# A Bayesian Framework for Automated Debugging

Sungmin Kang*
sungmin.kang@kaist.ac.kr
KAIST
Daejeon, South Korea

Wonkeun Choi*
anders@kaist.ac.kr
KAIST
Daejeon, South Korea

Shin Yoo
shin.yoo@kaist.ac.kr
KAIST
Daejeon, South Korea

## ABSTRACT

Debugging takes up a significant portion of developer time. As a result, automated debugging techniques including Fault Localization (FL) and Automated Program Repair (APR) have garnered significant attention due to their potential to aid developers in debugging tasks. With the recent advance in techniques that treat the two tasks as closely coupled, such as Unified Debugging, a framework to formally express these two tasks together would heighten our understanding of automated debugging and provide a way to formally analyze techniques and approaches. To this end, we propose a Bayesian framework of understanding automated debugging. We find that the Bayesian framework, along with a concrete statement of the objective of automated debugging, can recover maximal fault localization formulae from prior work, as well as analyze existing APR techniques and their underlying assumptions. As a means of empirically demonstrating our framework, we further propose BAPP, a Bayesian Patch Prioritization technique that incorporates intermediate program values to analyze likely patch locations and repair actions, with its core equations being derived by our Bayesian framework. We find that incorporating program values allows BAPP to identify correct patches more precisely: the rankings produced by BAPP reduced the number of required patch evaluations by 68% and consequently reduced the repair time by 34 minutes on average. Further, our Bayesian framework suggests a number of changes to the way fault localization information is used in program repair, which we validate is useful for BAPP. These results highlight the potential of value-cognizant automated debugging techniques, and further verifies our theoretical framework.

## CCS CONCEPTS

• **Software and its engineering → Software creation and management**.

## KEYWORDS

automated program repair, fault localization, automated debugging, bayesian statistics

*Both authors contributed equally to this research.


*ISSTA '23, July 17–21, 2023, Seattle, WA, USA*

ACM ISBN 979-8-4007-0221-1/23/07…$15.00
https://doi.org/10.1145/3597926.3598103

**ACM Reference Format:**
Sungmin Kang, Wonkeun Choi, and Shin Yoo. 2023. A Bayesian Framework for Automated Debugging. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '23), July 17–21, 2023, Seattle, WA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3597926.3598103

## 1 INTRODUCTION

Debugging plays a crucial role in the development process, as it is difficult to write a correct program on the first attempt, particularly when the program is large. As a result, significant resources are spent on debugging: Tassey notes that manual debugging can be tedious and demanding [33]. To aid developers in the debugging process, automated debugging tasks such as Fault Localization (FL) [10] or Automated Program Repair (APR) [15] were proposed to reduce developer burden when debugging issues. These techniques have matured enough to be applied in corporations and actively help developers [12, 23].

While FL and APR have been studied as separate research topics, they are in fact closely coupled activities. Most APR techniques require the guidance of an FL technique so that they can focus their repair efforts in specific locations within the program [18]. Some of the existing Mutation-Based Fault Localization (MBFL) techniques rely on discovering partial fixes to localize faults [9, 26]. More recently, Unified Debugging [5, 21] uses information from APR to refine and enhance FL accuracy. These efforts raise the following question: can we fuse FL and APR under a single framework to express them using a common language, and to derive future techniques?

This paper presents a novel Bayesian framework of automated debugging that incorporates both FL and APR. We posit that the purpose of automated debugging techniques is ultimately to jointly infer the likely location and repair action for a fix, and suggest that, by using Bayes' theorem, our framework can map probabilistic terms to well-known automated debugging concepts such as FL or APR. To firmly establish how Bayes' theorem can be useful in analyzing automated debugging techniques, we perform a number of theoretical analyses of existing literature. We start by looking at a popular family of fault localization techniques: spectrum-based fault localization (SBFL). Using our framework, we derive SBFL formulae based on a minimal set of assumptions and find that the resulting SBFL formulae are equivalent to the 'maximal' SBFL formulae as proven by Yoo et al. [41]. In addition, we analyze the behavior of well-known APR techniques, and find that they can also be expressed within our Bayesian framework; furthermore, our framework can analyze recent Unified Debugging [22] techniques which combine FL and APR, as well as provide concrete suggestions.

As a way of empirically testing our framework, we choose an important problem for APR techniques, patch prioritization, and

derive a solution from our framework. For many generate-and-validate APR techniques that generate a large number of patches, prioritizing the validation of promising patches is a key factor for better performance [14]. Inspired by a recent work that shows how humans heavily use program values during debugging [6], we propose BAPP (Bayesian Automated Patch Prioritization), a patch prioritization technique that incorporates program values. The core intuition is that patches must lead to program behavior change (i.e., a change in variable value or control flow) in failing tests while it is unlikely yet possible that they lead to behavior change in passing tests. That is, when comparing the value of the original expression and the new expression after the patch, there should be a difference. For example, when fixing control-flow statements, a change in the conditional expression *value* would lead to a change in path executed. As an empirical means of verifying our framework, we express the above principle in the language of our Bayesian framework, and directly derive precise expressions that are used by BAPP to rank patches, which at times deviates from APR practice. Using kPAR [18] as the APR tool, we empirically evaluate whether our approach can identify patches that pass regression tests efficiently.

Our results indicate that the incorporation of program states in ranking the patches successfully increased the efficiency from the original kPAR approach, with the median reduction of the plausible patch rank measured at 68%. Execution time also saw a significant improvement, resulting in an average reduction of 34 minutes. A new scheme for generating an FL ranking of suspicious locations with the consideration of program states was also shown to perform better than SBFL in finding the true buggy line; BAPP improved the identification of the buggy line within five attempts (acc@5) from SBFL's 8 to 11. Finally, we find that a slightly higher weight to the score from program states than to the SBFL score was best for identifying the plausible patch in our evaluation, indicating the incorporation of program state information was useful, and that the predictions made by our theoretic framework indeed contributed, underscoring the value of our framework.

Overall, our contributions are:

- A Bayesian framework that explains a number of known automated debugging results and practices;
- A patch prioritization technique, BAPP, which we derive directly from the Bayesian framework;
- Extensive empirical experiments demonstrating how BAPP can improve APR efficiency, and validate the theoretic predictions of our framework;
- A discussion of the possible implications of our theory.

The organization of this paper is as follows. We present our framework and its relationship to existing automated debugging literature in Section 2. Our approach is outlined in Section 3, while our evaluation setup is described in Section 4. Based on this, the results of our experiments are provided in Section 5. We discuss future work and threats to validity in Section 6, and related work in Section 7. Finally Section 8 concludes.

## 2 FRAMEWORK

We present a unified framework for automated debugging techniques.

### 2.1 Bayesian Inference

Bayesian inference is a way of updating probabilities or beliefs in response to new information, based on Bayes' theorem. In particular, given evidence or observations $E$, a hypothesis related to the evidence $H$, and the prior belief in the hypothesis $P(H)$, Bayesian inference postulates that the probability of a hypothesis given evidence, $P(H|E)$, can be calculated as the following:

$$P(H|E) = \frac{P(E|H)P(H)}{P(E)}$$

In Bayesian terminology, $P(H)$ is the *prior probability*; in contrast, $P(H|E)$ is the *posterior probability*, which is the updated belief after observing evidence $E$. To calculate the posterior, one needs a *statistical model* that can determine the probability of the evidence assuming that the hypothesis is correct, $P(E|H)$. The $P(E)$ term is a normalization term that does not influence the ranking of results, and thus may be ignored for our purposes.

Priors may be iteratively applied; in the face of new evidence $E'$, the probability of the hypothesis given both pieces of information (assuming $E$ and $E'$ are uncorrelated) , $P(H|E, E')$, is given as

$$P(H|E, E') = \frac{P(E'|H)P(H|E)}{P(E')}$$

showing that the previous posterior can be used as the prior when inferring given new evidence $E'$.

