtained the source code for all the studied APR systems from the authors to investigate their impact on unified debugging. Furthermore, we execute both APR systems with our modifications (to record detailed patch execution information required by unified debugging) and the original APR systems to ensure that they produce the same results (i.e., our modification does not change the APR behavior).

5.3.2 Construct Validity

This threat mainly lies in the dependent variables or metrics used in this study. To reduce such threats, we adopted the most widely used metrics in recent fault localization [12], [36], [44] and unified debugging studies [48], [49].

5.3.3 External Validity

To evaluate on real-world bugs, we choose the Defects4J dataset with hundreds of real-world bugs, which is the most widely used benchmark suite for recent APR and fault localization work. However, the study results may still not generalize to all possible systems in the wild. Furthermore, prior work has demonstrated that program repair tools might overfit the Defects4J dataset [69]. As unified debugging is based on patch repair information from program repair tools, it is indeed possible that the overfitting can be fed back to these fault localization results. However, as shown in our study, the effectiveness of unified debugging does not correlate with APR effectiveness. Also, the capability of UniDebug+, UniDebug++, and UniDebug+* to utilize multiple other program repair tools may also help mitigate overfitting issues – These techniques are designed such that the potential redundant sources of information from an increasingly amount of program repair tools will lower the chances of overfitting.

Since the study aims to investigate the impact of APR systems for unified debugging, different APR systems may also yield totally different results. Therefore, we simply studied all state-of-the-art APR systems that (1) have publicly available source code and (2) are applicable to the widely used Defects4J benchmark [58].

6 Related Work

As the studied unified debugging approach unifies traditional fault localization and automated program repair (APR) to boost both areas, in this section, we talk about the related work in both areas.

6.1 Fault Localization

The basic idea of fault localization is to automatically produce a ranking list of code elements (e.g., program methods or statements) based on the descending order of their suspiciousness values (i.e., the probability of being buggy)

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2021.3125203, IEEE Transactions on Software Engineering

to help developers in manual debugging or serve as the supplier for APR. Various fault localization techniques have been proposed over the past decades. Spectrum-based fault localization (SBFL) [70], [71], one of the most classic fault localization approaches, has been intensively studied due to its effectiveness and scalability. Its basic insight is that code elements primarily executed by failed tests are more suspicious than elements primarily executed by passed tests.

To date, various formulae (e.g., based on statistical analysis or other heuristics) have been proposed to compute code element suspiciousness, such as Tarantula [6], Ochiai [50], SBI [7], and so on. One main limitation for such traditional SBFL is that faulty code elements may be coincidentally executed by passed tests and elements executed by failed tests do not always have real impacts on the program failure. To bridge the gap between coverage and impact information, mutation-based fault localization (MBFL) [8], [9], [35], [36] has been proposed to transform program source code based on mutation testing [72] to check the impact of each code element on test outcomes. The basic idea of MBFL is that if one mutant incurs different failure outputs of failed tests before and after mutation, the corresponding code element of this mutant may have a high impact on program failures, and thus may be the buggy. MUSE [9] and Metallaxis [8] are two widely studied MBFL techniques targeting traditional application scenarios, while FIFL [35] is a MBFL technique specifically targeting evolving software systems. Compared with traditional MBFL techniques that were bound to mutation testing, unified debugging utilizes program repair information that aims to fix software bugs to *pass* more tests rather than mutation testing that was originally proposed to create new artificial bugs to *fail* more tests; furthermore, unified debugging has also been shown to substantially outperform state-of-the-art MBFL [48], [49]. In the literature, researchers have also proposed various other fault localization techniques, including techniques based on program slicing [55], [73], development history [74], and information retrieval [56], as well as techniques for combining various dimensions of information via machine learning [12], [37], [44].

