The Journal of Systems and Software xxx (xxxx) xxx

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Automated program repair for variability bugs in software product line systems[☆]

Thu-Trang Nguyen[a], Xiao-Yi Zhang[b], Paolo Arcaini[c], Fuyuki Ishikawa[c], Hieu Dinh Vo[a,*]

[a] *Faculty of Information Technology, VNU University of Engineering and Technology, Hanoi, Viet Nam*
[b] *University of Science and Technology Beijing, Beijing, China*
[c] *National Institute of Informatics, Tokyo, Japan*

ARTICLE INFO

Keywords:
Software product line
Variability bugs
Automated program repair

ABSTRACT

Software product line (SPL) systems are widely employed to develop industrial projects. For an SPL system, different products/variants are created by combining different subsets of the system features. Because of the interaction of the different features, a bug in the system could cause failures for some products (failing products), but not for others (passing products); such types of bugs are called *variability bugs*. Due to their variability characteristics, detecting and fixing bugs in SPL systems is challenging. There are several solutions for localizing buggy statements in these systems. However, there is still a lack of research on automatically fixing these bugs. In this work, we aim to make the first attempt at automatically fixing buggy statements in the source code of SPL systems. This paper proposes two approaches, single-product-based and multi-product-based, to repair the variability bugs in an SPL system to fix the failures of the failing products and not to break the correct behaviors of the passing products. For the single-product-based approach, each failing product is fixed individually, and the obtained patches are then propagated and validated on the other products of the system. For the multi-product-based approach, all the products are repaired simultaneously. The patches are generated and validated by all the sampled products of the system in each repair iteration. Moreover, to improve the repair performance of both approaches, we also introduce several heuristic rules for effectively and efficiently deciding where to fix (*navigating modification points*) and how to fix (*selecting suitable modifications*). These heuristic rules use intermediate validation results of the repaired programs as feedback to refine the fault localization results and evaluate the suitability of the modifications before actually applying and validating them by test execution. Our experimental results on a dataset of 318 variability bugs of five popular SPL systems show that the single-product-based approach is around 20 times better than the multi-product-based approach in the number of correct fixes. Notably, the heuristic rules could improve the performance of both approaches by increasing of 30%–150% the number of correct fixes, and decreasing of 30%–50% the number of attempted modification operations.

1. Introduction

Software product line (SPL) systems (or configurable systems in general) are becoming popular and widely employed to develop large industrial projects (Apel et al., 2013a; Pereira et al., 2021; Randrianaina et al., 2022). An SPL system is a product family containing a set of products sharing a common code base. Each product is identified by the selected features (Apel et al., 2013a). In other words, a project adopting the SPL methodology can tailor its functional and nonfunctional properties to the requirements of users (Apel et al., 2013a, 2016) by selecting different sets of features (Apel et al., 2016). For example, a popular SPL system, the Linux Kernel, can be configured to generate specific kernel variants for billions of scenarios (Abal et al., 2014; Acher et al., 2022).

In practice, bugs are an inevitable problem in software programs. Developers often need to spend about 50% of their time on addressing software bugs (Hamill and Goseva-Popstojanova, 2017). Detecting and fixing bugs in SPL systems could be very complicated due to their variability characteristics. Echeverría et al. (2016) conducted an empirical study to evaluate engineers' behaviors in fixing errors and propagating the fixes to other products in an industrial SPL system.

☆ Editor: Dr. Justyna Petke.
* Corresponding author.
E-mail addresses: trang.nguyen@vnu.edu.vn (T.-T. Nguyen), xiaoyi@ustb.edu.cn (X.-Y. Zhang), arcaini@nii.ac.jp (P. Arcaini), f-ishikawa@nii.ac.jp (F. Ishikawa), hieuvd@vnu.edu.vn (H.D. Vo).

https://doi.org/10.1016/j.jss.2024.112152
Received 6 February 2024; Received in revised form 7 June 2024; Accepted 25 June 2024

They showed that fixing buggy SPL systems is challenging, especially for large systems. Indeed, in an SPL system, each product is composed of a different set of features. Due to the interaction of different features, a bug in an SPL system could manifest itself in some products of the system but not in others, so called *variability bugs*. In order to fix variability bugs, we need to find patches which not only work for one product but also for all the products of the system, i.e., we need to fix the incorrect behaviors of all *failing products*, and do not break the correct behaviors of the *passing products*.

To reduce the cost of software maintenance and alleviate the heavy burden of manually debugging activities, multiple automated program repair (APR) approaches (Le Goues et al., 2012b; Martinez and Monperrus, 2016; Xuan et al., 2017; Li et al., 2022; Lutellier et al., 2020; Gazzola et al., 2019) have been proposed in recent decades. These approaches employ different techniques to automatically (i.e., without human intervention) synthesize patches that eliminate program faults and obtain promising results. However, these approaches focus on fixing bugs in a single non-configurable system.

In the context of SPL systems, there are several studies dealing with variability bugs at different levels, such as model or configuration. Arcaini et al. (2016, 2017) attempt to fix bugs in the variability models. Weiss et al. (2017) and Xiong et al. (2015) repair misconfigurations of the SPL systems. However, repairing variability bugs at the source code level still remains unexplored.

This work proposes two test suite-based program repair approaches, *single-product-based* and *multi-product-based*, for repairing buggy SPL systems at the source code level. The proposed approaches leverage the test suites of the products of the system as the specification to identify the existence of bugs, localize the bugs, and validate the correctness of the patches. For the *single-product-based approach* ($\texttt{SingleProd}_{\texttt{basic}}$), each failing product of the system is repaired individually, and then the obtained patches, which cause the product under repair to pass all its tests, are propagated and validated on the other products of the system. For the *multi-product-based approach* ($\texttt{MultiProd}_{\texttt{basic}}$), instead of repairing one individual product at a time, all the products are considered for repairing simultaneously. Specifically, the patches are generated and then validated by all the sampled products of the system in each repair iteration. For both approaches, the valid patches are the patches causing all the available tests of all the sampled products of the system to pass.

Furthermore, we also introduce several *heuristic rules* for improving the performance of the two approaches in repairing buggy SPL systems. We start from the observation that, in order to effectively and efficiently fix a bug, an APR tool must correctly decide (i) where to fix (*navigating modification points*) and (ii) how to fix (*selecting suitable modifications*). Our heuristic rules focus on enhancing the accuracy of these tasks by leveraging intermediate validation results of the repair process.

For *navigating modification points*, APR tools (Martinez and Monperrus, 2019; Li et al., 2022) often utilize the *suspiciousness scores*, which are often calculated before the repair process by fault localization (FL) techniques (Abreu et al., 2007; Wong et al., 2016; Moon et al., 2014). However, the additional information obtained during the repair process can provide valuable feedback for continuously refining the navigation of the modification points (Lou et al., 2020). Therefore, in this work, besides suspiciousness scores, the *fixing scores* of the modification points, which refer to the ability to fix the program by modifying the source code of the corresponding points, are used for modification point navigation. The intuition is that *if modifying the source code at a modification point mp causes (some of) the initial failed test(s) to be passed, mp could be the correct position of the fault or have relations with the fault.* Otherwise, modifying its source code cannot change the results of the failed tests. The modification point with a high fixing score and high suspiciousness score should be prioritized to attempt in each subsequent repair iteration.

After a modification point is selected, APR tools generate and *select suitable modifications* for that point. The modifications are evaluated by executing tests (Martinez and Monperrus, 2016; Li et al., 2022; Yang et al., 2023). This dynamic validation is time-consuming and costs a large amount of resources. To mitigate the wasted time of validating incorrect modifications, we introduce *modification suitability measurement* for quickly evaluating and eliminating unsuitable modifications. The suitability of a modification at position mp is evaluated by the similarity of that modification with the original source code and with the previous attempted modifications at mp. The intuition is that *the correct modification at mp is often similar to its original code and the other successful modifications at this point, while the modifications similar to the failed modifications are often incorrect*. Thus, the more similar a modification is to the original code and to the successful modifications, and the less similar it is to the failed modifications, then the more suitable that modification is for attempting at mp.

We embed these heuristic rules in single-product-based and multi-product-based approaches; these versions are called $\texttt{SingleProd}_{\texttt{heu}}$ and $\texttt{MultiProd}_{\texttt{heu}}$.

To evaluate the proposed approaches, we conduct several experiments with $\texttt{SingleProd}_{\texttt{basic}}$, $\texttt{MultiProd}_{\texttt{basic}}$, $\texttt{SingleProd}_{\texttt{heu}}$, and $\texttt{MultiProd}_{\texttt{heu}}$ on a dataset of 318 buggy versions of 5 SPL systems (i.e., 318 variability bugs). The experimental results show that the single-product-based approach is considerably better than the multi-product-based approach by **12 to 30 times** in the number of plausible fixes and about **20 times** in the number of correct fixes. Interestingly, our heuristics could help to boost the performance of both single-product-based and multi-product-based approaches by up to **200%**. Moreover, the repair performance could be negatively impacted by FL tools since the modification points are selected based on FL results which are often imperfect. To mitigate the impact of the third-party FL tool, we assess the effectiveness of the repair approaches in the setting in which correct FL results are provided. In this experiment, we observe that the single-product-based approach is better than the multi-product-based approach about **3 times** in effectiveness and **9 times** in efficiency. In addition, the proposed heuristic rules help to increase **30–150%** the number of correct fixes and decrease **30–70%** the number of attempted modification operations of the corresponding basic approaches.

This paper makes the following contributions:

- The single-product-based and multi-product-based approaches for repairing variability bugs in the source code of SPL systems.
- Heuristic rules for navigating modification points and selecting suitable modifications to improve the performance of APR tools.
- An extensive experimental evaluation showing the performance of the approaches.

The implementation of our approaches and the experimental results can be found at Nguyen et al. (2024).

Paper structure. Section 2 introduces the concepts of APR and SPL, and the problem of repairing variability bugs in SPL systems. Section 3 reviews some related works. Section 4 proposes the basic single-product-based and multi-product-based approach ($\texttt{SingleProd}_{\texttt{basic}}$ and $\texttt{MultiProd}_{\texttt{basic}}$) for repairing variability bugs. Then, the enhanced variants of these approaches ($\texttt{SingleProd}_{\texttt{heu}}$ and $\texttt{MultiProd}_{\texttt{heu}}$), which embed the heuristic rules, are introduced in Section 5. The experimental setup and results are presented in Section 6 and Section 7. Finally, Section 8 discusses threats that may affect the validity of the approach, and Section 9 concludes the paper.

2. Concepts and problem statement

2.1. Automated program repair

To reduce the software maintenance cost, multiple APR techniques have been proposed in the past. The most popular APR approach is *test-suite-based program repair* (Liu et al., 2020; Monperrus, 2014; Gazzola et al., 2019), such as GenProg (Le Goues et al., 2012b), Nopol (Xuan

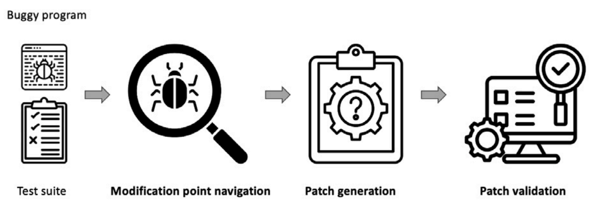


Fig. 1. Standard steps in test-suite-based program repair.

```
public int getGrade(int matrNr) throws
      ExamDataBaseException{
  int i = getIndex(matrNr);
  if(students[++i] != null &&
       !students[i].backedOut){
    //Patch:  if(students[i] != null &&
         !students[i].backedOut)
    return pointsToGrade(students[i].points, 0);
  }
  throw new ExamDataBaseException("Matriculation
       number not found");
}
```

Fig. 2. An example of buggy code snippet.

et al., 2017), and Cardumen (Martinez and Monperrus, 2018), which use test suites as the specification of the program's expected behaviors. For repairing a program failed by at least one test, these APR approaches attempt to generate candidate patches. Then, the available test cases are used to check whether the generated patches can fix the program.

In practice, the test-suite-based program repair tools are commonly implemented in three steps, as shown in Fig. 1. First, code elements of the program under repair are selected as the positions for attempting to fix by the **modification point navigation** step. In this step, to narrow down the search space, an FL technique can be applied to detect and rank suspicious code elements according to their suspiciousness. Then, the probability of being selected of the code elements is often decided based on their suspiciousness scores. Next, the **patch generation** step generates candidate patches for the selected code positions. A patch can be generated by multiple different techniques. For example, GenProg (Le Goues et al., 2012b) generates patches by using existing code from the program under repair, or Nopol (Xuan et al., 2017) collects running time information to build repair constraints and then uses a constraint solver to synthesize patches. Finally, a patch is validated by the test suites of the program to check whether the patched program meets the expected behaviors (**patch validation**).

The concepts of APR used in this paper are formally defined as follows:

Definition 1 (*Modification Point*)**.** A modification point $mp = (pos, c_o)$ is a code element that can be modified to repair the buggy program, in which pos is the position of the code element in the program under repair and c_o is its associated (original) code.

For example, in GenProg (Le Goues et al., 2012b; Martinez and Monperrus, 2016), which repairs the program at the statement level, a modification point is a suspicious statement in the program. For the buggy code in Fig. 2, a modification point of GenProg could be any suspicious statement in this code, such as $mp = (s_3, \texttt{if(students[++i]!=null\&\&!students[i].backedOut)})$. Instead, in Cardumen (Martinez and Monperrus, 2018), which fixes the program at the expression level, a modification point is an expression in a suspicious statement. For the suspicious statement s_3 in Fig. 2, each of its expressions could be a modification point in Cardumen, such as $mp = (s_3, \texttt{students[++i] != null})$.

Definition 2 (*Modification Operation*)**.** Given a modification point $mp = (pos, c_o)$, a *modification operation* $d = op(mp, c_n)$ is the transformation from the original code c_o to a new code by applying the repair operator op with the code c_n at the position pos. We consider $op \in \{rem, rep, ins\_bef, ins\_aft\}$, where rem, rep, ins_bef, and ins_aft are *remove*, *replace*, *insert before*, and *insert after* operators, respectively. The transformation of each modification operator is defined as follows:

- $rem(mp, c_n) = (pos,);$
- $rep(mp, c_n) = (pos, c_n);$
- $ins\_bef(mp, c_n) = (pos, c_n + c_o);$
- $ins\_aft(mp, c_n) = (pos, c_o + c_n);$

For a modification point mp, a modification operator can be applied to transform the source code at this point. Namely, the operator rem removes code at mp and the operator rep replaces the code c_o at mp with a new code c_n. In addition, the operator ins_bef adds the new code c_n before the original code c_o, while the operator ins_aft appends the new code c_n to the original code c_o; in ins_bef and ins_aft (in Definition 2), the concatenation of code is expressed via operator +.

In order to generate the new source code for applying insert/replace operators, several approaches (Le Goues et al., 2012b; Martinez and Monperrus, 2018; Yang et al., 2023; Li et al., 2022) leverage the *ingredients* from the program under repair or from the other projects. Instead, other approaches synthesize new code without using ingredients, such as jMutRepair (Martinez and Monperrus, 2016) or Nopol (Xuan et al., 2017).

Definition 3 (*Candidate Patch*)**.** A *candidate patch* (or *patch* for short) is the transformation result of a list of one or more modification operations.

In general, a patch could consist of one or more modification operations since a buggy program could be fixed by modifying one or several code statements. A *valid patch* is a candidate patch which passes all the available test cases of the program. Originally, the number of valid patches was a common metric to measure the performance of APR tools (Le Goues et al., 2012b,a). However, a test suite is often weak and inadequate (Ye et al., 2021b; Qi et al., 2015; Smith et al., 2015; Jiang et al., 2019), and it cannot cover all the behaviors of the program. Therefore, despite passing all the available test cases, a patch could still break other behaviors or introduce new faults, which are not covered by the given test suite (Smith et al., 2015). Such a valid patch is then referred to as a *plausible patch* or *test-adequate patch*, which needs to be further manually investigated by developers to ensure its correctness.

2.2. Software product line

A *software product line* (SPL) is a family of *products* which share a common code base. Each product is distinguished from the others in terms of its selected *features* (Apel et al., 2013a). In an SPL system, not all the combinations of the features are valid and result in a product. To create a valid combination, a *feature model* can define the dependencies and constraints between the features. A concrete product is constructed by composing the features based on the given configuration and the order of the features (Apel et al., 2013a; Thüm et al., 2014; Meinicke et al., 2017).

There are different ways to implement variability at the source code level (i.e., in the *solution space*), like *feature-oriented programming* (Apel et al., 2013a) and *delta-oriented programming* (Schaefer et al., 2010); they all allow to derive concrete products that implement valid feature configurations. In this work, our approach operates on concrete products of the SPL system. Therefore, we do not rely on any specific implementation of variability; in other words, we can use concrete products derived from any SPL system which can be implemented by different variability paradigms.

Table 1
The tested products of ExamDB system and their test results.