### 2.2 Application on Automated Debugging

We argue that the primary goal of automated debugging techniques is to find the likely fault location $l$ and the appropriate fix action $a$. Stated in probabilistic terms, the objective of automated debugging techniques is to infer the values of $P(l, a)$, or the likelihood that if we perform action $a$ at location $l$, the bug will be fixed.[1] Test-based automated debugging techniques may use dynamic information $D$, such as the results of individual tests, test suites, or (as we later do) program values to precisely infer the value of $P(l, a)$. Overall, we can say automated debugging techniques aim to infer $P(l, a|D)$, or the probability of certain patches given data $D$. As a result, we argue that the test-based automated debugging scenario can be effectively modeled using Bayesian inference, as the formula below denotes:

$$P(l, a|D) \propto P(D|l, a)P(l, a) \tag{1}$$

The formula above is in fact an application of Bayes' theorem without the denominator term, as the denominator is a normalization term that is the same for every patch $(l, a)$, and thus has no effect on the relative ranking between patches. The formula can be used to understand automated debugging techniques in various ways. For example, $P(l, a)$ can be decomposed to $P(l, a) = P(a|l)P(l)$; this can be thought to represent the separation of APR ($P(a|l)$) and FL ($P(l)$) techniques, as we describe in later sections. Additionally, we find that different families of automated debugging techniques differ in how they model the calculation of $P(D|l, a)$; concretely stating their models provides a useful window to inspect and compare techniques.

Finally, let us now turn to how fault localization fits in this model of automated debugging. We argue that FL is a special case

[1] We use $P(l, a)$ as a shorthand for $P(l = \text{fault location} \wedge a = \text{fix action})$ throughout the paper.

of automated debugging, for if one marginalizes Equation (1) over actions $a$, we end up with:

$$P(l|D) \propto P(D|l)P(l) \tag{2}$$

which can be used to derive maximal SBFL formulae, i.e. those that theoretically dominate other formulae of a particular group, as demonstrated in the next subsection.

## 2.3 Fault Localization

As a demonstration of our framework, we construct a statistical model based on the assumptions of prior theoretic work on FL [41] and show that, in conjunction with the Bayesian inference formula for fault localization (Equation (2)), we can recover formulae that were proven to be maximal, i.e. as close to optimal as an SBFL formula can get. Specifically, Yoo et al. analyze spectrum-based fault localization techniques, which use *program spectrum*. Program spectrum is a set of numbers that characterize how the test suite of the program interacts with each program element; in the paper, they notate spectrum with $e_f, e_p, n_f, n_p$ which denote the number of **f**ailing tests that **e**xecuted a location, the number of **p**assing tests that executed a location, the number of failing tests that did **n**ot execute a location, and the number of passing tests that did not execute a location, respectively. Further, the total number of failing tests is denoted as $F$.

Yoo et al. [41] make three assumptions about bugs in their analysis: (i) that there is a single fault in the code, (ii) that the code is deterministic, and (iii) that there is at least one failing test case. These assumptions are reasonable, as (i) even when there are multiple faults in the code, faults are typically clustered to treat each fault in isolation [32], and (ii, iii) without these assumptions it is difficult to start debugging at all. Upon these assumptions, we build the following statistical model that provides the probability a test will fail given coverage information and the true fault location:

$$P(t = \text{fail}|l = \text{fault} \wedge l \in_c t) = p \tag{3}$$

$$P(t = \text{fail}|l = \text{fault} \wedge l \notin_c t) = 0 \tag{4}$$

The first equation is simply stating that 'if the true fault location $l$ is covered by a test $t$, the probability that $t$ will fail is a nonzero $p$.' The second equation states that 'if the true fault location $l$ is *not* covered by $t$, it will never fail'. This model naturally follows from the previously stated assumptions.

Before we proceed further, we must set a prior $P(l)$ probability of each location being the true fault location. For simplicity we use the uniform prior: that is, all lines are equally suspicious when there is no information. Specifically, given the full set of statements $L$, $P(l) = \frac{1}{|L|}$. We note that one may opt to use different priors, such as differentiating based on statement type, to more closely represent the actual bug distribution, which is non-uniform [28].

With the prior and statistical model determined, we may now perform Bayesian inference. Suppose we observe a test $t$ that fails and does not cover $l$; how likely is it that $l$ is the true bug location? Bayesian inference asks the reverse question: assuming that $l$ is the true bug location and $t$ covers $l$, how likely is it that $t$ fails? Then, it combines this with the prior to answer our original question, the probability $l$ actually is the true bug location given that $t$ has failed.

$$\begin{aligned} P(l|t = \text{fail} \wedge l \notin_c t) &\propto P(t = \text{fail}|l = \text{fault} \wedge l \notin_c t)P(l) \\ &= 0 \times \frac{1}{|L|} = 0 \end{aligned}$$

Thus, through Bayesian inference, we can deduce that locations not covered by the failing test $t$ cannot be related to the bug. Similar principles can be applied to the other test scenarios as well; thus, given the first test, we update the probability that each location is the true fault location as:

$$P(l|t) \propto \begin{cases} p & (t = \text{fail} \wedge l \in_c t) \\ 1-p & (t = \text{pass} \wedge l \in_c t) \\ 0 & (t = \text{fail} \wedge l \notin_c t) \\ 1 & (t = \text{pass} \wedge l \notin_c t) \end{cases}$$

with the prior dropped because it is the same at every location. Using the fact that previous posteriors can be used as new priors, and that the four cases above neatly map to the $e_f, e_p, n_f, n_p$ spectra described earlier, we can iteratively derive the posterior probability that a location is a fault given the entire test suite:

$$P(l = \text{fault}|t_1, ..., t_n) \propto 0^{n_f} 1^{n_p} p^{e_f} (1-p)^{e_p}$$

While the formula above seems to have four variables, in terms of determining the ranking the formula can be further simplified. First, if $n_f \neq 0$, $P(l = \text{fault}|t_1, ..., t_n) = 0$, so the other factors are unimportant. For all statements for which $n_f = 0$, $e_f = F$ holds as well, so is irrelevant in terms of ranking; $1^{n_p} = 1$ as well, making $e_p$ the only deciding factor in determining suspiciousness. As a result, we derive Equation (5):

$$P(l = \text{fault}|t_1, ..., t_n) \propto \begin{cases} 0 & e_f < F \\ (1-p)^{e_p} & e_f = F \end{cases} \tag{5}$$

As long as $0 < p < 1$, the exact value of $p$ becomes irrelevant for ranking, and this leads to the same rankings as the Naish01 SBFL formula identified to be one of the maximal formulae [41]. Further, in the limit when $p \to 0$, this becomes equivalent to another maximal formula, Binary. Thus, using our framework, one can quickly arrive at maximal equations under a given set of assumptions. This also allows one to assess how good a statistical model describes real behavior of code; for example, if the Binary SBFL formula shows good performance, it would mean that $p \ll 1$, and thus there may be many tests that are passing due to coincidental correctness.

We close by making a few observations. First, while we make the single-fault assumption as it greatly simplifies matters, one can perform inference for multiple faults as well, as done by Barinel [3]. In addition, our framework can be used to derive other fault localization formulae; such as deriving MBFL formulae, e.g. we could derive a formula bearing similarity to MUSE [26].

## 2.4 Automated Program Repair

APR techniques can be analyzed using our framework as well. In our paper, we use the taxonomy of APR techniques proposed by Le Goues et al. [16], which divides APR techniques into two groups: heuristic-based and constraint-based. While there is growing interest in APR techniques that employ deep learning [7], their

principles are largely similar to other heuristics-based techniques, as we explore through this subsection.

*Heuristic-based*, or Generate and Validate (G&V), APR techniques generally first take the list of suspicious statements provided by a fault localization technique, then use heuristics to generate a number of patches at each location. Each patch is then evaluated against the tests that are present in the project: if a patch makes all tests pass, the patch is deemed *plausible* and becomes a candidate for suggestion to the developer. This process is naturally captured by the Bayesian formulation of automated debugging: along with the decomposition $P(l, a) = P(a|l)P(l)$, we may infer the posterior probability of $P(l, a)$ as

$$P(l, a|D_{(l,a)}) \propto P(D_{(l,a)}|l, a)P(a|l)P(l) \tag{6}$$

where $D_{(l,a)}$ term represents the test execution results after applying repair action $a$ on location $l$, while the $P(a|l)$ and $P(l)$ terms represent the patch generation heuristics and the fault localization processes, respectively. What, then, is the statistical model being used to update patch probabilities? We find that the validation process of G&V techniques is well-expressed by a simple conditional probability model. If we denote that a test $t$ passed under patch $(l, a)$ as $t_{(l,a)} = \text{pass}$, we can set the following statistical model which replicates the validation process:

$$P(\forall t.t_{(l,a)} = \text{pass}|(l, a) = \text{fix}) = 1 \tag{7}$$

$$P(\forall t.t_{(l',a')} = \text{pass}|(l, a) = \text{fix}) = p \tag{8}$$

where $(l, a) \neq (l', a')$. That is, if the patch is the true fix, it should make all tests pass; the second row indicates the possibility that a different patch may also lead to all tests passing. Expanding Bayesian rules as we did in the previous subsection leads to the usual validation criterion that tries each patch one by one and discards those that cause test failures. Meanwhile, we note that the statistical model above has no special cases when the patches are related, e.g. when $l = l'$; one may say that the addition of such special cases is what characterizes the unified debugging techniques, as explained later in this section.

