# Method-Level Bug Localization: A Hybrid Multi-objective Search Approach

Rafi Almhana[1], Marouane Kessentini[1] and Wiem Mkaouer[2]

ARTICLE INFO

*Keywords*:
Bugs localization
Fault localization
Search-based software engineering
Bug reports
Multi-objective search
Simulated annealing
Software maintenance

Abstract

**Context:** One of the time-consuming maintenance tasks is the localization of bugs especially in large software systems. Developers have to follow a tedious process to reproduce the abnormal behavior then inspect a large number of files. While several studies have been proposed for bugs localization, the majority of them are recommending classes/files as outputs which may still require high inspection effort. Furthermore, there is a significant difference between the natural language used in bug reports and the programming language which limits the efficiency of existing approaches since most of them are mainly based on lexical similarity.

**Objective:** In this paper, we propose an automated approach to find and rank the potential methods in order to localize the source of a bug based on a bug report description.

**Method:** Our approach finds a good balance between minimizing the number of recommended classes and maximizing the relevance of the proposed solution using a hybrid multi-objective optimization algorithm combining local and global search. The relevance of the recommended code fragments is estimated based on the use of the history of changes and bug-fixing, and the lexical similarity between the bug report description and the API documentation. Our approach operates on two main steps. The first step is to find the best set of classes satisfying the two conflicting criteria of relevance and the number of classes to recommend using a global search based on NSGA-II. The second step is to locate the most appropriate methods to inspect, using a local multi-objective search based on Simulated Annealing (MOSA) from the list of classes recommended by the first step.

**Results:** We evaluated our system on 6 open source Java projects, using the version of the project before fixing the bug of many bug reports. Our hybrid multi-objective approach is able to successfully locate the true buggy methods within the top 10 recommendations for over 78% of the bug reports leading to a significant reduction of developers' effort comparing to class-level bug localization techniques.

**Conclusion:** The experimental results show that the search-based approach significantly outperforms four state-of-the-art methods in recommending relevant files for bug reports.

## 1. Introduction

A software bug is a coding error that may cause abnormal behaviors and incorrect results when executing the system [1]. After identifying an unexpected behavior of the software project, a user or developer will report it in a document, called a bug report [2]. Thus, a bug report should provide useful information to identify and fix the bug. The number of these bug reports can be large. For example, MOZILLA had received more than 420,000 bug reports [3]. These reports are important for managers and developers during their daily development and maintenance activities including bug localization [4]. The process of finding the relevant source code fragments (methods, classes, etc.) that need to be modified to fix the bug according to a bug report description is defined as bug localization [5].

A developer always uses a bug report to reproduce the abnormal behavior to find the origin of the bug. However, the poor quality of bug reports can make this process tedious and time-consuming due to missing information. To find the cause of a bug, developers are not only using their own knowledge to investigate the bug report but interact with peer developers to collect additional information. An efficient automated approach for locating and ranking important code fragments for a specific bug report may lead to improve the productivity of developers by reducing the time to find the cause of a bug [4].

The majority of existing bug localization studies are mainly based on lexical matching scores between the statements of bug reports and the name of code elements in software systems [6, 7, 8]. However, there is a significant difference between the natural language used in bug reports and the programming language which limits the efficiency of existing approaches.

We considered, in this work, the following important observations. First, API documentation of the classes can be more useful than the name of code elements or comments to estimate the similarity between code fragments and bug reports [9]. Second, code fragments associated with previously fixed bug reports may be relevant also to the current report if these previously fixed bug reports are similar to a current bug report [10]. Third, a code fragment that was fixed recently is more likely to still contain bugs than another class that was last fixed a long time ago [2]. Fourth, a code fragment that has been frequently fixed, tend to be fault-prone and may cause more than one abnormal behavior in the future [11]. Finally, the recommendation of a large number of classes to inspect may make the process of finding the cause of a bug time-consuming.

ralmhana@umich.edu (R. Almhana); marouane@umich.edu (M. Kessentini); mwmvse@rit.edu (W. Mkaouer)

[1]Department of Computer and Information Science, University of Michigan.

[2]Department of Software Engineering, Rochester Institute of Technology.

To address some of these challenges, we proposed in our previous work [12] a comprehensive approach for bugs localization based on bug reports description. We utilized a multi-objective optimization algorithm [13] to find a balance between maximizing lexical and history-based similarity, and minimizing the number of recommended classes. The problem is formulated as a search for the best combination and sequence of classes from all the classes of the system that optimize as much as possible the balance between the above two conflicting objectives. The main feedback received from the participants of our experiments is that the file/class level recommendations are still time-consuming to explore and they very much prefer a precise localization at the method level.

In this work, we extended our previous work [12] to provide method-level bug localization, instead of the class-level bug localization. Our approach optimizes the trade-off between minimizing the number of recommended classes and maximizing the relevance of the proposed solution using a hybrid multi-objective optimization algorithm combining local and global search. The relevance of the recommended code fragments is estimated based on the use of the history of changes and bug-fixing, and the lexical similarity between the bug report description and the API documentation. Our approach includes two main steps: finding the best set of classes satisfying the two conflicting criteria of relevance and the number of classes to recommend and locating the most appropriate methods to inspect in those classes. We accomplish the former using a global search based on NSGA-II and the latter using a local multi-objective search based on Simulated Annealing (MOSA) [14].

We have executed an extensive empirical evaluation of 6 large open-source software projects with more than 22,000 bug reports in total based on an existing benchmark [15]. The experimental results show that the search-based approach significantly outperforms four state-of-the-art techniques in recommending relevant files for bug reports including our previous work at the class [7, 16, 17, 12] and method [16] levels. In particular, our hybrid multi-objective approach can successfully locate the true buggy methods within the top 10 recommendations at the methods level for over 78% of the bug reports.

The primary contributions of this paper can be summarized as follows:

- To the best of our knowledge and based on recent surveys [18], the paper proposes one of the first search-based software engineering approaches to address the problem of finding relevant code fragments for bug reports. The approach combines the use of lexical and history-based similarity measures to locate and rank relevant code fragments for bug reports while minimizing the number of recommended classes.
- We extended our previous work by proposing a new hybrid multi-objective formulation, using NSGA-II and MOSA, that combines global and local search to localize bugs at the method level instead of the class level.
- The paper reports the results of an empirical study with an implementation of our hybrid multi-objective approach. The obtained results provide evidence to support the claim that our proposal is more efficient, on average, than existing techniques for bugs localization at the method and class levels [7, 16, 17] based on a benchmark of 6 open source systems [19]. We also compared the results of our hybrid multi-objective approach with a mono-objective formulation to make sure that our objectives are conflicting and our previous work which based only on a global search.

The remainder of this paper is as follows: Section 2 is dedicated to the related work. Section 3 describes the proposed approach and the hybrid search algorithm. The evaluation of our approach and its results on some research questions are explained in Section 4 while Section 5 further discusses the obtained results. Section 6 describes the threats to validity related to our experiments. Finally, concluding remarks and future work are provided in Section 7.

## 2. Related Work

In this section, we survey different studies related to the areas of bug localization and search-based software engineering [20, 12, 21, 22, 23, 24, 25].

### 2.1. Bug Localization

The problem of bug localization can be considered as searching the source of a bug report given its description. To address this problem, the majority of existing studies is based on the use of Information-Retrieval (IR) techniques through the detection of textual and semantic similarities between a newly given report and source code entities [6]. Several IR techniques have been investigated, namely the Latent Semantic Indexing (LSI) [26], Latent Dirichlet Allocation (LDA) [27] and the Vector Space Model (VSM) [28]. Also, hybrid models extracted from these IRs techniques to tackle the problem of bug localization were proposed [15].

We summarize, in the following, the different tools and approaches proposed in the literature based on the above IR techniques. BugScout [7] is a topic-based approach using LDA to analyze the bug related information (description, comments, external links, etc.) to detect the source of a bug and duplicated bug reports. The main limitation of BugScout is the dependency of the results on the keywords entered by the user. DebugAdvisor [8] is a bug investigation system that takes as input a bug report in terms of text queries then uses them to mine existing fixed bug repository and generate a graph of possible reports. However, DebugAdvisor accuracy depends on the accuracy of the report's description and its accuracy when describing the bug and its related code entities.

BugLocator [17] combines several similarity scores from previous bug reports for bug localization. It generates a VSM

model to extract suspect source files for a given bug report. Then, BugLocator mines previously fixed bug reports along with related files involved to rank suspect code fragments. The main issue raised in this work is the proneness of the weight density to the noise in the large files. To overcome this limitation, the work of Wong et al. [29] added segmentation and stack trace analysis to improve the performance of the BugLocator approach. The limitation of this extension is that execution traces are not necessarily available in bug repositories.

BLUiR [30] has been proposed also to compare a bug report to the structure of source files. It decomposes reports into summaries and then uses the structural retrieval to calculate similarities between these
tokenized elements and source code ones to rank source code files. Saha et al. [31] extended BLUiR to consider similar reports information, similarly to BugLocator as an additional similarity score. DHbPd [32] incorporated code change information for bug localization. The main idea is to consider recently changed source code elements as potential candidates for hosting a bug.

Ye et al. [15] have modeled the similarity between bug reports and source code through several characteristics that are captured through the use of 6 similarity features that describe the projects' domain knowledge. The combination of these measures is fed to a ranking heuristic called learning-to-rank. The ranking model returns the top candidate source files to investigate for a given bug report. The main originality of their work is the use of project's API description and auto-generated documentation as one of the features to utilize to reduce the lexical gap between the human description and the source code.

Ye et al. [16] extended their previous work by extending their ranking features utilized by learning-to-rank from 6 to 19. Besides the existing surface lexical similarity, API-based lexical similarity, collaborative filtering, code elements naming similarity, fixed bugs frequency, they included other source code characteristics that can be extracted from the projects such as summaries, naming conventions, interclass dependencies, etc. Although taking these features into account has given better results in terms of better files ranking, such information may not be available in all projects and sometimes it may be outdated and that may deteriorate the localization accuracy.

AmaLgam [33] introduced the aggregation of relevant similarities extracted from source files, version history data and previously resolved bugs to calculate a global score for ranking files. This approach has given promising results compared to the previous techniques as it does not only combines code structure with previous reports but also it involves historical data to maximize the bug information coverage and enhance the localization accuracy.

Lamkanfi et al. developed a binary classifier to determine whether or not a bug is severe [34]. The report features were used as a training set to conduct a comparison between several classifiers, namely SVM, Naïve Bayes, Multinomial Naïve Bayes, and Nearest Neighbor. The experiments have shown that these classifiers outperform random severity assignment formulations. Similarly, the work of Lo et al. [35] presented a classification engine labeled GRAY that extends the linear regression to predict the priority of bugs, but not bugs localization, while taking into account various external and internal report characteristics, extracted as features, then used to train the model.