		Feature							
		ExamDB	BonusPoints	BackOut	Statistics	BonusPoints-BackOut	BonusPoints-Statistics	BackOut-Statistics	BonusPoints-BackOutStatistics
Product	p_1	T	F	F	F	F	F	F	F
	p_2	T	T	T	F	T	F	F	F
	p_3	T	T	F	T	F	T	F	F
	p_4	T	F	T	T	F	F	T	F
	p_5	T	T	T	T	T	T	T	T
	p_6	T	T	F	F	F	F	F	F
	p_7	T	F	F	T	F	F	F	F
	p_8	T	F	T	F	F	F	F	F

T means that the corresponding feature is enabled and F means that the corresponding feature is disabled in the product.
p_4 and p_8 fail at least one test (*failing products*). Other products pass all their tests (*passing products*).

2.3. Software product line testing

To test an SPL system $\mathfrak{S}$, the system is often sampled into a set of products $P = \{p_1, \ldots, p_n\}$, and each product p_i is tested against a corresponding test suite T_i. Given a variability bug in $\mathfrak{S}$, some products may fail their tests, but some other products may still pass all their test cases.

Definition 4 (*Variability Bug Nguyen et al., 2022*)**.** Given a buggy SPL system $\mathfrak{S}$ and a set of sampled products of the system P, a *variability bug* is an incorrect code statement in $\mathfrak{S}$ that causes the unexpected behaviors (failures) in a set of products which is a non-empty strict subset of P.

In other words, based on the test results, the sampled product set P of the *buggy SPL system* $\mathfrak{S}$ can be separated into two non-empty strict subsets $P = P_P \cup P_F$, where P_P is the set of *passing products* which pass all their tests, while P_F is the set of *failing products* in which each product failed at least one test.

Fig. 3 shows a variability bug in the ExamDB system. This system has 8 features, and the bug occurs in the feature named BackOut (line 5, Fig. 3(b)). The *feature model* (Apel et al., 2013a) and *feature order* (Prehofer, 2001) of this system are defined in Fig. 3(a). This system is sampled and tested by 8 products. The corresponding configurations and test results of these products are shown in Table 1. This variability bug causes products p_4 and p_8 to fail at least one test of their test suites, while the other products pass all their tests, i.e., $P_F = \{p_4, p_8\}$ and $P_P = \{p_1, p_2, p_3, p_5, p_6, p_7\}$. In this system, method getGrade of class ExamDataBaseImpl is implemented by both features BackOut and BonusPointBackOut. If BackOut is enabled and BonusPoints-BackOut is disabled, the buggy version of the method getGrade is included in the source code of the product and causes the product failure. Instead, if the feature BonusPointsBackOut is enabled, the correct version of getGrade implemented in this feature (Fig. 3(c)) will be composed in the product. Note that, even if both BackOut and BonusPointBackOut are enabled, the correct method getGrade of BonousPointBackOut will still be included in the product's source code, according to the defined feature order (shown in Fig. 3(a)). As a result, the products such as p_2 and p_5 (having both features enabled) have correct behaviors and pass all their tests.

The suspicious statements detected and ranked by the FL tool VarCop (Nguyen et al., 2022) for this variability bug are S including several statements such as $ExamDB.ExamDataBaseImpl.58$ at rank 1st and $BackOut.ExamDataBaseImpl.26$ at rank 2nd, etc. In particular, $ExamDB.ExamDataBaseImpl.58$ is the statement at line 58 in class ExamDataBaseImpl of the feature ExamDB (not shown in Fig. 3). Besides, the statement $BackOut.ExamDataBaseImpl.26$ is statement at line 26 in class ExamDataBaseImpl of the feature BackOut (i.e., the statement s_5 in Fig. 3(b)). The repair approach could select these statements as modification points for fixing. The modification of statement $ExamDB.ExamDataBaseImpl.58$ affects all the products of the system since the feature ExamDB is enabled in all of the products. Instead, the modification of statement $BackOut.ExamDataBaseImpl.26$ affects the products p_4 and p_8, whose feature BackOut is enabled, and they both contain this statement. Although feature BackOut is also enabled in two products p_2 and p_5, the statement $BackOut.ExamDataBaseImpl.26$ is not contained by these two products, so modifying it will not affect the behaviors of p_2 and p_5.

Repairing variability bugs in SPL systems means fixing the buggy statements to not only cause all the failing products to pass their tests but also not break the behaviors of any passing products of the systems.

Definition 5 (*Repairing Variability Bugs in an SPL System*)**.** Let us consider the 4-tuple $\langle \mathfrak{S}, P, \mathcal{T}, S \rangle$, where:

- $\mathfrak{S}$ is an SPL system containing variability bugs.
- $P = \{p_1, \ldots, p_n\}$ is the set of n sampled products, $P = P_P \cup P_F$, where P_P and P_F are the sets of *passing* and *failing* products of $\mathfrak{S}$.
- $\mathcal{T} = \{T_1, \ldots, T_n\}$ is a set of test suites, where $T_i \in \mathcal{T}$ is the test suite of product p_i (for each $i \in \{1, \ldots, n\}$).
- $S = \{s_1, \ldots, s_k\}$ is the ranked list of suspicious statements of the system $\mathfrak{S}$ which could be obtained by an FL technique.

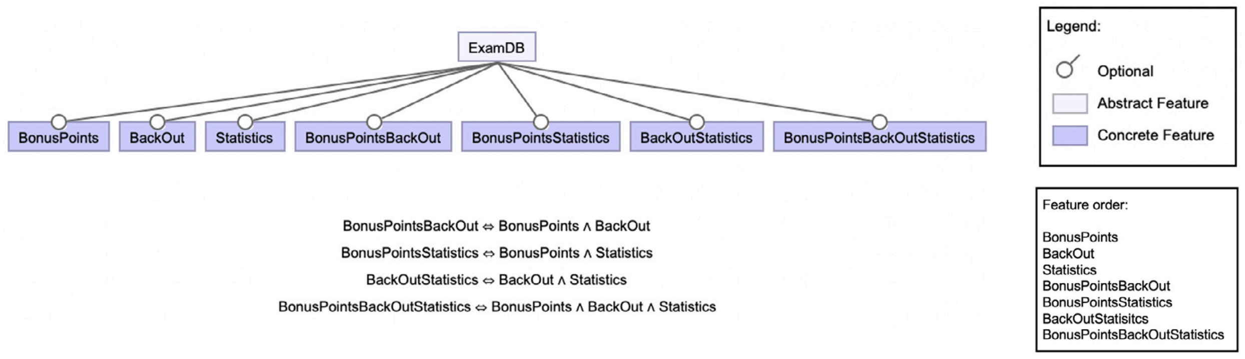
Repairing variability bugs in an SPL system consists in finding candidate patch(es) which make all the available tests in $\mathcal{T}$ pass.

3. Related work

3.1. Automated program repair

In recent years, APR has attracted a lot of attention from both industry and academia. Various APR techniques have been proposed. Goues et al. (2019) divided the APR techniques into three main groups: heuristic-based (Qi et al., 2014; Le Goues et al., 2012b; Yuan and Banzhaf, 2020a; Jiang et al., 2018; Xin and Reiss, 2017; Yang et al., 2023), constraint-based (Nguyen et al., 2013; Xuan et al., 2017), and learning-based repair (White et al., 2019; Li et al., 2022; Gupta et al., 2017; Jiang et al., 2021; Lutellier et al., 2020).

In *heuristic-based repair* direction, a search strategy such as random search (Qi et al., 2014), genetic programming (Le Goues et al., 2012b), or multi-objective genetic programming (Yuan and Banzhaf, 2020a) is leveraged to guide the search of valid patches. For example, GenProg (Le Goues et al., 2012b) is one of the most well-known program repair tools which uses a genetic programming technique to guide the patch generation process. This tool has been demonstrated to be able to fix real bugs in non-trivial programs. However, since the genetic algorithm needs to measure fitness values and distinguish better and worse patches, one general problem of APR tools using this technique is that they are computationally expensive. Besides, Qi


ExamDB
BonusPoints
BackOut
Statistics
BonusPointsBackOut
BonusPointsStatistics
BackOutStatistics
BonusPointsBackOutStatistics
Legend:
Optional
Abstract Feature
Concrete Feature
BonusPointsBackOut ⇔ BonusPoints ∧ BackOut
BonusPointsStatistics ⇔ BonusPoints ∧ Statistics
BackOutStatistics ⇔ BackOut ∧ Statistics
BonusPointsBackOutStatistics ⇔ BonusPoints ∧ BackOut ∧ Statistics
Feature order:
BonusPoints
BackOut
Statistics
BonusPointsBackOut
BonusPointsStatistics
BackOutStatisitcs
BonusPointsBackOutStatistics

(a) The feature model of the ExamDB system

```
public class ExamDataBaseImpl{

public int getGrade(int matrNr) throws
        ExamDataBaseException{
   int i = getIndex(matrNr);
   if(students[++i] != null &&
        !students[i].backedOut){
      //Patch:  if(students[i] != null &&
           !students[i].backedOut)
      return pointsToGrade(students[i].points, 0);
   }
   throw new ExamDataBaseException("Matriculation
        number not found");
}
}
```

(b) Feature BackOut

```
public class ExamDataBaseImpl{

public int getGrade(int matrNr) throws
        ExamDataBaseException{
    int i = getIndex(matrNr);
    if(students[i] != null &&
        !students[i].backedOut){
      return pointsToGrade( students[i].points,
            students[i].bonusPoints);
    }
    throw new ExamDataBaseException("Matriculation
        number not found");
}
}
```

(c) Feature BonusPointBackOut

Fig. 3. Feature model and class ExamDataBaseImpl in different features of ExamDB system.

et al. (2014) introduce RSRepair, in which a patch is generated by a random-search technique which is far less complicated compared to a genetic algorithm. Their experiments have shown that RSRepair is more efficient and effective than GenProg. Recently, various advanced APR tools in this search-based direction have been introduced, and they have shown their high performance, such as SimFix (Jiang et al., 2018), Arja (Yuan and Banzhaf, 2020a), and TransplantFix (Yang et al., 2023).

In *constraint-based repair* direction, repair constraints that the patched code should satisfy, are constructed. The patches are generated by solving such constraints. For example, SemFix (Nguyen et al., 2013) is a representative constraint-based APR. This tool generates repair constraints based on the expression in the buggy location using (controlled) symbolic execution program synthesis. In another research, Xuan et al. (2017) introduced Nopol, which collects runtime information, including variable and actual values, to construct constraints and then uses an SMT solver to synthesize patches for fixing buggy conditional statements.

In *learning-based repair* direction, the APR tools leverage machine learning/deep learning algorithms to learn from a large corpus of existing (correct) patches and generate candidate patches for a newly encountered program. For example, DeepRepair (White et al., 2019) uses code similarities, which are reasoned by a deep learning model, to select and transform ingredients. In addition, DEAR (Li et al., 2022) adopts tree-based LSTM models and uses a divide-and-conquer strategy to learn proper code transformations. It can generate fixes by modifying one or several statements at the same time.

Furthermore, due to advancements in deep learning methodologies, particularly the emergence of large language models, various APR approaches leveraging neural networks for acquiring bug-fixing patterns from large corpus of source code (Zhang et al., 2023; Fan et al., 2023; Xia et al., 2023; Joshi et al., 2023; Jin et al., 2023). These methods typically conceptualize APR as a machine translation task, wherein defective code snippets (source language) are automatically translated into corrected code snippets (target language). Leveraging the robust learning capacity of deep learning to discern concealed relationships within past bug-fixing datasets, these approaches have demonstrated impressive performance.

In this research, instead of introducing a new patch generation method, we focus on adapting the APR tools on a new context, i.e., fixing variability bugs of the SPL systems. Generally, most of these APR tools can be adopted to generate patches in our proposed approaches.

3.2. Software product line

In order to *guarantee the quality of SPL systems*, there are various studies about variability-aware analysis for the purpose of type checking (Kästner et al., 2012), testing (Wong et al., 2018), and control/data flow analysis (Bodden et al., 2013). Moreover, several approaches about configuration selection (Machado et al., 2014), configuration prioritization (Nguyen et al., 2020), and coincidental correctness detection (Nguyen et al., 2023; Nguyen and Vo, 2022), have been proposed to improve the efficiency of the SPL quality assurance process.

For *fault localization*, several approaches have been introduced for localizing variability bugs at different levels. Specifically, Arrieta et al. (2018) propose a method using SBFL metrics to localize bugs at the *feature-level*. To support debugging process, they also create valid products of minimum size containing the most suspicious features. In addition, VarCop (Nguyen et al., 2022) focuses on localizing variability bugs at the *statement-level*. This tool analyzes the interaction of the features to detect suspicious partial configurations, i.e., which is the minimum set of features needed to reveal the bugs. Based on the detected suspicious partial configurations, it detects suspicious statements and ranks them according to their suspiciousness scores.

For *repairing buggy* SPL systems, Arcaini et al. (2017) have proposed a solution to repair *variability models* to prevent incorrect configurations from being generated in the solution space. Besides, the wrong configurations, which violate the model constraints, are solved by Xiong et al. (2015). In that work, they use a constraint solver to automatically generate range fixes and ensure the desired properties of the generated fixes. Henard et al. (2013) proposed an approach to remove faults from feature models; given a feature model, through a cycle of test-and-fix, the approach tries to improve it by removing its wrong constraints. Furthermore, Echeverría et al. (2016) realize the importance and complexity of bug-fixing in the SPL system. They conducted an empirical study to analyze the patches of industrial SPL systems, which were manually fixed by engineers. This study also confirms that fixing variability bugs is challenging, especially propagating the fix when the source of the bug is in the feature interactions.

In this work, we attempt to fix variability bugs in the source code of SPL systems. To identify buggy elements, we use the output of VarCop (Nguyen et al., 2022), which can localize variability bugs at the fine-grain statement-level. Our work is aligned with the studies of Arcaini et al. (2017), Xiong et al. (2015), and Henard et al. (2013) regarding automated repairing SPL systems. However, our repairing targets are different. Indeed, Arcaini et al. (2017), Xiong et al. (2015), and Henard et al. (2013) target at wrong models and misconfigurations, while we focus on fixing incorrect code statements.

4. Automated program repair for variability bugs in SPL systems

This section introduces two approaches to repair variability bugs of an SPL system as defined in Definition 5: single-product-based (`SingleProd`$_{\texttt{basic}}$) and multi-product-based (`MultiProd`$_{\texttt{basic}}$), which not only try to fix the incorrect behaviors of the failing products but also try to not break the correct behaviors of the passing products.

The *single-product-based approach* individually repairs each failing product $p_i \in P_F$. Then, the obtained patches, which cause p_i to pass all its tests, are propagated and validated on the other products of the system. In this approach, *during the repair process, only the information of the product under repair p_i is considered for generating, evaluating, and evolving the patches*. The other products of the system are used to validate the obtained patches after the process of repairing p_i.

For the *multi-product-based approach*, instead of repairing one individual product at a time, all the products in P are considered at the same time during repair. By this method, *in each iteration of the repair process, the patches are generated, evaluated, and/or evolved based on the information of all the products in P of the system.*

For both approaches, the valid patches are those causing all the available tests of all the products in P to pass.

4.1. Single product-based approach (`SingleProd`$_{\texttt{basic}}$)

The single-product-based approach (`SingleProd`$_{\texttt{basic}}$) is shown in Alg. 1. Specifically, an APR tool R is adopted to generate patches for each failing product $p_i \in P_F$ with its contained suspicious statements S_i, $S_i \subseteq S$ (lines 4-5). Note that S is the list of suspicious statements of the whole system $\mathfrak{S}$. In this work, we employ a testing-based FL technique, VarCop (Nguyen et al., 2022), to identify suspicious statements. However, in general, one could use other FL techniques such as deep learning-based (Yang et al., 2024), code smell-based (Takahashi et al., 2021, 2018), or combine them to better localize the faults.

For each failing product p_i, we need to map each suspicious statement $s \in S$ to the corresponding statements in the product. In other words, S_i (line 4) consists of the suspicious statements which occur in p_i, $S_i \subseteq S$. During the repair of product p_i, only the suspicious statements contained in this product are considered. The process of generating patches `PatchGeneration` (i.e., line 5 in Alg. 1) is described in details in Section 4.1.1.

`PatchGeneration` gives as output the set of candidate patches C_i which make all the tests T_i of p_i to pass (line 5). Each candidate patch $c \in C_i$ is then propagated and validated on all the other products in P (line 6). If a candidate patch causes all the tests in $\mathcal{T}$ to pass, then the patch is *valid* for fixing the system $\mathfrak{S}$. If none of the candidate patches

Algorithm 1: `SingleProd`$_{\texttt{basic}}$ algorithm

```
1  Function ProductBasedApproach($P$, $\mathcal{T}$, $S$)
2      $validPatchSet \leftarrow \emptyset$
3      for $p_i \in P_F$ do
4          $S_i \leftarrow$ SuspStmtInProductMapping($S$, $p_i$)
5          $C_i \leftarrow$ PatchGeneration($p_i$, $T_i$, $S_i$)
6          $validPatchSet \leftarrow$ PatchGlobalValidation($P$,$\mathcal{T}$,$p_i$,$C_i$)
7          if $validPatchSet$.size() > 0 or timeout then
8              break
9          end
10     end
11     return $validPatchSet$
```

Algorithm 2: Patch generation algorithm in `SingleProd`$_{\texttt{basic}}$

```
1  Function PatchGeneration($p_i$, $T_i$, $S_i$)
2      $C_i \leftarrow \emptyset$ //valid patches of product $p_i$
3      $testResult \leftarrow$ TestResultInitialization($p_i$, $T_i$)
4      while $\neg searchStop$ do
5          $mp \leftarrow$ ModificationPointSelection($S_i$)
6          $d \leftarrow$ ModificationOperationGeneration($p_i$, $mp$)
7          $c \leftarrow$ PatchConstruction($d$, $testResult$)
8          $testResult \leftarrow$ PatchLocalValidation($p_i$, $T_i$, $c$)
9          if $\forall t \in testResult$, $t$ is a passed test then
10             $C_i.add(c)$
11         end
12     end
13     return $Ci$
```

in C_i causes all the tests in $\mathcal{T}$ to pass, the process continues by trying to fix the other failing products of the system (line 3).