6.2 Automated Program Repair

Automated program repair (APR) aims to directly fix program bugs without human intervention. Given a buggy project, APR techniques utilize various strategies to automatically generate potential patches and then validate those patches to check their correctness, e.g., based on regression tests [45], static analysis [75], or formal specifications [76]. To date, test-driven APR has been extensively studied due to the wide adoption of testing in practice. A typical test-driven APR technique first applies off-the-shelf fault localization techniques (e.g., Ochiai [50] has been widely used for APR [21], [45], [57]) to pinpoint potential buggy locations for patching. Then, any patches that can pass all the originally failing and passing tests are called *plausible patches*, while plausible patches semantically equivalent to corresponding developer patches are called *correct patches* (which are the final outcome for APR). Depending on how the patches are generated, APR [21], [27], [45], [51] can be categorized into the following categories [58], [77]: (1) *heuristic-based APR*, which investigates possible code modifications for patching by iterating a search space, e.g., GenProg [59] uses genetic programming algorithm to search donor code from existing code for generating patches; (2) *constraint-based APR*, which typically transforms the APR problem into a satisfiability problem by constructing a repair constraint that the patches should satisfy, e.g., Nopol [10] leverages an SMT solver to solve the condition synthesis problem; (3) *template-based APR*, which performs APR via predefined fixing patterns, e.g., FixMiner [52] automatically mines bug-fix patterns from existing code repositories; (4) *learning-based APR*, which uses machine learning techniques to learn correct code locations/snippets from a training code corpus, e.g., Prophet [27] and ELIXIR [78].

Recently, ProFL [48] initializes the idea of unified debugging to investigate the effectiveness of APR for fault localization. The experimental results show that ProFL is able to boost/outperform state-of-the-art SBFL [6], [7], [50], MBFL [8], [9], [35], [36], and unsupervised/supervised learning based fault localization [11], [12], [44] using the recent PraPR APR system [45], and also extends the application scope of APR to all possible bugs. However, it is not clear how other state-of-the-art APR techniques contribute to unified debugging and how to further advance unified debugging. This paper moves one step forward to that end, particularly with regards to assessing the impact of other APR tools in unified debugging.

7 Conclusion

In this paper, we have performed an extensive study of the impacts of different automated program repair systems on the recently proposed unified debugging approach [48], [49]. Our study results on the popular Defects4J benchmark suite have revealed various practical guidelines / findings for further advancing unified debugging, including: (1) nearly all studied 16 repair systems positively contribute to unified debugging despite their varying repair capabilities, (2) repair systems targeting multi-edit patches can bring extraneous noise into unified debugging, (3) repair systems with more executed/plausible patches tend to perform better for unified debugging, (4) unified debugging effectiveness does not rely on the availability of correct patches in automated repair, and (5) we propose an advanced unified debugging technique, UniDebug++, which localizes over 20% more bugs within Top-1 than state-of-the-art unified debugging technique ProFL (evaluated against four Defects4J subjects). Furthermore, we conduct more comprehensive studies to extend the above experiments to make the following additional unified debugging contributions: (6) we further perform an extensive study on additional subjects from Defects4J (Closure and Mockito) and confirm that UniDebug++ again outperforms ProFL by localizing 185 (out of 395 in total) bugs within Top-1, 14% more than ProFL, (7) we investigate the impact of 33 SBFL formulae on unified debugging and observe that UniDebug++ consistently improves upon all formulae, e.g., 61% and 53% average improvement on MFR / MAR, (8) we demonstrate that UniDebug++ can substantially boost state-of-the-art learning-based method-level fault localization techniques, (9) we extend unified debugging to the statement level for first time and observe

that UniDebug++ localizes 78 (out of 395 in total) bugs within Top-1 (22% more bugs than ProFL) and outperforms state-of-the-art learning-based fault localization techniques by 30%, and finally (10) we further propose a new technique, UniDebug+*, based on detailed patch statistics, to improve upon UniDebug++, e.g., further localizing 9% more bugs within Top-1 than UniDebug++ at the statement level.

In the near future, we intend to work on *tentative program repair*, a new direction enabled by unified debugging to allow fault localization and program repair to boost each other for more powerful debugging, e.g., patch execution results from an initial set of repair systems can enable precise fault localization for applying more advanced repair systems for cost-effective repair.

REFERENCES

[1] "Tricentis reports," 2020. [Online]. Available: https://www.tricentis.com/resources/software-fail-watch-5th-edition/

[2] U. Software, "Increasing software development productivity with reversible debugging," https://undo.io/media/uploads/files/Undo_ReversibleDebugging_Whitepaper.pdf, 2016, accessed: Jan. 21, 2019.

[3] C. Boulder, "University of cambridge study: Failure to adopt reverse debugging costs global economy $41 billion annually," https://www.roguewave.com/company/news/2013/university-of-cambridge-reverse-debugging-study, 2013, accessed: Jan. 8, 2019.

[4] M. Wen, J. Chen, Y. Tian, R. Wu, D. Hao, S. Han, and S.-C. Cheung, "Historical spectrum based fault localization," *IEEE Transactions on Software Engineering (TSE)*, 2020.

[5] R. Abreu, P. Zoeteweij, and A. J. Van Gemund, "On the accuracy of spectrum-based fault localization," in *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION (TAICPART-MUTATION 2007)*. IEEE, 2007, pp. 89–98.