Table 1 shows the most recent and relevant studies to our approach. It illustrates the differences among those studies in terms of input, output, and technique used to solve the bug localization problem. We notice that among all the studies listed in Table 1, the majority of them are related to recommend classes using Information Retrieval (IR) techniques. There are four different studies that addressed method-level bug localization. Youm et al. [36] utilized bug report description with code changes, source code, comments and stack traces and developer's log to find relevant methods.

Another approach is proposed by Lukins et al. [37] to localize bugs at the methods level using a static Latent Dirichlet allocation (LDA) technique which is solely based on source code. In another study [16], Ye et al. used learning to rank technique to develop a ranking model in which they assign a weight for each source code file as a result of several features such as source code, bug description, code changes history and bugs-fixing history. The proposed approach can be adopted for both class and method levels. We used this approach as one of the baselines to evaluate the performance of our approach since the authors provided a replication package.

## 2.2. Search-Based Software Engineering

Search-Based Software Engineering (SBSE) uses a computational search approach to solve optimization problems in software engineering [45]. Once a software engineering task is framed as a search problem, by defining it in terms of solution representation, fitness function, and solution change operators, there is a multitude of search algorithms that can be applied to solve that problem. Many search-based software testing techniques have been proposed for test cases generation [46, 47], mutation testing [48, 49], and regression testing [50]. However, the problem of bugs localization was not addressed using SBSE before our work [12].

The closest problem addressed using SBSE techniques is the bugs prioritization problem [51]. A mono-objective genetic algorithm was proposed to find the best sequence of bugs resolution that maximizes the relevance and importance of the bugs to fix while minimizing the cost. The authors formulated the bugs prioritization as an optimization problem. Each incoming bug is assigned three scores: the relevance score reflects developers' opinions about how significant is a bug in the repository compared to others, the importance score represents how fast the bug should be fixed, while the third score is the bug severity. These scores are aggregated in a fitness function. The authors deployed the genetic algorithm to randomly generate a population of possible orderings that evolve towards the best sequence of bugs resolution that maximizes the relevance and importance while

**Table 1**
Recent Studies

| Study | Input | Output | Technique | Date | Category |
|---|---|---|---|---|---|
| Huang, Qiao, et al. [38] | Bug Report, Source code | Package | IR, ML | Oct 2017 | Information Retrieval |
| Wen, Ming, et al. [39] | Developer's log, Software Changes. | Class | IR | Sep 2016 | |
| Youm, Klaus Changsun, et al. [36] | Comments, Stack Traces, Developer's Log and Code Changes. | Method | IR | Feb 2017 | |
| Lukins, Stacy K., et al. [37] | Source Code | Method | IR (LDA) | Sep 2010 | |
| Tantithamthavorn, Chakkrit, et al. [40] | Source Code, Bug Report | Method | IR-based Classifier | Oct 2018 | |
| Ye, Xin, et al. [41] | Source Code, Bug Description, Code Changes | Method | Ranking Model | Sep 2015 | Ranking Model |
| Pablo Loyola, et al. [42] | Code Changes | Class | Ranking Model | Oct 2018 | |
| An Ngoc Lam, et al. [43] | Source code, Bug Report | Class | IR (rVSM), neural network | May 2017 | Neural Network |
| Yan Xiao, et al. [44] | Bug Report | Class | deep learning translation | July 2018 | |
| Almhana, Rafi, et al. [12] | Source code, Bug Report, History of Bug Report | Class | Search Based Software Engineering | Sep 2016 | Search Based |

minimizing the severity. To overcome the limitation of aggregating two attributes that may experience conflicts, they extended their work [52] to better find the trade-off between bugs with low relevance to the users may have high severity scores.

## 3. The Hybrid Multi-Objective Formulation for Bug Localization

We first present an overview of our hybrid multi-objective approach to identify and prioritize relevant methods for bug reports, and then we describe the details of our formulation.

### 3.1. Approach Overview

Our approach aims at exploring a large search space to find relevant methods, to inspect by developers, given a description of a bug report. The search space is determined not only by the number of possible method combinations to recommend but also by the order in which they are proposed to the developer. In fact, bug reports may require the inspection of more than one method to identify and fix bugs.

Due to this large search space of potential solutions to explore, we propose a heuristic-based optimization method including two main steps. The first step, based on a global search, operates on the classes level while the second one, based on a local search, operates on the methods level of selected classes after the first step. A local search is used in the second step due to the reasonable size of the search space that consists of the methods of identified classes. We note that the search space at the method level after the first filtering step at the class level is still a large search problem due to the typical large size of classes. We noticed in our previous work for bugs localization at the class level that each file might have on average over 20 methods per class [12].

The difference between the two search strategies is not related to the objective space but to the change operators and population size/generation [53]. In global search, we use a population of several solutions at each iteration, and both crossover and mutation operators to create a significant perturbation of the solutions at each iteration. However, the local search is limited to one solution (not a population) at each iteration and only a limited change operator (mutation) to create a limited variation at each iteration when generating a new solution. The rationale behind the difference of both search strategies that the population and both change operators can help to explore a large search space to identify relevant solutions using the global search (all the classes of the systems). Once these relevant solutions are identified

then a local search can be applied to a smaller search space (limited number of classes that may contain the bug) using one solution at each iteration and only a mutation operator. The local search is faster than running another global search due to the limited search space [53].

The general structure of our approach is sketched in Figure 1. It takes 5 inputs as follows:

- The source code of the project to be inspected,
- The API specifications,
- The description of the bug report,
- A list of previous bug reports of the project,
- The history of the applied changes in previous releases of the project.

Our approach generates as output, in the first step, a near-optimal sequence of ranked classes that maximizes the relevance to the bug report and minimizes the number of recommended classes. The list of identified classes for inspection can be checked by the developer, as an optional step, to further reduce the number of class recommendations as shown in Figure 2. Then, the second step is executed to generate as output a near-optimal sequence of ranked methods that maximize the relevance to the bug report and minimizes the number of recommended methods out of the classes identified in the first step.

Both heuristic-based optimization steps are formulated based on two main conflicting objectives. The first objective is the correctness function that includes two routines:

- Maximizing the Lexical similarity between recommended code fragments (e.g. classes for the first step and methods for the second step) and the description of the bug report (including the API and name of code elements similarity);
- Maximizing the history-based function score that includes the number of recommended code fragments that have been fixed in the past, recent changes introduced by the developers to these code fragments and similarities with previous bug reports.

The second objective is to minimize the number of code fragments to recommend. We note that a code fragment for the global search algorithm is a class while it is a method for the local search algorithm. Thus, the total number of different fitness functions is four. The first fitness function for the global search algorithm evaluates the lexical similarity and history-based information between the classes and the bug reports/API while the second fitness function counts the number of classes to inspect by the developer. For the local search, the first fitness function calculates the lexical similarity and history-based information at the methods level while the second fitness function estimates the number of methods to be inspected by the developer. Thus, a total of four fitness functions are defined with two functions for each search algorithm.

**Definition 1: Pareto optimality**

A solution $x^* \in \Omega$ is Pareto optimal if $\forall x \in \Omega$ and $I = \{1, ..., M\}$ either $\forall m \in I$ we have $f_m(x) = f_m(x^*)$ or there is at least one $m \in I$ such that $f_m(x) > f_m(x^*)$.

**Definition 2: Pareto dominance**

A solution $x$ is said to dominate another solution $v$ ( denoted by $f(x) \prec f(v)$ ) if and only if $f(x)$ is partially less than $f(v)$. In other words, $\forall m \in \{1, ..., M\}$ we have $f_m(x) \leq f_m(v)$ and $\exists m \in \{1, ..., M\}$ where $f_m(x) < f_m(v)$.

**Definition 3: Pareto optimal set**

For a given MOP $f(x)$, the Pareto optimal set is $P^* = \{x \in \Omega | \neg \exists x' \in \Omega, f(x') \prec f(x)\}$.

It is obvious that those two objectives are conflicting since maximizing the relevance of recommended code fragments may lead to low precision and thus increase the number of recommended code fragments. Thus, we consider, in this paper, the task of bugs localization as a hybrid multi-objective optimization problem. We used the non-dominated sorting genetic algorithm (NSGA-II) as a global search for the class level [13] and the multi-objective Simulated Annealing algorithm (MOSA) [14] as a local search for the method level.

When comparing the relative fitness of generated solutions, both NSGA-II and MOSA utilize the idea of Pareto optimality (Definition 1) using dominance (Definition 2) as a basis for comparison [54]. The set of trade-off solutions is called Pareto optimal solutions or non-dominated solutions, and the image of this set in the objective space is called the Pareto front. Hence, the output of NSGA-II and MOSA consists in approximating the entire Pareto front (Definition 3) [54].

The definition of Pareto optimality states that $x^*$ is Pareto optimal if no feasible vector exists that would improve some objectives without causing a simultaneous worsening in at least one other objective.

In addition to Pareto Optimality and Pareto Dominance, we need to define Pareto Optimal set and Pareto Optimal Front.

In the following, we describe an overview of both algorithms, the solution representation, a formal formulation of the two objectives to optimize and the change operators.