The fixing process stops if one of these conditions is satisfied: (i) there is at least a candidate patch passing all the available tests of all products in P, (ii) all the failing products in P_F have been attempted to be fixed by R, or (iii) the time execution limitation is reached.

4.1.1. Patch generation

Alg. 2 shows the process of generating patches for fixing an individual failing product $p_i \in P_F$ (i.e., line 5 in Alg. 1). In general, any test-suite-based program repair APR tool R can be employed as the `PatchGeneration` procedure. For a selected modification point mp (line 5), depending on the employed APR tool, a modification operation can be generated by considering or not some specific information; such information could be taken from the program under repair or from outside the program. Without loss of generality, in this algorithm, we only set the product under repair p_i as an input for generating modification operations (line 6), and the other information is excluded.

The generated modification operations d are then used to construct candidate patches (line 7). A patch could contain one or multiple modification operations, and it could also be obtained through a crossover operation or evolved from the previous patches. In this algorithm, `PatchConstruction` is also an abstract function representing the corresponding function of any APR tool. We set two inputs for this function, including the modification operation d and the test results $testResult$ of the previous patch or of the original product if no patch has been found so far. In practice, several APR approaches such as GenProg (Le Goues et al., 2012b) need the $testResult$ to measure the fitness, navigate the search, and the evolution process during generating patches. Instead, some other APR approaches do not need the $testResult$ of the previous patch. For example, jMutRepair (Martinez and Monperrus, 2016) can synthesize a new patch in each iteration of the repair process without evolving from the other attempted patches. To keep the algorithm as simple as possible, some other additional information (e.g., the previous attempts) that could be needed in `PatchConstruction` is excluded in this algorithm.

Algorithm 3: Patch global validation algorithm in SingleProd$_{basic}$

```
Function PatchGlobalValidation($P$, $\mathcal{T}$, $p_i$, $C_i$)
    $validPatchSet \leftarrow \emptyset$
    for $c \in C_i$ do
        $valid \leftarrow true$
        for $p_k \in P \setminus \{p_i\}$ do
            $testResult_{p_k} \leftarrow$ PatchLocalValidation($p_k$, $T_k$,$c$)
            if $\exists t \in testResult_{p_k}$, $t$ is failed test then
                $valid \leftarrow false$
                break
            end
        end
        if $valid$ then
            $validPatchSet$.add($c$)
        end
    end
    return $validPatchSet$
```

After that, the function PatchLocalValidation (line 8) executes the patched product with the given test suite T_i and outputs the corresponding test results. If all the tests in T_i are passed, the corresponding patch is a valid patch of the product p_i (lines 9–10). Such valid patches will be globally validated on the other products of the systems (invocation at line 6 in Alg. 1, and definition in Alg. 3).

This patch generation process is iterated until the searchStop condition is satisfied (line 4). Without loss of generality, this condition could be the number of iterations, the number of attempted modifications, or the searching time, etc. In this work, we consider the searchStop condition in terms of searching time.

4.1.2. Patch validation

The process PatchGlobalValidation of validating the obtained patches of p_i on the other products of the system (line 6 in Alg. 1) is shown in Alg. 3. Different products of an SPL system could share multiple statements. Thus, a fix in one product needs to be propagated and validated by the other products of the system. Specifically, for validating the obtained patch $c \in C_i$ of p_i on the product p_k (being p_k a product in P different from p_i), all of the modifications in c are applied into the corresponding positions in p_k (lines 5–10). Note that there are cases that a modification in c cannot be applied in p_k since there does not exist a corresponding modification point in p_k. If all the modifications in c cannot be applied in p_k, this patch will not impact p_k's behaviors. This means that the validation result of p_k with such a patch corresponds to its original test result. If a patch $c \in C_i$ causes all the tests of all products in P to pass, c is a valid patch to fix the system $\mathfrak{S}$ (lines 12–14).

4.2. Multi product-based approach (MultiProd$_{basic}$)

The multi-product-based approach (MultiProd$_{basic}$) for repairing a buggy SPL system $\mathfrak{S}$ is shown in Alg. 4. This algorithm attempts to repair all the products of the system $\mathfrak{S}$ at the same time. For each repair iteration, the modification point mp is selected in the ranked list of all the suspicious statements S of the system (line 5). Then, any APR tool R can be used to generate a suitable modification operation for the selected modification point (line 6). Similarly to Alg. 2, the patch can be constructed by considering only the newly generated modification operation d or evolved from the previous attempts (line 7). After that, the generated patch is applied to all the products $p_i \in P$ of the system and validated by all the test suites $T_i \in \mathcal{T}$ (lines 9–12). For a product p_i which does not contain any corresponding modification points of the modifications in a patch c, $testResult_{p_i}$ (line 10) is exactly the original test results of this product. Indeed, if p_i does not contain the statements modified by c, its source code cannot be changed when c is applied.

Algorithm 4: MultiProd$_{basic}$ algorithm

```
Function SystemBasedApproach($\mathfrak{S}$, $P$, $\mathcal{T}$, $S$)
    $validPatchSet \leftarrow \emptyset$
    $testResultSet \leftarrow$ TestResultsInitialization($P$, $\mathcal{T}$)
    while $validPatchSet.size() = 0$ and $\neg timeout$ do
        $mp \leftarrow$ ModificationPointSelection($S$)
        $d \leftarrow$ ModificationOperationGeneration($\mathfrak{S}$, $m_p$)
        $c \leftarrow$ PatchConstruction($d$, $testResultSet$)
        $testResultSet \leftarrow \emptyset$
        for $p_i \in P$ do
            $testResult_{p_i} \leftarrow$ PatchLocalValidation($p_i$, $T_i$, $c$)
            $testResultSet$.add($testResult_{p_i}$)
        end
        $valid \leftarrow true$
        for $testResult_{p_i} \in testResultSet$ do
            if $\exists t \in testResult_{p_i}$, $t$ is a failed test then
                $valid \leftarrow false$
                break
            end
        end
        if $valid$ then
            $validPatchSet$.add($c$)
        end
    end
    return $validPatchSet$
```

To construct a patch, similarly to the single-product-based approach, the test result of the previous patch is passed to PatchConstruction function. However, instead of using the test result of one product, the multi-product-based approach employs the test results of all the products, $testResultSet$, for measuring fitness values and guiding the search process in the next generation (line 7). Any patch which causes all the tests in $\mathcal{T}$ to pass is a valid patch (line 13–22). The repair stops (guard at line 4) when: (i) at least a valid patch is found or (ii) the time budget is exhausted.

4.2.1. Computational improvement

In each repair iteration, the patches are generated, evaluated, and/or evolved based on the test information of all the sampled products P of the system. Thus, all the products must be validated after each patch c is constructed (lines 9–12). However, executing tests of all products P in each repair iteration is very time-consuming. To boost the efficiency of Alg. 4, one could improve the fitness function and stop validating a patch as soon as there is a patched product that failed. For example, lines 14–18 could be moved into the loop in line 9, and a break statement added to stop validating other products right after a product fails its test(s).

Although this method could enhance the patch validation efficiency, the precision of the fitness function could be negatively affected. Indeed, the fitness functions used to evaluate the patches are based on the number of passed and failed tests (Le Goues et al., 2012b; Martinez and Monperrus, 2019, 2018). Early stopping right after a test fails, would cause a loss of information about the non-executed test cases. Therefore, to effectively and efficiently validate patches in MultiProd$_{basic}$, it would require a different fitness function that can handle the lack of some test information. This is beyond the scope of this research, and we will investigate it in our future work.

4.3. Single product-based approach vs. multi product-based approach

The main difference between the multi-product-based and single-product-based approaches is the way of generating and validating patches. Namely, for each repair iteration, *in the multi-product-based approach, the PatchConstruction function synthesizes patches, measures the fitness values, and/or evolves patches by considering the test results* *of* ***all the products*** P *of the system* (line 7, Alg. 4). Instead, *in the single-product-based approach, the test results of* ***only the product under repair*** *are considered* (line 7, Alg. 2).

For example, the APR approaches which adopt evolutionary algorithms, such as jGenProg (Martinez and Monperrus, 2016), often use a fitness function to guide the evolution of a population of patches throughout some generations. Specifically, in a given generation k, the patches with better fitness values will be selected for evolving at generation $k+1$. The default fitness function in jGenProg counts the number of failed tests. The lower number of failed tests, the better a patch. By adopting this type of APR tool, in the single-product-based approach, the fitness value is the number of failed tests in T_i of the product under repair p_i only. Instead, in the multi-product-based approach, the fitness value is the total number of failed tests of all the tests in $\mathcal{T}$ of all the products of the system.

For the single-product-based approach, the APR tool R can quickly evaluate the patches as the number of tests to execute is much less compared to the multi-product-based approach. However, in the single-product-based approach, the candidate patches could be biased by the product under repair because they are evaluated using the tests of only one product. Consequently, it could be less effective in generating a good patch for the whole system. In contrast, in the multi-product-based approach, the generated patches could be better in the general context of the system, as they are evaluated by the test suites of all the products. However, the number of tests that must be executed in each iteration could be very large. This could be inefficient in practice.

5. Heuristic rules for improving the repair performance of APR tools

This section introduces several heuristic rules to improve the APR tools' performance by guiding them to effectively and efficiently navigate modification points and select suitable modifications. We show how to apply these heuristic rules on both single-product-based and multi-product-based approaches to boost their performance in repairing variability bugs.

5.1. Heuristic rules

5.1.1. Modification point navigation ($ModNav$)

In order to decide the positions in the program for synthesizing patches, APR tools often select modification points based on their *suspiciousness scores* measured in advance by FL techniques. In previous studies (Ghanbari et al., 2019; Jiang et al., 2018; Wen et al., 2018), FL techniques, such as Ochiai (Abreu et al., 2007), have shown their effectiveness in creating the initial ranked list of suspicious statements and narrowing the search space for APR tools. However, the list of suspicious statements could be continuously better (re-)ranked, and the precision of modification point navigation could also be gradually refined by leveraging the intermediate results of program repair (Lou et al., 2020; Benton et al., 2022).

Indeed, the validation results of the modification operations can provide valuable feedback for correctly finding the positions of the faulty code elements. The key idea of heuristic rule $ModNav$ is that *for a modification point* mp*, if a modification operation at that point causes the initial failed test(s) to be passed,* mp *could be the correct position of the fault or have relations with the fault.* Instead, if modifying its code cannot change the failure states of the failed tests, probably mp is not the correct position of the fault. For example, Table 2 shows several modification operations generated to fix the bug in Fig. 3(b) and their corresponding validation results. In this table, the numbers of initial failed tests and passed tests are shown in the header of corresponding columns. For each row, we show the number of initial failed/passed tests which are still failed/passed after applying the modification operation. Although the modification operation $MO3$ at `BackOut.ExamDataBaseImpl.26` (i.e., statement s_5 in Fig. 3(b)) could not make all the tests passed, it helps to decrease the number of failed tests. Initially, there were 2 failed tests, and after applying this modification operation, the number of failed tests decreased to 1. This indicates that the modification operation $MO3$ is still incorrect in fixing the buggy statement, but the selected statement seems to be the correct position to be fixed. It should be attempted to continuously fix in the next iterations.

When applying rule $ModNav$, the navigation is guided by not only the *suspiciousness scores* of the code elements but also by their *fixing scores* measured based on the validation results of the attempted modification operations. The fixing score of a modification point $mp = (pos, c_o)$ indicates the ability to fix the program by modifying source code c_o at the position pos. In each repair iteration, *both suspiciousness and fixing scores are combined to navigate modification points*. Suspiciousness scores are calculated once in advance by off-the-shelf FL tools. Instead, fixing scores are initialized with the same values for all modification points and then continuously updated in each repair iteration after a modification operation is applied and validated.

For a modification point mp, if a modification operation at mp causes all the initial failed tests to pass and does not break any initial passed tests, that modification is the valid patch of the program under repair. However, in the case that not all of the tests are passed, the modification can still reflect the ability to fix the program at the corresponding point. In this study, we measure the fixing score of a modification point based on the information of to what extent the modification at that point fixes unexpected behaviors (i.e., whether some initial failed tests are fixed (changed to "passed")) and breaks correct behaviors (i.e., whether some initial passed tests are broken (changed to "failed")).

To measure the fixing scores and use them for guiding modification point navigation in the next iterations, we divide the modification operations into three groups based on the test results of the corresponding modified programs: *SomeFixedNoneBroken*, *SomeFixedSomeBroken*, and *NoneFixed*. Specifically, *SomeFixedNoneBroken* refers to the modifications causing some initial failed tests to be passed and do not break any initial passed tests. *SomeFixedSomeBroken* refers to the modifications causing some initial failed tests to be passed but breaking some initial passed tests. Finally, *NoneFixed* refers to the modifications which cannot change the state of any initial failed tests. For example, the modification operation $MO1$ in Table 2 is categorized in the group *NoneFixed*, since none of the failed tests is fixed. The modification operations $MO2$ and $MO3$ are in group *SomeFixedSomeBroken* and *SomeFixedNoneBroken*, respectively. Both $MO2$ and $MO3$ can fix one initial failed test. However, $MO3$ does not break any initial passed test, while $MO2$ breaks one. This intuitively shows that the ability to fix the program at the corresponding modification point of $MO3$ is better than that of $MO2$.

According to the state of fixing the initial failed tests and breaking the initial passed tests, the order of these groups in terms of ability to fix the program is as follows: (1) *SomeFixedNoneBroken*, (2) *SomeFixedSomeBroken*, and (3) *NoneFixed*. We follow this order to prioritize the selection of a modification point in the next iterations. This means that the modification point whose modification is in group *SomeFixedNoneBroken* will have the highest priority of being selected (i.e., highest fixing score). Particularly, the fixing score of a modification point is measured as follows:

Definition 6 (*Fixing Score*)**.** Given a program with test suite $T = T_f \cup T_p$, where T_f is the set of initial failed tests and T_p is the set of initial passed tests, after applying a modification operation $d = op(mp, c_n)$, the $fixing\_score$ of the modification point mp is:

$$fixing\_score(mp) = \begin{cases} 2 & \text{if } \exists t \in T_f,\ t \text{ becomes a passed test} \\ & \text{and } \nexists t' \in T_p,\ t' \text{ becomes a failed test} \\ 1 & \text{if } \exists t \in T_f,\ t \text{ becomes a passed test} \\ & \text{and } \exists t' \in T_p,\ t' \text{ becomes a failed test} \\ 0 & \text{otherwise} \end{cases} \quad (1)$$

Table 2
Example of modification operations for fixing the bug at statement s_5 in Fig. 3(b).

Statement ID	Modification operation	#failed tests (2)	#passed tests (116)
ExamDB.ExamDataBaseImpl.58	MO1	2	90
BackOut.ExamDataBaseImpl.26	MO2	1	115
BackOut.ExamDataBaseImpl.26	MO3	1	116

In each repair iteration, the modification point with the highest fixing score will be selected. If the modification points have the same fixing scores, the possibility of being selected is decided based on their suspiciousness scores. In this work, the suspiciousness scores are measured by the state-of-the-art FL tool VarCop (Nguyen et al., 2022).

Note that the detailed number of failed and passed tests could also somewhat reflect the fixing ability of the modification points. However, we do not consider such detailed information in measuring fixing scores. The reason is that, for a modification operation, the detailed number of passed and failed tests depends on not only the modification point but also on the specific modification (i.e., modified code). Such detailed test results could provide wrong indication for evaluating the correctness of the modification points. For example, a modification operation that attempts the correct position but using a wrong modification, could fix only some initial failed tests and break multiple passed tests. Instead, the sign of changing the state of a test from failed to passed is a visible evidence showing that the modification point is a buggy element or related to the bug. Thus, we prioritize that position to continuously try to attempt modifications there (fixing score is 2 or 1).

5.1.2. Modification suitability measurement ($ModSuit$)

After selecting a modification point, an APR tool generates and selects suitable modification operations to attempt to fix that point. To validate the suitability of a modification operation, all the tests (or, at least, all the failed tests) of the program under repair need to be re-executed. This dynamic validation is repeated multiple times during the repair process. This validation step is time-consuming and costs a large amount of resources. We here propose the heuristic rule $ModSuit$ that measures the *suitability* of a modification operation without executing tests. This could enhance both the efficiency and the effectiveness of APR tools.