If the statistical model is this simple, what are G&V techniques improving? Particularly with the advent of deep learning-based techniques, we may say that latest APR techniques are improving the *prior distribution* of patches, in particular the $P(a|l)$ term that describes which repair actions are likely given a specific location. While this probability is implicit in techniques such as template-based APR, in deep learning-based APR techniques the probabilistic nature is explicit, as the neural models will generate probabilities for each of the patches that they generate. While neural APR techniques are showing rapid improvements [44], this analysis shows that they rely on the same dynamic update model as earlier APR work [11].

*Constraint-based* APR techniques often rely on constructing *constraints* that patches should satisfy in order to fix the patch. Many techniques use SMT solvers to solve these constraints; as a result, they rely less on having strong prior distributions $P(l, a)$. For example, Angelix [25] uses SBFL results and has a less restrictive $P(a|l)$, while DirectFix [24] does not use external FL results at all, essentially using a uniform prior $P(l)$.

We analyze Angelix as an example to show how constraint-based techniques can be understood under our framework. To simplify the operations of Angelix, for each test $t$ an angelic value $v_t$ is derived for fix expression $a$ at a location $l$; the values are 'angelic' because if the value of the expression $a$ at $t$ becomes equivalent to $v_t$, the test will pass. For example, Angelix might derive that a certain predicate must evaluate to `true` for a previously failing test to pass. For passing tests, $v_t$ is set to maintain the existing behavior, while for failing tests a value that makes the test pass is found, e.g. using SMT solvers [25]. The value of the fix expression when executing test $t$, $[\![a]\!]_t$, is expected to be $v_t$ on all tests,

$$P([\![a]\!]_t = v_t|(l, a) = \text{fix}) = 1 \tag{9}$$

and any patches that deviate from the angelic values at any test are discarded. Note that there is no distinction between passing and failing tests in Equation (9), which distinguishes the constraint-based techniques from the update rules of BAPP introduced in Section 3.

## 2.5 Unified Debugging

Recently, *unified debugging* has been proposed as a way to integrate the FL and APR process [22]. While there are a number of proposed techniques, we use our Bayesian framework to analyze the recent SeAPR [5] technique, as it provides an approach in which our framework can re-derive the core assumptions and make recommendations on the equation form.

The SeAPR technique defines 'high-quality patches' as patches that make at least one previously failing test pass when applied. Based on this, SeAPR assigns higher priority to patches that modify the same locations as high-quality patches. Their assumptions can be transformed into a statistical model under our framework; adding the single fault assumption for simplification we can formulate the model as

$$P(\exists t.(t = \text{fail} \wedge t_{(l,a')} = \text{pass})|(l, a) = \text{fix}) = p_1 \tag{10}$$

$$P(\exists t.(t = \text{fail} \wedge t_{(l',a')} = \text{pass})|(l, a) = \text{fix}) = p_2 \tag{11}$$

where $l \neq l' \wedge p_1 > p_2$. In particular, Equation (10) describes the special rule for related patches which was not in the statistical model of prior G&V approaches. Along with the 'discard patches that fail tests' criterion provided in Equation (7), the statistical model can be used to derive probabilities of each patch being the true fix based on our framework. Noteworthy in the statistical model that we build based on the SeAPR settings is that fail-to-pass tests can appear in patches unrelated to the true fix (Equation (11)), unlike in the FL model where tests could not fail without covering the true fault location, leading to a different suspiciousness formulation. In fact, after simplification, we find that

$$\log(P((l, a) = \text{fix}|D)) \propto p^+ - \gamma p^- \tag{12}$$

where $p^+$ is the number of high-quality patches at $l$, while $p^-$ is the number of low-quality patches at $l$, and $\gamma = \frac{\log((1-p_2)/(1-p_1))}{\log(p_1/p_2)}$. This is in fact equivalent to the Wong2 [37] SBFL formula when $\gamma = 1$. Unfortunately, the SeAPR publication [5] did not experiment with the Wong2 formula, so it is unclear to what extent the empirical results presented in that paper support our framework.

Nonetheless, we believe this analysis demonstrates the utility of our framework. Benton et al. [5] argued that the use of APR results can be mapped to coverage spectrum analogues and thus made the assumption that SBFL formulae may be similarly used in unified debugging. However, our framework allows an inspection of the

assumptions behind the model, and further shows that the assumptions are different from SBFL, where we could confidently exclude locations not covered by tests. Finally, our framework suggests a formula not studied in the original work, showing its capability of making practical suggestions that may not be considered without the use of a theoretic framework.

## 3 BAYESIAN PATCH PRIORITIZATION

So far, we have explored various branches of the automated debugging field and shown that a multitude of prior results can be understood through our Bayesian framework. In this section, distinctly from the aforementioned approaches, we use our framework as the theoretic basis to derive a novel, fully automatic patch prioritization technique, BAPP, to incorporate values and improve the efficiency of automated program repair techniques by efficiently identifying promising patches quickly.

### 3.1 Assumptions and Derivation

To contribute towards solving the important problem of patch prioritization, we use our Bayesian framework to derive a formula for our tool, BAPP. First, we construct a statistical model based on the principle of *behavior change*. Specifically, we observe that (i) correct patches must alter the behavior of failing tests, and (ii) that it is unlikely, yet possible, that they may alter the behavior of passing tests. For example, when adding the statement `if (v == null) return;` to a location, the behavior would change if there is at least one test execution in which `v == null`; otherwise the patch would not alter the behavior of the test. We note that if a location is executed multiple times, it is sufficient for the behavior to be changed at just one point to alter test behavior. Thus, we can formally specify these assumptions into the following statistical model, with $\mathrm{Ch}(t,(l,a))$ denoting that patch $(l,a)$ would alter the behavior of program under test $t$:

$$P(t = \text{failing}|(l,a) = \text{fix} \wedge \mathrm{Ch}(t,(l,a))) = p \tag{13}$$

$$P(t = \text{failing}|(l,a) = \text{fix} \wedge \neg\mathrm{Ch}(t,(l,a))) = 0 \tag{14}$$

where $0 < p < 1$; namely, if a patch changes failing test behavior it has a chance to be the true patch, while if a patch does not change failing test behavior there is no chance it is the true patch.

We denote the following 'spectrum' to represent program change for a specific patch $(l,a)$, similarly to what was done when using the Bayesian framework to analyze existing techniques. $c_f$ denotes the number of failing tests for which $(l,a)$ changes behavior; $c_p$ denotes the number of changed passing tests, $n_f$ denotes the number of unchanged failing tests, and $n_p$ finally denotes the number of unchanged passing tests. Further, we use the decomposition $P(l,a) = P(a|l)P(l)$, for which $P(a|l)$ and $P(l)$ may be any patch-generating and FL technique, respectively. From this we can derive:

$$P((l,a) = \text{fix}|D) \propto (0^{n_f} 1^{n_p} p^{c_f} (1-p)^{c_p}) P(a|l)P(l) \tag{15}$$

Handling the $c_f < F$ case separately and removing terms that are unrelated to ranking similarly to the SBFL case, we end up with the following:

$$P((l,a) = \text{fix}|D) \propto \begin{cases} 0 & (c_f < F) \\ (1-p)^{c_p} P(a|l)P(l) & (c_f = F) \end{cases} \tag{16}$$

This can be more easily calculated in practice using log terms as follows, where $\alpha = -\log_2(1-p)$ (note that $\alpha > 0$, as $1 - p < 1$):

$$\log_2 P((l,a) = \text{fix}|D) \propto \begin{cases} -\infty & (c_f < F) \\ \log_2(P(a|l)P(l)) - \alpha c_p & (c_f = F) \end{cases} \tag{17}$$

We later use $\alpha$ to control how much to weigh the dynamic information: when $\alpha$ is large, $c_p$ will have significant sway on the ranking results, while when $\alpha$ is small, $c_p$ will have less influence. The impact of $\alpha$ corresponds to the strength of the assumption of the statistical model in Equation (14). Thus, by inspecting whether test behavior would change (locally) when a patch is applied, we can obtain a more precise posterior probability regarding which patch is likely to be correct. This technique may also be used to obtain more precise fault localization results: we may simply marginalize over the space of repair actions as follows:

$$P(l|D) = \sum_a P((l,a) = \text{fix}|D) \tag{18}$$

Note that even without considering program values, Equation 17 differs from how existing APR techniques prioritize their patches in two ways. First, unlike existing template based APR techniques, it explicitly considers the patch probability at a location, $P(a|l)$; thus in our case, even if two locations have the same FL score, patches that are likely as indicated by $P(a|l)$ will be prioritized. Second, the suspiciousness of a location, $P(l)$, is directly multiplied to the patch probability, $P(a|l)$, to produce the final ranking. In contrast, existing APR techniques validate each patch generated at a likely fault location before moving on to the next likely location, no matter how improbable the patch is (as indicated by $P(a|l)$). Our results in Section 5 demonstrate that these small tweaks lead to an increase in performance in our tool.

