Interactive Patch Filtering as Debugging Aid

Jingjing Liang
Key Lab of High Confidence Software
Technologies, Ministry of Education
Department of Computer Science and
Technology, EECS, Peking University
Beijing, China
jingjingliang@pku.edu.cn

Ruyi Ji
Key Lab of High Confidence Software
Technologies, Ministry of Education
Department of Computer Science and
Technology, EECS, Peking University
Beijing, China
jiruyi910387714@pku.edu.cn

Jiajun Jiang
Key Lab of High Confidence Software
Technologies, Ministry of Education
Department of Computer Science and
Technology, EECS, Peking University
Beijing, China
jiajun.jiang@pku.edu.cn

Yiling Lou
Key Lab of High Confidence Software
Technologies, Ministry of Education
Department of Computer Science and
Technology, EECS, Peking University
Beijing, China
louyiling@pku.edu.cn

Yingfei Xiong
Key Lab of High Confidence Software
Technologies, Ministry of Education
Department of Computer Science and
Technology, EECS, Peking University
Beijing, China
xiongyf@pku.edu.cn

Gang Huang
Key Lab of High Confidence Software
Technologies, Ministry of Education
Department of Computer Science and
Technology, EECS, Peking University
Beijing, China
hg@pku.edu.cn

ABSTRACT

It is widely recognized that program repair tools need to have a high precision to be useful, i.e., the generated patches need to have a high probability to be correct. However, it is fundamentally difficult to ensure the correctness of the patches, and many tools compromise other aspects of repair performance such as recall for an acceptable precision.

In this paper we ask a question: can a repair tool with a low precision be still useful? To explore this question, we propose an interactive filtering approach to patch review, which filters out incorrect patches by asking questions to the developers. Our intuition is that incorrect patches can still help understand the bug. With proper tool support, the benefit outweighs the cost even if there are many incorrect patches.

We implemented the approach as an Eclipse plugin tool, *InPaFer*, and evaluated it with a simulated experiment and a user study with 30 developers. The results show that our approach improve the repair performance of developers, with 62.5% more successfully repaired bugs and 25.3% less debugging time in average. In particular, even if the generated patches are all incorrect, the performance of the developers would not be significantly reduced, and could be improved when some patches provide useful information for repairing, such as the faulty location and a partial fix.

KEYWORDS

Interactive debugging, Patch filtering, User study, Program repair

1 INTRODUCTION

In the past decades, automatic program repair (APR) attracted a lot of research efforts and significant progress has been made [1, 10, 12, 19, 23, 29, 39, 44, 45]. Many of the proposed APR approaches are test-based program repair approaches, which take as input a buggy program and a test suite with at least a failed test, and automatically generate a set of patches that make all tests pass.

It is commonly recognized that an APR tool needs to have a high precision to be useful, i.e., the generated patches should have a high probability to be correct. For example, Tao et al. [36] have shown that the repair performance of developers significantly improves when they are given a high-quality patch, and becomes worse when they are given a low-quality patch. If a repair approach produces more incorrect patches than correct patches, the overall repair performance would become even lower.

However, it is fundamentally difficult to achieve a high precision. Most programs are not developed with a formal and complete specification, and we usually only have a test suite as a specification. Since tests cannot guarantee the correctness of the program, it is therefore fundamentally impossible to guarantee the correctness of patches. Furthermore, in practice the test suites are usually weak, such that many more incorrect patches can pass the tests than correct patches [25], making achieving a high precision very difficult. This problem is often known as "weak test suite" [30] or "overfitting" [34].

As a result, many approaches take conservative strategies [31], compromising performance in other aspects to reach an acceptable precision. For example, many APR tools return only one most probable patch for each bug [12, 32, 39, 42, 44]. Some approaches repair only bugs satisfying strict conditions, e.g., frequently recurring bugs [2, 24] or bugs with a reference implementation [14, 28]. These conservative strategies inevitably compromise the performance in other aspects. In the above examples, returning more patches or targeting more bugs could potentially help generate correct patches for more bugs, i.e., increasing recall, but the conservative strategies disallow this possibility. In fact, all state-of-the-art repair approaches have a low recall, i.e., only a small portion of bugs can be correctly fixed. For example, Hercules [33], one of the newest repair approach proposed in 2019, repairs only 13% bugs on the Defects4J benchmark.

Given the fundamental difficulty in achieving a high precision, in this paper we explore from a different perspective: can a repair tool with a low precision still be useful? Our exploration is based on the following observation: The practical use of ARP tools consists of two steps [8], patch generation, where the ARP tool proposes candidate patches, and patch review, where the developer examines

the patches to ensure their correctness and other quality attributes. While the patch generation has been extensively studied, we still lack understanding and tool support for patch review. We conjecture that the review of incorrect patches also helps understand the bug, and with proper tool support, the review of incorrect patches would at least not reduce the repair performance of the developers. If this conjecture holds, we have the liberty to build ARP tools with lower precisions, e.g., by generating more patches per bugs, and potentially increase other aspects of tool performance, e.g., recall.

We propose an interactive patch filtering approach to provide tool support for patch review, verifying the conjecture constructively. Given multiple patches for a bug, where most of the patches are expected to be incorrect, an interactive patch filtering tool asks the developer questions about attributes of the system that could distinguish between different patches. The attributes could be about the behavior of a test, e.g., whether a statement should be executed during a test or not, or could be the confidence of the developer on the system, e.g., whether a method should be patched or not. The developer picks a question and provides an answer, and the tool filters the patches based on the answer. The process continues until the developer figures out a correct patch (by picking an automatically generated one or by contriving a new one) or no more patches can be filtered out. We design a two-stage algorithm to implement this interactive system, introducing an offline preparing stage to optimize the response time of the online interactive stage.

We have implemented our approach as an Eclipse plugin called *InPaFer*, which stands for an **In**teractive **Pa**tch **Fil**ter. The plugin includes a user interface to allow the developer to easily browse the patches and the questions, as well as a diff view to visualize the effect of a patch on a test execution. The plugin implements three types of questions: (1) whether a statement should be executed in a test execution, (2) whether the assignment to a variable is correct in a test execution, (3) whether a method should be patched or not.

Based on the plugin, we conducted two experiments to verify the conjecture. We collected the patches generated by 13 different repair tools on the Defects4J benchmark [13] and assume that the patches are generated by one tool. In this way, we get a combined tool whose precision is lower than every single tool but the recall is higher than every single tool. Based on this tool, the first experiment is a simulated experiment that measures the average number of questions to distinguish the patches, where a computer-simulated user randomly chooses questions and provides answers. The result suggests that a relatively small number of questions are needed to finish the filtering process, 3.1 per bug on average, and the number of needed questions is not related to the number of patches.

The second experiment is a user study involving 30 participants divided into three groups. The three groups repair bugs without any patch, with the generated patches, and with *InPaFer*, respectively. The results show that, compared with the group with no patch, the group with *InPaFer* correctly repair 63% more bugs and uses 25% less time on average; compared with the group with the generated patches, the group with *InPaFer* correctly repair 39% more bugs and uses 28% less time on average. Furthermore, even if all generated patches are incorrect, the group with *InPaFer* still performs slightly better than the group with no patches, repairing 27% more bugs and uses 13% less time. This confirms our conjecture: with proper

tool support, the patch review process helps understand the bug, and eventually contribute to debugging.

In summary, this paper makes the following contributions:

(1) An interactive patch filtering approach to supporting the patch review step, and a two-stage algorithm to implement the approach.
(2) An Eclipse plugin with a carefully designed interface for the user to easily browse the questions and the patches.
(3) Two experiments, including a user study, to investigate the usefulness of *InPaFer* in aiding developers in debugging.