The rule is based on the observation that the previous modifications of a modification point mp can provide empirical evidence for APR tools about which modifications should or should not be attempted. The reason is that the previous modifications of mp are the modifications that have been attempted and validated by tests. Their validation results can reveal whether similar modifications are suitable for that point or not. Additionally, a correct patch is often similar to the original code (Csuvik et al., 2020; Kim et al., 2023). Thus, for the modification point mp, we can leverage its original code and the previous modifications to quickly evaluate the suitability of a newly generated modification. The intuition is that a suitable modification d at mp could be similar to the original code of mp as well as the successful modifications (Definition 7) at this point, while it should not be similar to the failed modifications (Definition 8) which were tested and shown to be not effective.

Definition 7 (*Successful Modification*)**.** Given a program with the test suite T, a modification operation $d = op(mp, c_n)$ is a *successful modification* if after applying d, each test $t \in T$ is a passing test.

Definition 8 (*Failed Modification*)**.** Given a program with the test suite T, a modification operation $d = op(mp, c_n)$ is a *failed modification* if after applying d, there exists a test $t \in T$ that is failing.

In this work, *the suitability of a new modification operation $d = op(mp, c_n)$ is measured based on the similarity of the modified code c_m (if d is applied) with the original code at mp, and with the previous successful and failed modifications at mp.* The intuition is that a correct modification at mp is often similar to its original code and the other successful modifications at this point, while the modifications similar to the failed modifications are often incorrect. Thus, the suitability of a modification operation $d = op(mp, c_n)$ at mp depends on:

(i) the similarity between c_m and the original code c_o,
(ii) the similarity between c_m and the modified code of the previous successful modifications, and
(iii) the diversity between c_m and the modified code of the previous failed modifications.

If the suitability score of d is greater than a given threshold θ, we apply it to the program and validate it by tests. Otherwise, we can discard it without executing any test.

Given a modification point mp and its original source code c_o, let $D = D_f \cup D_s$ be the previous modifications at mp, where D_f is the set of failed modifications, and D_s is the set of successful modifications. For a modification operation $d = op(mp, c_n)$, the *suitability score* of attempting d at mp is measured as follows:

$$suitScore(d) = \frac{\alpha(suit(c_m)) + \beta(1 - unsuit(c_m))}{\alpha + \beta} \quad (2)$$

where c_m is the modified code obtained by applying d. In Eq. (2), $suit(c_m)$ tells how d is suitable to be applied at mp. It depends on the similarity of c_m with the original code, as well with the code obtained by successful modifications. $unsuit(c_m)$, instead, shows how d is unsuitable to be applied at mp, and it is given by the similarity of c_m and the failed modifications. Specifically, $suit(c_m)$ and $unsuit(c_m)$ are measured as follows:

$$suit(c_m) = \max(similarity(c_m, c_o), \max_{d_s \in D_s} similarity(c_m, c_{d_s})) \quad (3)$$

$$unsuit(c_m) = \max_{d_f \in D_f} similarity(c_m, c_{d_f}) \quad (4)$$

In Eqs. (3) and (4), c_{d_s} and c_{d_f} are the modified code of a successful operation $d_s \in D_s$ and of a failed operation $d_f \in D_f$, respectively. Overall, if $suitScore(d, mp) > \theta$, d is applied to the program and validated by executing tests (θ is a hyperparameter of the approach). Otherwise, we eliminate d without actually executing tests. Without loss of generality, any function which can measure the similarity of two sequences can be used in these equations. The impact of different similarity functions is experimented and shown in Section 7.2.2.

5.2. Applying the heuristic rules in the single-product-based and multi-product-based approaches

Fig. 4 shows the general process of APR when the proposed heuristic rules are applied. In general, for correctly selecting modification points, both the suspicious scores (measured by FL techniques) and fixing scores (continuously updated based on the intermediate validation results) are leveraged in the modification point navigation step. After that, the APR tool generates modification operations for the selected points. In this work, we do not propose any new modification generation approach, but we use those proposed by existing APR tools; however, we modify the tools by adding mechanisms to decide whether to apply the generated modification to the program based on its suitability score. The suitability of a modification is calculated according to its similarity with the original code and the previously attempted modifications of the selected point. If the suitability score exceeds a threshold, we apply that modification. Otherwise, we exclude it.

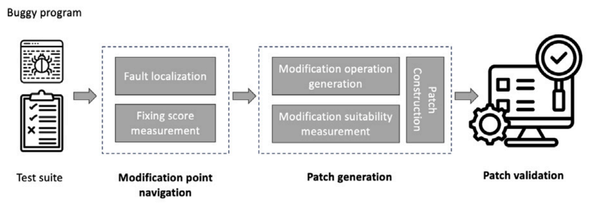


Fig. 4. The process of APR with the two proposed heuristic rules.

Algorithm 5: Patch generation algorithm in $\mathtt{SingleProd}_{\mathtt{heu}}$

```
Function PatchGeneration($p_i$, $T_i$, $S_i$)
    $C_i \leftarrow \emptyset$ //valid patches of product $p_i$
    $testResult \leftarrow$ TestResultInitialization($p_i$, $T_i$)
    while $\neg searchStop$ do
        $mp \leftarrow$ ModificationPointSelection($S_i$)
        $d \leftarrow$ ModificationOperationGeneration($p_i$, $mp$)
        $score \leftarrow$ ModificationSuitabilityMeasurement($d$, $mp$)
        if $score > \theta$ then
            $c \leftarrow$ PatchConstruction($d$, $testResult$)
            $testResult \leftarrow$ PatchLocalValidation($p_i$, $T_i$, $c$)
            if $\forall t \in testResult$, $t$ is a passed test then
                $C_i.add(c)$
            end
            FixingScoreUpdate($mp$, $testResult$)
        end
    end
    return $Ci$
```

Algorithm 6: $\mathtt{MultiProd}_{\mathtt{heu}}$ algorithm

```
Function SystemBasedAproach($\mathfrak{S}$, $P$, $\mathcal{T}$, $S$)
    $validPatchSet \leftarrow \emptyset$
    $testResultSet \leftarrow$ TestResultsInitialization($P$, $\mathcal{T}$)
    while $validPatchSet.size() = 0$ and $\neg timeout$ do
        $mp \leftarrow$ ModificationPointSelection($S$)
        $d \leftarrow$ ModificationOperationGeneration($\mathfrak{S}$, $m_p$)
        $score \leftarrow$ ModificationSuitabilityMeasurement($d$, $mp$)
        if $score > \theta$ then
            $c \leftarrow$ PatchConstruction($d$, $testResultSet$)
            $testResultSet \leftarrow \emptyset$
            for $p_i \in P$ do
                $testResult_{p_i} \leftarrow$ PatchLocalValidation($p_i$, $T_i$, $c$)
                $testResultSet$.add($testResult_{p_i}$)
            end
            $valid \leftarrow true$
            for $testResult_{p_i} \in testResultSet$ do
                if $\exists t \in testResult_{p_i}$, $t$ is a failed test then
                    $valid \leftarrow false$
                    break
                end
            end
            FixingScoreUpdate($mp$, $testResultSet$)
            if $valid$ then
                $validPatchSet$.add($c$)
            end
        end
    end
    return $validPatchSet$
```

5.2.1. $\mathtt{SingleProd}_{\mathtt{heu}}$

For the single-product-based approach, Alg. 5 shows how we apply the proposed heuristic rules in repairing an individual product of the SPL system, i.e., the procedure of PatchGeneration for a product p_i. The algorithm is a modification of the one shown in Alg. 2 and described in Section 4.1; the new operations are highlighted in yellow, while the other functions are the same. In this variant of PatchGeneration, after each repair iteration, the fixing scores of the modification points are updated (line 14) and then they are combined with the suspiciousness scores for precisely selecting modification points in the next iterations (line 5). In addition, the suitability score of each modification operation is measured (line 7) and checked with a threshold (line 8) before actually applying it and validating it by executing tests (lines 9–13). Similarly to Alg. 2, the generation process stops when the condition searchStop (line 4) is satisfied; in this work, we use the maximum searching time as stopping condition.

In $\mathtt{SingleProd}_{\mathtt{heu}}$, instead of gradually selecting failing products (in a random manner as $\mathtt{SingleProd}_{\mathtt{basic}}$) for repairing, we sort and select the failing products of the system based on their complexity (***Failing product navigation*** ($ProdNav$)). We hypothesize that the less complex a product is, the easier the product can be fixed. In this work, we measure the complexity of a failing product $p_i \in P_F$ based on its source code size and its tests. The intuition is that the bigger the product is, the more complex it is. Also, the more failed tests the product has, the more incorrect behaviors it has. Particularly, the complexity of a product $p_i \in P_F$ having test suite $T_i = T_f \cup T_p$ is measured as follows:

$$complexity(p_i) = \frac{\frac{productSize_{p_i}}{systemSize} + \frac{|T_f|}{|T_i|}}{2} \tag{5}$$

where $productSize_{p_i}$ and $systemSize$ are measured at the statement level. $systemSize$ is the total number of all the statements in all the features of system $\mathfrak{S}$. $productSize_{p_i}$ is the number of statements contained in the product p_i.

5.2.2. $\mathtt{MultiProd}_{\mathtt{heu}}$

Alg. 6 shows the details of how we apply the heuristics in the multi-product-based approach. The algorithm is a modification of $\mathtt{MultiProd}_{\mathtt{basic}}$ shown in Alg. 4, and described in Section 4.2; the new operations are highlighted in yellow, while the other functions are the same. The main difference of this variant compared to Alg. 4 is the addition of the fixing scores and modification suitability measurement. After each repair iteration, the fixing scores of the modification points are updated (line 22), and then they are combined with the suspiciousness scores for precisely selecting modification points in the next iterations (line 5). In $\mathtt{MultiProd}_{\mathtt{heu}}$, the fixing scores are measured based on the results of all the tests in $\mathcal{T}$ of all the products. Additionally, the suitability score of each modification operation is measured (line 7) and checked with a threshold (line 8) before actually applying and validating by executing tests.

6. Experiment methodology

To evaluate our approaches in repairing variability bugs of SPL systems, we aim to answer the following research questions:

RQ1. Performance analysis. How effective and efficient are $\mathtt{SingleProd}_{\mathtt{basic}}$, $\mathtt{MultiProd}_{\mathtt{basic}}$, $\mathtt{SingleProd}_{\mathtt{heu}}$, $\mathtt{MultiProd}_{\mathtt{heu}}$ in repairing buggy SPL systems?

RQ2. Intrinsic analysis. How do the heuristic rules *failing product navigation* ($ProdNav$), *modification point navigation* ($ModNav$), *modification suitability measurement* ($ModSuit$) contribute to the repair performance? How do the hyperparameters α and β of modification suitability measurement ($ModSuit$) impact the repair performance?

RQ3. Sensitivity analysis. How do the characteristics of the SPL systems (i.e., the different systems, the number of failing products, and the number of suspicious statements) affect the repair performance?

6.1. Benchmarks

To evaluate the proposed variability bug repair approaches, we conducted experiments on a public set of benchmarks of variability bugs (Ngo et al., 2021). Currently, this is the only public dataset containing the versions of SPL systems that are affected by variability

Table 3
Benchmarks.

SPL system	#Products	#Avg. tests/product	#Buggy versions of the SPL
ZipMe	25	255	55
GPL	99	87	105
ExamDB	8	166	49
Email	27	86	36
BankAccount	34	20	73

bugs and have been found through testing. Table 3 shows the detailed information the dataset that we used in the experiments of this research. There are 5 SPL systems. Each SPL system has a different number of buggy versions, and each buggy version contains a single bug; for example, `ZipMe` has 55 buggy versions. In total, there are 318 buggy SPL systems, and each of them is considered as input of the proposed repair approaches. The table also reports the number of sampled products of each buggy system by using 4-wise combinatorial testing and the average number of tests per product. For example, `ZipMe` has been sampled into 25 products, and each product is tested by 255 test cases on average.

There are other datasets of variability bugs, but either they are not published, or they do not provide the test suites along with the bugs. Thus, they are not suitable for our experiments. For example, Arrieta et al. (2018) introduced a set of artificial bugs to evaluate their approach in localizing bugs in SPL systems; however, this dataset is not published. Abal et al. (2018) and Mordahl et al. (2019) published their datasets of real-world variability bugs, but, in these datasets, most of the bugs are compile-time bugs and they are not provided with tests.

6.2. Evaluation procedure and metrics

6.2.1. Evaluation procedure

Performance analysis.

Experiment setting: We conducted the experiments and analyzed the repair performance of all four proposed approaches($\texttt{SingleProd}_{\texttt{basic}}$, $\texttt{MultiProd}_{\texttt{basic}}$, $\texttt{SingleProd}_{\texttt{heu}}$, and $\texttt{MultiProd}_{\texttt{heu}}$) in two settings:

`withFL` (*with fault localization*). In this setting, we apply the FL approach *VarCop* (Nguyen et al., 2022) to detect and rank suspicious statements. Then, the output of VarCop is used as the input for all four repair approaches.

`withoutFL` (*without fault localization*). In this setting, we aim to analyze the performance of the APR approaches without the possible negative impact of the third-party FL tool; therefore, we provide as input to the repair approaches only the statements which are actually buggy.

In both settings `withFL` and `withoutFL`, the time execution limit of all the approaches (i.e., *timeout* in Algs. 1, 4, and 6) for fixing a buggy SPL system is 60 min. The *searchStop* condition of `PatchGeneration` (Algs. 2 and 5) is also set to 60 min.[1]

APR tool selection: We select two representative APR tools R, jGenProg (Martinez and Monperrus, 2016) and Cardumen (Martinez and Monperrus, 2018), to be used as underlying repair tools of our proposed approaches, as explained in Section 4. The two APR tools generate modifications at different levels of code elements in Java programs. Specifically, jGenProg is the implementation of the popular APR tool GenProg (Le Goues et al., 2012b) for Java programs. It repairs buggy programs at the *statement-level*. It synthesizes patches by taking code statements from the program under repair and uses an evolutionary strategy to navigate the search space. This means that, for each generation, a number of *best* patches (assessed by a fitness function) are selected for evolution in the next generation. Cardumen, instead, aims at fixing fine-grained code elements, i.e., it operates at the *expression-level*. It mines the source code of the program under repair to create repair templates and then uses such templates for synthesizing patches. In order to generate concrete patches, the placeholders in the templates are replaced by variables which frequently occur in the modification points. In Cardumen, a selective search strategy is used to explore the search space. Particularly, to speed up the search, Cardumen utilizes a probability model to prioritize candidate patches.

There are several reasons for choosing jGenProg and Cardumen. First, they are popular and representative APR tools for Java programs which are widely used in the related studies (Liu et al., 2020; Durieux et al., 2019; Ye et al., 2021a; Wang et al., 2021; Benton et al., 2022). Second, they target fixing buggy code at different levels, statement and expression levels. Third, they leverage different search strategies, evolutionary and selective. Finally, they are standalone tools which can be executed if provided with the Java program source code and a test suite, without requiring any additional data/components (e.g., ssFix (Xin and Reiss, 2017) needs to connect to a private engine or LSRepair (Liu et al., 2018) requires run-time code search over Github repositories).

Statistical analysis: The performance of the approaches could be affected by the randomness of APR tools, e.g., the random selection of modification operators and of the templates. To mitigate the impact of such randomness, each experiment is executed five times. The statistical analysis is conducted in the experimental results of the setting `withFL` whose search space is large and could be highly impacted by the randomness. Note that the FL phases of these experiments are done only once by VarCop (Nguyen et al., 2022), as its result is deterministic.

Specifically, for each execution of each experiment, we record the metric *#Correct fixes* (Section 6.2.2) as dependent variable. Then, for each APR tool (i.e., jGenProg and Cardumen), we compare the distributions of *#Correct fixes* across the five repetitions, for the two versions of the single-product-based approach ($\texttt{SingleProd}_{\texttt{heu}}$ vs. $\texttt{SingleProd}_{\texttt{basic}}$), and the distributions of *#Correct fixes* for the two versions of the multi-product-based approach ($\texttt{MultiProd}_{\texttt{heu}}$ vs. $\texttt{MultiProd}_{\texttt{basic}}$). We have first checked the distributions for normality using the Shapiro–Wilk test; since six out of the eight distributions that we need to compare are not normal, we select the non-parametric Mann–Whitney U test (McKnight and Najab, 2010) to compare them. The null hypothesis is there is no significant difference; we take $\alpha = 0.05$ as confidence value. If the p-value returned by the Mann–Whitney U test is lower than α, we reject the null hypothesis that there is no significant difference, and we accept the alternative hypothesis that they are different; otherwise, we claim that there is no significant difference. In case of significant difference, we use the Vargha and Delaney's $\hat{A}_{12}$ effect size to assess the strength of the significance: if $\hat{A}_{12}$ of approach A and B is greater than 0.5, the results of A are significantly higher (and so better) than those of the other.

Intrinsic analysis. We studied the impacts of the heuristic rules: *Failing product navigation* ($ProdNav$), *Modification point navigation* ($ModNav$), *Modification suitability measurement* ($ModSuit$) by alternatively disabling one of them in $\texttt{SingleProd}_{\texttt{heu}}$. In addition, we created different variants of $\texttt{SingleProd}_{\texttt{heu}}$ with different similarity functions, suitability parameters, and suitability thresholds (Eq. (2), Section 5.1.2) in measuring the suitability of the modification operations. We conducted these experiments on the different variants of $\texttt{SingleProd}_{\texttt{heu}}$ since this approach obtained the best performance compared to the other approaches (Section 7.1). Moreover, due to the time limitation and the need to repeat each experiment multiple times, we conducted these experiments on the variability bugs of three systems, `BankAccount`, `Email`, and `ExamDB`.