Up to this point, we have used our Bayesian framework to derive mathematical principles for the BAPP technique. Based on this, we describe how program states may be efficiently evaluated for this technique for practical implementation of BAPP, and about the specific choices of $P(l)$ and $P(a|l)$.

### 3.2 Implementation Overview

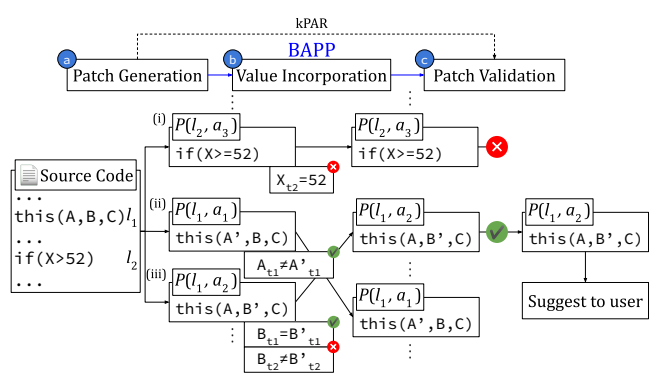



**Figure 1: Overview of BAPP.**

In the rest of this section, we will explain the implementation details of BAPP, a patch re-ranking tool built upon the derivations

presented in the previous subsection. As shown in Figure 1, BAPP can be broadly divided into three steps. First is the generation of all possible patches (Figure 1(a)), the implementation of which is closely based on kPAR [18], the open-source implementation of the original pattern-based APR, PAR [11]. In the second step, BAPP uses the Java debugger, JDB, to extract values of expressions relevant to the patches generated in the previous step (Figure 1(b)). This step is our main contribution to the overall technique, as the original kPAR simply comprises of the first and the third step. As will be discussed later in the section, this stage also involves the removal of patches with syntax errors saving the compilation cost from the original kPAR implementation. Using the extracted values, a likelihood score is calculated for each possible patch in accordance with the derivations presented in the previous subsection, and these scores are used to comprehensively rank the patches for the next and final step: patch validation (Figure 1(c)).

**Listing 1: Abbreviated patch for Chart-8.**

```
 public Week (Date time, TimeZone zone) {
-    this(time, RTP.DEFAULT_TIME_ZONE, ...);
+    this(time, zone, ...);
 }
```

The following subsections provide further details for each of the steps: patch generation, value extraction, and validation. In order to provide a clear picture of the entire process, we will use the correctly generated patch for Defects4J Chart-8 as a running example. The context of the buggy line in the source code is presented in Listing 1. This simple patch is shown abbreviated in Figure 1 (iii), along with alternative patches in the same project.

## 3.3 Patch Generation

Our patch generation shares repair templates with kPAR, as shown in Table 1. BAPP first generates the AST of all the files covered by the failing tests using the `javalang` library [34]. Then, using this AST, BAPP finds matching templates for each of the lines executed by the failing tests. The template types and the possible patches that can be generated for each of the templates are presented in Listing 1. Looking at the buggy line in our example code in Listing 1, BAPP would detect a method invocation node in the AST at this location. Traversing the AST also allows us to detect the `zone` variable that can be used to replace the second argument in this method invocation. Thus, BAPP would be able to conclude that a Parameter Replacer template could be applied to this location.

After the AST analysis, BAPP generates all possible patches for each variant of the identified matching templates. In our example, in accordance with the description for the Parameter Replacer template, a patch will be generated in which the original argument `RegularTimePeriod.DEFAULT_TIME_ZONE` would be replaced with the variable in the scope with the appropriate type, `zone`. Considering a field of type `Date` declared in this class (not shown in Listing 1), another Parameter Replacer patch could be generated as shown in Figure 1 (ii). The output of this stage is the list of all possible patches for each of the locations under consideration.

## 3.4 Value Incorporation

To obtain program values, BAPP used JDB to execute tests on the original unchanged source code, extract both the values of the original expression and the new expression of each patch generated in the previous stage. For the patch $a_2$ in Figure 1 (iii), whenever the breakpoint is triggered at location $l_1$, BAPP extracted the value of the original argument `RegularTimePeriod.DEFAULT_TIME_ZONE`, as well as the value of the new argument `zone`, illustrated under Figure 1 (b) as `B` and `B'`, respectively. After the execution of each test, BAPP analyzes these values to either filter out implausible patches or assign a likelihood score for the remaining patches. Equation (17) derived in the previous section summarizes how the values extracted are processed: note that values from failing tests are processed differently from values of the passing tests.

All failing tests are executed before any of the passing tests are executed. In accordance with the assumption that the fix must change the behavior of the failing test, as specified in Equation (17), the value of the original expression is compared with the value of the new expression for each patch. Any patch for which the two values are identical is discarded as implausible. In our running example, if `RegularTimePeriod.DEFAULT_TIME_ZONE` and `zone` have equal values whenever this particular line is executed in a failing test, then this particular patch would be removed from the pool of possible patches after the execution of that failing test.

After all failing tests are executed, passing tests are run in order to assign a likelihood score to each of the remaining patches after the implausible patches have been filtered out. In our implementation, the $c_p$ term in Equation (17) is set to the number of passing tests in which the original value and the replacement value are different at any instance in which the location in question is executed during the passing test. More intuitively, if the patch does not change the behavior of the passing test, the likelihood score increases, and vice versa. With the incorporation of normalized Ochiai SBFL scores represented in Equation (17) as $P(l)$, the final score is calculated after the execution of passing tests based on the equation.

Although argument values are evaluated for Parameter Replacement patches as shown in our example, the return values of the method invocations are not evaluated. We empirically find that invoking methods for value extraction often leads to various side effects, threatening the integrity of value extraction in other patches and thus the accuracy of the tool. For similar reasons, return values are not evaluated for Parameter Adder, Parameter Remover, and Method Replacer.

To improve efficiency, we apply the following optimization to this stage. First, we only consider the top 200 locations in the SBFL ranking. To prevent JDB stopping at breakpoints within loops at every iteration, we limit each breakpoint to 100 hits, before which the corresponding values are not extracted. Values are only extracted for the *last* 100 hits of a statement,[2] based on prior work showing that failing values that induce test failures appear in shorter execution traces [1]. We also impose a 15-minute timeout to the value-extraction stage, which we found to be reasonable across all bugs we studied. With the timeout of 15 minutes, BAPP often cannot execute all passing tests, especially for projects like Closure which has a large number of test cases. To address this issue, we prioritize passing test execution based on the current likelihood

[2]Statement execution counts can be retrieved from coverage profilers, which are used by the SBFL technique.

**Table 1: Fix templates used in our study.**

| Template Type | Description |
|---|---|
| Parameter Replacer | Replace an argument with another variable of the appropriate type. |
| Parameter Adder | Switch to an overloaded method by adding a variable of the appropriate type as an additional argument. |
| Parameter Remover | Switch to an overloaded method by removing an existing argument. |
| Method Replacer | Replace the method name to another method of the same type from the same class. |
| Conditional Replacer | Replace a conditional expression with another boolean expression. |
| Conditional Adder | Append a new component to a conditional expression using `\|\|` or `&&`. |
| Conditional Remover | Remove a component of a conditional expression. |
| Null Checker | Insert a null checker before a referenced variable. |
| Cast Checker | Insert a type checker before a typecasted variable. |

score of all the lines covered by each remaining passing test. While on as a matter of theory prioritization would be better without these optimizations, as a practical tool BAPP needs to balance analysis time with performance improvement, which is why we opt to include these optimizations.

## 3.5 Patch Validation

Through the previous steps, BAPP has generated all the possible patches – which essentially means that all the information necessary to apply the patches have been collected – and these patches have been ranked based on the relevant information including the program states and the Ochiai SBFL results. The final application and evaluation of the patches in the specified order is performed by replacing the original expression in the source code with the new expression.