The remainder of the paper is organized as follows. Section 2 introduces the framework of our approach. Section 3 illustrates our implementation in detail. Section 4 evaluates the effectiveness of our approach on Defects4J, while Section 5 and 6 validates the threats and related work, respectively. Finally, Section 7 concludes the paper.

2 APPROACH

In this section, we will first use a running example to introduce the overview of our approach in Section 2.1. Then, the next two sections (i.e., Section 2.2 and 2.3) will demonstrate the two stages of our approach. Figure 2 presents the overview of our approach.

2.1 Overview

In this section, we will introduce our approach with a running example. Figure 1 shows a code snippet from the buggy program Math41 in Defects4J [13] benchmark, which invokes the buggy method `evaluate()` when the condition `length>1` is satisfied. Besides, the following three patches are generated by existing APR techniques and can make all the test cases pass. In particular, p_1 and p_2 are incorrect patches that change the condition in line 320 (in Figure 1) to a new one, while p_3 is the correct patch that updates a `for` statement in the buggy method `evaluate()`.

```
313    public double eval(double values, ...) {
           ...
320        if(length == 1){  //incorrect patches change here
321            var = 0.0;
322        }else if(length > 1){ ...
323            // buggy code resides in method evaluate()
324            var=evaluate(values,weights,m,begin,length);
325        }
           ...
329    }
```

Figure 1: A code snippet from Math41

✗ p_1: `if(length==1)`→ `if(length==5&&length!=0)`
✗ p_2: `if(length==1)`→ `if((length&1)==1)`
✓ p_3: `for(i=0;i<weights.length;)`→ `for(i=begin;i<begin+length;)`

By analyzing the program and the patches, we can collect a set of program attributes related to each patch. For example, when applying the first two patches (i.e., p_1 and p_2), the failed test case executes the statement in line 321. On the contrary, when applying patch p_3, the statement in line 321 is not executed. Therefore, checking the correctness of the attributes can help to filter incorrect patches. As a consequence, our approach will select the attributes that have the

ability to distinguish different candidate patches, and treat them as questions to ask for developers' confirmation. For example, for the above three candidate patches, the first two questions listed below correspond to the attributes related to program execution trace and change location, respectively.

> q_1: *Whether the statement in line 321 should be covered?*
> q_2: *Whether the method* `evaluate()` *should be patched?*
> q_3: ...

Then, we can present all the questions to developers. The developer could pick a question and provide an answer. For example, suppose the developer first select the first question, and regards it as incorrect (i.e., the answer is No.). The patches related to this attribute can be filtered (i.e., p_1 and p_2 in the example), because the patches cause the incorrect program attributes. However, the other patch (i.e., p_3) will be remained since it does not make the program have the attribute. In fact, each time a question (not the last one) is answered, there must be some patches filtered: when an attribute is refuted, the corresponding patches can be filtered. Otherwise, the other patches can be filtered. For example, the answer to question q_2 is confirmed, the patches p_1 and p_2 can be filtered, because they change the code in method `eval()` but not `evaluate()`.

In summary, by answering questions, the number of candidate patches will monotonously decrease. Based on this insight, we proposed our approach called *InPaFer*, which leverages program attributes as questions to ask developers and filters patches generated by APR techniques according to the answers.

There is a challenge in implementing *InPaFer*: since collecting some kinds of attributes may take a lot of time, such as program execution trace, it will be impractical to provide a timely response for online debugging. To overcome this challenge, we utilize the fact that repair approaches are assumed to work offline, e.g., after a daily build and before the working time of the next day. In fact, current repair techniques often require hours to fix a bug, and cannot be used online. Based on this fact, we design a two-stage approach. The first stage (i.e., Preparing Stage) is an offline process that collects program attributes, while the second stage (i.e., Interactive Stage) is an online process that only needs to filter patches based on the collected attributes, which can be achieved within a short response time.

2.2 Preparing Stage

In this section, we will describe the first stage in our approach, which is called preparing stage. As explained, this stage is an offline process that performs data preparation for the next interactive stage. In particular, when given a set of patches related to a bug, our approach automatically collects program attributes for different patches, which will be finally leveraged to construct a set of questions for interaction.

Generally, many kinds of attributes can be used in our approach as long as they can distinguish the candidate patches from some perspective. However, the attributes that can distinguish more patches and are easy to understand for developers should be preferred, because they potentially can decrease the number of interactions and reduce the burden of developers. As a result, the current implementation of our approach employs three kinds of attributes of programs, which include both static code property (`Modified Method`)

and dynamic runtime features (`Execution Trace` and `Variable Value`). The followings describe the details of the attributes.

- **`Modified Method`** denotes the specific method in the program, which the patches modify the code in. This kind of attribute is described as "*The method m should be patched*", where *m* represents the name of some method. Therefore, according to the given patches, our approach automatically analyzes the change locations for each patch, i.e., which methods are modified by the patch. Particularly, when a patch changes multiple methods, our approach will record all of them.
- **`Execution Trace`** means the executed statements while running the failing test case over the patched program. For simplicity, we only consider execution traces in methods that are modified by the given patches. That is we first collect all methods that are modified by at least one patch, and then record the traces of the test execution in all those methods over the patched program. In particular, the traces are collected at the line level, i.e., which lines of code are executed. Additionally, since the same method may also be executed multiple times in one execution, we leverage a hierarchical aligning algorithm to compare the difference of execution traces. That is when given two traces, we first align them at method level and obtain pair-wise methods, and next we compare the traces in a pair of methods. In these two processes, we greedily align the traces based on the execution order. Finally, we identify the differences between traces at line level as attributes, i.e., some lines of code are uniquely executed over a part of patched programs. In general, this attribute is described as "*The statement at line n in method m should be executed*", where *n* and *m* represent the line number and method, respectively. Therefore, each attribute corresponds to a unique line of code in the program.
- **`Variable Value`** indicates a variable is assigned some value at specific locations during the execution. Particularly, we consider all local variables and class fields with primitive types at the entry and exit locations of the modified methods. More concretely, for each method that is modified by at least one patch, we collect all values assigned to variables at the entry and exit of all invocations to the method. Therefore, the attribute of "*the value val assigned to var*" denotes the variable *var* is assigned the value of *val* in the execution at least once over the patched program.

Therefore, in the current implementation of *InPaFer*, in total we use three kinds of attributes. They are described as (1) the method *m* should be patched, (2) the statement at line *n* in method *m* should be executed, and (3) the value *val* assigned to *var* is correct, respectively, where the *m*, *n*, *val* and *var* correspond to some method, line number, variable value and variable name.

In order to store the attributes and corresponding patches, we treat each attribute as a question and define the following data structure.

Definition 2.1. (*Interactive Question (IQ)*.) An interactive question is a pair $\langle q_{attr}, patches \rangle$, where q_{attr} is a question about whether the attribute *attr* of the program is correct or not, and *patches* is a
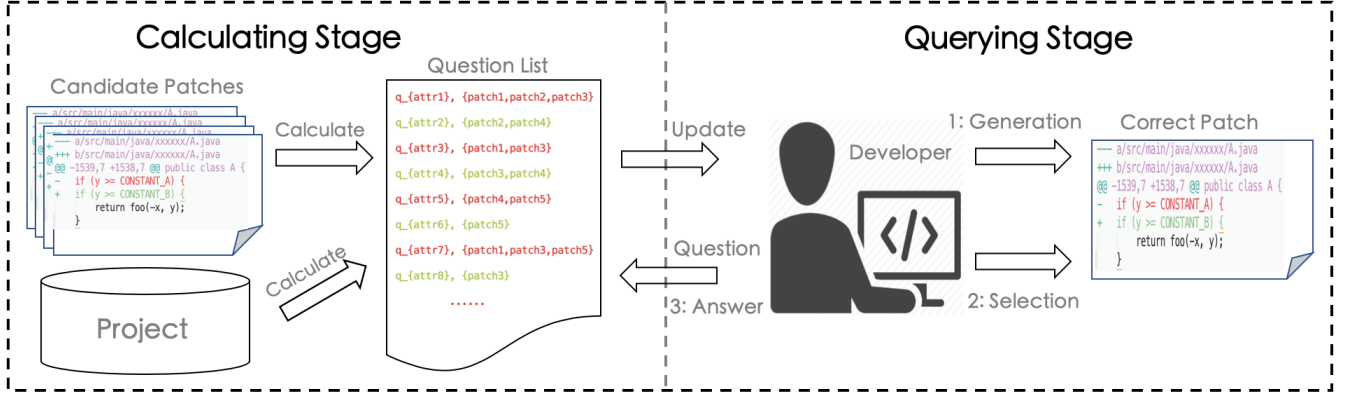
Figure 2: The workflow overview of the proposed approach.

set of patches that make the answer to question q_{attr} *yes*, i.e, after applying the patch in *patches*, the program attribute *attr* holds.