Sensitivity analysis. We studied how $\texttt{SingleProd}_{\texttt{heu}}$ repairs variability bugs in different SPL systems. We also studied the impact of the following factors on the performance of $\texttt{SingleProd}_{\texttt{heu}}$: *the number of failing products* and *the number of suspicious statements* of each

[1] The comparison between single-product-based and multi-product-based approaches is fair, as both can use at most 60 min.

buggy SPL system. In this experiment, we categorize the buggy systems into different groups based on their *number of failing products* (resp. *number of suspicious statements*) and analyze the repair performance of $\texttt{SingleProd}_{\texttt{heu}}$ of each group.

6.2.2. Metrics

For evaluating the effectiveness of the approaches, we adopt *#Plausible fixes* and *#Correct fixes* as in the related studies (Li et al., 2022; Durieux et al., 2019; Yuan and Banzhaf, 2020b; Ye et al., 2022). *#Plausible fixes* represents the number of buggy systems obtaining valid patches that make the systems pass all their tests. Instead, *#Correct fixes* indicates the number of buggy systems obtaining valid patches that are also equivalent to the ground-truth patches provided by the benchmark. Indeed, the number of plausible fixes was widely used to show how effective an APR technique is: the higher the number of plausible fixes, the better the APR tool (Le Goues et al., 2012b,a). However, a plausible patch could still be incorrect due to the inadequacy of the provided test suites (Ye et al., 2021b; Qi et al., 2015; Smith et al., 2015; Jiang et al., 2019). Thus, in recent studies (Li et al., 2022; Ye et al., 2022), the number of correct fixes has become a more popular metric: the higher the number of correct fixes, the better the APR approach.

In addition, *#Mods/system* and *Runtime/system* are used for evaluating the efficiency of the approaches. *#Mods/system* shows the average number of modification operations that a repair approach tries to fix a buggy SPL system. *Runtime/system* reports the average running time taken by a repair approach to fix a buggy SPL system. Indeed, the running time has been widely used to compare the efficiency of the APR tools (Liu and Zhong, 2018; Wen et al., 2018; Ghanbari et al., 2019). However, this metric could be unstable and dependent on many variables that are unrelated to the APR approaches (Liu et al., 2020). Thus, we report both the average number of attempts and the average runtime that an APR approach takes to repair a buggy SPL system to show how efficient the approach is. The less number of modifications and the less runtime, the better APR approach.

7. Experimental results

7.1. RQ1. Performance analysis

7.1.1. Setting `withoutFL`

The performance of the FL technique has a major impact on the effectiveness and efficiency of the APR approaches. The reason is that the APR tools often select the modification points based on the output of the FL technique (i.e., the suspicious statement ranked lists) and this could contain statements that are not actually faulty. In this experiment, to mitigate the impact of the FL phase, we conducted an experiment in which, for each buggy SPL system, the APR approaches are given the correct position of the faulty statements in advance. Table 4 shows the performance of the APR approaches in this setting on the total of 318 variability bugs. Overall, *the performance of the single-product-based approaches is better than that of the multi-product-based approaches, about 3 times in effectiveness and 9 times in efficiency*. In addition, *our heuristic rules help to enhance the corresponding basic approaches from 30 to 150% in the number of correct fixes and from 30 to 70% in the number of modification operations*.

For example, by using jGenProg to generate patches, $\texttt{SingleProd}_{\texttt{basic}}$ can correctly fix 38 buggy systems. Instead, only 12 buggy systems can be correctly fixed by $\texttt{MultiProd}_{\texttt{basic}}$. By using Cardumen, $\texttt{MultiProd}_{\texttt{basic}}$ can correctly fix 9 buggy systems, while $\texttt{SingleProd}_{\texttt{basic}}$ can do that for a significantly larger number of systems, i.e., 40. Moreover, the single-product-based approach is also much more efficient than the multi-product-based approach. Particularly, the *Runtime/system* of $\texttt{SingleProd}_{\texttt{basic}}$ is 5–14 times lower than $\texttt{MultiProd}_{\texttt{basic}}$. These results show that $\texttt{SingleProd}_{\texttt{basic}}$ can correctly fix many more variability bugs in significantly less running time than $\texttt{MultiProd}_{\texttt{basic}}$.

One of the main reasons why the single-product-based approach obtained better results than the multi-product-based approach is that it can attempt more modification operations, which helps increase its possibility of reaching plausible/correct patches. On average, $\texttt{SingleProd}_{\texttt{basic}}$ tried about 73–95 modification operations on a buggy system. Instead, $\texttt{MultiProd}_{\texttt{basic}}$ tried 13–18 modification operations, around five times less than $\texttt{SingleProd}_{\texttt{basic}}$. Indeed, for each modification, the single-product-based approach can quickly evaluate it by the test suite of one product. Instead, the multi-product-based approach needs to validate the modification over all the test suites of all the sampled products. This dynamic validation process is time-consuming. As a result, given the same limitation of execution time, $\texttt{SingleProd}_{\texttt{basic}}$ can attempt many more modifications and obtain more plausible/correct fixes.

Although attempting a larger number of modifications, the single-product-based approach is still more efficient in terms of running time. In particular, to repair an SPL system, $\texttt{SingleProd}_{\texttt{basic}}$ takes about 7 min, while $\texttt{MultiProd}_{\texttt{basic}}$ takes about 59 min. This demonstrates that executing all the test suites of the system to validate the modifications in each repair iteration is really ineffective. To improve the performance of the multi-product-based approach, modifications could be validated by a subset of selected test cases.

Regarding the heuristic rules, the performance of $\texttt{SingleProd}_{\texttt{heu}}$ and $\texttt{MultiProd}_{\texttt{heu}}$ is considerably better than the corresponding basic approaches in both effectiveness and efficiency. For example, with Cardumen, $\texttt{SingleProd}_{\texttt{heu}}$ needs to attempt 20 modification operations for each buggy SPL system to correctly fix 65 bugs. Instead, $\texttt{SingleProd}_{\texttt{basic}}$ must attempt 73 modification operations in each system and correctly fixes only 40 bugs. Moreover, $\texttt{MultiProd}_{\texttt{heu}}$ can correctly fix 24 bugs by trying 10 modifications per system. In contrast, with a larger number of modifications (i.e., 13 modifications per system), $\texttt{MultiProd}_{\texttt{basic}}$ can correctly fix a lower number of bugs, i.e., only 9 bugs.

In this setting, the suspicious statement lists of the SPL systems contains only buggy statements. Thus, the modification points are always correctly selected by all the approaches in *modification point navigation* step. However, for the same selected modification points, the $ModSuit$ heuristics helps $\texttt{SingleProd}_{\texttt{heu}}$ and $\texttt{MultiProd}_{\texttt{heu}}$ to avoid attempting a lot of incorrect modification operations and then quickly find the correct ones. Instead, various incorrect modifications are attempted by $\texttt{SingleProd}_{\texttt{basic}}$ and $\texttt{MultiProd}_{\texttt{basic}}$. Consequently, $\texttt{SingleProd}_{\texttt{heu}}$ and $\texttt{MultiProd}_{\texttt{heu}}$ can plausibly/correctly fix more bugs than $\texttt{SingleProd}_{\texttt{basic}}$ and $\texttt{MultiProd}_{\texttt{basic}}$. In addition, by avoiding attempting incorrect modifications, $\texttt{SingleProd}_{\texttt{heu}}$ and $\texttt{MultiProd}_{\texttt{heu}}$ can also reduce the number of modifications and consume less running time to fix a buggy SPL system compared to the corresponding basic approaches.

7.1.2. Setting `withFL`

Table 5 shows the repairability performance of the proposed APR approaches on the total of 318 variability bugs in the setting `withFL`. In this setting, *the single-product-based approaches also obtain considerably better results than the multi-product-based approaches*. They obtain significantly higher values for *#Correct fixes* with a significantly lower *Runtime/system*. Furthermore, *the heuristic rules of $ModNav$ and $ModSuit$ can help to significantly boost the effectiveness of both single-product-based and multi-product-based directions up to five times*.

By both APR tools jGenProg and Cardumen, the effectiveness of $\texttt{SingleProd}_{\texttt{basic}}$ is higher than that of $\texttt{MultiProd}_{\texttt{basic}}$ from 12 to 30 times. For example, by leveraging jGenProg to generate patches, $\texttt{SingleProd}_{\texttt{basic}}$ can plausibly fix 49 systems. Instead, only 4 systems can be plausibly fixed by $\texttt{MultiProd}_{\texttt{basic}}$. By using Cardumen, $\texttt{MultiProd}_{\texttt{basic}}$ cannot correctly fix any buggy system, while $\texttt{SingleProd}_{\texttt{basic}}$ can do that for 13 systems. In addition, the single-product-based approach is also much more efficient than the multi-product-based approach. Particularly, the *Runtime/system* of $\texttt{SingleProd}_{\texttt{basic}}$ is 2–3 times lower than $\texttt{MultiProd}_{\texttt{basic}}$. Similarly to the discussion for the

Table 4
RQ1 — The performance of repairing variability bugs of the approaches in the setting `withoutFL` (i.e., the correct positions of buggy statements are given).

APR tool	Metrics	Single product-based approach		Multi product-based approach	
		$\texttt{SingleProd}_{\texttt{basic}}$	$\texttt{SingleProd}_{\texttt{heu}}$	$\texttt{MultiProd}_{\texttt{basic}}$	$\texttt{MultiProd}_{\texttt{heu}}$
jGenProg	#Correct fixes	38	40	12	28
	#Plausible fixes	64	49	37	34
	#Mods/system	95	27	18	14
	Runtime/system (min)	8.7	6.7	42.1	22
Cardumen	#Correct fixes	40	65	9	24
	#Plausible fixes	112	92	42	41
	#Mods/system	73	20	13	10
	Runtime/system (min)	5.3	3.9	75.1	52

Table 5
RQ1 — The performance of repairing variability bugs of the approaches in the setting `withFL`.

APR tool	Metrics	Single product-based approach		Multi product-based approach	
		$\texttt{SingleProd}_{\texttt{basic}}$	$\texttt{SingleProd}_{\texttt{heu}}$	$\texttt{MultiProd}_{\texttt{basic}}$	$\texttt{MultiProd}_{\texttt{heu}}$
jGenProg	#Correct fixes	20	37	1	5
	#Plausible fixes	49	49	4	7
	#Mods/system	144	179	20	30
	Runtime/system (min)	19.3	16.2	41.8	44.3
Cardumen	#Correct fixes	13	40	0	1
	#Plausible fixes	92	80	3	4
	#Mods/system	133	122	23	22
	Runtime/system (min)	13.8	15.3	41.3	52.2

setting `withoutFL`, the single-product-based approach locally validates each modification by the tests of one product before globally validating it by the whole test suites of the system. Thus, it can attempt more modification operations in less running time and efficiently obtain a higher number of plausible/correct fixes.

Notably, *by applying the heuristic rules,* $\texttt{SingleProd}_{\texttt{heu}}$ *improves the number of correct fixes of* $\texttt{SingleProd}_{\texttt{basic}}$ *from 85% to 207%, and* $\texttt{MultiProd}_{\texttt{heu}}$ *improves the performance of* $\texttt{MultiProd}_{\texttt{basic}}$ *up to 5 times.* For example, using Cardumen, $\texttt{SingleProd}_{\texttt{heu}}$ can correctly fix 40 systems, and $\texttt{MultiProd}_{\texttt{heu}}$ can correctly fix 1 system; instead, the corresponding numbers of correct fixes of $\texttt{SingleProd}_{\texttt{basic}}$ and $\texttt{MultiProd}_{\texttt{basic}}$ are 13 and 0, respectively. Indeed, the validation results of the previous modification operations are very valuable in guiding the APR approaches to correctly select modification points and avoid attempting unsuitable modifications. For example, with the bug in Fig. 3, after attempting $MO3$ (Table 2), the associated modification point of the buggy statement mp = (`BackOut.ExamDataBaseImpl.26, if(students[++i] != null && !student[i].backedOut)`) is prioritized by both approaches, $\texttt{SingleProd}_{\texttt{heu}}$ and $\texttt{MultiProd}_{\texttt{heu}}$, to be continuously fixed. Also, the previous attempted modification operations at this point, yet incorrect such as $d = ins\_aft(mp,$ `this.students = new main.Student[100]`$)$ or $d = ins\_bef(mp,$ `main.Student[]; oldStudents = students`$)$ provide feedback to help the repair approaches avoid attempting similar incorrect modifications in the next iterations. As a result, the correct modification operation for this buggy statement is sooner attempted by $\texttt{SingleProd}_{\texttt{heu}}$ and $\texttt{MultiProd}_{\texttt{heu}}$.

Moreover, *the efficiency of the enhanced approaches and the corresponding basic approaches is not significantly different, and it greatly depends on the employed APR tools.* For example, using jGenProg, $\texttt{SingleProd}_{\texttt{heu}}$ takes less time to try more modifications to fix a buggy SPL system than $\texttt{SingleProd}_{\texttt{basic}}$. Specifically, for a buggy SPL system, $\texttt{SingleProd}_{\texttt{heu}}$ attempts 179 modifications in 16 min, while $\texttt{SingleProd}_{\texttt{basic}}$ takes 19 min to attempt 144 modifications. In contrast, leveraging Cardumen, compared to $\texttt{SingleProd}_{\texttt{basic}}$, $\texttt{SingleProd}_{\texttt{heu}}$ takes a slightly higher runtime to attempt less number of modifications. $\texttt{SingleProd}_{\texttt{basic}}$ spends about 14 min on 133 modifications to fix a variability bug, while $\texttt{SingleProd}_{\texttt{heu}}$ spends about 15 min to attempt 122 modifications.

In fact, the number of modification operations is not always proportional to the runtime. The reason is that for a system, the modification operations of the approaches could try to modify code of different modification points. In setting `withFL`, the modification points are selected on the list of suspicious statements returned by VarCop, which could contain not only buggy but also non-buggy statements. With the guidance of the heuristics $ModNav$, the modification points selected by the enhanced approaches are different from the corresponding basic ones. The differences in the selected modification points could lead to the number of affected products being different, which leads to the variation in the runtime of the approaches; this is because only the products affected by the modifications need to re-execute the tests. However, by not being significantly different in *#Mods/system* and *Runtime/system*, $\texttt{SingleProd}_{\texttt{heu}}$ and $\texttt{MultiProd}_{\texttt{heu}}$ can correctly fix a much higher number of buggy systems compared to the basic approaches.

7.1.3. Statistical analysis

Overall, *the enhanced variants of both single-product-based and multi-product-based approaches consistently obtain significantly better effectiveness than the corresponding basic variants*. Table 6 shows the statistical analysis of the comparison of $\texttt{SingleProd}_{\texttt{heu}}$ to $\texttt{SingleProd}_{\texttt{basic}}$, and $\texttt{MultiProd}_{\texttt{heu}}$ to $\texttt{MultiProd}_{\texttt{basic}}$, in terms of number of correct fixes. For single-product-based approaches, the number of correct fixes of $\texttt{SingleProd}_{\texttt{heu}}$ and $\texttt{SingleProd}_{\texttt{basic}}$ is significantly different (p-value = 0.01). Also, in all the experiments, $\texttt{SingleProd}_{\texttt{heu}}$ always obtains higher results ($\hat{A}_{12} = 1$). Similarly, the performance of $\texttt{MultiProd}_{\texttt{heu}}$ is considerably better than $\texttt{MultiProd}_{\texttt{basic}}$ if jGenProg is used. However, by using Cardumen, the results of these multi-product-based approaches are not significantly different. The reason is that the results of Cardumen in these experiments are not good, only 1 or none of the systems can be fixed. Thus, it cannot show the differences in the approaches' results.

Answer to RQ1: The single-product-based approach, in both settings `withFL` and `withoutFL`, is more effective and efficient than the multi-product-based approach in repairing variability bugs. The heuristic rules can significantly boost the performance of both single-product-based and multi-product-based approaches.

Table 6
RQ1 – Statistical analysis regarding *#Correct fixes* of `SingleProd`$_{heu}$ vs. `SingleProd`$_{basic}$ and `MultiProd`$_{heu}$ vs. `MultiProd`$_{basic}$ in different experiment executions – `withFL` setting.

	Single product-based approach		Multi product-based approach	
	p-value	$\hat{A}_{12}$	p-value	$\hat{A}_{12}$
jGenProg	0.01	1	0.02	1
Cardumen	0.01	1	0.08	–

Table 7
RQ2 — Impact of disabling each heuristic rule in `SingleProd`$_{heu}$.