For each patch in the ranking, BAPP first applies the patch to the source code. All the failing tests are run before the passing tests, and during the runs, if any of the tests fail, the patch in question is considered as faulty and the next patch is considered. When a patch is found which passes all the failing and passing tests, the repair process is terminated, as illustrated under Figure 1(c). It is important to note that some bugs may have multiple plausible patches. Because BAPP simply terminates after finding the first plausible patch, an incorrect plausible patch might be output instead of the correct patch.

# 4 EXPERIMENTAL SETUP

This section describes the settings of our empirical studies.

## 4.1 Configurations

As mentioned in the previous sections, we use `javalang` [34] for the generation of AST for the source code, and JDB for the extraction of values of relevant expressions during test executions. During the implementation of BAPP, we encountered inconveniences that motivated us to make changes to the `javalang` and JDB modules, in order to fix bugs or add features. For instance, we added a feature to convert a part of the AST tree back into code, which was not originally provided in `javalang`. Other changes include adding the position information to node types for which the information was originally omitted. When using the JDB, it was necessary to make changes to the module in order to ensure that JDB has the same execution semantics as the native Java runtime: for example, JDB originally lacks support for short-circuit evaluation.

Although BAPP's patch generation was based on kPAR's implementation, our results have several differences with kPAR's results that are worth noting. First, while kPAR results obtained by Liu et al. [19] were based on Defects4J v1, some bugs were modified when the benchmark was updated to Defects4J v2; the bugs that were modified were excluded from our study, as this change implies that the original buggy versions were not correctly captured in the earlier Defects4J version. For the sake of our experiment, we have also excluded bugs that kPAR was capable of patching via multi-hunk patches, as they violate the single-fault assumption made earlier; one could still apply BAPP after fault clusterization [32]. Finally, kPAR uses information about methods defined in external modules in order to generate patches for templates such as Method Replacers and Parameter Replacers. Due to the limitations of `javalang`, we omit support for patching invocations to external methods.

We used the Ochiai [27] suspiciousness order for our SBFL ranking ($P(l)$) as it is one of the most commonly used SBFL formulae [19, 44], and use the uniform distribution for $P(a|l)$; that is, if the number of patches generated by kPAR at a location is $N_l$, $P(a|l) = \frac{1}{N_l}$. This causes our ranking to be different from that of kPAR even when there is no dynamic information, as we described in Section 3.1. Ties in both SBFL and our FL technique were broken using the max-rank tiebreaker, as is done in prior FL research [31]. To evaluate FL results, we use the acc@$k$ metric, which evaluates how many bugs can be localized within $k$ inspections, as this metric was suggested to be practical by prior work [29]. The $\alpha$ parameter was set to 0.3 in RQ1 and RQ2 as it empirically showed the best performance. The experiments were run on machines with Intel(R) Core(TM) i7-6700 CPU @ 3.40GHz and 32GB of DDR4 RAM @ 2133MHz.

## 4.2 Research Questions

We aim to answer the following research questions with our empirical evaluation.

**RQ1. Efficiency Improvement:** How much more efficient is BAPP in comparison to kPAR in finding the first plausible patch? For this question we consider execution time and the overall patch rank of the first plausible patch.

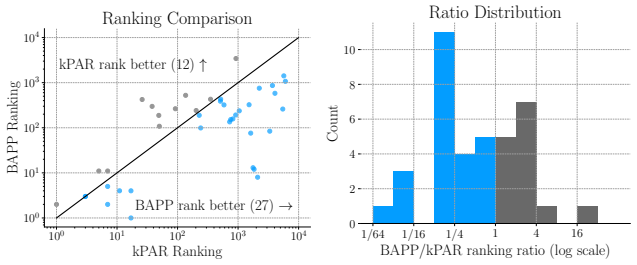
**RQ2. FL Improvement:** How much improvement can be made to the SBFL ranking of the true buggy line by incorporating the likelihood score calculated from the value extraction?
**RQ3. Configuration:** What is the optimal configuration for finding the first plausible patch? For this question we consider different values of $\alpha$ in Equation (17). We also evaluate patch rankings prioritizing SBFL results while using program states only as a tiebreaker, and vice versa. Finally, we compare the efficiency of runs with and without the $P(a|l)$ term in Equation (17).
**RQ4. Qualitative Analysis:** When does BAPP perform well, and when does it not? We analyze the reasons behind the successes and failures of BAPP, providing a breakdown of cases.

## 5 RESULTS

This section presents the results from our empirical evaluation.

### 5.1 RQ1: Efficiency Improvement

Out of 41 Defects4J bugs successfully patched by kPAR with our experimental setup, all 41 bugs are successfully patched with BAPP. This indicates that the patches filtered out during the value incorporation stage did not include plausible patches necessary to fix the bugs. As mentioned in Section 3.4, the patches that do not change program behavior during the execution of failing tests are filtered out. Furthermore, when using the debugger to evaluate expressions, we can filter out expressions that cause syntax errors or out-of-scope errors, which would only cause compilation errors if applied to the source code.




(a) Raw Ranks. (b) Rank Ratios.

**Figure 2: Performance of BAPP at re-ranking patches.**

The improvement in patch validation efficiency when using BAPP to prioritize patches is shown in Figure 2. Figure 2(a) plots kPAR and BAPP's ranks of the first plausible patch that was evaluated, illustrating the difference in efficiency for bugs from projects of different sizes. Overall, we observe consistent improvements across bugs of all project sizes. This is noteworthy because bugs from large projects also include a large number of tests that need to be run during the value-incorporation stage. However, because of the 15-minute timeout set on the execution of this stage as described in Section 3.4, only tens or hundreds of tests out of thousands can be executed to extract the program states. The fact that these bugs saw significant improvements in efficiency indicates the effectiveness of the optimization described in Section 3.4.

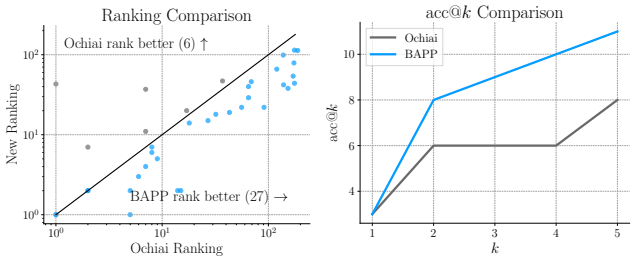
Of the 41 bugs studied, 13 are related to method invocation (i.e. Parameter Replacer/Adder/Remover and Method Replacer). As explained in Section 3.4, we do not extract return values for lines that fit these templates, because of the side effects from the duplicate method invocation necessary to extract these values. While one may wonder if our technique will also show improvements in those cases, many of these patches in fact show significant improvements in the rank of the plausible patch. In fact, one of the biggest improvements that can be seen in Figure 2(a) is for Closure-10, whose patch is of the Method Replacer type; BAPP improves the rank of the plausible patch to 83, from kPAR's 3338. From such examples, we infer that even if values cannot be extracted for the plausible patch, the value extraction and processing for the rest of the patches can yield a ranking that ultimately improves the efficiency for many bugs.

Figure 2(b) plots the BAPP/kPAR ratio for the ranks of plausible patches: the peak of the distribution is between $\frac{1}{8}$ and $\frac{1}{4}$, indicating that the efficiency improvement with respect to patch ranking for BAPP is within four- to eight-fold for a large portion of the bugs under consideration. The median ratio is 0.32.

So far, we have compared the efficiency of BAPP with respect to kPAR in terms of the rank of the plausible patch. However, execution time comparison gives a better picture of the practical effectiveness of BAPP, as BAPP has a patch analysis overhead, and there is the possibility that BAPP mostly filters out patches that fail to compile, which take a smaller time to validate and remove from the patch list. The mean execution-time reduction across the 41 bugs is 34 minutes, despite the overhead of value extraction which is on average 11 minutes. Thus, we argue that the improvement in efficiency outweighed the overhead cost of the extraction and evaluation of program states.

**Answer to RQ1:** BAPP could successfully reduce the patch validation effort by 68% in the median case. The mean execution-time reduction of 34 minutes demonstrates the practical efficiency improvement outweighed the overhead cost of value incorporation.

### 5.2 RQ2: FL Improvement




(a) Raw Ranks. (b) acc@$k$.