For instance, one interactive question for the example presented in Section 2.1 can be $\langle q_{attr1}, \{p_1, p_2\}\rangle$, where, q_{attr1} is "*Whether the statement at line 321 in method* eval() *should be covered?*". In this way, for each attribute, we can construct an interactive question, which will be used on the next interactive stage. In particular, we will delete *IQ*s whose *patches* includes all candidate patches ahead of time to reduce the number of questions.

2.3 Interactive Stage

As it is introduced above, the interactive stage is an online process with developers (shown in Figure 2) using the interactive questions constructed in the preparing stage. The input of this stage is a list of *IQ*s and the complete project under debugging. Each time, our approach collects the feedback from developers for some questions and update the candidate questions and patches in accordance. More concretely, in the interactive debugging process, there are in total three kinds of actions that a developer can take.

- **Answer** Answering an *IQ* to filter out some plausible but incorrect patches.
- **Selection** Selecting a patch from the candidates in *IQ*s as the correct patch.
- **Generation** Generating a correct patch by themselves to fix the bug.

The **Answer** action is the main procedure in the interactive querying process, which interactively refine the candidate patches with removing incorrect ones according to the answers of some question from developers. Algorithm 1 demonstrates the updating process for each round of interaction after developers answering a question. In this process, we maintain a list of *IQ*s (i.e., $\mathbb{Q}$) waiting for answers and a set of candidate patches (i.e., $\mathbb{P}$). In the beginning, $\mathbb{P}$ contains all candidate patches. In each round, we update these two parts according to the developer's answer. Particularly, when an attribute is correct (i.e., the answer is *yes* to the corresponding question), the patches related to it will form the new candidate patch set (lines 3-4), otherwise, they will be filtered from candidates (line 6). Finally, the candidate questions $\mathbb{Q}$ will be updated according

to remaining patches (lines 8-13). In particular, if all patches are filtered in an *IQ*, it will be deleted and not require developers to answer in the future (line 10).

Additionally, in the debugging process, developers may also check the correctness of candidate patches when $\mathbb{P}$ is not too large. Therefore, the other two actions denote that developers can *select* the correct ones directly from candidate patches or *generate* patches manually and finalize the debugging process.

Algorithm 1 Update Algorithm

Input: $\mathbb{Q}$: question list, $\mathbb{P}$: all candidate patches
        q: an answered question, a: answer to q
Output: $\mathbb{Q}'$: updated question list, $\mathbb{P}'$: updated patch list
1: $\mathbb{Q}' \leftarrow \emptyset$, $\mathbb{P}' \leftarrow \emptyset$
2: **if** $\mathbb{Q}! = \emptyset$ && $\mathbb{P}! = \emptyset$ **then**
3:      **if** $a ==$ yes **then**   // *q.attr* is correct
4:          $\mathbb{P}' \leftarrow q.patches$ // patches satisfy *q.attr*
5:      **else**
6:          $\mathbb{P}' \leftarrow \mathbb{P} \setminus q.patches$ // patches do not satisfy *q.attr*
7:      **end if**
8:      **for** each $q' \in \mathbb{Q}$ **do**   // update candidate questions
9:          $q'.patches \leftarrow q'.patches \cap \mathbb{P}'$
10:         **if** $q'.patches \neq \emptyset$ **then**
11:            $\mathbb{Q}' \leftarrow \mathbb{Q}' \cup \{q'\}$
12:         **end if**
13:      **end for**
14: **end if**

---

3 ECLIPSE PLUGIN

To evaluate the effectiveness of our approach, we have developed a prototype tool called *InPaFer*, which is a plugin program for Eclipse with a graphical user interface (GUI). Figure 3 shows a snapshot of the plugin during a debugging process. Specifically, it consists of two embedded views, *Query View* and *Diff View*, for collecting developers' feedback and displaying information to developers.

Query View is the main component of our approach, which presents the details of *interactive question*s and corresponding candidate patches. In order to separately display different kinds of

information, it is further subdivided into three panels. As shown in the figure, the first panel shows failing test cases and the number of candidate patches thus far. The second panel shows the details of questions that developers can selectively answer. As introduced in Section 2.2, currently we employed three kinds of attributes that corresponds to the three groups of questions in the view. For each question, it not only displays the attribute details, but also shows the number of related patches and the state the of question. When a question is answered (*Yes* or *No*), the state of the question will be updated from UNCLEAR to YES/NO. As a separate panel, we display the candidate patches when a question is selected. Additionally, the plugin also provides a one-click rollback to reset all the answers.

Diff View is an auxiliary view to visualize the differences of execution traces before and after applying a patch to the buggy program, where the green lines of code are commonly covered by the failing test before and after repair, while the red lines of code are particularly covered by one of them. Finally, the other lines of code are changed by the patch or not covered by any of them. In this way, the developers can clearly understand the impact of the patch on the program execution, and possibly feel easy to answer the questions.

Please note that all views or panels are logically interrelated to each other, the selection of one part may trigger the update of the display in other places. For example, when a patch is selected, the *Diff View* will refresh the trace difference immediately. Moreover, developers can locate the changed code in the editor by simply selecting a patch. Most importantly, when a question is answered, the candidate patches of all other questions will be updated according to Algorithm 1 and refreshed on the view.

4 EVALUATION

To evaluate the effectiveness of *InPaFer*, we have conducted two experiments. The first one is a simulation experiment, which investigates the effectiveness and efficiency of *InPaFer* when applied to a large number of real-world bugs. Besides, in this experiment, we also study the impacts of different kinds of *interactive questions*. The other experiment is a user study to evaluate the usefulness of *InPaFer* in a realistic program repair scenario. Specifically, we investigate whether it can improve the efficiency and correctness of developers when debugging.

4.1 Simulation Experiment

In this experiment, we designed the study to answer the following research questions:

- **RQ1**: How effective is *InPaFer* in debugging real-world bugs?
- **RQ2**: How effective are different kinds of *interactive questions* of *InPaFer*?

RQ1 investigates the effectiveness and efficiency of *InPaFer* via the number of remaining patches and queries. Ideally, all incorrect patches can be correctly filtered out and only correct patches (can be empty) are left after several rounds of queries. RQ2 compares the effectiveness of questions built with different kinds of constraints.