	`SingleProd`$_{heu}$	Not applied rule		
	(all rules applied)	$ProdNav$	$ModNav$	$ModSuit$
#Correct fixes	34	32	32	14
#Plausible fixes	54	51	48	63
#Mods/system	148	150	147	158
Runtime/system (min)	2.2	1.8	1.8	2.3

7.2. RQ2. Intrinsic analysis

7.2.1. Impact of the heuristic rules

Table 7 shows how the heuristic rules impact the performance of `SingleProd`$_{heu}$. As expected, `SingleProd`$_{heu}$ *obtains the highest performance when all the heuristic rules are enabled*. Its effectiveness can decrease from 6% to 60% if one of the rules is disabled. Particularly, `SingleProd`$_{heu}$ can correctly fix 34 bugs in three experimental systems, `BankAccount`, `Email`, and `ExamDB`. If $ProdNav$ or $ModNav$ are disabled, the number of correct fixes is 32. Instead, if $ModSuit$ is disabled, only 14 variability bugs can be correctly fixed. Indeed, if $ModSuit$ is disabled, more modifications are attempted and so the number of plausible fixes can significantly increase. For example, `SingleProd`$_{heu}$ obtains 54 plausible fixes, while this number is increased by 17% (63 plausible fixes) if $ModSuit$ is disabled. This result indicates that without lightweight evaluating and eliminating incorrect modifications, there is a high rate of over-fitting or false-positive patches. Indeed, in this variant of `SingleProd`$_{heu}$, multiple incorrect modification operations have been attempted. These modification operations can cause the system to pass all the available tests, but they are incorrect patches.

Moreover, the running time of `SingleProd`$_{heu}$ does not significantly vary when disabling the heuristic rules. For example, `SingleProd`$_{heu}$ takes about 2.2 min to repair a buggy SPL system. If one of the heuristic rules is disabled, the fixing process of this approach consumes from 1.8 to 2.3 min. Indeed, enabling and disabling the heuristic rules affect the selected modification points in the repair process. This leads to the numbers of affected products of each system being different. Thus, the runtime slightly fluctuates in these experiments due to the differences in the number of the affected and tested products.

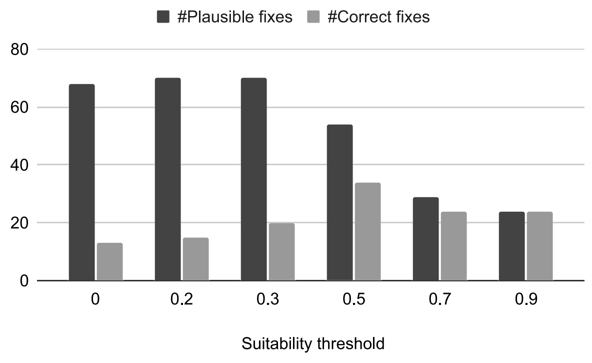
7.2.2. Impact of the similarity functions in $ModSuit$

We build different variants of `SingleProd`$_{heu}$ with different similarity functions for measuring the suitability of the modification operations in $ModSuit$ (Eqs. (3) and (4), Section 5.1.2). As seen in Table 8, `SingleProd`$_{heu}$ *obtains the highest number of correct fixes when the Jaccard is used as the similarity function (36)*. Instead, *when Cosine or N-gram is leveraged,* `SingleProd`$_{heu}$*'s effectiveness is the lowest (28 and 31), although the number of modification operations is the highest*.

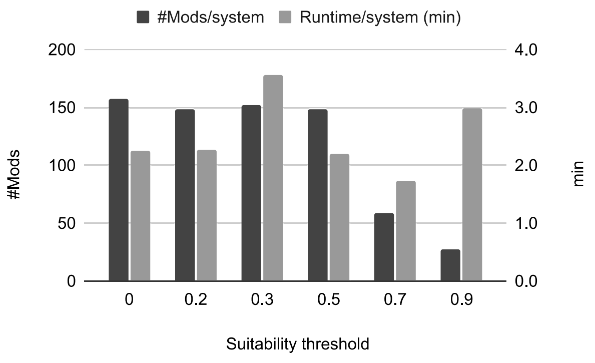
There are two reasons for these results of the different similarity functions. First, multiple bugs can be fixed by minor changes (Kim et al., 2023). That is the reason why similarity functions like Levenshtein or Jaccard, which reflect the edit distances of the sentences, could be helpful. Second, the order of the tokens in the modified code of the modifications could not be an important factor in deciding the similarity in the program repair topic. For example, the correct modification and the buggy source code could share several variables, operators, etc., but these tokens are in different orders in the code statements. Metrics such as Cosine or N-gram, which consider the order of tokens in measuring the similarity, could not work well in this case.

Table 8
RQ2 — Impact of the similarity functions in modification suitability measurement.

Metrics	Cosine	N-gram	Longest common subsequence	Levenshtein	Jaccard
#Correct fixes	28	31	32	34	36
#Plausible fixes	52	47	58	54	56
#Mods/system	154	154	153	148	145
Runtime/system (min)	2.2	2.3	2.0	2.1	2.3



(a) Effectiveness (*#Plausible fixes*, *#Correct fixes*)



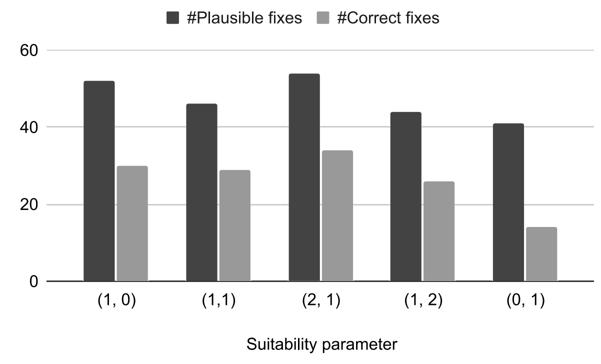
(b) Efficiency (*#Mods/system*, *Runtime/system*)

Fig. 5. RQ2 — Impact of the suitability threshold θ on `SingleProd`$_{heu}$'s performance.

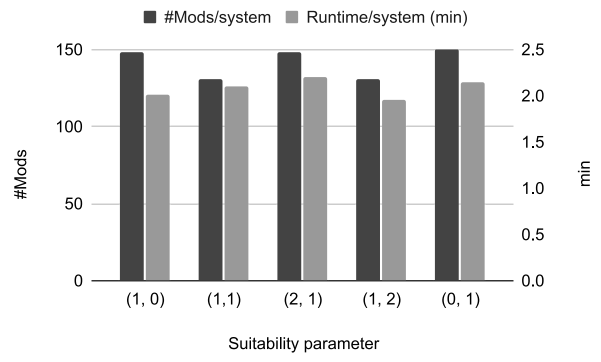
7.2.3. Impact of the suitability threshold θ in $ModSuit$

Fig. 5 shows the repair performance of `SingleProd`$_{heu}$ with different values of threshold θ in deciding whether a modification is suitable for the application into the program (rule $ModSuit$). In general, *the number of plausible fixes decreases when the threshold increases* (see Fig. 5(a)). Specifically, if θ is equal to 0, the number of plausible fixes is 68, while if θ is 0.9, this number is 24. This is because the higher the threshold, the stricter $ModSuit$ is to evaluate the suitability of the modifications. This leads to multiple modifications being eliminated without being applied to the system and validated against the tests. Indeed, as shown in Fig. 5(b), increasing the threshold leads to a considerable decrease in the number of modification operations, from 157 modifications when θ is 0, to 27 modifications when θ is 0.9. Hence, the number of plausible fixes decreased.

The previous observations can be explained as follows. `SingleProd`$_{heu}$ *obtains the highest number of correct fixes with* $\theta = 0.5$ (see Fig. 5(a)). For example, if the threshold increases from 0 to 0.5, the number of correct fixes increases from 13 to 34. However, if the threshold continuously increases from 0.5 to 0.9, the number of correct fixes decreases from 34 to 24. Indeed, with the small value of the threshold $\theta = 0$, multiple incorrect modification operations could be attempted, and then APR tools stop searching when a plausible fix of the system is reached. However, that fix could be still incorrect. Thus,

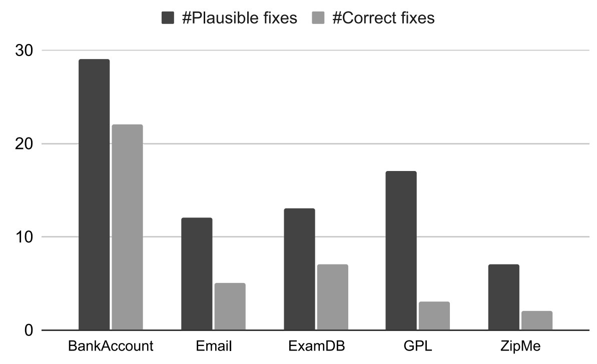


(a) Effectiveness (*#Plausible fixes*, *#Correct fixes*)

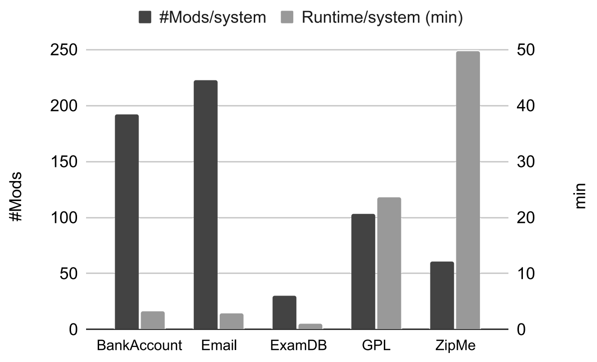


(b) Efficiency (*#Mods/system*, *Runtime/system*)

Fig. 6. RQ2 — Impact of the suitability parameters (α, β) on SingleProd$_{\texttt{heu}}$'s performance.



(a) Effectiveness (*#Plausible fixes*, *#Correct fixes*)



(b) Efficiency (*#Mods/system*, *Runtime/system*)

Fig. 7. RQ3 — The performance of SingleProd$_{\texttt{heu}}$ in fixing variability bugs of different SPL systems.

although the number of plausible fixes is high, multiple of those fixes are over-fitting and incorrect. On the other hand, with the large threshold value $\theta = 0.9$, the approach could eliminate multiple promising modification operations. This leads to a decrease in the numbers of both plausible and correct fixes. *Overall, to ensure the best performance of* SingleProd$_{\texttt{heu}}$*, the threshold θ should be 0.5.*

7.2.4. Impact of hyperparameters α and β in $ModSuit$

In order to analyze how the suitability parameters impact SingleProd$_{\texttt{heu}}$'s performance, we conducted experiments with five different values of the pair (α, β) in Eq. (2), as shown in Fig. 6. Specifically, for a modified code c_m, if the pair (α, β) is $(1, 0)$ or $(0, 1)$, it means that we use the value of only $suit(c_m)$ or $unsuit(c_m)$ to evaluate its suitability. Instead, when both α and β are greater than 0, it means that we consider both $suit(c_m)$ and $unsuit(c_m)$ in the evaluation, with different importance; using $(2, 1)$ gives more importance to $suit(c_m)$, while using $(1, 2)$ gives more importance to $unsuit(c_m)$. Finally, when we use $(1, 1)$, it means that we give the same importance to the two aspects.

These different parameters do not significantly affect SingleProd$_{\texttt{heu}}$*'s efficiency, as the numbers of modification operations and running time just vary slightly*, as shown in Fig. 6(b). However, SingleProd$_{\texttt{heu}}$'s *effectiveness is considerably affected*; indeed, as seen in Fig. 6(a), SingleProd$_{\texttt{heu}}$ obtains the highest *#Correct fixes* when both $suit(c_m)$ and $unsuit(c_m)$ are considered and $suit(c_m)$ has a higher priority ($\alpha = 2$, $\beta = 1$). In addition, the repairing result of SingleProd$_{\texttt{heu}}$ is the lowest when $\alpha = 0$ and $\beta = 1$. This variant of SingleProd$_{\texttt{heu}}$ can correctly fix 14 buggy systems, half of those fixed when both $suit(c_m)$ and $unsuit(c_m)$ are considered. Indeed, when $\alpha = 0$, the suitability of a new modification is evaluated by only considering the feedback of the previous failed attempts without considering the original code of the corresponding modification point. The previous attempts could guide SingleProd$_{\texttt{heu}}$ to decide incorrect modifications, i.e., the modifications similar to the failed modifications can be eliminated. However, by only previous failed attempts, there is little or even no evidence about correct modifications, which is reflected by the similarity with the original code (Csuvik et al., 2020; Kim et al., 2023).

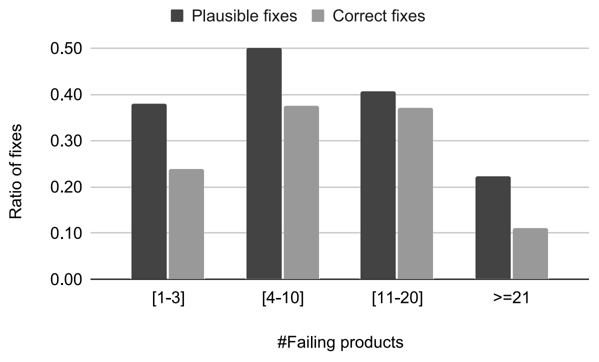
Answer to RQ2: Each heuristic rule has a different impact on the performance of SingleProd$_{\texttt{heu}}$. Moreover, the effectiveness of SingleProd$_{\texttt{heu}}$ is significantly affected by the used similarity function, parameters, and threshold in $ModSuit$.

7.3. RQ3. Sensitivity analysis

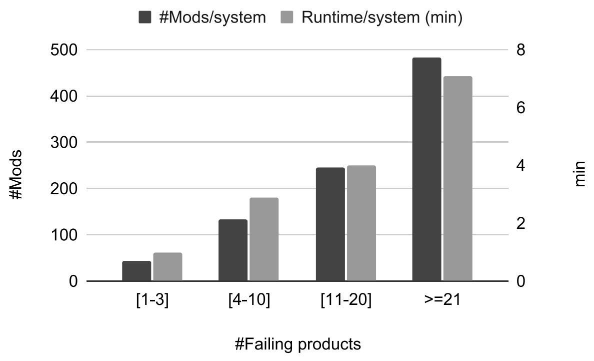
7.3.1. Impact of the characteristics of the different SPL systems

Overall, *the repair performance of* SingleProd$_{\texttt{heu}}$ *varies by varying the characteristics of the SPL system under repair.* As shown in Fig. 7, `BankAccount` is the system obtaining the highest number of correct patches. The reason is that `BankAccount` is a small system with multiple similar functions. This could be helpful for redundancy-based APR tools such as jGenProg or Cardumen to find patches in the product under repair itself. Instead, variability bugs in the `ExamDB` system are fixed by the least number of modification operations. The reason is that the sampled set of this system is small (i.e., 8 products) and only about one or two failing products for each buggy version of this system. Thus, by the single-product-based approach, the number of failing products of `ExamDB` system that have been attempted to be fixed is much less than the other systems.

Moreover, *the running time of the APR approach depends on not only the number of attempted modification operations but also the size of the system and the test suites*. For example, although each bug in `BankAccount` is attempted by 185 modification operations, which is

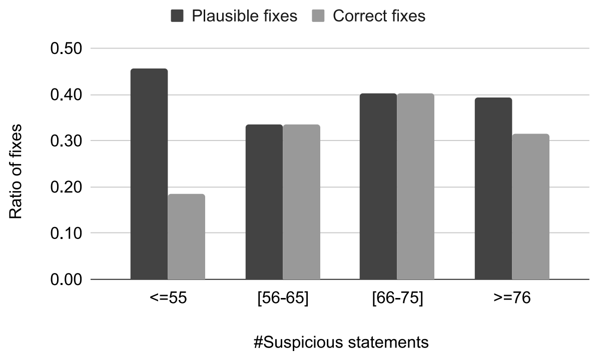


(a) Effectiveness (*#Plausible fixes*, *#Correct fixes*)

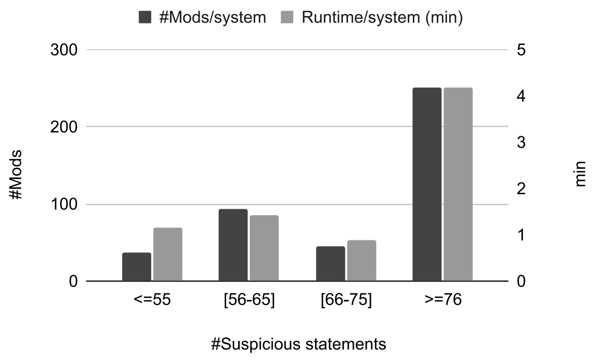


(b) Efficiency (*#Mods/system*, *Runtime/system*)

Fig. 8. RQ3 – Impact of the number of failing products on `SingleProd`$_{\texttt{heu}}$'s performance – `BankAccount`.



(a) Effectiveness (*#Plausible fixes*, *#Correct fixes*)



(b) Efficiency (*#Mods/system*, *Runtime/system*)

Fig. 9. RQ3 – Impact of the number of suspicious statements on `SingleProd`$_{\texttt{heu}}$'s performance – `BankAccount`.

three times larger than the number of modifications attempted to fix a bug in `ZipMe`, the fixing time of a bug in `ZipMe` is 15 times larger than that of `BankAccount`. The reason is that `ZipMe` contains 3460 statements which is much larger than `BankAccount`. Also, the test suite of each sampled product of `ZipMe` is 13 times larger than that of a product of `BankAccount`. As a result, although less modifications are attempted for it, `ZipMe` still costs more execution time.