[6] J. A. Jones and M. J. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, 2005, pp. 273–282.

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

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

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

[10] J. Xuan, M. Martinez, F. Demarco, M. Clement, S. L. Marcote, T. Durieux, D. Le Berre, and M. Monperrus, "Nopol: Automatic repair of conditional statement bugs in java programs," *IEEE Transactions on Software Engineering*, vol. 43, no. 1, pp. 34–55, 2016.

[11] M. Zhang, X. Li, L. Zhang, and S. Khurshid, "Boosting spectrum-based fault localization using pagerank," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2017, pp. 261–272.

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

[13] L. Zhang, M. Kim, and S. Khurshid, "Localizing failure-inducing program edits based on spectrum information," in *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, 2011, pp. 23–32.

[14] C. Gouveia, J. Campos, and R. Abreu, "Using html5 visualizations in software fault localization," in *2013 First IEEE Working Conference on Software Visualization (VISSOFT)*, 2013, pp. 1–10.

[15] P. Zoeteweij, R. Abreu, R. Golsteijn, and A. J. van Gemund, "Diagnosis of embedded software using program spectra," in *14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS'07)*, 2007, pp. 213–220.

[16] 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 *the 35th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2020, 2020.

[17] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé, "Tbar: Revisiting template-based automated program repair," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, p. 31–42.

[18] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé, "AVATAR: fixing semantic bugs with fix patterns of static analysis violations," in *SANER*, 2019, pp. 456–467.

[19] T. Durieux and M. Monperrus, "Dynamoth: Dynamic code synthesis for automatic program repair," in *Proceedings of the 11th International Workshop on Automation of Software Test*, ser. AST '16. Association for Computing Machinery, 2016, p. 85–91.

[20] M. Martinez and M. Monperrus, "Ultra-large repair search space with automatically mined templates: The cardumen mode of astor," in *SSBSE*, 2018, pp. 65–86.

[21] J. Jiang, Y. Xiong, H. Zhang, Q. Gao, and X. Chen, "Shaping program repair space with existing patches and similar code," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2018, pp. 298–309.

[22] M. Martinez and M. Monperrus, "Astor: A program repair library for java," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, 2016, pp. 441–444.

[23] Y. Yuan and W. Banzhaf, "Arja: Automated repair of java programs via multi-objective genetic programming," *IEEE Transactions on Software Engineering*, 2018.

[24] Y. Lou, J. Chen, L. Zhang, D. Hao, and L. Zhang, "History-driven build failure fixing: how far are we?" in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 43–54.

[25] M. Wu, L. Zhang, C. Liu, S. H. Tan, and Y. Zhang, "Automating cuda synchronization via program transformation," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019, pp. 748–759.

[26] J. Jiang, L. Ren, Y. Xiong, and L. Zhang, "Inferring program transformations from singular examples via big code," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 255–266.

[27] F. Long and M. Rinard, "Automatic patch generation by learning correct code," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2016, pp. 298–312.

[28] Z. Qi, F. Long, S. Achour, and M. Rinard, "An analysis of patch plausibility and correctness for generate-and-validate patch generation systems," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, 2015, p. 24–36.

[29] F. Long and M. Rinard, "Staged program repair with condition synthesis," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015, pp. 166–178.

[30] F. Long, P. Amidon, and M. Rinard, "Automatic inference of code transforms for patch generation," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 727–739.

[31] F. Long and M. Rinard, "An analysis of the search spaces for generate and validate patch generation systems," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 2016, pp. 702–713.

[32] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. J. C. van Gemund, "A practical evaluation of spectrum-based fault localization," *J. Syst. Softw.*, vol. 82, no. 11, pp. 1780–1792, 2009. [Online]. Available: https://doi.org/10.1016/j.jss.2009.06.035

[33] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund, "Spectrum-based multiple fault localization," in *ASE 2009, 24th IEEE/ACM International Conference on Automated Software Engineering, Auckland, New Zealand, November 16-20, 2009*. IEEE Computer Society, 2009, pp. 88–99. [Online]. Available: https://doi.org/10.1109/ASE.2009.25

[34] ——, "An observation-based model for fault localization," in *Proceedings of the 2008 International Workshop on Dynamic Analysis: held in conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2008), WODA 2008, Seattle, Washington, USA, July 21, 2008*, B. Liblit and A. Rountev, Eds. ACM, 2008, pp. 64–70. [Online]. Available: https://doi.org/10.1145/1401827.1401841

[35] L. Zhang, L. Zhang, and S. Khurshid, "Injecting mechanical faults to localize developer faults for evolving software," in *OOPSLA*, 2013, p. 765–784.