4.1.1 Experiment Setup.

Dataset. In order to simulate the scenario, where there are multiple patches for a bug, we can apply patches produced by multiple existing APR techniques. We consider all existing automatic program tools, which work on Java language and are available thus far. As a matter of fact, different APR techniques may produce similar or even the same patches, to improve the efficiency and clarity for interactive debugging, we will remove duplicate patches ahead of time, which are the same in syntax. In total, we selected 13 program repair tools and the details of each tool are presented in Table 2. All of them were evaluated on the commonly used Defects4J [13] benchmark and the results are available. We collected the patch data from previous studies [1, 12, 19, 23, 26, 39, 44, 45]. In total, we have collected 8654 patches for 85 bugs, and the details are listed in Table 1. In the table, the first two columns present the project names and line numbers of source code. Columns "**Bug**" and "**AvgPatch**" show the number of bugs in each project and the average number of patches for each bug. Finally, the column "**C/NC**" denotes the number of bugs that have or do not have correct patches collected.

Table 1: Dataset in experiments.

Project	kLoC	Bug	AvgPatch	C/NC
JFree**Chart**	96	17	225	9/8
Closure Compiler	90	13	6	3/10
Apache Commons **Lang**	22	13	66	8/5
Apache Commons **Math**	85	42	92	15/27
Total	321	85	101	35/50

Table 2: APR tools included by *InPaFer*.

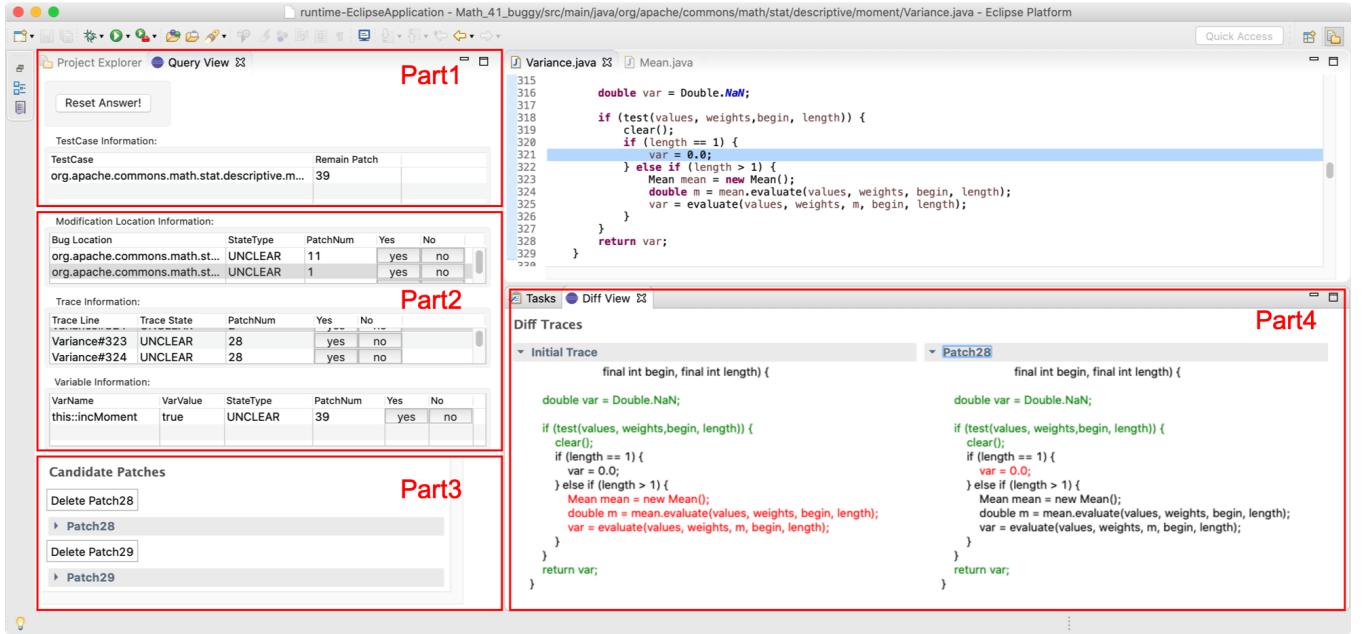| Name	Description
jKali	The Java implementation of Kali [30], which only performs functionality deletion.
jGenProg	The Java implementation of GenProg [20, 38], which repair bugs with genetic programming algorithm.
kPAR	The Java implementation of PAR [15], which generate patches based on predefined fix patterns.
Nopol [45]	Relying on constraint solving to fix incorrect conditions.
jMutRepair [26]	A mutation based program repair tool.
Cardumen [27]	Generating patches based on mined templates.
Avatar [23]	A repair tool based on the fix patterns of static analysis violations.
HDrepair [19]	A repair tool based on historical bug-fix information.
ACS [44]	Learning statistical information from open source programs for fixing incorrect conditions.
3sfix [7] CapGen [39] SimFix [12]	Repair approaches based on similar code match.
DeepRepair [40]	An extension of jGenProg, which leverages code similarity.

Figure 3: The screenshot of *InPaFer*.

Procedure. To automatically simulate the interaction process with developers, each time *InPaFer* randomly selects one question from all candidates and then automatically gets the answer via analyzing the fixed programs. Moreover, to alleviate the impact of randomness, we repeat the interaction process for each bug five times and take the mean number of queries as the final result.

4.1.2 Results for Remaining Patches and Query Number (RQ1). Table 3 shows the results of this research question. In the table, columns "**None**" and "**All Correct**" show the number of bugs, which do not have any candidate patch left and only have correct patches left, respectively. The following columns present the percentage ranges of patches left among all candidates. For example, "≤ 40%" denotes that the percentage is from 20% to 40%. Each cell shows the number of corresponding bugs. Particularly, we separately display the number of bugs that contain (**Con Bug**) and do not contain (**NotCon Bug**) correct patches. Finally, row "**Query Number**" shows the average number of queries.

From the table, our approach can correctly filter out all incorrect patches and only make the correct ones left for 49.4% (42/85) bugs in total. Particularly, when there is no correct patch, it can filter all incorrect patches for about 78% (29/50) bugs. Additionally, after analyzing these bugs we find that the number of candidate patches ranges from 2 to 1248, and on average more than 60 per bug, which may potentially cost a lot of time of developers to manually review. However, in this process, our approach on average only requires about 3.2 queries for each bug after filtering all of them. As explained in our user study (Section 4.2), the interactive debugging process will not cause a big burden to human developers and significantly improve the efficiency of manual review. On the other hand, when the correct patches are given, our approach can help correctly remove all incorrect patches while still save the correct

ones for about 37.1% bugs (13/35), and the number of queries is even smaller, i.e., three queries on average. The result indicates that our approach is effective for patch filtering.

> **Finding 1.** *For about 49% bugs, InPaFer can filter out all incorrect patches and save all correct patches within 3.1 questions on average.*

However, from the table we can see that there are still some incorrect patches that cannot be completely filtered out in the experiment. For example, there are 3.5% (3) bugs having less than 20% (or 20%-40%) candidate patches left. However, the queries needed are still not too many, usually less than six per bug apart from the one which needs 20 queries (in ≤ 20%). The main reason is the candidate patches are semantically too similar to each other, causing the program executions the same. For example, Figure 4 shows two candidate patches, where the first one (left) is the correct patch while the other is incorrect. However, both these two patches change the `if` condition (line 679) and have the same execution path, making our approach cannot better distinguish them. However, more attributes can be added to further improve the effectiveness of our approach. Additionally, even though some patches cannot be filtered out by our approach, they are possibly easy for developers to review (e.g., the incorrect patch in Figure 4 compares two constant values). As we will explain in the user study (Section 4.2), even if 12/26 (>40%) candidate patches left in Task1, our approach can still improve the efficiency of manual review process.

Finally, for about 23.5% (20/85) bugs, *InPaFer* cannot generate any questions to filter candidate patches. We further reviewed these bugs and found the reason is that most of the patches are modified the same location and have similar program attributes,

```
//correct patch          //incorrect patch
679- if(dataset!=null){   679- if(dataset!=null){
679+ if(dataset==null){   679+ if(AbsRenderer.ZERO==null){
680    return result;     680    return result;
681  }                    681  }
```

Figure 4: Example candidate patches left after filtering.