7.3.2. Impact of the number of failing products

Fig. 8 shows the impact of the number of failing products on the effectiveness and efficiency of `SingleProd`$_{\texttt{heu}}$. The performance of `SingleProd`$_{\texttt{heu}}$ could be impacted by various factors such as the nature of code, the size of systems, etc. Thus, to focus on the impact of the number of failing products, in this experiment, we analyze the performance of `SingleProd`$_{\texttt{heu}}$ on the buggy versions of one system, `BankAccount`. In this experiment, we separate buggy versions of `BankAccount` system into different *groups* according to their numbers of failing products (as reported on x-axis of the plots in Fig. 8). Moreover, as the numbers of buggy SPL systems in different groups are different, for ease of comparison, we report the ratio of fixes (Fig. 8(a)), the average number of modification operations, and the average running time (Fig. 8(b)) of the bugs in each group. Overall, `SingleProd`$_{\texttt{heu}}$ *obtains the best effectiveness when the bugs cause failures for about 4–20 failing products (12%–60% total sampled products of the system)* (see Fig. 8(a)). *If the number of failing products increases, the number of modification operations and the running time will also increase* (see Fig. 8(b)).

Indeed, when multiple sampled products are affected by the bugs (i.e., the number of failing products is large), the APR tool may need to try fixing multiple of them. This increases the attempted modification operations and running time. Intuitively, attempting to repair more products and trying with more modification operations increases the probability of reaching plausible/correct fixes. For the systems containing only one or three failing products, 24% of the bugs are correctly fixed. If the bugs affect about 11–20 products of the system, 37% of the bugs are correctly fixed. However, if there are too many products affected by the bugs, such bugs could be difficult to be fixed. Specifically, if the bugs cause failures for more than 21 products of the system, only 11% of the bugs are correctly fixed.

7.3.3. Impact of the number of suspicious statements

Similarly to the previous experiment in Section 7.3.2, to focus on the impact of the number of suspicious statements, we analyze the repair performance of bugs in one system, `BankAccount`, and report the ratio of fixes. Fig. 9 shows how `SingleProd`$_{\texttt{heu}}$'s performance is impacted by the size of suspicious statements. Overall, *the number of modification operations and running time increases proportionally with the increase of the number of suspicious statements* (see Fig. 9(b)). Specifically, if there are less than 55 suspicious statements, 36 modification operations are attempted to fix a bug. Instead, if the number of suspicious statements increases 1.5 times, i.e., more than 76 suspicious statements, about 251 modifications are attempted to repair a bug.

The ratio of plausible fixes decreases, yet the ratio of correct fixes increases when the size of suspicious space increases (see Fig. 9(a)). For systems with less than 55 suspicious statements, 45% of the bugs are plausibly fixed, and 18% of them are correctly fixed. Instead, if the SPL system has more than 76 suspicious statements, the values of these metrics are 39% and 31%, respectively. This result shows that with a small number of suspicious statements, the APR tool has a high possibility of selecting the buggy statements correctly. Then, it can quickly find a plausible patch and stop searching (high ratio of plausible fixes, yet low ratio of correct fixes). However, with a larger set of suspicious statements, by multiple attempts, $\texttt{SingleProd}_{\texttt{heu}}$ has more evidence from previous attempts to find correct modification points and suitable modifications. This could help to increase the ratio of correct fixes. Note that the search process could be much more complicated (i.e., a large amount of modification operations have been attempted) when the search space is too large. This can also negatively impact the repair performance; indeed, the ratio of correct fixes declines from 40% to 31% when the suspicious statements set increases from 66–75 statements to more than 76 statements.

Answer to RQ3: The effectiveness and the efficiency of $\texttt{SingleProd}_{\texttt{heu}}$ are impacted by various factors such as system size, the number of sampled products, the number of failing products, the number of suspicious statements, etc. $\texttt{SingleProd}_{\texttt{heu}}$'s performance decreases when the complexities of these factors increase, such as large system, high number of failing products, and large suspicious statement sets.

8. Threats to validity

The main threats to the validity of our approaches are: internal, construct, conclusion, and external validity threats (Wohlin et al., 2012).

Threats to internal validity mainly lie in the correctness of the implementation of our approaches. To reduce this threat, we implemented our approaches on top of a popular APR framework, Astor (Martinez and Monperrus, 2019). Also, we carefully reviewed our code and made it public (Nguyen et al., 2024) so that other researchers can double-check and reproduce our experiments.

Threats to construct validity mainly lie in the rationality of the assessment metrics. To reduce this threat, we chose the popular metrics that have been widely used in previous studies (Li et al., 2022; Durieux et al., 2019; Yuan and Banzhaf, 2020b; Ye et al., 2022; Liu and Zhong, 2018; Wen et al., 2018; Ghanbari et al., 2019; Liu et al., 2020) for evaluating both effectiveness and efficiency of the approaches.

Threats to conclusion validity are related to the ability to draw definitive conclusions. Since the repair approaches are affected by randomness, this must be taken into consideration in the assessment. For this reason, we repeated each experiment five times. Arcuri and Briand (2011), in their guideline on running experiments with randomized algorithms, suggest to execute 30 repetitions of the experiments. However, they also recognize that this is not feasible for expensive problems, as the one considered in this work; in this case, they advise to take particular care in doing statistical analysis, by accounting for both statistical significance and effect size, as we have done in our analysis (see Section 6.2.1).

Threats to external validity mainly lie in the benchmark used in our experiments. There is only one dataset of artificial bugs is used in the experiments. Thus, the obtained results on artificial faults cannot be generalized for large-scale SPL systems containing real faults. To mitigate the threat, we chose five widely-used systems in existing studies (Apel et al., 2013c; Meinicke et al., 2016; Apel et al., 2013b), which target different application domains. In future work, we are planning to collect more real-world variability bugs in larger SPL systems to evaluate our approaches to address these threats. As another external threat, all systems in the benchmark are developed in Java. Therefore, we cannot claim that similar results would have been observed in other programming languages. This is a common threat of several studies on configurable software systems (Wong et al., 2018; Souto et al., 2017). Another threat lies in our selected APR tools and experiments in only single-bug systems. To reduce the threat, we chose two representative APR tools targeting at different levels, and they are widely used in the existing studies (Liu et al., 2020; Durieux et al., 2019; Ye et al., 2021a; Wang et al., 2021; Benton et al., 2022). In the future, we also plan to conduct experiments with more APR tools.

9. Conclusion

Although SPL systems are popular and their bugs could cause severe damage in multiple products, automatically addressing the bugs in these systems has not been investigated thoroughly. In this paper, we introduce two approaches, single-product-based and multi-product-based, for fixing variability bugs in SPL systems. For the single-product-based method, each failing product is fixed individually, and the obtained patches are propagated and validated on the other products of the system. For multi-product-based, the whole SPL system is considered for repair at the same time, i.e., the APR tool is employed to repair the whole system in each iteration. In addition, to improve the performance of APR tools in *navigating modification points* and *selecting suitable modifications*, we propose several heuristic rules leveraging intermediate validation results of the repaired programs. Our experimental results on a dataset of 318 variability bugs of 5 popular SPL systems show that the single-product-based adaptation method is better than the multi-product-based adaptation method about 20 times in the number of correct fixes. Notably, our heuristic rules could improve the performance of both adaption methods by 30%–150% in the number of correct fixes and 30%–50% in the number of attempted modification operations.

As future work, we plan to extend the approach in different directions. First, in order to further improve the effectiveness of the approach, we could adopt other techniques to localize faults, like deep learning-based (Yang et al., 2024), or code smell-based (Takahashi et al., 2021, 2018). Moreover, as explained in Section 4.2.1, the multi-product-based approach is computationally expensive, as it requires execute the test suites of each product; in Section 4.2.1, we discuss that a possible computational improvement would be to avoid running some test suites once a product fails its test suite. However, this would require to identify new fitness functions that are able to handle the lack of information; we leave this as future work.

CRediT authorship contribution statement

Thu-Trang Nguyen: Writing – review & editing, Writing – original draft, Validation, Software, Methodology, Formal analysis, Conceptualization. **Xiao-Yi Zhang:** Writing – review & editing, Writing – original draft, Validation, Methodology, Conceptualization. **Paolo Arcaini:** Writing – review & editing, Writing – original draft, Validation, Project administration, Methodology, Formal analysis, Conceptualization. **Fuyuki Ishikawa:** Writing – review & editing, Supervision, Project administration, Methodology, Formal analysis, Conceptualization. **Hieu Dinh Vo:** Writing – review & editing, Supervision, Methodology, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

Acknowledgments

P. Arcaini and F. Ishikawa are supported by Engineerable AI Techniques for Practical Applications of High-Quality Machine Learning-based Systems Project (Grant Number JPMJMI20B8), JST-Mirai. X. Zhang is supported by Youth Fund of the National Natural Science Foundation of China (No. 62302035). This research is supported by Vietnam National Foundation for Science and Technology Development (NAFOSTED) under grant number 102.03-2023.14.

References

Abal, I., Brabrand, C., Wasowski, A., 2014. 42 Variability bugs in the Linux kernel: A qualitative analysis. In: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering. ASE '14, Association for Computing Machinery, New York, NY, USA, pp. 421–432. http://dx.doi.org/10.1145/2642937.2642990.

Abal, I., Melo, J., Stănciulescu, Ş., Brabrand, C., Ribeiro, M., Wąsowski, A., 2018. Variability bugs in highly configurable systems: A qualitative analysis. ACM Trans. Softw. Eng. Methodol. (TOSEM) 26 (3), 1–34.

Abreu, R., Zoeteweij, P., Van Gemund, A.J., 2007. On the accuracy of spectrum-based fault localization. In: Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION. TAICPART-MUTATION 2007, IEEE, pp. 89–98. http://dx.doi.org/10.1109/TAIC.PART.2007.13.

Acher, M., Martin, H., Lesoil, L., Blouin, A., Jézéquel, J.-M., Khelladi, D.E., Barais, O., Pereira, J.A., 2022. Feature subset selection for learning huge configuration spaces: The case of Linux kernel size. In: Proceedings of the 26th ACM International Systems and Software Product Line Conference - Volume A. SPLC '22, Association for Computing Machinery, New York, NY, USA, pp. 85–96. http://dx.doi.org/10.1145/3546932.3546997.

Apel, S., Batory, D., Kästner, C., Saake, G., 2013a. Software product lines. In: Feature-Oriented Software Product Lines: Concepts and Implementation. pp. 3–15.

Apel, S., Batory, D., Kästner, C., Saake, G., 2016. Feature-Oriented Software Product Lines. Springer.

Apel, S., Kästner, C., Lengauer, C., 2013b. Language-independent and automated software composition: The FeatureHouse experience. IEEE Trans. Softw. Eng. 39 (1), 63–79. http://dx.doi.org/10.1109/TSE.2011.120.

Apel, S., Rhein, A.v., Wendler, P., Größlinger, A., Beyer, D., 2013c. Strategies for product-line verification: Case studies and experiments. In: Proceedings of the 2013 International Conference on Software Engineering. ICSE '13, IEEE Press, pp. 482–491.

Arcaini, P., Gargantini, A., Vavassori, P., 2016. Automatic detection and removal of conformance faults in feature models. In: 2016 IEEE International Conference on Software Testing, Verification and Validation. ICST, pp. 102–112. http://dx.doi.org/10.1109/ICST.2016.10.

Arcaini, P., Gargantini, A., Vavassori, P., 2017. Automated repairing of variability models. In: Proceedings of the 21st International Systems and Software Product Line Conference - Volume A. SPLC '17, Association for Computing Machinery, New York, NY, USA, pp. 9–18. http://dx.doi.org/10.1145/3106195.3106206.

Arcuri, A., Briand, L., 2011. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In: Proceedings of the 33rd International Conference on Software Engineering. ICSE '11, Association for Computing Machinery, New York, NY, USA, pp. 1–10. http://dx.doi.org/10.1145/1985793.1985795.

Arrieta, A., Segura, S., Markiegi, U., Sagardui, G., Etxeberria, L., 2018. Spectrum-based fault localization in software product lines. Inf. Softw. Technol. 100, 18–31. http://dx.doi.org/10.1016/j.infsof.2018.03.008.

Benton, S., Li, X., Lou, Y., Zhang, L., 2022. Evaluating and improving unified debugging. IEEE Trans. Softw. Eng. 48 (11), 4692–4716. http://dx.doi.org/10.1109/TSE.2021.3125203.

Bodden, E., Tolêdo, T., Ribeiro, M., Brabrand, C., Borba, P., Mezini, M., 2013. SPLLIFT: Statically analyzing software product lines in minutes instead of years. SIGPLAN Not. 48 (6), 355–364. http://dx.doi.org/10.1145/2499370.2491976.

Csuvik, V., Horváth, D., Horváth, F., Vidács, L., 2020. Utilizing source code embeddings to identify correct patches. In: 2020 IEEE 2nd International Workshop on Intelligent Bug Fixing. IBF, pp. 18–25. http://dx.doi.org/10.1109/IBF50092.2020.9034714.

Durieux, T., Madeiral, F., Martinez, M., Abreu, R., 2019. Empirical review of Java program repair tools: A large-scale experiment on 2,141 bugs and 23,551 repair attempts. In: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. In: ESEC/FSE 2019, Association for Computing Machinery, New York, NY, USA, pp. 302–313. http://dx.doi.org/10.1145/3338906.3338911.

Echeverría, J., Pérez, F., Abellanas, A., Panach, J.I., Cetina, C., Pastor, Ó., 2016. Evaluating bug-fixing in software product lines: An industrial case study. In: Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement. ESEM '16, Association for Computing Machinery, New York, NY, USA, http://dx.doi.org/10.1145/2961111.2962635.

Fan, Z., Gao, X., Mirchev, M., Roychoudhury, A., Tan, S.H., 2023. Automated repair of programs from large language models. In: Proceedings of the 45th International Conference on Software Engineering. ICSE '23, IEEE Press, pp. 1469–1481. http://dx.doi.org/10.1109/ICSE48619.2023.00128.

Gazzola, L., Micucci, D., Mariani, L., 2019. Automatic software repair: A survey. IEEE Trans. Softw. Eng. 45 (1), 34–67. http://dx.doi.org/10.1109/TSE.2017.2755013.

Ghanbari, A., Benton, S., Zhang, L., 2019. Practical program repair via bytecode mutation. In: Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis. In: ISSTA 2019, Association for Computing Machinery, New York, NY, USA, pp. 19–30. http://dx.doi.org/10.1145/3293882.3330559.

Goues, C.L., Pradel, M., Roychoudhury, A., 2019. Automated program repair. Commun. ACM 62 (12), 56–65. http://dx.doi.org/10.1145/3318162.

Gupta, R., Pal, S., Kanade, A., Shevade, S., 2017. DeepFix: Fixing common c language errors by deep learning. In: Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence. AAAI '17, AAAI Press, pp. 1345–1351.

Hamill, M., Goseva-Popstojanova, K., 2017. Analyzing and predicting effort associated with finding and fixing software faults. Inf. Softw. Technol. 87, 1–18. http://dx.doi.org/10.1016/j.infsof.2017.01.002.

Henard, C., Papadakis, M., Perrouin, G., Klein, J., Le Traon, Y., 2013. Towards automated testing and fixing of re-engineered feature models. In: Proceedings of the 2013 International Conference on Software Engineering. ICSE '13, IEEE Press, pp. 1245–1248.

Jiang, N., Lutellier, T., Tan, L., 2021. CURE: Code-aware neural machine translation for automatic program repair. In: Proceedings of the 43rd International Conference on Software Engineering. ICSE '21, IEEE Press, pp. 1161–1173. http://dx.doi.org/10.1109/ICSE43902.2021.00107.

Jiang, J., Xiong, Y., Xia, X., 2019. A manual inspection of Defects4J bugs and its implications for automatic program repair. Sci. China Inf. Sci. 62 (10), 200102:1–200102:16. http://dx.doi.org/10.1007/S11432-018-1465-6.

Jiang, J., Xiong, Y., Zhang, H., Gao, Q., Chen, X., 2018. Shaping program repair space with existing patches and similar code. In: Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis. In: ISSTA 2018, Association for Computing Machinery, New York, NY, USA, pp. 298–309. http://dx.doi.org/10.1145/3213846.3213871.

Jin, M., Shahriar, S., Tufano, M., Shi, X., Lu, S., Sundaresan, N., Svyatkovskiy, A., 2023. InferFix: End-to-end program repair with LLMs. In: Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. In: ESEC/FSE 2023, Association for Computing Machinery, New York, NY, USA, pp. 1646–1656. http://dx.doi.org/10.1145/3611643.3613892.

Joshi, H., Sanchez, J.C., Gulwani, S., Le, V., Radiček, I., Verbruggen, G., 2023. Repair is nearly generation: multilingual program repair with LLMs. In: Proceedings of the Thirty-Seventh AAAI Conference on Artificial Intelligence and Thirty-Fifth Conference on Innovative Applications of Artificial Intelligence and Thirteenth Symposium on Educational Advances in Artificial Intelligence. In: AAAI'23/IAAI'23/EAAI'23, AAAI Press, http://dx.doi.org/10.1609/aaai.v37i4.25642.

Kästner, C., Apel, S., Thüm, T., Saake, G., 2012. Type checking annotation-based product lines. ACM Trans. Softw. Eng. Methodol. 21 (3), http://dx.doi.org/10.1145/2211616.2211617.

Kim, J., Park, J., Yoo, S., 2023. The inversive relationship between bugs and patches: An empirical study. In: 2023 IEEE International Conference on Software Testing, Verification and Validation Workshops. ICSTW, IEEE, pp. 314–323.