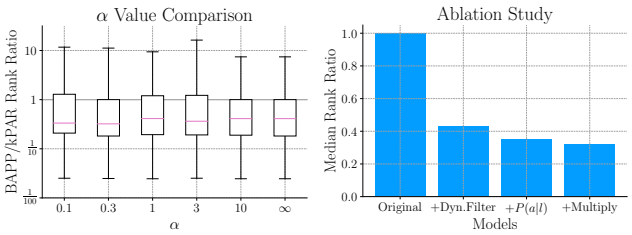
**Figure 3: Performance of BAPP at FL.**

In accordance with Equation (18), we built a new FL ranking of the covered locations for each of the Defects4J bugs under consideration. Figure 3(a) shows the changes in the ranks of the true buggy line between SBFL and BAPP's FL across bugs from projects

of different sizes. The differences in the FL rankings resemble the differences in patch rankings shown in Figure 2. The median of the BAPP-FL/SBFL ranking ratio is 0.57, indicating general improvements. Additionally, Figure 3(b) shows the acc@k comparison between SBFL and BAPP-FL: we find that BAPP-FL generally outperforms SBFL. Existing work has shown that identifying the true fault location within a few tries is important for the developer trust in FL techniques [13]. We believe these results indicate that BAPP shows promise in improving practical fault localization as well, with a relatively small computational budget of at most 15 minutes.

**Answer to RQ2:** BAPP's FL performed better than Ochiai overall, with the reduction in rank of the true buggy line at a median of 0.43, and identified the true buggy location at top-k rankings more often as well.

## 5.3 RQ3: Configuration




(a) $\alpha$ value comparison. (b) Ablation.

**Figure 4: Performance of BAPP under different settings. Lower is better.**

Figure 4(a) depicts the patch ranking reduction ratio of BAPP as $\alpha$ changes. When we evaluated the efficiency of different values of $\alpha$ up to $\infty$ (which strictly prioritizes the $c_p$ term over the SBFL score), $\alpha = 0.3$ saw the greatest reduction in the plausible patch ranking, with the median rank ratio of 0.32. While not in the figure, we found that FL performance was best when $\alpha = 0.3$ as well. Nonetheless, the result distributions over different $\alpha$ values show little difference, indicating the performance of BAPP is resilient to specific values of $\alpha$. Thus, in general the incorporation of program values is enough to enhance APR and FL performance.

Figure 4(b) isolates the different components of our technique and demonstrates the contribution of each component to BAPP's overall efficiency improvement. The first bar plot represents the performance of our original kPAR implementation. The second bar shows the performance of using the dynamic information to filter out patches that do not change the program states in failing tests; in the next step, the $P(a|l) = \frac{1}{N_l}$ term is introduced, but instead of the scores being multiplied as described in Equation (17), the probabilities are used as a tiebreaker among patches that have the same SBFL score. Finally, instead of prioritizing SBFL score over patch probability when ranking, we multiply the SBFL score with the dynamic patch probability using the best $\alpha$ value of 0.3, thus leading to the full BAPP technique as formally described in Equation (17). As evident in the figure, the failing test filter significantly improved the efficiency with a percentage point decrease of 57 in the median rank ratio. The $P(a|l)$ term and the component multiplication further improved the efficiency with a percentage point decrease of 8 and 3, respectively. This supports our earlier point in Section 3.1 that each component in our derivation contributes to the improvement in the performance of locating a plausible patch, demonstrating that our theoretical derivation can suggest ways to improve performance.

**Answer to RQ3:** The $\alpha$ value of 0.3 yielded the highest efficiency for identifying plausible patches. We verify that multiplying FL and patch likelihood score is better than prioritizing one over the other, and that the inclusion of the $P(a|l)$ term also contributed to the improved efficiency.

## 5.4 RQ4: Qualitative Analysis

We first present a breakdown of the individual cases in which BAPP underperformed its counterpart technique. First, when performing patch ranking, there were two main reasons plausible patches were ranked lower than the initial ranking from kPAR. In some cases, the statistical model that we used did not favor the patch: certain plausible patches would change passing test behavior often, or even whenever they were executed. For example, we found that the state would always change for the Chart-8 bug; nonetheless the patch itself is correct. In other cases, due to the large number of patches generated at certain locations, the likelihood of patches at those locations would drop due to the $P(a|l) = \frac{1}{N_l}$ term. As a result, all patches from such locations would be de-prioritized, leading to worse results. For example, a plausible patch for the Math-15 bug shared patch location with 205 other patches, and as a result dropped in ranking. However, as shown in Section 5.3, the incorporation of the $P(a|l)$ term resulted in an overall improvement of the efficiency of finding a plausible patch.

Our analysis for FL similarly reveals two reasons our technique yielded worse results. First, due to the nature of our FL technique which is closely related to patch templates, our technique could not suggest statements for which no patch was generated. For example, in Closure-22, one of the top-ranked actual buggy locations is simply a `continue;` statement, for which our technique generates no patches, and consequently fails to rank. We believe such issues can be overcome by adopting patch generation techniques that can generate a greater variety of patches in future work. A second issue was that for certain statements with conditions, a large number of patches that would always change the state would be generated, and as a result the likelihood of the statement (which is the sum of the likelihood of patches) would drop. Closure-115 had this issue, and as a result our technique showed poor performance in both FL and patch prioritization for this bug.

On the other hand, when such pitfalls are not met, our technique performs well; in Math-85, for example, the correct patch replaces the conditional expression `fa * fb >= 0.0` with `fa * fb > 0.0`. For every instance in which this buggy line was hit during failing test executions, `fa * fb` was equal to `0.0`, meaning the patch would change the program behavior. On the other hand, `fa * fb` was never equal to `0.0` during the execution of any of the passing

tests in this project, leading BAPP to improve the FL ranking by 76% (91 → 22) and the APR ranking by 81% (820 → 159).

**Answer to RQ4:** We broadly identify issues hindering better performance of BAPP, such as patches that do not closely match the statistical model we use.

# 6 DISCUSSION

We discuss threats to validity and the limitations of our framework, and present potential future research directions.

## 6.1 Threats to Validity

Threats to **internal validity** concern whether the results presented in the paper are sound. We have replicated a widely studied APR technique, kPAR, and verified that it could replicate patches reported to be successfully generated by kPAR in the literature. Our implementation is also publicly available for further scrutiny. BAPP depends on `javalang`, a widely used open source Java parser.

Threats to **external validity** concern whether the results would generalize to new subjects. In the case of BAPP, we take account of the potential idiosyncrasy of different bugs by experimenting over 41 bugs from the widely-used Defects4J benchmark of real-world faults. Further, we perform a search over the parameter $\alpha$ in RQ3, showing that performance gradually changes as the parameter changes. We have also attempted formulating techniques from a broad cross-section of the automated debugging literature using our proposed theoretical framework; as long as a technique shares the goal of inferring the posterior likelihood of the correct patch $P(l, a)$ we believe our framework will continue to be applicable. Meanwhile, we have presented results of BAPP re-ranking patches generated by kPAR; while our analysis shows that our simple statistical model works well for kPAR-generated patches, further experimentation is needed to decide whether our assumptions work for other patch generation techniques.

## 6.2 Limitations & Future Work

A major limitation of our framework is the single fault assumption, limiting the cases to which our theories can be applied. While it is possible to overcome these issues by reasoning over *sets* of solutions instead of single solutions as we have done in our work, when there are $N$ possible solutions this requires reasoning over $O(2^N)$ combinations of solutions, which quickly becomes impractical. Barinel [3] uses a heuristic named Staccato [2] to generate a smaller group of candidates to perform Bayesian inference over; more experiments are required to determine whether such heuristics would be scalable for automated debugging in general, and not just fault localization.

Throughout our paper, we assumed that test results are independent from each other as well, which can be a problematic assumption: it is undeniable that some tests are more similar to each other than others (as can be seen in failure clusterization work, for example [4]). Despite this, our theoretic framework could recreate formulae and practices from a broad range of automated debugging techniques, indicating that they make the same 'test independence' assumption as well. A strength of our theoretic framework is that it also provides ways to model test dependence: for example, the likelihood model provided in Equation (3) could have an adaptive $p$ based on prior test results, which would increase the complexity of subsequent equations for better modeling of the interaction between faults and tests.

Our framework also directs us towards future research directions that we hope to pursue further. To start off, we consider how existing techniques deal with the prior probability of patches, $P(l, a)$. While in almost all APR work it is decomposed to $P(a|l)P(l)$ and thus fault localization precedes patch generation, it does not necessarily need to be this way. Under our framework, one can equally decompose $P(l, a)$ to $P(l|a)P(a)$ instead, identifying the repair operation prior to performing fault localization. In certain cases, this formulation is closer to human practice: for example, in Defects4J Lang-29, the error message shows 'expected: [0] but was: [0.0]', from which one can infer that (i) a type needs to be changed somewhere, but (ii) which location to fix is unknown. Indeed, some existing techniques have actually pioneered this concept in a restricted way: VFix [39] notably focuses on null pointer exception fixes, and searches for fix locations given the types of fixes it can do. Such a direction is particularly promising given the recent improvements in using error messages for generating patches [40].