[36] X. Li and L. Zhang, "Transforming programs and tests in tandem for fault localization," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, Oct. 2017. [Online]. Available: https://doi.org/10.1145/3133916

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

[38] T.-D. B. Le, D. Lo, C. Le Goues, and L. Grunske, "A learning-to-rank based fault localization approach using likely invariants," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, 2016, pp. 177–188.

[39] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair," *Ieee transactions on software engineering*, vol. 38, no. 1, pp. 54–72, 2011.

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

[41] M. Monperrus, "Automatic software repair: a bibliography," *ACM Computing Surveys (CSUR)*, vol. 51, no. 1, pp. 1–24, 2018.

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

[43] P. S. Kochhar, X. Xia, D. Lo, and S. Li, "Practitioners' expectations on automated fault localization," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, 2016, pp. 165–176.

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

[45] A. Ghanbari, S. Benton, and L. Zhang, "Practical program repair via bytecode mutation," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 19–30.

[46] S. Saha *et al.*, "Harnessing evolution for multi-hunk program repair," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 13–24.

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

[48] Y. Lou, A. Ghanbari, X. Li, L. Zhang, D. Hao, and L. Zhang, "Can automated program repair refine fault localization?" *arXiv preprint arXiv:1910.01270*, 2019.

[49] Y. Lou, A. Ghanbari, X. Li, L. Zhang, H. Zhang, D. Hao, and L. Zhang, "Can automated program repair refine fault localization? a unified debugging approach," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020, pp. 75–87.

[50] R. Abreu, P. Zoeteweij, and A. J. Van Gemund, "An evaluation of similarity coefficients for software fault localization," in *2006 12th Pacific Rim International Symposium on Dependable Computing (PRDC'06)*. IEEE, 2006, pp. 39–46.

[51] Y. Xiong, J. Wang, R. Yan, J. Zhang, S. Han, G. Huang, and L. Zhang, "Precise condition synthesis for program repair," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 2017, pp. 416–426.

[52] A. Koyuncu, K. Liu, T. F. Bissyandé, D. Kim, J. Klein, M. Monperrus, and Y. Le Traon, "Fixminer: Mining relevant fix patterns for automated program repair," *Empirical Software Engineering*, pp. 1–45, 2020.

[53] K. Liu, A. Koyuncu, T. F. Bissyandé, D. Kim, J. Klein, and Y. Le Traon, "You cannot fix what you cannot find! an investigation of fault localization bias in benchmarking automated program repair systems," in *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, 2019, pp. 102–113.

[54] S. Benton, X. Li, Y. Lou, and L. Zhang, "On the effectiveness of unified debugging: An extensive study on 16 program repair systems," in *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2020.

[55] H. Agrawal, J. R. Horgan, S. London, and W. E. Wong, "Fault localization using execution slices and dataflow tests," in *Proceedings of Sixth International Symposium on Software Reliability Engineering. ISSRE'95*. IEEE, 1995, pp. 143–151.

[56] J. Zhou, H. Zhang, and D. Lo, "Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports," in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 14–24.

[57] M. Wen, J. Chen, R. Wu, D. Hao, and S. Cheung, "Context-aware patch generation for better automated program repair," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, 2018, pp. 1–11.

[58] K. Liu, S. Wang, A. Koyuncu, K. Kim, T. F. D. A. Bissyande, D. Kim, P. Wu, J. Klein, X. Mao, and Y. Le Traon, "On the efficiency of test suite based program repair: A systematic assessment of 16 automated repair systems for java programs," in *42nd ACM/IEEE International Conference on Software Engineering (ICSE)*, 2020.

[59] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, "Automatically finding patches using genetic programming," in *2009 IEEE 31st International Conference on Software Engineering*. IEEE, 2009, pp. 364–374.

[60] K. Pearson, "Notes on regression and inheritance in the case of two parents proceedings of the royal society of london, 58, 240-242," 1895.

[61] L. Chen, Y. Ouyang, and L. Zhang, "Fast and precise on-the-fly patch validation for all," in *ICSE*, 2021, pp. 1123–1134.

[62] M. Zhang, Y. Li, X. Li, L. Chen, Y. Zhang, L. Zhang, and S. Khurshid, "An empirical study of boosting spectrum-based fault localization via pagerank," *IEEE Transactions on Software Engineering*, 2019.