## 3.2. NSGA-II

In this paper, we adapted one of the widely used multi-objective algorithms called NSGA-II [13]. NSGA-II is a powerful global search method stimulated by natural selection that is inspired by the theory of Darwin. Hence, the basic idea of NSGA-II is to make a population of candidate solutions evolve toward the near-optimal solution in order to solve a multi-objective optimization problem. NSGA-II is designed to find a set of optimal solutions, called non-dominated solutions, also Pareto set. A non-dominated solution is the one which provides a suitable compromise be-

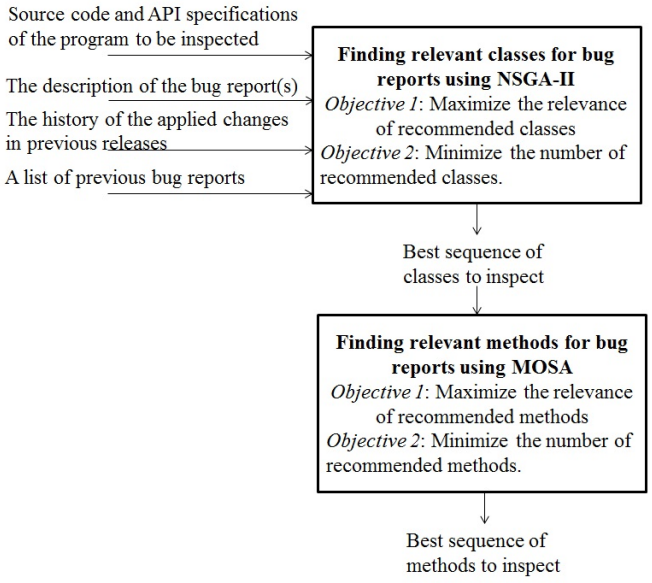


**Figure 1:** Approach Overview

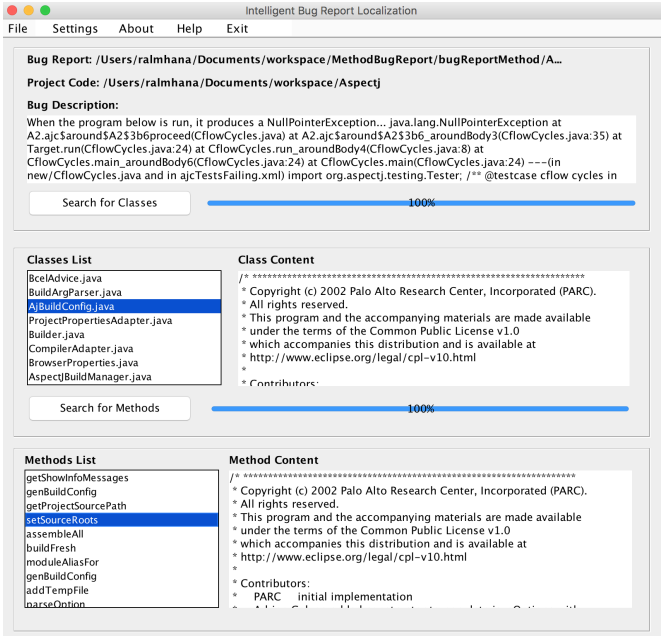


**Figure 2:** The proposed bugs localization tool.

tween all objectives without degrading any of them. The first step in NSGA-II is to create randomly a population $P_0$ of individuals encoded using a specific representation. Then, a child population $Q_0$ is generated from the population of parents $P_0$ using genetic operators such as crossover and mutation. Both populations are merged into a population $R_1$ of size N. NSGA-II starts by generating an initial population based on a specific representation that will be discussed later in the Solution Representation section of this paper (section 3.5.1 and section 3.6.1). This initial population consists

**Definition 4: Pareto optimal front**

For a given MOP $f(x)$ and its Pareto optimal set $P^*$, the Pareto front is $PF^* = \{f(x), x \in P^*\}$.

of a list of code fragments from the studied system. Thus, this population stands of a set of solutions represented as sequences of code fragments to be evaluated by the fitness functions for a specific bug report description taken as input.

The whole population that contains N individuals (solutions) is sorted using the dominance principle into several fronts. The dominance level becomes the basis of a selection of individual solutions for the next generation. Fronts are added successively until the parent population $P_{t+1}$ is filled with N solutions. When NSGA-II has to cut off a front $F_i$ and select a subset of individual solutions with the same dominance level, it relies on the crowding distance to make the selection. This front $F_i$ to be split, is sorted in descending order, and the first $(N - |P_{t+1}|)$ elements of $F_i$ are chosen. Then a new population $Q_{t+1}$ is created using selection, crossover, and mutation. This process repeats until reaching the last iteration according to stop criteria. The following three subsections describe more precisely our adaption of NSGA-II to the model change detection problem.

### 3.3. Multi-Objective Simulated Annealing (MOSA)

Multi-objective Simulated Annealing is a local search heuristic inspired by the concept of annealing in metallurgy where metal is heated, raising its energy and relieving it of defects due to its ability to move around more easily [55]. As its temperature drops, the metal's energy drops and eventually it settles in a more stable state and becomes rigid. The local search algorithm of the Simulated Annealing is very suitable for exploring reasonable search spaces in terms of the size like in our case [55].

The first step of the MOSA algorithm is to initialize a total of five parameters: temperature parameter $T_0$, cooling factor $\alpha$ and cooling step $N_{Step}$, final temperature $T_{Stop}$ and the maximum number of iteration $N_{Stop}$.

In MOSA, the mutated solution will be kept and used for the next iteration if it dominates or is in the same non-dominating front as the solution from the previous iteration. To determine the probability that the mutated solution dominated by the solution from the previous iteration will be kept and used for the next iteration of MOSA, there are several possible acceptance probability functions that can be utilized.

Since the previous works [50, 55] have noted that the average cost criteria yields good performance we have utilized this metric. The average cost criteria simply takes the average of the differences of each objective value between two solutions, $i$ and $j$, over all objectives $D$, as shown in Equation

1. The final acceptance probability function used in MOSA is shown in Equation 2.

$$c(i,j) = \frac{\sum_{k=1}^{|D|} c_k(j) - c_k(i)}{|D|} \quad (1)$$

$$AcceptProb(i,j,temp) = e^{\frac{-abs(c(i,j))}{temp}} \quad (2)$$

Where *temp* is the current temperature and $c_k(i)$ is the objective value for solution $i$. As explained in the next sections, MOSA will be used at the method level of our bug localization approach in order to recommend the most relevant methods of the classes identified by the first step of our approach based on NSGA-II.

### 3.4. Fitness Functions

Both steps of our approach use two main fitness functions that are applied at the class level (global search) and the method level (local search). The first objective consists of the size of the solution which corresponds to the number of recommended classes or methods. The second objective of correctness is defined as the average of two functions: lexical-based similarity (LS) and history-based similarity (HS). Thus, we formally define this function as:

$$f_1 = \frac{LS + HS}{2} \quad (3)$$

The lexical-based similarity (LS) consists of an average of two functions. The first function is based on a cosine similarity [56] between the description of a bug report and the source code while the second one checks the similarity between the description of a bug report and the API documentation. We used the whole content of a source code file (the code and comments). The vocabulary was extracted from the names of variables, classes, methods, parameters, types, etc. We used the Camel Case Splitter to perform the Tokenization for prepossessing the identifiers [57].

During the tokenization process, we used a standard information retrieval stop words to eliminate irrelevant information such as punctuation, numbers, etc. In addition, the words are reduced to their stem based on a Porter Stemmer. This operation reduces the deviation between related words such as "designing" and "designer" to the same stem "design". Then, the cosine similarity measure is used to compare between the description of a bug report and the source code (classes or methods).

Equation 4 calculates the cosine similarity between two vectors. Each actor is represented as an n dimensional vector, where each dimension corresponds to a vocabulary term. The cosine of the angle between two vectors is considered as an indicator of similarity. Using cosine similarity, the conceptual similarity between two vectors $c_1$ and $c_2$ is determined as follows:

$$\begin{aligned} Sim(c_1,c_2) &= \cos(\overrightarrow{c_1},\overrightarrow{c_2}) = \frac{\overrightarrow{c_1}.\overrightarrow{c_2}}{\|\overrightarrow{c_1}\| \times \|\overrightarrow{c_2}\|} \\ &= \frac{\sum_{i=1}^{n}(w_{i,1} \times w_{i,2})}{\sqrt{\sum_{i=1}^{n}(w_{i,1})^2 \times \sum_{i=1}^{n}(w_{i,2})^2}} \in [0,1] \end{aligned} \quad (4)$$

where $\overrightarrow{c_1} = (w_{1,1}, ..., w_{n,1})$ is the term vector corresponding to actor $c_1$ and $\overrightarrow{c_2} = (w_{1,2}, ..., w_{n,2})$ is the term vector corresponding to $c_2$. The weights $w_{i,j}$ is computed using information retrieval based techniques such as the Term Frequency - Inverse Term Frequency (TF-IDF) method.

The second component of the correctness objective is the history-based similarity. This measure is an average of three functions. The first function counts the number of times that a code snippet (i.e. classes or methods) was fixed to eliminate bugs based on the history of bug reports. In fact, a source code that was fixed several times has a high probability of being buggy and includes new bugs. Formally, this function will be normalized between [0,1] and defined as:

$$H_1 = \frac{\sum_{i=1}^{Size(S)} NbFixedBugs(report, C_i)}{Size(S) \times Max(NbFixedBugs(report, C_i))} \quad (5)$$

Where $S$ is a solution containing a number of recommended classes $S = \{c_1, c_2, \ldots, c_{size(S)}\}$. The second function checks if a code snippet (i.e. classes or methods) was recently changed or fixed. In fact, a source code that was modified recently has a higher probability of containing a bug. Thus, this function compares between the date of the bug report and the last date where the source code was modified. If a suggested code snippet was modified on the same day of the bug report then the value of this function is 1. We define this normalized function, normalized in the range of [0,1] as following:

$$H_2 = \frac{\sum_{i=1}^{Size(S)} \frac{1}{report.data - last(report, c_i) + 1}}{Size(S)} \quad (6)$$

The third function evaluates the consistency between the recommended source code based on previous bug reports. The code snippets that are recommended together for similar previous bug reports have a high probability to include a bug evolving most of them. To this end, this function calculates the cardinality, Cbr, of the largest intersection set of code snippets (i.e. classes or methods) between the solution S and the sets of code snippets (i.e. classes or methods) recommended for each of previous bug reports. Then, this measure is normalized between [0,1] and defined as follows:

$$H_3 = \frac{Cbr}{Size(S)} \quad (7)$$

**Bug ID:** 101751
**Commit Summary:** Bug 101751 Enhance **IImagehandler** interface to allow full customization of **image handling** mechanism
**Bug Description:** BIRT currently stores temp chart/**image** into a directory that the viewer provides. It assumes that the **image** can be accessed then as a static resource, bypassing the application server. This is achieved by generating chart/**image** URL that points to the **image** directly. In a WAR deployment environment, the **image** directory can no longer be specified under the web-application, because the web app installation directory can not always be found. The modified mechanism is to have the **image** directory as a hard-coded directory, instead of retrieved by getRealPath() call now. Because the **images** are no longer stored in the web app, they may not be accessible directly through URL without engine's help. The proposed solution is to enhance **ImageHandler**, so that it not only stored **image**, but returns **image** too. This way, the web application (viewer) could set the **image handler**, the engine writes the **images**, then the viewer, given a reference to the imahe **handler**, could call the get functions to retrieve **images** and send back to client based on the URL. **IImageHandler** therefore needs to be enhanced. So do the default **image handler** implementations.