Table 3: Remaining patches in bugs.

Remain Patches	None	All Correct	≤ 20%	≤ 40%	> 40%	= 100%	Total
Con Bug	0	13	2	1	13	6	35
NotCon Bug	29	0	1	2	4	14	50
Total Bug	29	13	3	3	17	20	85
Query Number	3.2	3.0	9.8	1.7	2.2	–	3.1

which cannot be better distinguished by the current implementation of *InPaFer*. In fact, 16 out of the 20 bugs contain less than 5 candidate patches. When the number of patches is small, it may be easy for the developer to review.

4.1.3 Results for Different Questions (RQ2). To investigate the effectiveness of different attributes in our approach, we conduct a controlled experiment, where each time we apply questions based on only one kind of attribute. In addition, as explained above that the current attributes do not have sufficient ability to distinguish all candidate patches. In the comparison, we only focus on the patches which can be filtered by *InPaFer*.

Figure 5 shows the experimental results when applying different attributes. The x-axis denotes the number of queries while the y-axis denotes the percentage of remaining patches. From the figure, the performances of different kinds of attributes vary greatly, and there is no such attribute that can filter all incorrect patches. The reason is incorrect patches tend to be similar to each other on a certain attribute while different on some other attributes. For example, the attribute of `Variable Value` can at most distinguish about 82% incorrect patches. After combining other two attributes, it can filter up to 97% incorrect patches in ten queries.

In addition, the performance when using `Variable Value` is even better than that using all attributes within three queries. This is because of the randomness in the simulation process. However, though `Variable Value` is effective, developers seldom use it in practice based on the developersâĂŹ feedback, since it is usually hard to answer. On the contrary, the questions related to the attribute of `Modified Method` is the most frequently selected as it is easy to answer and performs relatively well in all cases.

Nevertheless, the results show that the number of queries is usually no larger than 6 to achieve the best performance for each attribute. Besides, usually the first two queries contribute most.

> **Finding 2.** *The performance of* `Variable Value` *is better than the performance of other two attributes, but it is worse than that of combining all attributes.*
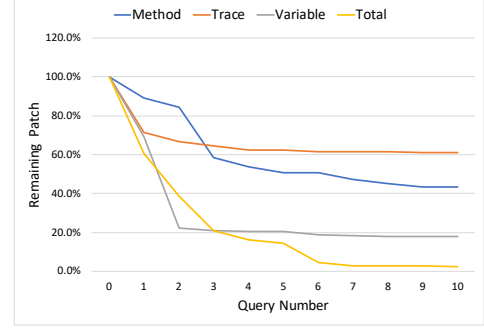

Figure 5: Remaining patches and query numbers.

4.2 User Study

Since *InPaFer* was designed as an interactive debugging approach, to investigate whether it helps developers in practice, we further conducted a user-in-the-loop study, which focused on the following research questions:

- **RQ3**: How do developers perform while debugging with *InPaFer*?
- **RQ4**: What is the feedback of developers after using *InPaFer*?

To investigate the impacts of our approach to developers' debugging performance in practice, we configured three debugging settings: ManuallyFix, FixWithPatches and FixWith*InPaFer*. ManuallyFix and FixWithPatches denote that the developers manually repair bugs without and with patches, which are produced by the 13 APR tools used in our paper, respectively, while FixWith*InPaFer* denotes that the developers repair bugs with the assistance of *InPaFer*. Additionally, after the experiment, we have interviewed all developers and collected their feedbacks for further analysis.

4.2.1 Study Design. We will introduce our study design from three aspects: *Tasks*, *Participants* and *Procedure*.

Tasks. We selected four bugs as debugging tasks that are not straightforward to repair. Additionally, we also considered the diversity of programs, i.e. from different projects, which perform different types of tasks. Table 4 shows the task details, including the bug id and patch number for each bug, etc. All bugs are from the Defects4J dataset shown in Table 1. Particularly, Task1 and Task2 contain the correct patches for the bugs while Task3 and Task 4 do not. Besides, the participants would not be informed whether the candidate patch set contains a correct patch.

Table 4: Tasks in user study.

Task ID	Bug ID	Patch	Correct Patch	Query Number	Remain Patch
Task1	Chart9	26	3	3	12
Task2	Math41	48	1	6	1
Task3	Lang14	8	0	2.4	0
Task4	Lang22	24	0	2.4	0

Note: In this table, column **"Patch"** and column **"Correct Patch"** denote the number of original patches and the number of correct patches in original patches, respectively, and column **"Remain Patch"** denotes the number of remaining patch after all questions are answered.

Participants. In total, we recruited 30 participants to conduct our user study. They are all students who majored in computer science from our department. Besides, they have at least three-year programming experience and are familiar with debugging in Eclipse. Additionally, the participants have no prior experience of repairing those bugs in the study.

Procedure. In the study, participants were evenly divided into three separate groups (i.e., A, B and C) with each group including 10 participants. Each participant would finish four tasks in the corresponding group as shown in Table 5. For example, the participants in Group A should manually repair the bugs in Task1 and Task3 and manually repair bugs in Task2 and Task4 with the help of patches. As a result, from the table, each participant would finish all four tasks under two different debugging scenarios. Therefore, our study consists of 120 (30 × 4) individual debugging processes and each debugging process is called one *debugging session.*

Table 5: Groups in user study.

Group	ManuallyFix	FixWithPatches	FixWith*InPaFer*
Group A	Task1+3	Task2+4	–
Group B	Task2+4	–	Task1+3
Group C	–	Task1+3	Task2+4

In addition, to make the participants familiar with our tool, before the formal user study, participants in Group B and C were required to debug an irrelevant bug using *InPaFer* until they got familiar with *InPaFer*. Since there are too many debugging sessions, we assigned each session 30 minutes. If the participants cannot finish the debugging within the given time slot, we considered the bug failed to be repaired. After the participants finished a session, we would manually check whether the patch, figured out by the developers, was right.

After finishing the debugging, we interviewed each participant and collected their feedbacks. In specific, for each group, we carried out the interview in terms of the question: *what was the difference between the two settings you have experienced?* for the participants who have debugged with *InPaFer*, we would ask more questions about the function of *InPaFer*, such as, (i) *Which kinds of attribute related questions were most useful in InPaFer?* (ii) *Was Diff View useful for debugging?*

4.2.2 Results for Repaired Bugs and Repair Time (RQ3). To measure the performance of developers, we consider both the number of debugging sessions where the bugs were correctly repaired and the time used during debugging.

Table 6 shows the number of sessions in which the bugs were successfully repaired in three different debugging scenarios. From the table, our approach (FixWith*InPaFer*) significantly outperformed ManuallyFix and FixWithPatches with respectively 62.5% and 39.3% improvements. Especially, when the correct patches existed in the candidate patches, developers could always fix the bug in the study. However, it was not the case for FixWithPatches even though the same patches were given. The result demonstrated the effectiveness of our approach.

Additionally, when considering the debugging time, our approach achieved better performance as well. Particularly, it could significantly shorten the debugging time of developers in all tasks compared with FixWithPatches and in three out of four tasks compared with ManuallyFix. Overall, our approach could reduce the debugging time by 25.3% and 28.0% against the other two, respectively. Therefore, our approach can improve the efficiency of human debugging.

> **Finding 3.** *Overall, FixWithInPaFer can reduce the debugging time by 25.3% and 28.0% on average, and increase the success rate by 62.5% and 39.3% on average, compared to ManuallyFix and FixWithPatches, respectively.*

Table 6: Number of successful debugging sessions in user study.