[63] 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.

[64] "Pit mutation testing system," 2018. [Online]. Available: http://pitest.org/

[65] T. A. Budd, "Mutation analysis of program test data," Ph.D. dissertation, New Haven, CT, USA, 1980, aAI8025191.

[66] V. Musco, M. Monperrus, and P. Preux, "A large-scale study of call graph-based impact prediction using mutation testing," *Software Quality Journal*, pp. 1–30, 2016.

[67] L. Zhang, T. Xie, L. Zhang, N. Tillmann, J. De Halleux, and H. Mei, "Test generation via dynamic symbolic execution for mutation testing," in *ICSM*, 2010, pp. 1–10.

[68] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE transactions on software engineering*, vol. 37, no. 5, pp. 649–678, 2010.

[69] T. Durieux, F. Madeiral, M. Martinez, and R. Abreu, "Empirical review of java program repair tools: A large-scale experiment on 2,141 bugs and 23,551 repair attempts," in *FSE*, 2019, p. 302–313.

[70] L. Naish, H. J. Lee, and K. Ramamohanarao, "A model for spectra-based software diagnosis," *ACM Transactions on software engineering and methodology (TOSEM)*, vol. 20, no. 3, pp. 1–32, 2011.

[71] R. Santelices, J. A. Jones, Y. Yu, and M. J. Harrold, "Lightweight fault-localization using multiple coverage types," in *2009 IEEE 31st International Conference on Software Engineering*. IEEE, 2009, pp. 56–66.

[72] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE TSE*, vol. 37, no. 5, pp. 649–678, 2011.

[73] M. Renieres and S. P. Reiss, "Fault localization with nearest neighbor queries," in *18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings.* IEEE, 2003, pp. 30–39.

[74] S. Kim, T. Zimmermann, E. J. Whitehead Jr, and A. Zeller, "Predicting faults from cached history," in *29th International Conference on Software Engineering (ICSE'07)*. IEEE, 2007, pp. 489–498.

[75] R. van Tonder and C. L. Goues, "Static automated program repair for heap properties," in *Proceedings of the 40th International Conference on Software Engineering*, 2018, pp. 151–162.

[76] Y. Wei, Y. Pei, C. A. Furia, L. S. Silva, S. Buchholz, B. Meyer, and A. Zeller, "Automated fixing of programs with contracts," in *Proceedings of the 19th international symposium on Software testing and analysis*, 2010, pp. 61–72.

[77] C. L. Goues, M. Pradel, and A. Roychoudhury, "Automated program repair," *Communications of the ACM*, vol. 62, no. 12, pp. 56–65, 2019.

[78] R. K. Saha, Y. Lyu, H. Yoshida, and M. R. Prasad, "Elixir: Effective object-oriented program repair," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2017, pp. 648–659.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2021.3125203, IEEE Transactions on Software Engineering

Samuel Benton received his B.S. in Software Engineering and B.S. in Computer Engineering in 2016 and Masters in Software Engineering in 2019 from the University of Texas at Dallas. He is a currently Software Engineering doctoral candidate attending the University of Texas at Dallas under the supervision of Dr. Lingming Zhang and Dr. Adrian Marcus.

Xia Li is an Assistant Professor in the Department of Software Engineering and Game Design and Development at Kennesaw State University. He received his PhD degree in Computer Science at University of Texas at Dallas supervised by Dr. Lingming Zhang. His research interests focus on Software Engineering, in particular: software testing and debugging involving dynamic/static program analysis.

Yiling Lou is currently a postdoctoral fellow in the Department of Computer Science at Purdue University. She received her B.S. in 2016 and Ph.D. in 2021 from the Department of Computer Science and Technology at Peking University. Her research interests mainly focus on Software Testing and Debugging, and its synergy with Artificial Intelligence and Program Analysis.

Lingming Zhang is an Associate Professor in Department of Computer Science at the University of Illinois at Urbana-Champaign, where he leads the Intelligent Software Engineering (iSE) group. He obtained PhD in Electrical and Computer Engineering at the University of Texas at Austin in 2014. He received BSc in Computer Science from Nanjing University and MSc in Computer Science from Peking University. His main research interests lie in Software Engineering, and its synergy with Machine Learning, Programming Languages, and Formal Methods. His research has been recognized with a number of awards, including the ACM SIGSOFT Early Career Researcher Award, NSF CAREER Award, Google Faculty Research Award, SAMSUNG GRO Award, and multiple distinguished/best paper awards.