**Figure 3:** BIRT Bug Report Example (ID 101751)

```
public class HTMLServerImageHandler implements IHTMLImageHandler
{
    protected Logger log = Logger.getLogger(HTMLServerImageHandler.class.getName());

    private static int count = 0;

    private static HashMap map = new HashMap();

    public HTMLServerImageHandler()
    {}

    public String onDesignImage(IImage image, Object context)
    {}

    public String onDocImage(IImage image, Object context)
    {}

    public String onURLImage(IImage image, Object context)
    {}

    public String onCustomImage(IImage image, Object context)
    {}

    protected String createUniqueFileName(String imageDir, String prefix, String postfix)
    {}

    protected String createUniqueFileName(String imageDir, String prefix)
    {}

    public String onFileImage(IImage image, Object context)
    {}

    protected String handleImage(IImage image, Object context, String prefix, boolean needMap)
    {}

    protected String getImageMapID(IImage image)
    {}
}
```

**Figure 4:** A code fragment from the class HTMLServerImageHandler

org.eclipse.birt.report.engine.api

**Interface IHTMLImageHandler**

**All Known Implementing Classes:**

HTMLCompleteImageHandler, HTMLImageHandler, HTMLServerImageHandler

```
public interface IHTMLImageHandler
```

Defines the image handler interface for use in HTML format

**Figure 5:** API Specification of the interface IHTMLImageHandler

## 3.5. Class-Level Solution Approach

### *3.5.1. Solution Representation*

To represent a candidate solution (individual), we used a vector representation. Each dimension of the vector represents a class to recommend for a specific bug report. Thus, a solution is defined as a sequence of classes to recommend for inspection by the developer to locate the bug.

When created, the order of recommended classes corresponds to their positions in the vector. The classes to recommend are dependent since a bug can be located in different classes. In addition, the goal is to recommend a minimum set of classes while maximizing the correctness objective.

For instance, A solution to find possible relevant classes for the bug report of Figure 3, extracted from our experiments, that shows an example of a bug report from BIRT project (ID 101751) is a vector of several ranked classes to be inspected. This bug report describes a defect in the image handling mechanism. The solution consists of a sequence of classes to inspect extracted from the BIRT project.

### *3.5.2. Fitness Functions*

The first lexical similarity function is defined as the sum of the cosine similarity scores between a description of a bug report and the source code of each of the suggested classes divided by the total number of recommended classes. As described in Figures 4 and 5, the description of the bug report example includes several similar words with one of the recommended classes to inspect, the class *HTMLServerImageHandler*. Thus, the cosine similarity function applied between the source code of that class and the description of the bug report will detect such similarities. However, using only this similarity function may not be enough.

The text of a bug report is expressed in a natural language; however, a large part of the content of source code is described in a programming language (except comments). Thus, the similarity score between a bug report description and a source code can be low when there is not enough comments in the code, therefore we added a new similarity function between the bug reports and the APIs description.

The second lexical function is based on the use of cosine similarity between the bug report description and the API specification of each method of a recommended buggy class. Thus, it is defined as the sum of the maximum of the cosine similarity scores between a description of a bug report and each of the methods composing the suggested class divided by the total number of recommended classes. Figure 5 shows the API specification of the *IHTMLImageHandler* interface that includes different terms such as image and handler that also exists in the bug report description of Figure 4. Thus, the lexical similarity between the API specification and the description of a bug report may also help to better identify relevant buggy classes.

In addition to lexical functions, we also add another component to represent the historical measure which composes of three functions. The first function counts the number of times a particular class was fixed. Normally, the more times developers make changes in a class, the more defects could

**Table 2**
List of commits reported on the same file (HTMLServerImageHander) for Birt Project

| Bug# | Commit Description | Date |
|---|---|---|
| Bug 243553 | HTMLServerImageHander returns wrong ImageUrl when using HTMLRenderOption | Aug 2008 |
| Bug 101751 | Enhance IImagehandler interface to allow full customization of birt image handling mechanism | July 2005 |
| Bug 200187 | Deprecated HTMLServerImageHandler methods are not marked as deprecated | July 2007 |

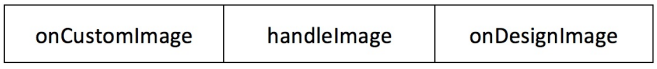


**Figure 6:** A simplified method-level solution representation

be introduced in this particular file in the future. The second history-based function is to measure how recent a particular class has been changed because making some changes today might cause some defects to happen tomorrow. The last function in this category is to evaluate the consistency between what we recommend in terms of classes to what has been touched by the developers in the past to fix a particular defect. Table 2 shows a list of commits reported on the same file in Figure 4, we use the history of bug reports to find the similarity in the description of several commits/bug reports and therefore find the classes or methods that were fixed in the past in order to build our solution for the current reported bug.

## 3.6. Method-Level Solution Approach

### *3.6.1. Solution Representation*

We used a vector of elements to represent a candidate solution, each element represents a method along with its class name in order to recommend for a given bug. Thus, a solution is defined as a sequence of methods to recommend for inspection by the developer to locate the bug. The recommended methods are sorted and ranked in their vector to represent their degree of importance to be reviewed by the developers. The methods to recommend are dependent since a bug can be located in different methods among different classes while maintaining the balance between minimizing the set of methods to recommend and maximizing the value of the correctness objective.

Figure 6 shows a simplified solution generated to find possible relevant methods for the bug report of Figure 3 extracted from the BIRT project (ID 101751).

### *3.6.2. Fitness Functions*

We adapted the fitness functions defined at the class level to calculate the new method level measures. Lexical similarity functions are used to weigh the similarity between the source code of a suggested method from one side or the description of a bug report and the API specification of each method from the other side. As highlighted in Figure 7, the description of the bug report includes a few keywords that already exist in the *handleImage* method. Therefore, the similarity measure between the source code of that method and the description of the bug report is high. History-based fitness functions are used on a particular method by looking at its recently applied changes; along with the consistency between our recommended methods to previously fixed methods of similar bugs in the past.

```
protected String handleImage(IImage image, Object context, String prefix, boolean needMap)
{
        String mapID = null;
        if(needMap)
        {
                mapID = getImageMapID(image);
                if(map.containsKey(mapID))
                {
                        return (String)map.get(mapID);
                }
        }
        String ret = null;
        if (context != null
                        && (context instanceof HTMLRenderContext))
        {
                HTMLRenderContext myContext = (HTMLRenderContext) context;
                String imageURL = myContext.getBaseImageURL();
                String imageDir = myContext.getImageDirectory();
                if(imageURL==null || imageURL.length()==0
                                || imageDir==null || imageDir.length()==0)
                {
                        log.log(Level.SEVERE, "imageURL or ImageDIR is not set!"); //$NON-NLS-1$
                        return null;
                }

                String fileName;
                File file;
```

**Figure 7:** A code fragment from the method handleImage

## 4. Evaluation

In order to evaluate our approach for recommending relevant methods to inspect for bug reports, we conducted a set of experiments based on different versions of 6 open source systems listed in Table 3. Each experiment is repeated 30 times, and the obtained results are subsequently statistically analyzed. Our aim is to compare our hybrid multi-objective approach with a variety of existing approaches:

- Approaches not based on heuristic search such as [7, 16, 17, 16] at the class and methods level.
- Our previous multi-objective work [12], a one step multi-objective formulation based on NSGA-II to identify relevant classes
- A mono-objective formulation.

**Table 3**
Studied Projects

| Project | # bugs | Time | # API | # files in the project (average per version) | # methods in the project (median) | # fixed files/-classes per bug report (median) | # fixed methods per bug report (median) |
|---|---|---|---|---|---|---|---|
| Eclipse UI | 6495 | 10/2001-01/2014 | 1314 | 3454 | 29582 | 2 | 2 |
| Birt | 4178 | 06/2005-12/2013 | 957 | 6841 | 57329 | 1 | 3 |
| JDT | 6274 | 10/2001-01/2014 | 1329 | 8184 | 30240 | 2 | 2 |
| AspectJ | 593 | 03/2002-01/2014 | 54 | 4439 | 21346 | 2 | 2 |
| Tomcat | 1056 | 07/2002-01/2014 | 389 | 1552 | 17970 | 1 | 2 |
| SWT | 4151 | 02/2002-01/2014 | 161 | 2056 | 28355 | 3 | 5 |

In this section, we present our research questions followed by experimental settings and parameters. Finally, we discuss our results for each of those research questions.

## 4.1. Research Questions

In our study, we assess the performance of our approach by finding out whether it could identify the most relevant classes and methods to inspect for bug reports. Our study aims at addressing the following research questions outlined below. We also explain how our experiments are designed to address these questions. The main question to answer is to what extent the proposed approach can propose meaningful bug localization solutions based on the description of a bug report? To this end, we defined the following research questions:

- **RQ1.** (Effectiveness) To what extent can the proposed approach identify relevant methods to localize bugs based on bug reports description?
- **RQ2.** (Comparison to search techniques) How does the proposed hybrid approach performs comparing to our previous multi-objective work [12], a one step multi-objective formulation based on NSGA-II to identify relevant methods, random search, and a mono-objective formulation?
- **RQ3.** (Comparison to state-of-the-art) How does our approach perform compared to existing bugs localization techniques not based on heuristic search?

To answer RQ1, we validate the proposed multi-objective technique on six medium to large-size open-source systems, as detailed in the next section, to evaluate the correctness of the recommended methods to inspect for a bug report. To this end, we used the following evaluation metrics:

- **Precision@k** is the fraction of two components. The numerator component is the number of correct recommended methods in the top k of recommended methods (or files) in the solution. The denominator component is the minimum number of methods (or files), between k and the number of recommended methods/files to inspect in the ranked recommendations list.
- **Recall@k** is the fraction of two components. The numerator component is the number of correct recommended methods in the top k of recommended methods (or files) in the solution. The denominator component is the total number of expected methods (or files) to recommend that contain the bug.
- **Accuracy@k** measures the percentage of bug reports for which at least one correct recommendation was provided in the top k ranked methods (or files).

To answer RQ2, we compared, using the above metrics, the performance of our hybrid multi-objective approach, called HMOA, with our previous multi-objective work [12], a one step multi-objective formulation based on NSGA-II to identify relevant methods (called hNSGA-II), random search and a mono-objective formulation, based on a Genetic Algorithm, aggregating all the objectives into one objective with equal weight. We note that hNSGA-II is using only NSGA-II using the same fitness functions but applied directly at the methods level (minimizing the number of recommended methods and maximizing the relevance of recommended methods). Furthermore, we implemented three mono-objective formulations: 1.with an equal aggregation of both objectives (GA); 2. a mono-objective algorithm with the only objective of lexical similarity (LS); and 3. a mono-objective algorithm with the only objective of history simi-

larity (HS). Random search and the mono-objective formulation are applied for both levels (class and method levels) similar to our approach so we can ensure a fair comparison. hNSGA-II is used to show the value of using a two levels approach.