Task ID	ManuallyFix	FixWithPatches	FixWith*InPaFer*
Task1	1	9	10
Task2	8	6	10
Task3	5	8	10
Task4	10	5	9
Total	**24**	**28**	**39**

From Table 6 and Figure 6, we observed that the relative performances of the developers in three debugging settings were different in four tasks. Specifically, in Task1 and Task3, FixWithPatches had more successful debugging sessions but less debugging time compared with ManuallyFix, while it would reverse in Task2 and Task4. Therefore, we further investigated the reasons for this difference. Based on our data, we suspected that they were mainly due to the number and quality of candidate patches. More concretely, we make the following observations from the data:

(1) On the one hand, as shown in Table 4, Task1 contains 3 correct patches in 26 candidate patches, while Task2 contains only 1 correct in 48 patches. Although both Task3 and Task4 do not contain correct patches, Task3 contains only 8 incorrect patches, while Task4 contains 24 incorrect patches. This observation suggests that the number of incorrect patches is negatively related to the repair performance.

(2) On the other hand, we found that almost all candidate patches in Task1 exactly changed the faulty code, even if the patches are incorrect, and the candidate patches in Task3 provided partially correct code. The fault location and referable code potentially could provide guidance for developers to better understand the bugs. In contrast, the candidate patches in Task2 changed the code in different locations, and the candidate patches in Task4 provided meaningless code. These may mislead developers. In summary, different incorrect patches may have different quality and high-quality incorrect patches still guide the developers.
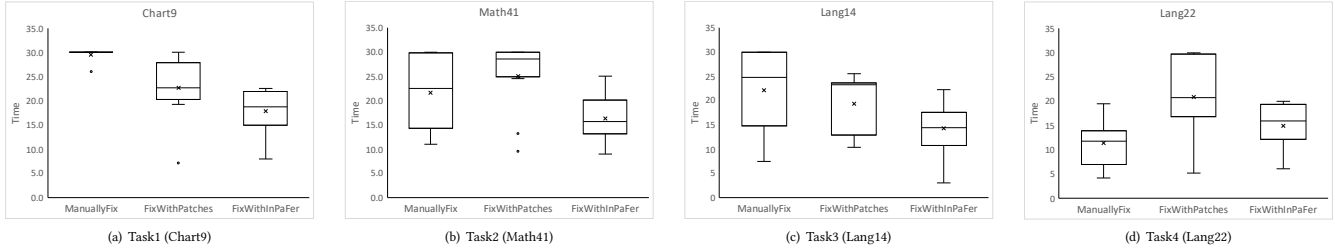
Figure 6: Debugging time in user study.

Finding 4. *High-quality incorrect patches with partial correct code at faulty location can still be helpful to developers.*

In addition, in Task1 and Task3, FixWith*InPaFer* shows small improvement than FixWithPatches, while it shows greater improvement than FixWithPatches in Task2 and Task4. This suggests that *InPaFer* reduces the negative effect of the low-quality incorrect patches, and this reduction is more significant when the negative effect is larger.

Finding 5. *The number and quality of incorrect patches affect the debugging performance of developers when they are provided with patches, and InPaFer helps reduce the negative effect from low-quality incorrect patches.*

Besides, for Task1, Task2 and Task3, the performances of developers in FixWith*InPaFer* are significantly better than that in ManuallyFix, while the performance of developers in FixWith*InPaFer* is a little worse than that in ManuallyFix. This suggests that when containing correct or incorrect but high-quality patches, *InPaFer* could significantly improve the performance of developers. On the other hand, according to the result of Task4, when providing patches with low quality, *InPaFer* will not affect developers' performance too much and still can improve the manual patch review.

4.2.3 Results for Feedback (RQ4). To better understand the debugging process and the attitude of developers to our approach in practice, we conducted an interview with all participants after the experiments. The details are listed below.

Group C: compare FixWith*InPaFer* and FixWithPatches
All participants of Group C gave a positive feedback on *InPaFer* when repairing Task2. Based on the interview, we summarized the help from the following two aspects. (i) *Patch Filtering*. Most participants said that *InPaFer* could help them to filter most of incorrect patches by answering a few questions, like *"It can narrow the range of correct patches after a few simple questions."*. Therefore, they only needed to check several candidate patches. (ii) *Bug Understanding*. Since the developers were not familiar with the bugs they were debugging, it was hard to identify whether a given patch is correct or not. However, for Task2, the interactive questions were not hard to answer, and they provided a clue to help the developers to understand the bugs step by step. Finally, they could understand the bugs better and select the correct patches easier.

Besides, for Task4, the participants thought that *InPaFer* only help them to filter out all incorrect patches, but it cannot help them to repair the bugs, because the incorrect patches are meaningless code, which cannot provide other help.

Group B: compare FixWith*InPaFer* and ManuallyFix
The participants in Group B indicated that *InPaFer* could provide the correct or partially correct patches, which provided guideline for debugging. Specifically, for Task1, they could find the correct patch after answering only a few questions. Besides, although Task3 does not contain the correct patch, the partially correct patches also help them a lot.

The participants also referred to a limitation that *InPaFer* does not support single answer cancellation. Specifically, the developers may misunderstand Task1 and select a wrong answer for some questions. This would lead to the correct patches wrongly filtered. When the developers realized that they selected a wrong answer, they need to reset all questions and restart the interaction again, which would waste the debugging time. This is a limitation of our current tool implementation but not our approach. A better tool implementation could make the answer to a question reversible, and potentially further boost the performance of *InPaFer*.

Group A: compare FixWithPatches and ManuallyFix
On the one hand, small part of participants thought that candidate patches could provide the *Fault Location* for Task4, which could provide guidance when debugging. On the other hand, almost all participants complained that it was difficult to find the correct patch as there are too many patches to review when repairing Task2. Reviewing many incorrect patches would disturb their debugging.

Finding 6. *Answering the questions of InPaFer helps filter out incorrect patches as well as understanding the bug.*

For all participants who fixed bugs with *InPaFer*, we carried out the interview in terms of the following questions:

(1) *Which kind of attributes related questions were most useful in InPaFer?* Though we have already compared the impacts of different kinds of attributes related questions in the study, the result is still unclear to us from the developers' perspective. In the interview, almost all participants regarded that the questions related to Execution Trace helped them most. Particularly, a developer explained that *"developers know where the programs should execute, but they don't know what's wrong. InPaFer knows all executed location, but it does not know whether the execution flow is correct or not. It is very helpful to combine these two kinds of information."* Besides,

a small number of participants also agreed that `Modified Method` was also somehow helpful. They said that when the method was easy to understand, the question related to `Modified Method` could be helpful, otherwise, it would be hard to answer this question as a much deeper understand of the bug was needed. However, they suggested that it would be more useful to developers who was familiar with the project. Finally, the questions related to `Variable Value` were the least to be selected as useful, because methods are often invoked multiple times during an execution, and the current question type does not allow us to locate a specific invocation.

(2) *Was Diff View useful for debugging?* About a third of participants thought that it was useful to understand the bug and correct the misunderstanding. For example, one participant said that, *"The Diff View shows which branch that the execution get into makes the test case pass. It corrects the previous misunderstanding."*

> **Finding 7.** *In our user study, the questions related to* `Execution Trace` *helped the developers most, while* `Variable Value` *could be improved by distinguishing different invocations.*

5 THREATS TO VALIDITY

The *internal* threat to validity lies in the recruited participants. On the one hand, all of them are not familiar with the projects, which may cost them more time to understand the program. However, our findings are from the comparison of debugging time in three groups, while the absolute total debugging time will not affect. On the other hand, to mitigate the bias from grouping, we randomly grouped participants according to their programming experience, and made each group of participants have close debugging capability in terms of developing years, i.e., one average about 6 years of developing experience each group. Besides, the participants from two groups finished four tasks for in debugging setting, which can alleviate the threats from participants.