Le Goues, C., Dewey-Vogt, M., Forrest, S., Weimer, W., 2012a. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In: Proceedings of the 34th International Conference on Software Engineering. ICSE '12, IEEE Press, pp. 3–13.

Le Goues, C., Nguyen, T., Forrest, S., Weimer, W., 2012b. GenProg: A generic method for automatic software repair. IEEE Trans. Softw. Eng. 38 (1), 54–72. http://dx.doi.org/10.1109/TSE.2011.104.

Li, Y., Wang, S., Nguyen, T.N., 2022. DEAR: A novel deep learning-based approach for automated program repair. In: Proceedings of the 44th International Conference on Software Engineering. ICSE '22, Association for Computing Machinery, New York, NY, USA, pp. 511–523. http://dx.doi.org/10.1145/3510003.3510177.

Liu, K., Koyuncu, A., Kim, K., Kim, D., F. Bissyandé, T., 2018. LSRepair: Live search of fix ingredients for automated program repair. In: 2018 25th Asia-Pacific Software Engineering Conference. APSEC, pp. 658–662. http://dx.doi.org/10.1109/APSEC.2018.00085.

Liu, K., Wang, S., Koyuncu, A., Kim, K., Bissyandé, T.F., Kim, D., Wu, P., Klein, J., Mao, X., Traon, Y.L., 2020. On the efficiency of test suite based program repair: A systematic assessment of 16 automated repair systems for Java programs. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering. ICSE '20, Association for Computing Machinery, New York, NY, USA, pp. 615–627. http://dx.doi.org/10.1145/3377811.3380338.

Liu, X., Zhong, H., 2018. Mining stackoverflow for program repair. In: 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering. SANER, pp. 118–129. http://dx.doi.org/10.1109/SANER.2018.8330202.

Lou, Y., Ghanbari, A., Li, X., Zhang, L., Zhang, H., Hao, D., Zhang, L., 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. In: ISSTA 2020, Association for Computing Machinery, New York, NY, USA, pp. 75–87. http://dx.doi.org/10.1145/3395363.3397351.

Lutellier, T., Pham, H.V., Pang, L., Li, Y., Wei, M., Tan, L., 2020. CoCoNuT: Combining context-aware neural translation models using ensemble for program repair. In: Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis. In: ISSTA 2020, Association for Computing Machinery, New York, NY, USA, pp. 101–114. http://dx.doi.org/10.1145/3395363.3397369.

Machado, I.D.C., Mcgregor, J.D., Cavalcanti, Y.C., De Almeida, E.S., 2014. On strategies for testing software product lines: A systematic literature review. Inf. Softw. Technol. 56 (10), 1183–1199. http://dx.doi.org/10.1016/j.infsof.2014.04.002.

Martinez, M., Monperrus, M., 2016. ASTOR: A program repair library for Java (Demo). In: Proceedings of the 25th International Symposium on Software Testing and Analysis. In: ISSTA 2016, Association for Computing Machinery, New York, NY, USA, pp. 441–444. http://dx.doi.org/10.1145/2931037.2948705.

Martinez, M., Monperrus, M., 2018. Ultra-large repair search space with automatically mined templates: The Cardumen mode of Astor. In: Colanzi, T.E., McMinn, P. (Eds.), Search-Based Software Engineering. Springer International Publishing, Cham, pp. 65–86.

Martinez, M., Monperrus, M., 2019. Astor: Exploring the design space of generate-and-validate program repair beyond GenProg. J. Syst. Softw. 151 (C), 65–80. http://dx.doi.org/10.1016/j.jss.2019.01.069.

McKnight, P.E., Najab, J., 2010. Mann–Whitney U test. Corsini Encycl. Psychol. 1.

Meinicke, J., Thm, T., Schrter, R., Benduhn, F., Leich, T., Saake, G., 2017. Mastering Software Variability with FeatureIDE, first ed. Springer Publishing Company, Incorporated.

Meinicke, J., Wong, C.-P., Kästner, C., Thüm, T., Saake, G., 2016. On essential configuration complexity: Measuring interactions in highly-configurable systems. In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering. ASE '16, Association for Computing Machinery, New York, NY, USA, pp. 483–494. http://dx.doi.org/10.1145/2970276.2970322.

Monperrus, M., 2014. A critical review of "automatic patch generation learned from human-written patches": Essay on the problem statement and the evaluation of automatic software repair. In: Proceedings of the 36th International Conference on Software Engineering. In: ICSE 2014, Association for Computing Machinery, New York, NY, USA, pp. 234–242. http://dx.doi.org/10.1145/2568225.2568324.

Moon, S., Kim, Y., Kim, M., Yoo, S., 2014. Ask the mutants: Mutating faulty programs for fault localization. In: 2014 IEEE Seventh International Conference on Software Testing, Verification and Validation. pp. 153–162. http://dx.doi.org/10.1109/ICST.2014.28.

Mordahl, A., Oh, J., Koc, U., Wei, S., Gazzillo, P., 2019. An empirical study of real-world variability bugs detected by variability-oblivious tools. In: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. In: ESEC/FSE 2019, Association for Computing Machinery, New York, NY, USA, pp. 50–61. http://dx.doi.org/10.1145/3338906.3338967.

Ngo, K.-T., Nguyen, T.-T., Nguyen, S., Vo, H.D., 2021. Variability fault localization: A benchmark. In: Proceedings of the 25th ACM International Systems and Software Product Line Conference - Volume A. SPLC '21, Association for Computing Machinery, New York, NY, USA, pp. 120–125. http://dx.doi.org/10.1145/3461001.3473058.

Nguyen, T.-T., Ngo, K.-T., Nguyen, S., Vo, H.D., 2022. A variability fault localization approach for software product lines. IEEE Trans. Softw. Eng. 48 (10), 4100–4118. http://dx.doi.org/10.1109/TSE.2021.3113859.

Nguyen, T.-T., Ngo, K.-T., Nguyen, S., Vo, H.D., 2023. Detecting false-passing products and mitigating their impact on variability fault localization in software product lines. Inf. Softw. Technol. 153 (C), http://dx.doi.org/10.1016/j.infsof.2022.107080.

Nguyen, S., Nguyen, H., Tran, N., Tran, H., Nguyen, T.N., 2020. Feature-interaction aware configuration prioritization for configurable code. In: Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering. ASE '19, IEEE Press, pp. 489–501. http://dx.doi.org/10.1109/ASE.2019.00053.

Nguyen, H.D.T., Qi, D., Roychoudhury, A., Chandra, S., 2013. SemFix: Program repair via semantic analysis. In: Proceedings of the 2013 International Conference on Software Engineering. ICSE '13, IEEE Press, pp. 772–781.

Nguyen, T.-T., Vo, H.D., 2022. Detecting coincidental correctness and mitigating its impacts on localizing variability faults. In: 2022 14th International Conference on Knowledge and Systems Engineering. KSE, pp. 1–6. http://dx.doi.org/10.1109/KSE56063.2022.9953777.

Nguyen, T.-T., Zhang, X.-Y., Arcaini, P., Ishikawa, F., Vo, D.H., 2024. Automated program repair for variability bugs in software product line systems. http://dx.doi.org/10.5281/zenodo.11350970.

Pereira, J.A., Acher, M., Martin, H., Jézéquel, J.-M., Botterweck, G., Ventresque, A., 2021. Learning software configuration spaces: A systematic literature review. J. Syst. Softw. 182 (C), http://dx.doi.org/10.1016/j.jss.2021.111044.

Prehofer, C., 2001. Feature-oriented programming: A new way of object composition. Concurr. Comput.: Pract. Exper. 13 (6), 465–501. http://dx.doi.org/10.1002/cpe.583.

Qi, Z., Long, F., Achour, S., Rinard, M., 2015. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In: Proceedings of the 2015 International Symposium on Software Testing and Analysis. In: ISSTA 2015, Association for Computing Machinery, New York, NY, USA, pp. 24–36. http://dx.doi.org/10.1145/2771783.2771791.

Qi, Y., Mao, X., Lei, Y., Dai, Z., Wang, C., 2014. The strength of random search on automated program repair. In: Proceedings of the 36th International Conference on Software Engineering. In: ICSE 2014, Association for Computing Machinery, New York, NY, USA, pp. 254–265. http://dx.doi.org/10.1145/2568225.2568254.

Randrianaina, G.A., Tërnava, X., Khelladi, D.E., Acher, M., 2022. On the benefits and limits of incremental build of software configurations: An exploratory study. In: Proceedings of the 44th International Conference on Software Engineering. ICSE '22, Association for Computing Machinery, New York, NY, USA, pp. 1584–1596. http://dx.doi.org/10.1145/3510003.3510190.

Schaefer, I., Bettini, L., Bono, V., Damiani, F., Tanzarella, N., 2010. Delta-oriented programming of software product lines. In: Software Product Lines: Going beyond: 14th International Conference, SPLC 2010, Jeju Island, South Korea, September 13-17, 2010. Proceedings 14. Springer, pp. 77–91.

Smith, E.K., Barr, E.T., Le Goues, C., Brun, Y., 2015. Is the cure worse than the disease? Overfitting in automated program repair. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering. In: ESEC/FSE 2015, Association for Computing Machinery, New York, NY, USA, pp. 532–543. http://dx.doi.org/10.1145/2786805.2786825.

Souto, S., d'Amorim, M., Gheyi, R., 2017. Balancing soundness and efficiency for practical testing of configurable systems. In: Proceedings of the 39th International Conference on Software Engineering. ICSE '17, IEEE Press, pp. 632–642. http://dx.doi.org/10.1109/ICSE.2017.64.

Takahashi, A., Sae-Lim, N., Hayashi, S., Saeki, M., 2018. A preliminary study on using code smells to improve bug localization. In: Proceedings of the 26th Conference on Program Comprehension. ICPC '18, Association for Computing Machinery, New York, NY, USA, pp. 324–327. http://dx.doi.org/10.1145/3196321.3196361.

Takahashi, A., Sae-Lim, N., Hayashi, S., Saeki, M., 2021. An extensive study on smell-aware bug localization. J. Syst. Softw. 178, 110986. http://dx.doi.org/10.1016/j.jss.2021.110986.

Thüm, T., Kästner, C., Benduhn, F., Meinicke, J., Saake, G., Leich, T., 2014. FeatureIDE: An extensible framework for feature-oriented software development. Sci. Comput. Program. 79, 70–85. http://dx.doi.org/10.1016/j.scico.2012.06.002.

Wang, S., Wen, M., Lin, B., Wu, H., Qin, Y., Zou, D., Mao, X., Jin, H., 2021. Automated patch correctness assessment: How far are we? In: Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering. ASE '20, Association for Computing Machinery, New York, NY, USA, pp. 968–980. http://dx.doi.org/10.1145/3324884.3416590.

Weiss, A., Guha, A., Brun, Y., 2017. Tortoise: interactive system configuration repair. In: Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering. ASE '17, IEEE Press, pp. 625–636. http://dx.doi.org/10.1109/ASE.2017.8115673.

Wen, M., Chen, J., Wu, R., Hao, D., Cheung, S.-C., 2018. Context-aware patch generation for better automated program repair. In: Proceedings of the 40th International Conference on Software Engineering. ICSE '18, Association for Computing Machinery, New York, NY, USA, pp. 1–11. http://dx.doi.org/10.1145/3180155.3180233.

White, M., Tufano, M., Martínez, M., Monperrus, M., Poshyvanyk, D., 2019. Sorting and transforming program repair ingredients via deep learning code similarities. In: 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering. SANER, pp. 479–490. http://dx.doi.org/10.1109/SANER.2019.8668043.

Wohlin, C., Runeson, P., Hst, M., Ohlsson, M.C., Regnell, B., Wessln, A., 2012. Experimentation in Software Engineering. Springer Publishing Company, Incorporated.

Wong, W.E., Gao, R., Li, Y., Abreu, R., Wotawa, F., 2016. A survey on software fault localization. IEEE Trans. Softw. Eng. 42 (8), 707–740.

Wong, C.-P., Meinicke, J., Lazarek, L., Kästner, C., 2018. Faster variational execution with transparent bytecode transformation. Proc. ACM Program Lang. 2 (OOPSLA), http://dx.doi.org/10.1145/3276487.

Xia, C.S., Wei, Y., Zhang, L., 2023. Automated program repair in the era of large pre-trained language models. In: Proceedings of the 45th International Conference on Software Engineering. ICSE '23, IEEE Press, pp. 1482–1494. http://dx.doi.org/10.1109/ICSE48619.2023.00129.

Xin, Q., Reiss, S.P., 2017. Leveraging syntax-related code for automated program repair. In: Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering. ASE '17, IEEE Press, pp. 660–670.

Xiong, Y., Zhang, H., Hubaux, A., She, S., Wang, J., Czarnecki, K., 2015. Range fixes: Interactive error resolution for software configuration. IEEE Trans. Softw. Eng. 41 (6), 603–619. http://dx.doi.org/10.1109/TSE.2014.2383381.

Xuan, J., Martinez, M., DeMarco, F., Clement, M., Marcote, S.L., Durieux, T., Le Berre, D., Monperrus, M., 2017. Nopol: Automatic repair of conditional statement bugs in Java programs. IEEE Trans. Softw. Eng. 43 (1), 34–55. http://dx.doi.org/10.1109/TSE.2016.2560811.

Yang, A.Z.H., Le Goues, C., Martins, R., Hellendoorn, V., 2024. Large language models for test-free fault localization. In: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering. ICSE '24, Association for Computing Machinery, New York, NY, USA, http://dx.doi.org/10.1145/3597503.3623342.

Yang, D., Mao, X., Chen, L., Xu, X., Lei, Y., Lo, D., He, J., 2023. TransplantFix: Graph differencing-based code transplantation for automated program repair. In: Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering. ASE '22, Association for Computing Machinery, New York, NY, USA, http://dx.doi.org/10.1145/3551349.3556893.

Ye, H., Martinez, M., Durieux, T., Monperrus, M., 2021a. A comprehensive study of automatic program repair on the QuixBugs benchmark. J. Syst. Softw. 171, 110825. http://dx.doi.org/10.1016/j.jss.2020.110825.

Ye, H., Martinez, M., Monperrus, M., 2021b. Automated patch assessment for program repair at scale. Empir. Softw. Eng. 26 (2), http://dx.doi.org/10.1007/s10664-020-09920-w.

Ye, H., Martinez, M., Monperrus, M., 2022. Neural program repair with execution-based backpropagation. In: Proceedings of the 44th International Conference on Software Engineering. ICSE '22, Association for Computing Machinery, New York, NY, USA, pp. 1506–1518. http://dx.doi.org/10.1145/3510003.3510222.

Yuan, Y., Banzhaf, W., 2020a. ARJA: Automated repair of Java programs via multi-objective genetic programming. IEEE Trans. Softw. Eng. 46 (10), 1040–1067. http://dx.doi.org/10.1109/TSE.2018.2874648.

Yuan, Y., Banzhaf, W., 2020b. Toward better evolutionary program repair: An integrated approach. ACM Trans. Softw. Eng. Methodol. 29 (1), http://dx.doi.org/10.1145/3360004.

Zhang, Q., Fang, C., Ma, Y., Sun, W., Chen, Z., 2023. A survey of learning-based automated program repair. ACM Trans. Softw. Eng. Methodol. 33 (2), http://dx.doi.org/10.1145/3631974.

Thu-Trang Nguyen is a Ph.D. candidate in Software Engineering at University of Engineering and Technology, Vietnam National University, Hanoi (VNU - UET). She received her M.Sc. degree from Japan Advanced Institute of Science and Technology in 2019 and received her B.Sc. degree from VNU - UET in 2016. Her research interests include program analysis, software product line systems, software vulnerability detection, and software testing.

Xiao-Yi Zhang received the B.Sc. degree and Ph.D. degree from Beihang University (BUAA), in 2010 and 2018, respectively. He is currently an associate professor with the School of Computer and Communication Engineering at University of Science and Technology Beijing (USTB), China. His research interests include software testing and safety analysis for smart software, such as cyber–physical systems, autonomous driving systems, and deep learning models. https://researchmap.jp/xiaoyi-zhang

Paolo Arcaini is an associate professor at the National Institute of Informatics, Japan. He received a Ph.D. in Computer Science from the University of Milan in 2013. Before joining NII, he held an assistant professor position at Charles University, Czech Republic. His current main research interests are related to testing of autonomous driving systems, testing of quantum programs, automatic repair of neural networks, and falsification of hybrid systems. More information is available at https://group-mmm.org/~arcaini/

Fuyuki Ishikawa is an associate professor at National Institute of Informatics and at Sokendai, Japan. His research interests include software engineering techniques for trustworthy smart systems, including testing, verification, and repair techniques for automated driving systems and learning-based systems. Ph.D. (information science and technology, The University of Tokyo, 2007).

Hieu Dinh Vo is the head of the Department of Software Engineering at the Faculty of Information Technology, the University of Engineering and Technology, VNU Hanoi. He earned a Bachelor's degree in Computer Engineering from the Ho Chi Minh City University of Technology, a Master's degree in Distributed Multimedia Systems from the University of Leeds, and a Ph.D. in Information Science from the Japan Advanced Institute of Science and Technology. His research interests include source code analysis, software testing, service computing, and software architecture.