If Random Search outperforms a guided search method thus, we can conclude that our problem formulation is not adequate. It is important also to determine if our objectives are conflicting and outperform a mono-objective technique. The comparison between a multi-objective technique with mono-objective ones is not straightforward. The first one returns a set of non-dominated solutions while the second one returns a single optimal solution. To this end, we choose the nearest solution to the Knee point [13] (i.e., the vector composed of the best objective values among the population members) as a candidate solution to be compared with the single solution returned by the mono-objective algorithm. We did not invent the knee point method and we used it as recommended by the current literature [58, 59, 60]. The two common ways are the use of the reference point and the knee point. The definition of the reference point (best region of the Pareto front) can be subjective for most real-world problems since it depends on the preferences [58]. The knee point represents the maximum trade-off between the objectives thus it is reasonable to compare it with a mono-objective solution with equal weights of the different objectives aggregated in one fitness function. The fact that we are comparing a mono-objective formulation with equal weights to a knee point (representing the maximum possible trade-off) ensures a fair comparison.

The hNSGA-II algorithm identifies relevant methods for bug reports using the same fitness functions, applied at the methods level, of the proposed approach but only using one step. Thus, the solutions of hNSGA-II are a sequence of methods that are generated and evolved using NSGA-II. The comparison with hNSGA-II is important to confirm the relevance of using a hybrid approach combining both a global and local search algorithms. In comparison with our previous class level work [12], we considered the files of the methods to ensure a fair comparison. This comparison can evaluate the impact of adding the MOSA component on the quality of the results especially in terms of finding the best ranking of the classes/files.

To answer RQ3, we compared our multi-objective approach to different existing techniques not based on heuristic search:

- BugScout [7] identifies relevant classes based on the use of Latent Dirichlet Allocation measure [27].
- BugLocator [17] ranks classes using both textual and structural similarity.
- Learning-to-rank (LRank) [16] technique ranks methods using a machine learning technique to learn from the history of previous bug reports. While this technique can be adopted to both class and methods level like our approach, we configured the implementation to recommend methods as output. Also, we compared our work with two additional baselines. The first one is only based on the use of the lexical measure (LS) to rank classes and the second one is based on the only use of the history measure (HS). These two baselines may justify or not the need of considering complementary information from both the lexical and history similarities in our multi-objective formulation.

We considered the files of the methods to ensure a fair comparison with class level recommendation tools (BugScout, BugLocator and our previous work [12]) and we considered comparisons of the results at the methods level with [16]. Thus, we have two categories of comparison: 1) the class-level approaches are compared to our approach using the evaluation metrics applied at the files (precision, recall and accuracy); and 2) the methods-level approaches are compared to our approach using the evaluation metrics applied at the methods (precision, recall and accuracy).

### 4.2. Software Projects and Experimental Setting

As described in Table 3, we extended a benchmark data sets for six open-source systems [15, 16] from the class to the methods level and we are making this new benchmark available to the community [19]. The data is a spreadsheet where rows are bugs and columns are attributes of the bug. Columns are bug id, bug description, bug summary/commit, bug commit id, bug resolved date. Besides, we added two more columns, the first column contains a list of classes that have been changed in order to resolve the bug, and the second contains a list of methods that have been fixed. Both of those columns (classes list and method list) have been generated using Git (version code system) which provides us the ability to extract a list of files or methods that have been changed in each commit along with date and developer (committer). The whole data has been generated using data from Git or Bugzilla (bug tracking system).

- **Eclipse UI** is the user interface of the Eclipse development framework.
- **Tomcat** implements several Java EE specifications.
- **AspectJ** is an aspect-oriented programming (AOP) extension created for the Java programming language.
- **Birt** provides reporting and business intelligence capabilities.
- **SWT** is a graphical widget toolkit.
- **JDT** provides a set of tool plug-ins for Eclipse.

In Table 3 shows the different statistics of the analyzed systems including the time range of the bug reports, the number of bug reports, the number of classes and methods in a project, the number of APIs, the number of fixed classes per bug report, and the number of fixed methods per bug report. The total number of collected bug reports and associated classes and methods is more than 22,000 bug reports

for the six open source systems. All these projects are using BugZilla tracking system and GIT as a version control system. The ground truth used in our evaluation is the bug location and its respective bug report. To avoid using a fixed version of the source code, we associated a before-fixed version of the source code to each bug report. Therefore, for each bug report in our evaluation, we used the version of the source code just before the fix was committed. Based on the collected data, we created two sets: one for the training data and the other for the test data. The bug reports for each system were sorted chronologically based on the time dimension. The sorted bug reports are then split into 10 folds with equal sizes, where $\text{fold}_1$ contains the most oldest bug reports and the last fold, $\text{fold}_{10}$, contains the recent ones. In addition, the oldest fold is split into 70% training (history of bug reports) and 30% validation. The approach is trained on fold $i + 1$ and tested on $\text{fold}_i$, for all $i$ from 1 to 10. The best recommended solution is then compared with the expected solution of classes and methods that contain the bug. Thus, $\text{fold}_1$ contains the oldest bug reports whereas $\text{fold}_{10}$ contains the latest bug reports. Since the folds are arranged chronologically, this means that the system is always trained on the most recent bug reports with respect to the testing fold.

### 4.3. Parameters Tuning and Statistical Tests

Since metaheuristic algorithms are stochastic optimizers, they can provide different results for the same problem instance from one run to another.

We used the Wilcoxon rank sum test [61] in a pairwise fashion in order to detect significant performance differences between the algorithms (HMOA vs each of the competitors) under comparison based on 30 independent runs. BugScout and BugLocator are both deterministic thus we did not perform 30 independent runs.The Wilcoxon test allows testing the null hypothesis H0 that states that both algorithms medians' values for a particular metric are not statistically different against H1 which states the opposite. The Wilcoxon test does not require that the data sets follow a normal distribution since it operates on values' ranks instead of operating on the values themselves. Since we are comparing more than two different algorithms, we performed several pairwise comparisons based on Wilcoxon test to detect the statistical difference in terms of performance. To compare two algorithms based on a particular metric, we record the obtained metric's values for both algorithms over 30 runs. For deterministic techniques, we considered one value of each metric on each system. After that, we compute the metric's median value for each algorithm. Besides, we executed the Wilcoxon test with a 95% confidence level ($\alpha = 5\%$) on the recorded metric's values using the Wilcoxon MATLAB routine. If the returned p-value is less than 0.05 then we reject H0 and we can state that one algorithm outperforms the other, otherwise we cannot say anything in terms of performance difference between the two algorithms.

The Wilcoxon test allows verifying whether the results are statistically different or not. However, it does not give any idea about the difference in magnitude. To this end, we used the Vargha and Delaney's A statistics which is a non-parametric effect size measure. In our context, given the different performance metrics (such as Precision and Recall), the A statistics measures the probability that running an algorithm B1 (HMOA) yields better performance than running another algorithm B2 (such as GA). If the two algorithms are equivalent, then A = 0.5.

An often-omitted aspect in metaheuristic search is the tuning of algorithm parameters. In fact, the parameter setting significantly influences the performance of a search algorithm on a particular problem. For this reason, for each search algorithm and each system, we performed a set of experiments using several population sizes: 10, 20, 30, 40 and 50. The stopping criterion was set to 100,000 fitness evaluations for all search algorithms to ensure fairness of comparison. We used a high number of evaluations as a stopping criterion since our approach requires multiple objectives. Each algorithm was executed 30 times with each configuration and then the comparison between the configurations was performed based on different metrics described previously using the Wilcoxon test. The other parameters values were fixed by trial and error and are as follows: (1) crossover probability = 0.4; mutation probability = 0.3 where the probability of gene modification is 0.1. In fact, we decided to reduce the diversity of the generated solutions at each iteration since a local search exploration will be executed as well as a second step.

MOSA was performed with a starting temperature of 0.0002 and an alpha value of 0.99995. The starting temperature and alpha values were chosen because they yielded the best results in empirical preliminary tests. All probability distributions used by the search process (e.g., to determine the type of mutation to execute or code fragments to select) were such that each discrete possibility had an equal chance of being selected.

### 4.4. Results

#### *4.4.1. Results for RQ1*

The results of Table 4 and Figures 8-13 confirm the effectiveness of our hybrid multi-objective approach (HMOA) to identify the most relevant classes and methods for bug reports that include the bugs on the 6 open source systems. Table 4 shows the average precision@k results of our HMOA technique on the different six systems, with k ranging from 5 to 20. For example, most of the recommended methods to inspect in the top 5 (k=5) are relevant with a precision of 83%. The lowest precision is around 71% for k=20 which still could be considered acceptable due to the low granularity/abstraction level (methods). In terms of recall, Table 4 confirms that the majority of the expected methods to recommend are located in the top 20 (k=20) with an average recall score of 83%. An average of more than 73% of methods recommended in the top 5 covers the expected buggy methods. The average accuracy@k results on the different six systems are described in Table 4 showing that an average of 67%, 74%, 86%, and 91% are achieved for k = 5, 10, 15, and 20 respectively.