The *external* threat to validity lies in the four tasks used in user study, which may cause the findings may be not generalizable to all other cases. To mitigate this threat, we selected the tasks from different projects, covering different types. Additionally, we evenly distribute participants over different projects, which can mitigate the impact of projects to debugging process.

6 RELATED WORK

6.1 Interactive Repair

Some recent studies tried to involve developers in the program repair process. Cashin et al. [6] proposed PATCHPART, which clusters a set of generated patches by program invariants. Patches in a cluster are likely to be all correct or all incorrect, such that the developers ideally need only examine one patch per cluster. Compared with their approach, our approach actively asks questions rather than clusters the patches. We also present an empirical study showing that this process could increase the repair performance of developers.

Böhme et al. [3] introduced LEARN2FIX, which queries the developers to build a test oracle before patch generation to overcome

overfitting. Compared with them, our study focuses on the patch review process after patch generation, and give evidences that our approach boosts the overall repair performance of developers.

6.2 Interactive Debugging

A lot of interactive debugging techniques [4, 11, 16–18, 21, 22] leverage user feedback to localize fault. Algorithm debugging [4] initially was proposed to resolve the functional programming debugging problem. Algorithm debugging first builds a debugging tree to reflect the method invocation for a failed test. Then, it repeatedly asks the developer to answer whether the input-output is correct for a method invocation in the debugging tree, and prunes the tree based on the answer until the fault is localized. Li et al. [21] improved algorithm debugging by leveraging spectrum based fault localization (SBFL) and dynamic dependences to decide the order of method invocations to be questioned. Similarly, Gong et al. [11] proposed an interactive fault localization technique to improve SBFL by asking developers to label the statements as faulty or clean, and updating the suspicious statement list.

Ko and Myers proposed Whyline [16–18], which first records the program execution trace and allows developers to select some questions about program output. Whyline can give possible explanations according to the dynamic slicing until the developer finds the root cause of the fault. Lin et al. proposed Microbat [22] to improve Whyline by allowing developers to select the trace execution pattern, such as *Correct Step, Wrong Variable Value*, and using developer's feedback to recommend some suspicious trace.

Different from these techniques, our approach aims to help patch review by interactively patch filtering, instead of improving the fault localization. Besides, our approach provided the differences among patched programs, while others do not.

6.3 Patch Correctness Identification

Since the weak test suites, a lot of researches focus on automatically identifying the correctness of patches. Some approaches adopt a deterministic way. Xin and Reiss [41] assume that there is an oracle, which can give the corresponding output for a newly generated input. If the output produced by the patched program violates the output produced by the oracle, the patch is incorrect. Yang et al. [46] identify the correctness of patches by generating new test cases, which can obviously validate the oracle, such as crash and memory leak problems. Similarly, Gao et al. [9] use crash-freedom as the oracle to discard patches, which crash on the new tests.

Other approaches adopt a heuristic way to identify correctness of patches. Tan et al. [35] use anti-patterns to discard the patches which accord with the pre-defined patterns. Xiong et al. [43] determine the patches correctness by the behavior similarity of test case executions. Besides, Yu et al. [47] indicate that test case generation can filter a part of incorrect patches, but cannot turn incorrect patches into correct ones.

Different from these approaches, our approach identifies the patch correctness by the developers. We adapted an interaction with the developers by asking questions related to the attributes of different patched programs. The developers can understand the bug and figure out the correct patches through the interaction.

6.4 Effect of Patches

Tao et al. [37] have investigated the effect of automatic patch generation in realistic debugging scenarios. They observed that high-quality patches significantly improve debugging correctness while low-quality patches influence participantsâĂŹ debugging correctness. Our user study also has a similar observation that a small number of high quality candidate patches could improve the performance of manual debugging.

Cambronero et al. [5] conducted a similar experiment except that the developers were provided with five candidate patches. The results show that when given candidate patches, the efficiency and correctness of developers did not be improved. Different from their observation, our study found that providing a large number of low quality patches would affect the performance of developers.

7 CONCLUSION

In this paper, we proposed an interactive patch filtering approach, which contains a two-stage algorithm, to provide tool support for patch review. We also implemented our approach as an Eclipse plugin called *InPaFer*. The evaluation results show that our tool would significantly boost the repair performance of developers when the patch set contains high-quality patches, and would not significantly reduce the repair performance even when the patches are all of low-quality. These findings give many implications to future repair tool building, for example, (1) the repair tool of low precision can still useful, as the interactive process helps filter out incorrect patches without significantly affecting the repair performance; (2) a generated patch does not have to be fully correct to be useful, as incorrect patches could also provide good hints such as repair location and partial repair to the developer.