Finally, while the automated debugging work that we cover do not incorporate dependency information and updates at most based on the evaluation results of patches generated at the same location, we believe our model could facilitate the derivation of FL and APR technologies that leverage dependency information such as call stacks, and thus enhance precision; we hope to pursue such research areas in future work.

# 7 RELATED WORK

This section presents related work in different subfields.

## 7.1 Theories for Automated Debugging

Most existing work on automated debugging focuses on designing techniques; relatively little has been done to examine the theoretical aspect of automated debugging. Weimer et al. observed the duality between APR and mutation testing [35], which can be thought to have provided the foundations for the subsequent work on Mutation Based Fault Localization (MBFL) [21, 26]. Xie et al. [38] proposed a theoretical framework for proving hierarchy between Spectrum Based Fault Localization (SBFL) formulas, which eventually resulted in the no existence proof for the greatest formula (i.e., there is no single formula that is guaranteed to outperform all the other formulas) [42], prompting the FL research community to focus on the aggregated use of multiple formulas and extra input features [17], rather than designing new single formula. However, both of existing theoretical results on APR and FL are limited to relatively specific domains, i.e., mutation and spectrum-based fault localization approaches, respectively. We still lack a general framework that can express automated debugging as a whole, including FL techniques such as MBFL and SBFL, as well as various APR techniques. We believe that such a general framework may allow us to rigorously reflect on existing techniques and propose new and interesting future research directions.

### 7.2 Automated Debugging

Automated debugging has a long history [10] and thus it is difficult to summarize all techniques in the scope of this paper. In this work we focus on test-based automated debugging techniques, which we find ideal when applying the Bayesian inference toolkit. Test-based automated debugging techniques can be roughly categorized into FL and APR. Test-based FL generally seeks to identify the part of a project that needs to be fixed given a number of failing tests and potentially passing tests. Researchers have identified multiple ways to do this, including the use of program spectrum [10], mutation testing [26], project history [36], and more. Test-based APR seeks to change the source code of a project so that all tests pass, and ideally so that the patch is semantically equivalent to the patch that the developer would have made. As with FL, there are multiple approaches: while the first APR technique, GenProg [15], used genetic algorithms to create patches, other ways to generate patches subsequently emerged, such as using templates as with PAR [11], generating constraints that patches should meet then solving those constraints with SMT solvers, as with Angelix [25], or by using deep neural networks [7].

### 7.3 Program Values in Automated Debugging

A large number of automated debugging techniques do not explicitly use concrete program values in any way; for example, most FL techniques do not explicitly use program values [10, 17, 26], and a significant number of APR techniques focus on generating the correct patch given the static context rather than incorporating values. Nonetheless, there have been attempts to incorporate values into the automated debugging process, as humans do [6]. SmartFL [43] generates a detailed probabilistic graph of a program that incorporates values into its inference process, but due to the potentially large graphs that are generated, inference can be slow. For APR, Angelix [25] identifies angelic values that allow a test to pass, while Dynamoth [8] used a debugger to similarly check if certain predicates met angelic value conditions. As our theory provides a holistic view of APR and FL, our tool paves a way to consider values and tackle the automated debugging problem as a whole.

### 7.4 Patch Prioritization

As generate-and-validate APR techniques improved and increased their search space, the importance of patch prioritization has also grown, and multiple techniques have been suggested; as it is difficult to give a full overview of all techniques within this paper we introduce a cross-section of explored approaches. To improve fault localization during the patch validation process, Unified Debugging [22] techniques have been proposed to improve the precision of FL while doing patch validation. Meanwhile, some techniques seek to optimize the patch template to apply: for example, Prophet [20] mines statistics of patches to precisely apply templates. Other techniques prioritize patches based on the specific code snippet they introduce: ELIXIR [30] uses manually constructed features to identify patch ingredients to be used when applying a patch template. Our prioritization approach differs as it uses program values as a means of calculating patch ranking, and as a result is orthogonal with the aforementioned techniques.

## 8 CONCLUSION

We propose a Bayesian framework of automated debugging, postulating that the ultimate goal of automated debugging techniques is to infer the posterior likelihood over the space of fault locations and repair actions, $P(l, a)$. We find that this formulation can recover previously proven results, such as the maximal Op2/Binary SBFL formulae, as well as have specific probability terms neatly mapped to specific automated debugging concepts and allow an inspection of the assumptions behind automated debugging techniques. To demonstrate the utility of the framework, we propose a novel value-incorporating patch prioritization technique for APR, whose core principles are derived from our Bayesian framework. Along with the use of debuggers which allows the efficient implementation of the recommendations of the framework, we find that overall our tool BAPP can improve the patch ranking by 68%, leading to an average execution time reduction of 34 minutes. BAPP also improves the FL ranking in two-thirds of the inspected bugs, leading to an increase in acc@k values. In addition, our ablation study reveals BAPP is resilient to the choice of $\alpha$ values. We believe that our Bayesian framework also suggests interesting research directions, such as choosing the repair operator before identifying where to fix, that have not been thoroughly explored, and thus hope to perform related research in the future.

## ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their valuable feedback. This research was supported by the Undergraduate Research Project programme at KAIST, as well as the Institute for Information & communications Technology Promotion grant funded by the Korean government (MSIT) (No.2021-0-01001).

## REFERENCES

[1] Rawad Abou Assi, Chadi Trad, Marwan Maalouf, and Wes Masri. 2019. Coincidental correctness in the Defects4J benchmark. *Software Testing, Verification and Reliability* 29 (03 2019). https://doi.org/10.1002/stvr.1696

[2] Rui Abreu and Arjan J. C. van Gemund. 2009. A Low-Cost Approximate Minimal Hitting Set Algorithm and its Application to Model-Based Diagnosis. In *SARA*.

[3] R. Abreu, P. Zoeteweij, and A.J.C. van Gemund. 2009. Spectrum-Based Multiple Fault Localization. In *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering (ASE 2009)*. 88–99. https://doi.org/10.1109/ASE.2009.25

[4] Gabin An, Juyeon Yoon, Jeongju Sohn, Jingun Hong, Dongwon Hwang, and Shin Yoo. 2022. Automatically Identifying Shared Root Causes of Test Breakages in SAP HANA. In *Proceedings of the 44th IEEE/ACM International Conference on Software Engineering - Software Engineering In Practice Track (ICSE SEIP 2022)*. 65–74.

[5] Samuel Benton, Yuntong Xie, Lan Lu, Mengshi Zhang, Xia Li, and Lingming Zhang. 2022. Towards Boosting Patch Execution On-the-Fly. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*. 2165–2176. https://doi.org/10.1145/3510003.3510117

[6] Mariano Ceccato, Alessandro Marchetto, Leonardo Mariani, Cu D. Nguyen, and Paolo Tonella. 2012. An empirical study about the effectiveness of debugging when random test cases are used. In *2012 34th International Conference on Software Engineering (ICSE)*. 452–462. https://doi.org/10.1109/ICSE.2012.6227170

[7] Zimin Chen, Steve Kommrusch, Michele Tufano, L. Pouchet, D. Poshyvanyk, and Monperrus Martin. 2019. SequenceR: Sequence-to-Sequence Learning for End-to-End Program Repair. *ArXiv* abs/1901.01808 (2019).

[8] Thomas Durieux and Martin Monperrus. 2016. DynaMoth: Dynamic Code Synthesis for Automatic Program Repair. In *2016 IEEE/ACM 11th International Workshop in Automation of Software Test (AST)*. 85–91. https://doi.org/10.1145/2896921.2896931

[9] Shin Hong, Taehoon Kwak, Byeongcheol Lee, Yiru Jeon, Bongsuk Ko, Yunho Kim, and Moonzoo Kim. 2017. MUSEUM: Debugging Real-World Multilingual Programs Using Mutation Analysis. *Information and Software Technology* 82 (2017), 80–95.

[10] James A. Jones, Mary Jean Harrold, and John Stasko. 2002. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering*. ACM, New York, NY, USA, 467–477.

[11] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic Patch Generation Learned from Human-written Patches. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13)*. IEEE Press, Piscataway, NJ, USA, 802–811.