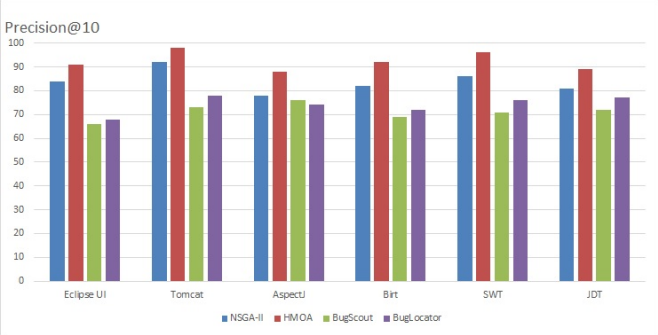


**Figure 8:** Median Precision@10 at the class level on the different systems for 30 independent runs.

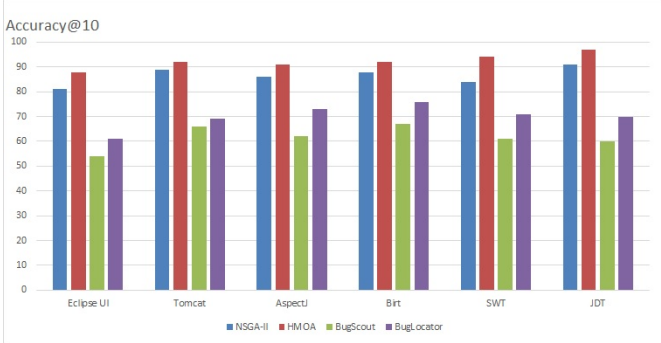


**Figure 10:** Median Accuracy@10 at the class level on the different systems for 30 independent runs.

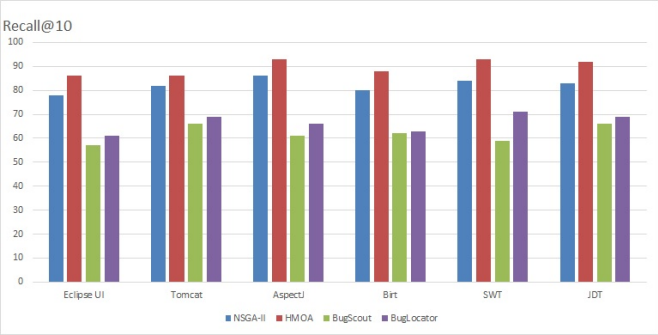


**Figure 9:** Median Recall@10 at the class level on the different systems for 30 independent runs.

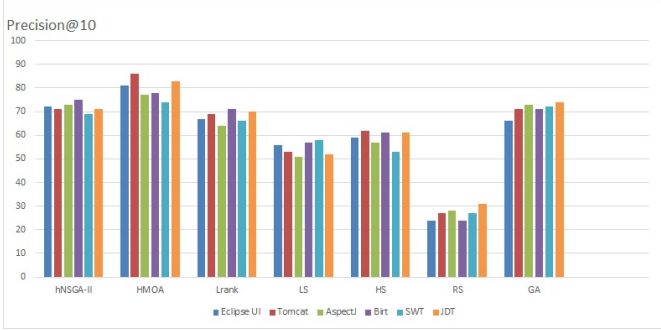


**Figure 11:** Median Precision@10 at the methods level on the different systems for 30 independent runs.

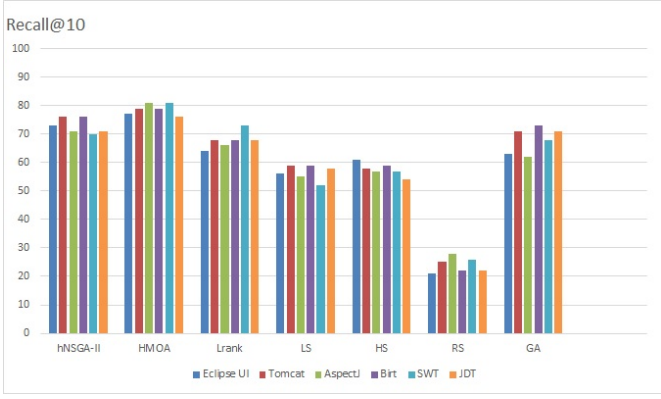


**Figure 12:** Median Recall@10 at the methods level on the different systems for 30 independent runs.

Figures 11-13 summarize the results of the precision@10, recall@10 and accuracy@10 for each of the studied systems. The obtained results clearly show that most of the buggy methods were recommended correctly by our hybrid multi-objective approach in the top 10 with a minimum precision of 82% for AspectJ, a minimum recall of 84% for Eclipse and a minimum accuracy of 81% for Eclipse as well. Thus, we noticed that our technique does not have a bias towards the evaluated system. As described in Figures 11-13, in all systems, we had almost similar average scores of precision, recall, and accuracy. All these results based on the different measures were statistically significant on 30 independent runs using the Wilcoxon test with a 95% confidence level ($\alpha < 5\%$) as detailed in Table 6.

To answer RQ1, the obtained results on the six open source systems using the different evaluation metrics of precision, recall, and accuracy clearly validate the hypotheses that our hybrid multi-objective approach can recommend efficiently relevant buggy methods to inspect for each bug report.

### *4.4.2. Results for RQ2*

Concerning RQ2, we have two categories of comparison. The first category is dedicated to the comparison of HMOA with other method level approaches (LS, LRank, HS, GA, RS, hNSGA-II) to our approach. The second category is related to the comparison of HMOA with our previous multi-objective work [12] for bugs localization at the class level. Thus, the comparison in the second category is performed at the class level (similar to RQ3).

Tables 4-5 and Figures 8-10 confirm that HMOA is better, in average, than random search, the one-step multi-objective methods level formulation (hNSGA-II), and the three mono-objective formulations (LS, HS and GA) based on the three metrics of precision, recall and accuracy on all the 6 systems.

**Table 4**
Median Precision@k, Recall@k and Accuracy@k on 30 independent runs at the methods level.

| K | Precision @ K | | | | | | |
|---|---|---|---|---|---|---|---|
| | hNSGA-II | **HMOA** | LR | LS | HS | RS | GA |
| 5 | 76 | **83** | 72 | 62 | 66 | 32 | 69 |
| 10 | 71 | **79** | 68 | 54 | 58 | 26 | 71 |
| 15 | 68 | **76** | 61 | 51 | 54 | 28 | 63 |
| 20 | 64 | **71** | 52 | 44 | 49 | 21 | 54 |
| K | Recall @ K | | | | | | |
| | hNSGA-II | **HMOA** | LR | LS | HS | RS | GA |
| | 69 | **73** | 61 | 49 | 51 | 21 | 58 |
| | 72 | **78** | 67 | 54 | 56 | 24 | 63 |
| | 75 | **81** | 69 | 59 | 62 | 27 | 71 |
| | 79 | **83** | 72 | 63 | 66 | 21 | 74 |
| K | Accuracy @ K | | | | | | |
| | hNSGA-II | **HMOA** | LR | LS | HS | RS | GA |
| 5 | 64 | **67** | 58 | 39 | 34 | 23 | 51 |
| 10 | 69 | **74** | 64 | 52 | 48 | 27 | 57 |
| 15 | 81 | **86** | 77 | 61 | 57 | 29 | 63 |
| 20 | 86 | **91** | 83 | 68 | 66 | 33 | 72 |

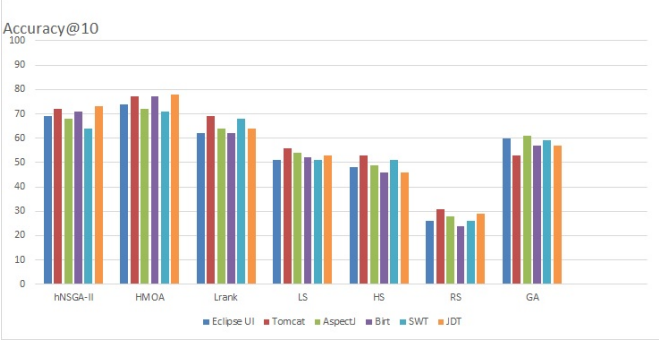


**Figure 13:** Median Accuracy@10 at the methods level on the different systems for 30 independent runs.

The average accuracy, precision, and recall values of random search (RS) on the six systems are lower than 32% as described in Table 4. This can be explained by the huge search space to explore to identify the best order of methods to inspect for bugs localization. The performance of the three mono-objective algorithms was much better than random search but lower than the multi-objective formulations. The aggregation of both objectives into one objective generates better results on all the six systems than the two other algorithms considering each objective separately. Thus, an interesting observation is the clear complementary between the history-based similarity function and the lexical-based measure. In fact, we found that the buggy methods that are not detected by one of the two algorithms were identified by the other algorithm. The average precision, recall, and accuracy of each of the two algorithms (LH and HS) was between 61% and 73% but the aggregation of both objectives into one in our multi-objective formulations improve a lot the obtained results. In addition, since the three multi-objective formulations (NSGA-II, MOHA, and hNSGA-II) outperform the mono-objective GA then it is clear that the two objectives of correctness/relevance and the number of recommended methods are conflicting.

Table 5 confirms also the outperformance of our hybrid multi-objective algorithm comparing to the remaining multi-objective formulations (hNSGA-II and NSGA-II). It is clear that HMOA results are better than hNSGA-II in terms of precision, recall and accuracy. This may confirm that the use of MOSA as a local search to identify methods helped for a better exploration of the large space of possible method comparing to the one-step NSGA-II approach. Furthermore, the results of Figures 8-10 show that both HMOA have better precision, recall and accuracy, on average, than previous work [12]. Thus, it is also clear that adaptation of the methods level fitness functions is more adequate than our previous

**Table 5**
Median Precision@k, Recall@k and Accuracy@k on 30 independent runs at the class/files level.

| K | Precision @ K | | | |
|---|---|---|---|---|
| | NSGA-II | **HMOA** | Bug Scout | Bug Locator |
| 5 | 89 | **100** | 76 | 78 |
| 10 | 82 | **92** | 71 | 74 |
| 15 | 74 | **84** | 63 | 69 |
| 20 | 68 | **81** | 48 | 51 |
| **K** | **Recall @ K** | | | |
| | NSGA-II | **HMOA** | Bug Scout | Bug Locator |
| | 72 | **84** | 59 | 62 |
| | 81 | **86** | 64 | 67 |
| | 87 | **89** | 69 | 72 |
| | 94 | **100** | 74 | 80 |
| **K** | **Accuracy @ K** | | | |
| | NSGA-II | **HMOA** | Bug Scout | Bug Locator |
| 5 | 68 | **83** | 41 | 44 |
| 10 | 86 | **86** | 62 | 69 |
| 15 | 94 | **97** | 74 | 78 |
| 20 | 97 | **100** | 79 | 82 |

work to localize bugs and their impact on the ranking of the classes to be explored by the developers in a positive way.

All these results were statistically significant on 30 independent runs using the Wilcoxon test with a 95% confidence level ($\alpha < 5\%$) as described in Table 6. We have also found the following results of the Vargha Delaney $A_{12}$ statistic : a) On large and medium scale systems (Birt, JDT, Eclipse UI, and AspectJ ) HMOA is better than all the other algorithms based on all the performance metrics with an A effect size higher than 0.89; b) On small scale systems (Tomcat, SWT), HMOA is better than all the other algorithms with an A effect size higher than 0.91.

We conclude that there is empirical evidence that our hybrid multi-objective formulation surpasses the performance of random search and other search-based approaches thus our formulation is adequate (this answers RQ2).

### *4.4.3. Results for RQ3*

Since it is not sufficient to compare our approach with only search-based algorithms, we compared the performance of NSGA-II with three different bug localization techniques not based on heuristic search [7, 16, 17]. Similar to the comparison with NSGA-II, we used class-level comparison measures for [7, 17] and method-level comparison for [16]. Tables 4 and 5, and Figures 8-13 present the precision@k, recall@k and accuracy@k results for the 3 implemented methods, with k ranging from 5 to 20. HMOA achieves better results, on average, than the other three methods on all six projects. For example, our approach achieved, on average, Precision@k of 92%, 87%, 79% and 76% are achieved for k= 5, 10, 15 and 20 respectively as described in Table 5. In comparison, BugLocator achieved an average Precision@k of 68%. BugScout and Lrank achieved an average Precision@k of 66% and 72%, respectively. Similar observations are also valid for the recall@k and accuracy@k.

Based on the results of Figures 11-13 Birt and Tomcat are two projects where Lrank performs close to the HMOA approach. For many bug reports in Birt, most of the buggy methods are those that have been frequently fixed in previous bug reports which explain the relatively high performance obtained by Lrank and HMOA. Since the bug fixing information is exploited by both the NSGA-II approach and Lrank, it is expected that they obtain the best performance results.