REFERENCES

[1] [n.d.]. https://github.com/SerVal-DTF/FL-VS-APR/tree/master/kPAR.
[2] Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. 2019. Getafix: Learning to Fix Bugs Automatically. *Proc. ACM Program. Lang.* 3 (10 2019), 27. https://doi.org/10.1145/3360585
[3] Marcel Böhme, Charaka Geethal, and Van-Thuan Pham. 2019. Human-In-The-Loop Automatic Program Repair. *CoRR* abs/1912.07758 (2019).
[4] Rafael Caballero, Adrián Riesco, and Josep Silva. 2017. A Survey of Algorithmic Debugging. *ACM Comput. Surv.* (2017), 60:1–60:35. https://doi.org/10.1145/3106740
[5] José Pablo Cambronero, Jiasi Shen, Jürgen Cito, Elena Glassman, and Martin Rinard. 2019. Characterizing Developer Use of Automatically Generated Patches. In *2019 IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2019, Memphis, Tennessee, USA, October 14-18, 2019*, Justin Smith, Christopher Bogart, Judith Good, and Scott D. Fleming (Eds.). IEEE Computer Society, 181–185.
[6] Padraic Cashin, Carianne Martinez, Westley Weimer, and Stephanie Forrest. 2019. Understanding Automatically-Generated Patches Through Symbolic Invariant Differences. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*. IEEE, 411–414.
[7] Zimin Chen and Martin Monperrus. [n.d.]. The Remarkable Role of Similarity in Redundancy-based Program Repair. ([n. d.]).
[8] Zachary P. Fry, Bryan Landau, and Westley Weimer. 2012. A human study of patch maintainability. In *International Symposium on Software Testing and Analysis, ISSTA 2012, Minneapolis, MN, USA, July 15-20, 2012*, Mats Per Erik Heimdahl and Zhendong Su (Eds.). ACM, 177–187.
[9] Xiang Gao, Sergey Mechtaev, and Abhik Roychoudhury. 2019. Crash-avoiding program repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019*, Dongmei Zhang and Anders Møller (Eds.). ACM, 8–18.
[10] Luca Gazzola, Daniela Micucci, and Leonardo Mariani. 2017. Automatic Software Repair: A Survey. *TSE* PP, 99 (2017), 1–1. https://doi.org/10.1109/TSE.2017.2755013
[11] Liang Gong, David Lo, Lingxiao Jiang, and Hongyu Zhang. 2012. Interactive fault localization leveraging simple user feedback. In *28th IEEE International Conference on Software Maintenance, ICSM 2012, Trento, Italy, September 23-28, 2012*. IEEE Computer Society, 67–76.
[12] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. 2018. Shaping Program Repair Space with Existing Patches and Similar Code. In *ISSTA*.
[13] René Just, Darioush Jalali, and Michael D Ernst. 2014. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *ISSTA*. 437–440.
[14] Y. Ke, K. T. Stolee, C. L. Goues, and Y. Brun. 2015. Repairing Programs with Semantic Code Search (T). In *ASE*. 295–306. https://doi.org/10.1109/ASE.2015.60
[15] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic patch generation learned from human-written patches. In *ICSE*. 802–811.
[16] A. J. Ko and Brad A. Myers. 2004. Designing the whyline: a debugging interface for asking questions about program behavior. In *Proceedings of the 2004 Conference on Human Factors in Computing Systems, CHI 2004, Vienna, Austria, April 24 - 29, 2004*, Elizabeth Dykstra-Erickson and Manfred Tscheligi (Eds.). ACM, 151–158.
[17] A. J. Ko and Brad A. Myers. 2008. Debugging reinvented: asking and answering why and why not questions about program behavior. In *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008*, Wilhelm Schäfer, Matthew B. Dwyer, and Volker Gruhn (Eds.). ACM, 301–310.
[18] A. J. Ko and Brad A. Myers. 2009. Finding causes of program output with the Java Whyline. In *Proceedings of the 27th International Conference on Human Factors in Computing Systems, CHI 2009, Boston, MA, USA, April 4-9, 2009*, Dan R. Olsen Jr., Richard B. Arthur, Ken Hinckley, Meredith Ringel Morris, Scott E. Hudson, and Saul Greenberg (Eds.). ACM, 1569–1578.
[19] Xuan-Bach D Le, David Lo, and Claire Le Goues. 2016. History Driven Program Repair. In *SANER*. 213–224. https://doi.org/10.1109/SANER.2016.76
[20] C. Le Goues, ThanhVu Nguyen, S. Forrest, and W. Weimer. 2012. GenProg: A Generic Method for Automatic Software Repair. *TSE* 38, 1 (Jan 2012), 54–72.
[21] Xiangyu Li, Shaowei Zhu, Marcelo d'Amorim, and Alessandro Orso. 2018. Enlightened debugging. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman (Eds.). ACM, 82–92.
[22] Yun Lin, Jun Sun, Yinxing Xue, Yang Liu, and Jin Song Dong. 2017. Feedback-based debugging. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, Sebastián Uchitel, Alessandro Orso, and Martin P. Robillard (Eds.). IEEE / ACM, 393–403.
[23] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. 2019. AVATAR: Fixing Semantic Bugs with Fix Patterns of Static Analysis Violations. In *Proceedings of the 26th IEEE International Conference on Software Analysis, Evolution, and Reengineering*. IEEE, 456–467.
[24] Fan Long, Peter Amidon, and Martin Rinard. 2017. Automatic Inference of Code Transforms for Patch Generation. In *ESEC/FSE*. 727–739. https://doi.org/10.1145/3106237.3106253
[25] Fan Long and Martin Rinard. 2016. An Analysis of the Search Spaces for Generate and Validate Patch Generation Systems. In *ICSE*. 702–713. https://doi.org/10.1145/2884781.2884872
[26] Matias Martinez and Martin Monperrus. 2016. ASTOR: A Program Repair Library for Java. In *Proceedings of ISSTA*. https://doi.org/10.1145/2931037.2948705
[27] Matias Martinez and Martin Monperrus. 2018. Ultra-Large Repair Search Space with Automatically Mined Templates: The Cardumen Mode of Astor. In *Search-Based Software Engineering - 10th International Symposium, SSBSE 2018, Montpellier, France, September 8-9, 2018, Proceedings (Lecture Notes in Computer Science)*, Thelma Elita Colanzi and Phil McMinn (Eds.), Vol. 11036. Springer, 65–86.
[28] Sergey Mechtaev, Manh-Dung Nguyen, Yannic Noller, Lars Grunske, and Abhik Roychoudhury. 2018. Semantic program repair using a reference implementation. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman (Eds.). ACM, 129–139. https://doi.org/10.1145/3180155.3180247
[29] Martin Monperrus. 2017. *Automatic Software Repair: a Bibliography*. Technical Report. 1–24 pages. https://doi.org/10.1145/3105906
[30] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. 2015. An Analysis of Patch Plausibility and Correctness for Generate-and-validate Patch Generation Systems *(ISSTA)*. 24–36.
[31] Abhik Roychoudhury and Yingfei Xiong. 2019. Automated program repair: a step towards software automation. *Sci. China Inf. Sci.* 62, 10 (2019), 200103:1–200103:3. https://doi.org/10.1007/s11432-019-9947-6
[32] Ripon K. Saha, Yingjun Lyu, Hiroaki Yoshida, and Mukul R. Prasad. 2017. ELIXIR: Effective Object Oriented Program Repair. In *ASE*. IEEE Press. http://dl.acm.org/citation.cfm?id=3155562.3155643
[33] Seemanta Saha, Ripon K. Saha, and Mukul R. Prasad. 2019. Harnessing evolution for multi-hunk program repair. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, Joanne M. Atlee, Tevfik Bultan, and Jon Whittle (Eds.). IEEE / ACM, 13–24.
[34] Edward K Smith, Earl T Barr, Claire Le Goues, and Yuriy Brun. 2015. Is the cure worse than the disease? overfitting in automated program repair. In *FSE*.

532–543.

[35] Shin Hwei Tan, Hiroaki Yoshida, Mukul R Prasad, and Abhik Roychoudhury. 2016. Anti-patterns in Search-Based Program Repair. In *FSE*. https://doi.org/10.1145/2950290.2950295

[36] Yida Tao, Jindae Kim, Sunghun Kim, and Chang Xu. 2014. Automatically Generated Patches As Debugging Aids: A Human Study. In *FSE*. 64–74.

[37] Yida Tao, Jindae Kim, Sunghun Kim, and Chang Xu. 2014. Automatically generated patches as debugging aids: a human study. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, Shing-Chi Cheung, Alessandro Orso, and Margaret-Anne D. Storey (Eds.). ACM, 64–74.

[38] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically finding patches using genetic programming. In *ICSE*. 364–374. https://doi.org/10.1109/ICSE.2009.5070536

[39] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. 2018. Context-Aware Patch Generation for Better Automated Program Repair. In *ICSE*.

[40] M. White, M. Tufano, M. Martinez, M. Monperrus, and D. Poshyvanyk. 2017. Sorting and Transforming Program Repair Ingredients via Deep Learning Code Similarities. *ArXiv e-prints* (July 2017). arXiv:cs.SE/1707.04742

[41] Qi Xin and Steven Reiss. 2017. Identifying Test-Suite-Overfitted Patches through Test Case Generation. In *ISSTA*. 226–236. https://doi.org/10.1145/3092703.3092718

[42] Qi Xin and Steven P. Reiss. 2017. Leveraging Syntax-related Code for Automated Program Repair *(ASE)*. http://dl.acm.org/citation.cfm?id=3155562.3155644

[43] Yingfei Xiong, Xinyuan Liu, Muhan Zeng andz Lu Zhang, and Gang Huang. 2018. Identifying patch correctness in test-based program repair. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman (Eds.). ACM, 789–799.

[44] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, and Lu Zhang. 2017. Precise Condition Synthesis for Program Repair. In *ICSE*. https://doi.org/10.1109/ICSE.2017.45

[45] Jifeng Xuan, Matias Martinez, Favio Demarco, Maxime Clément, Sebastian Lamelas, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. 2017. Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs. *TSE* (2017).

[46] Jinqiu Yang, Alexey Zhikhartsev, Yuefei Liu, and Lin Tan. 2017. Better Test Cases for Better Automated Program Repair. In *FSE*. 831–841. https://doi.org/10.1145/3106237.3106274

[47] Zhongxing Yu, Matias Martinez, Benjamin Danglot, Thomas Durieux, and Martin Monperrus. 2017. Test Case Generation for Program Repair: A Study of Feasibility and Effectiveness. *CoRR* abs/1703.00198 (2017).