[12] Serkan Kirbas, Etienne Windels, Olayori McBello, Kevin Kells, Matthew Pagano, Rafal Szalanski, Vesna Nowack, Emily Rowan Winter, Steve Counsell, David Bowes, Tracy Hall, Saemundur Haraldsson, and John Woodward. 2021. On The Introduction of Automatic Program Repair in Bloomberg. *IEEE Software* 38, 4 (2021), 43–51. https://doi.org/10.1109/MS.2021.3071086

[13] Pavneet Singh Kochhar, Xin Xia, David Lo, and Shanping Li. 2016. Practitioners' Expectations on Automated Fault Localization. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA 2016)*. Association for Computing Machinery, New York, NY, USA, 165âĂŞ176.

[14] Anil Koyuncu, Kui Liu, Tegawendé F. Bissyandé, Dongsun Kim, Jacques Klein, Martin Monperrus, and Yves Le Traon. 2020. FixMiner: Mining Relevant Fix Patterns for Automated Program Repair. *Empirical Softw. Engg.* 25, 3 (may 2020), 1980–2024. https://doi.org/10.1007/s10664-019-09780-z

[15] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A Generic Method for Automatic Software Repair. *IEEE Transactions on Software Engineering* 38, 1 (2012), 54–72. https://doi.org/10.1109/TSE.2011.104

[16] Claire Le Goues, Michael Pradel, Abhik Roychoudhury, and Satish Chandra. 2021. Automatic Program Repair. *IEEE Software* 38, 4 (2021), 22–27. https://doi.org/10.1109/MS.2021.3072577

[17] Xia Li, Wei Li, Yuqun Zhang, and Lingming Zhang. 2019. DeepFL: Integrating Multiple Fault Diagnosis Dimensions for Deep Fault Localization. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 169–180. https://doi.org/10.1145/3293882.3330574

[18] Kui Liu, Anil Koyuncu, TegawendÃĺ F. BissyandÃĺ, Dongsun Kim, Jacques Klein, and Yves Le Traon. 2019. 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)*. 102–113. https://doi.org/10.1109/ICST.2019.00020

[19] Kui Liu, Shangwen Wang, Anil Koyuncu, Kisub Kim, Tegawendé F. Bissyandé, Dongsun Kim, Peng Wu, Jacques Klein, Xiaoguang Mao, and Yves Le Traon. 2020. On the Efficiency of Test Suite based Program Repair A Systematic Assessment of 16 Automated Repair Systems for Java Programs. *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)* (2020), 615–627.

[20] Fan Long and Martin Rinard. 2016. Automatic Patch Generation by Learning Correct Code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*. Association for Computing Machinery, New York, NY, USA, 298–312. https://doi.org/10.1145/2837614.2837617

[21] Yiling Lou, Ali Ghanbari, Xia Li, Lingming Zhang, Haotian Zhang, Dan Hao, and Lu Zhang. 2020. 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 (ISSTA 2020)*. Association for Computing Machinery, New York, NY, USA, 75–87.

[22] Yiling Lou, Ali Ghanbari, Xia Li, Lingming Zhang, Haotian Zhang, Dan Hao, and Lu Zhang. 2020. 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*. ACM. https://doi.org/10.1145/3395363.3397351

[23] Alexandru Marginean, Johannes Bader, Satish Chandra, Mark Harman, Yue Jia, Ke Mao, Alexander Mols, and Andrew Scott. 2019. SapFix: Automated End-to-End Repair at Scale. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. 269–278. https://doi.org/10.1109/ICSE-SEIP.2019.00039

[24] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2015. Directfix: Looking for simple program repairs. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. IEEE, 448–458.

[25] S. Mechtaev, J. Yi, and A. Roychoudhury. 2016. Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. 691–701.

[26] Seokhyeon Moon, Yunho Kim, Moonzoo Kim, and Shin Yoo. 2014. Ask the Mutants: Mutating Faulty Programs for Fault Localization. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*. 153–162. https://doi.org/10.1109/ICST.2014.28

[27] Akira Ochiai. 1957. Zoogeographic studies on the soleoid fishes found in Japan and its neighbouring regions. *Bulletin of Japanese Society of Scientific Fisheries* 22 (1957), 526–530.

[28] Kai Pan, Sunghun Kim, and E. James Whitehead. 2009. Toward an Understanding of Bug Fix Patterns. *Empirical Softw. Engg.* 14, 3 (jun 2009), 286âĂŞ315. https://doi.org/10.1007/s10664-008-9077-5

[29] Chris Parnin and Alessandro Orso. 2011. Are automated debugging techniques actually helping programmers?. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis (ISSTA 2011)*. ACM, New York, NY, USA, 199–209.

[30] Ripon K. Saha, Yingjun Lyu, Hiroaki Yoshida, and Mukul R. Prasad. 2017. Elixir: Effective object-oriented program repair. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 648–659. https://doi.org/10.1109/ASE.2017.8115675

[31] Jeongju Sohn and Shin Yoo. 2017. FLUCCS: Using Code and Change Metrics to Improve Fault Localization. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2017)*. Association for Computing Machinery, New York, NY, USA, 273âĂŞ283. https://doi.org/10.1145/3092703.3092717

[32] Yi Song, Xiaoyuan Xie, Xihao Zhang, Quanming Liu, and Ruizhi Gao. 2023. Evolving Ranking-Based Failure Proximities for Better Clustering in Fault Isolation. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22)*. Association for Computing Machinery, New York, NY, USA, Article 41, 13 pages.

[33] Gregory Tassey. 2002. The Economic Impacts of Inadequate Infrastructure for Software Testing. (05 2002).

[34] Chris Thunes. 2022. javalang: Pure Python Java parser and tools. https://github.com/c2nes/javalang. (2022).

[35] W. Weimer, Z. P. Fry, and S. Forrest. 2013. Leveraging program equivalence for adaptive program repair: Models and first results. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 356–366.

[36] Ming Wen, Junjie Chen, Yongqiang Tian, Rongxin Wu, Dan Hao, Shi Han, and Shing-Chi Cheung. 2021. Historical Spectrum Based Fault Localization. *IEEE Transactions on Software Engineering* 47, 11 (2021), 2348–2368. https://doi.org/10.1109/TSE.2019.2948158

[37] W. Eric Wong, Yu Qi, Lei Zhao, and Kai-Yuan Cai. 2007. Effective Fault Localization using Code Coverage. In *31st Annual International Computer Software and Applications Conference (COMPSAC 2007)*, Vol. 1. 449–456. https://doi.org/10.1109/COMPSAC.2007.109

[38] Xiaoyuan Xie, Tsong Yueh Chen, Fei-Ching Kuo, and Baowen Xu. 2013. A Theoretical Analysis of the Risk Evaluation Formulas for Spectrum-based Fault Localization. *ACM Transactions on Software Engineering Methodology* 22, 4, Article 31 (October 2013), 40 pages.

[39] Xuezheng Xu, Yulei Sui, Hua Yan, and Jingling Xue. 2019. VFix: Value-Flow-Guided Precise Program Repair for Null Pointer Dereferences. In *Proceedings of the 41st International Conference on Software Engineering (ICSE '19)*. IEEE Press, 512âĂŞ523. https://doi.org/10.1109/ICSE.2019.00063

[40] He Ye, Matias Martinez, Xiapu Luo, Tao Zhang, and Martin Monperrus. 2022. SelfAPR: Self-supervised Program Repair with Test Execution Diagnostics. (2022). https://doi.org/10.48550/ARXIV.2203.12755

[41] Shin Yoo, Xiaoyuan Xie, Fei-Ching Kuo, Tsong Yueh Chen, and Mark Harman. 2014. *No Pot of Gold at the End of Program Spectrum Rainbow: Greatest Risk Evaluation Formula Does Not Exist*. Technical Report RN/14/14. University College London.

[42] Shin Yoo, Xiaoyuan Xie, Fei-Ching Kuo, Tsong Yueh Chen, and Mark Harman. 2017. Human Competitiveness of Genetic Programming in SBFL: Theoretical and Empirical Analysis. *ACM Transactions on Software Engineering and Methodology* 26, 1 (July 2017), 4:1–4:30.

[43] Muhan Zeng, Yiqian Wu, Zhentao Ye, Yingfei Xiong, Xin Zhang, and Lu Zhang. 2022. Fault Localization via Efficient Probabilistic Modeling of Program Semantics. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*. 958–969. https://doi.org/10.1145/3510003.3510073

[44] Qihao Zhu, Zeyu Sun, Yuan-an Xiao, Wenjie Zhang, Kang Yuan, Yingfei Xiong, and Lu Zhang. 2021. A Syntax-Guided Edit Decoder for Neural Program Repair. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 341âĂŞ353. https://doi.org/10.1145/3468264.3468544


Received 2023-02-16; accepted 2023-05-03