To answer RQ3, the obtained results on the six open source systems using the different evaluation metrics of precision, recall and accuracy clearly validate the hypotheses that our hybrid multi-objective approach outperforms several bugs localization techniques not based on heuristic search both at the method and class levels.

**Table 6**
The Wilcoxon rank sum test results in a pairwise fashion (HMOA vs each of the competitors) to detect significant performance differences between the algorithms under comparison using the Precision, Recall and Accuracy measures.

| **Precision** | **hNSGA-II** | **NSGA-II** | **BugScout** | **BugLocator** | **Lrank** | **LS** | **HS** | **RS** | **GA** |
|---|---|---|---|---|---|---|---|---|---|
| Eclipse UI | 0.012 | 0.024 | 0.014 | 0.021 | 0.032 | 0.023 | 0.001 | 0.003 | 0.027 |
| Tomcat | 0.038 | 0.013 | 0.017 | 0.011 | 0.017 | 0.017 | 0.013 | 0.012 | 0.014 |
| AspectJ | 0.022 | 0.024 | 0.021 | 0.017 | 0.037 | 0.021 | 0.004 | 0.017 | 0.011 |
| Birt | 0.016 | 0.047 | 0.018 | 0.003 | 0.032 | 0.031 | 0.012 | 0.031 | 0.023 |
| SWT | 0.038 | 0.014 | 0.022 | 0.014 | 0.024 | 0.011 | 0.024 | 0.004 | 0.014 |
| JDT | 0.021 | 0.035 | 0.017 | 0.019 | 0.017 | 0.023 | 0.012 | 0.014 | 0.027 |
| **Recall** | **hNSGA-II** | **NSGA-II** | **BugScout** | **BugLocator** | **Lrank** | **LS** | **HS** | **RS** | **GA** |
| Eclipse UI | 0.023 | 0.020 | 0.027 | 0.023 | 0.027 | 0.026 | 0.017 | 0.002 | 0.031 |
| Tomcat | 0.031 | 0.017 | 0.004 | 0.007 | 0.032 | 0.011 | 0.031 | 0.012 | 0.023 |
| AspectJ | 0.014 | 0.019 | 0.016 | 0.016 | 0.018 | 0.007 | 0.014 | 0.006 | 0.014 |
| Birt | 0.022 | 0.014 | 0.011 | 0.012 | 0.019 | 0.024 | 0.022 | 0.011 | 0.017 |
| SWT | 0.031 | 0.023 | 0.016 | 0.032 | 0.031 | 0.016 | 0.016 | 0.013 | 0.023 |
| JDT | 0.023 | 0.011 | 0.021 | 0.037 | 0.043 | 0.018 | 0.027 | 0.014 | 0.019 |
| **Accuracy** | **hNSGA-II** | **NSGA-II** | **BugScout** | **BugLocator** | **Lrank** | **LS** | **HS** | **RS** | **GA** |
| Eclipse UI | 0.026 | 0.032 | 0.026 | 0.018 | 0.034 | 0.007 | 0.013 | 0.024 | 0.011 |
| Tomcat | 0.028 | 0.017 | 0.017 | 0.022 | 0.021 | 0.016 | 0.017 | 0.008 | 0.023 |
| AspectJ | 0.031 | 0.024 | 0.032 | 0.016 | 0.038 | 0.023 | 0.022 | 0.013 | 0.017 |
| Birt | 0.017 | 0.019 | 0.021 | 0.024 | 0.027 | 0.009 | 0.031 | 0.011 | 0.032 |
| SWT | 0.024 | 0.027 | 0.019 | 0.019 | 0.021 | 0.017 | 0.024 | 0.021 | 0.037 |
| JDT | 0.006 | 0.021 | 0.024 | 0.027 | 0.013 | 0.023 | 0.011 | 0.017 | 0.021 |

## 5. Discussion

We executed our hybrid multi-objective algorithm on a desktop computer with CPU Intel(R) Core(TM) i7 3.2 GHz and 20G RAM. Figure 14 presents the average execution time of our approach on 30 independent runs for the different six systems. This average execution time is to parse all bug reports for single system and generate the recommended solutions. We have also compared the HMOA execution time to our previous work based on NSGA-II to evaluate the cost of adding the new MOSA component to localize bugs at the method level. The average execution time on the different systems was around 23 minutes. The highest execution time was observed on the Eclipse system with 28 minutes and the lowest one was around 19 minutes for AspectJ. We believe that the execution is reasonable since bug localization is not a real-time problem. We also found that the execution time depends on the number of files to parse and the history of bug reports. Furthermore, the cost of adding the MOSA local search is low with an average of 6 minutes comparing to our previous work based on NSGA-II at the class level. Furthermore, we compared the execution time between our approach and hNSGA-II which shows that the local search based on MOSA is actually faster than applying NSGA-II for the methods-level search (an average of around 3 mins per system). In fact, the hNSGA-II formulation is executed at the methods level which is a much larger search space than the use of local search on a smaller search space of classes identified after a number of iterations of NSGA-II at the class level.

To evaluate the impact of increasing the size of the data used (history of previous bug reports and changes), we executed a scenario on the JDT project in which we increased the size of the dataset incrementally fold by fold until we include all the 9 folds in the dataset. It is clear from Figure 15 that for all the three metrics of Precision@k, Recall@k and Accuracy@k that increasing the size of the previous bug reports does not improve all the three metrics. This can be explained by the fact that recent bug reports and history of changes are the most important part of the data. The obtained results confirm also that our hybrid multi-objective approach did not require a large set of data to generate good

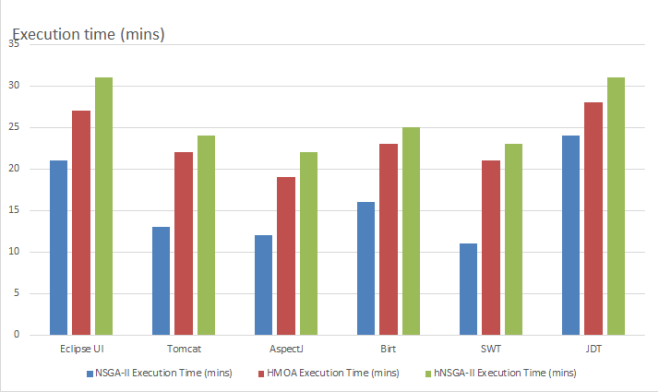


**Figure 14:** Average execution time (in minutes) of NSGA-II, hNSGA-II and HMOA, on the different systems for 30 independent runs on the different systems

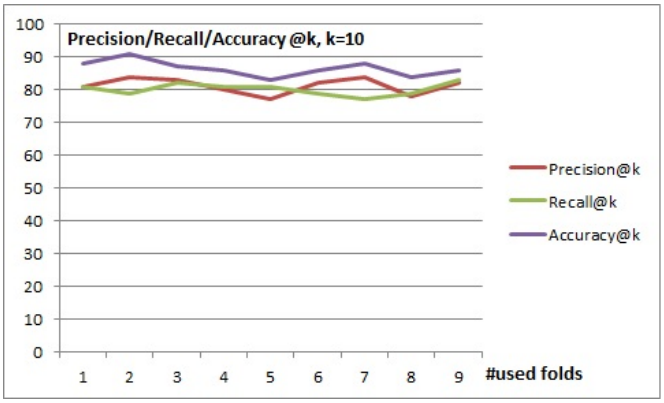


**Figure 15:** Impact of the data training size (folds) on the three evaluation metrics based on the JDT project for the HMOA algorithm.

results in terms of finding possible buggy methods for bug reports. One interesting observation from the recall results is that this measure did not decrease when more bugs reports are added to the datasets. It could be explained by the fact that the the history-based part of the fitness function is only part of the objective, thus the noise introduced by older bug reports is not very impactful. Futhermore, our approach is not based on machine learning to learn from all the dataset. It is based on metaheuristics search guided by fitness functions thus the results are likely less susceptible to noise.

## 6. Threats to Validity

We explore, in this section, the factors that can bias our empirical study. These factors can be classified into three categories: construct internal and external validity. Construct validity concerns the relation between the theory and the observation. Internal validity concerns possible bias with the results obtained by our proposal. Finally, external validity is related to the generalization of observed results outside the sample instances used in the experiment.

In our experiments, construct validity threats are related to the absence of similar work that uses search-based techniques for bug localization expect our previous work. For that reason, we compared our proposal with different mono-objective formulations to check the need for a multi-objective approach and a one-level multi-objective formulation to evaluate the performance of our hybrid approach. A construct threat can also be related to the corpus of manually localized bugs for every bug report. A limitation related to our experiments is the difficulty to set the thresholds for some of the parameters of Bug Locator. In fact, we used the default thresholds used by the authors that can have an impact on the quality of the generated results. Another possible threat is related to the use of the knee point to compare the mono-objective search to our approach.

We take into consideration the internal threats to validity in the use of stochastic algorithms since our experimental study is performed based on 30 independent simulation runs for each problem instance, and the obtained results are statistically analyzed by using the statistical test with a 95% confidence level ($\alpha = 5\%$). The parameter tuning of the different optimization algorithms used in our experiments creates another internal threat that we need to evaluate in our future work by additional experiments to evaluate the impact of the parameters on the quality of the results.

External validity refers to the generalization of our findings. In this study, we performed our experiments on six different widely-used open-source systems belonging to the different domains and with different sizes. However, we cannot assert that our results can be generalized to other applications, other programming languages, and other practitioners.

## 7. Conclusion and future work

We propose, in this paper, an automated approach to localize and rank potential relevant methods for bug reports as an extension of our previous work limited to class level recommendations. Our approach finds a trade-off between minimizing the number of recommended methods and maximizing the correctness of the proposed solution using a hybrid multi-objective algorithm. The correctness of the recommended methods is estimated based on the use of the history of changes and bug-fixing, and the lexical similarity between the bug report description and the API documentation. Our approach uses the main steps, the first step finds the best set of classes satisfying the two conflicting criteria of relevance and number of classes to recommend using a global search based on NSGA-II. The second step is to locate the most appropriate methods to inspect, using a local multi-objective search based on Simulated Annealing (MOSA) from the list of classes identified in the first step.

The paper presents the results of an empirical study with an implementation of our hybrid multi-objective approach based on 22,000 bug reports. The obtained results provide evidence to support the claim that our proposal is more efficient, on average, than state of the art techniques on 6 open source systems. As part of our future work, we plan to ex-

tend our work to consider the severity of the bugs when identifying relevant files. Furthermore, we are planning to address the problem of finding the qualified developers to fix the bugs based on the outputs of our bug localization approach. Finally, we will extend our work to handle multiple bugs reports at the same time and consider the dependency between them when recommending code fragments to the developers.

## References


[1] B. Bruegge and A. H. Dutoit, *Object-Oriented Software Engineering Using UML, Patterns and Java-(Required)*. Prentice Hall, 2004, vol. 2004.

[2] T. Zimmermann, R. Premraj, N. Bettenburg, S. Just, A. Schroter, and C. Weiss, "What makes a good bug report?" *IEEE Transactions on Software Engineering*, vol. 36, no. 5, pp. 618–643, 2010.

[3] N. Bettenburg, R. Premraj, T. Zimmermann, and S. Kim, "Duplicate bug reports considered harmful… really?" in *Software maintenance, 2008. ICSM 2008. IEEE international conference on*. IEEE, 2008, pp. 337–345.

[4] M. Fischer, M. Pinzger, and H. Gall, "Analyzing and relating bug report data for feature tracking," in *WCRE*, vol. 3, 2003, p. 90.

[5] S. Wang and D. Lo, "Version history, similar report, and structure: Putting them together for improved bug localization," in *Proceedings of the 22Nd International Conference on Program Comprehension*, ser. ICPC 2014. New York, NY, USA: ACM, 2014, pp. 53–63. [Online]. Available: http://doi.acm.org/10.1145/2597008.2597148

[6] C. Sun, D. Lo, X. Wang, J. Jiang, and S.-C. Khoo, "A discriminative model approach for accurate duplicate bug report retrieval," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. ACM, 2010, pp. 45–54.

[7] A. T. Nguyen, T. T. Nguyen, J. Al-Kofahi, H. V. Nguyen, and T. N. Nguyen, "A topic-based approach for narrowing the search space of buggy files from a bug report," in *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 2011, pp. 263–272.

[8] B. Ashok, J. Joy, H. Liang, S. K. Rajamani, G. Srinivasa, and V. Vangala, "Debugadvisor: a recommender system for debugging," in *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. ACM, 2009, pp. 373–382.

[9] H. Aman, S. Amasaki, T. Yokogawa, and M. Kawahara, "Empirical study of abnormality in local variables and its application to fault-prone java method analysis," *Journal of Software: Evolution and Process*, p. e2220, 2019.

[10] X. Ye, H. Shen, X. Ma, R. Bunescu, and C. Liu, "From word embeddings to document similarities for improved information retrieval in software engineering," in *Proceedings of the 38th international conference on software engineering*. ACM, 2016, pp. 404–415.

[11] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan, "Bug isolation via remote program sampling," in *ACM Sigplan Notices*, vol. 38, no. 5. ACM, 2003, pp. 141–154.

[12] R. Almhana, W. Mkaouer, M. Kessentini, and A. Ouni, "Recommending relevant classes for bug reports using multi-objective search," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2016, pp. 286–295.

[13] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: Nsga-ii," *IEEE transactions on evolutionary computation*, vol. 6, no. 2, pp. 182–197, 2002.

[14] P. Czyżżak and A. Jaszkiewicz, "Pareto simulated annealing—a metaheuristic technique for multiple-objective combinatorial optimization," *Journal of Multi-Criteria Decision Analysis*, vol. 7, no. 1, pp. 34–47, 1998.

[15] X. Ye, R. Bunescu, and C. Liu, "Learning to rank relevant files for bug reports using domain knowledge," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 689–699.

[16] ——, "Mapping bug reports to relevant files: A ranking model, a fine-grained benchmark, and feature evaluation," *IEEE Transactions on Software Engineering*, vol. 42, no. 4, pp. 379–402, 2016.

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

[18] M. Harman, S. A. Mansouri, and Y. Zhang, "Search-based software engineering: Trends, techniques and applications," *ACM Computing Surveys (CSUR)*, vol. 45, no. 1, p. 11, 2012.

[19] "Methods level data for bugs localization," http://www-personal.umd.umich.edu/~marouane/tsedata.zip, accessed: 2018-10-01.

[20] M. Kessentini and A. Ouni, "Detecting android smells using multi-objective genetic programming," in *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. IEEE, 2017, pp. 122–132.

[21] A. Ghannem, G. El Boussaidi, and M. Kessentini, "On the use of design defect examples to detect model refactoring opportunities," *Software Quality Journal*, vol. 24, no. 4, pp. 947–965, 2016.

[22] B. Amal, M. Kessentini, S. Bechikh, J. Dea, and L. B. Said, "On the use of machine learning and search-based software engineering for ill-defined fitness function: a case study on software refactoring," in *International Symposium on Search Based Software Engineering*. Springer, Cham, 2014, pp. 31–45.

[23] M. Kessentini, A. Ouni, P. Langer, M. Wimmer, and S. Bechikh, "Search-based metamodel matching with structural and syntactic measures," *Journal of Systems and Software*, vol. 97, pp. 1–14, 2014.

[24] A. Ghannem, G. El Boussaidi, and M. Kessentini, "Model refactoring using examples: a search-based approach," *Journal of Software: Evolution and Process*, vol. 26, no. 7, pp. 692–713, 2014.

[25] A. Ghannem, M. Kessentini, and G. El Boussaidi, "Detecting model refactoring opportunities using heuristic search," in *Proceedings of the 2011 Conference of the Center for Advanced Studies on Collaborative Research*, 2011, pp. 175–187.

[26] S. T. Dumais, "Latent semantic analysis," *Annual review of information science and technology*, vol. 38, no. 1, pp. 188–230, 2004.

[27] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent dirichlet allocation," *Journal of machine Learning research*, vol. 3, no. Jan, pp. 993–1022, 2003.

[28] G. Salton, A. Wong, and C.-S. Yang, "A vector space model for automatic indexing," *Communications of the ACM*, vol. 18, no. 11, pp. 613–620, 1975.

[29] C.-P. Wong, Y. Xiong, H. Zhang, D. Hao, L. Zhang, and H. Mei, "Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis," in *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*. IEEE, 2014, pp. 181–190.

[30] R. K. Saha, M. Lease, S. Khurshid, and D. E. Perry, "Improving bug localization using structured information retrieval," in *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*. IEEE, 2013, pp. 345–355.

[31] R. K. Saha, J. Lawall, S. Khurshid, and D. E. Perry, "On the effectiveness of information retrieval based bug localization for c programs," in *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*. IEEE, 2014, pp. 161–170.

[32] S. Rao and A. Kak, "Retrieval from software libraries for bug localization: a comparative study of generic and composite text models," in *Proceedings of the 8th Working Conference on Mining Software Repositories*. ACM, 2011, pp. 43–52.

[33] S. Wang and D. Lo, "Version history, similar report, and structure: Putting them together for improved bug localization," in *Proceedings of the 22nd International Conference on Program Comprehension*. ACM, 2014, pp. 53–63.

[34] A. Lamkanfi, S. Demeyer, E. Giger, and B. Goethals, "Predicting the severity of a reported bug," in *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*. IEEE, 2010, pp. 1–10.

[35] Y. Tian, D. Lo, and C. Sun, "Drone: Predicting priority of reported bugs by multi-factor analysis," in *2013 IEEE International Conference on Software Maintenance*. IEEE, 2013, pp. 200–209.

[36] K. C. Youm, J. Ahn, and E. Lee, "Improved bug localization based on code change histories and bug reports," *Information and Software Technology*, vol. 82, pp. 177–192, 2017.

[37] S. K. Lukins, N. A. Kraft, and L. H. Etzkorn, "Bug localization using latent dirichlet allocation," *Information and Software Technology*, vol. 52, no. 9, pp. 972–990, 2010.

[38] Q. Huang, D. Lo, X. Xia, Q. Wang, and S. Li, "Which packages would be affected by this bug report?" in *2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2017, pp. 124–135.

[39] M. Wen, R. Wu, and S.-C. Cheung, "Locus: Locating bugs from software changes," in *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2016, pp. 262–273.

[40] C. Tantithamthavorn, S. L. Abebe, A. E. Hassan, A. Ihara, and K. Matsumoto, "The impact of ir-based classifier configuration on the performance and the effort of method-level bug localization," *Information and Software Technology*, vol. 102, pp. 160–174, 2018.

[41] X. Ye, R. Bunescu, and C. Liu, "Mapping bug reports to relevant files: A ranking model, a fine-grained benchmark, and feature evaluation," *IEEE Transactions on Software Engineering*, vol. 42, no. 4, pp. 379–402, 2015.

[42] P. Loyola, K. Gajananan, and F. Satoh, "Bug localization by learning to rank and represent bug inducing changes," in *Proceedings of the 27th ACM International Conference on Information and Knowledge Management*. ACM, 2018, pp. 657–665.

[43] A. N. Lam, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "Bug localization with combination of deep learning and information retrieval," in *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*. IEEE, 2017, pp. 218–229.

[44] Y. Xiao, J. Keung, K. E. Bennin, and Q. Mi, "Machine translation-based bug localization technique for bridging lexical gap," *Information and Software Technology*, vol. 99, pp. 58–61, 2018.

[45] M. Harman and B. F. Jones, "Search-based software engineering," *Information and software Technology*, vol. 43, no. 14, pp. 833–839, 2001.

[46] M. Harman, Y. Jia, and Y. Zhang, "Achievements, open problems and challenges for search based software testing," in *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2015, pp. 1–12.

[47] A. Núñez, M. G. Merayo, R. M. Hierons, and M. Núñez, "Using genetic algorithms to generate test sequences for complex timed systems," *Soft Computing*, vol. 17, no. 2, pp. 301–315, 2013.

[48] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. Le Traon, and M. Harman, "Mutation testing advances: an analysis and survey," in *Advances in Computers*. Elsevier, 2019, vol. 112, pp. 275–378.

[49] C. Henard, M. Papadakis, and Y. Le Traon, "Mutation-based generation of software product line test configurations," in *International Symposium on Search Based Software Engineering*. Springer, 2014, pp. 92–106.

[50] J. Shelburg, M. Kessentini, and D. R. Tauritz, "Regression testing for model transformations: A multi-objective approach," in *International Symposium on Search Based Software Engineering*. Springer, 2013, pp. 209–223.

[51] D. Dreyton, A. A. Araújo, A. Dantas, Á. Freitas, and J. Souza, "Search-based bug report prioritization for kate editor bugs repository," in *International Symposium on Search Based Software Engineering*. Springer, 2015, pp. 295–300.

[52] D. Dreyton, A. A. Araújo, A. Dantas, R. Saraiva, and J. Souza, "A multi-objective approach to prioritize and recommend bugs in open source repositories," in *International Symposium on Search Based Software Engineering*. Springer, 2016, pp. 143–158.

[53] E.-G. Talbi, *Metaheuristics: from design to implementation*. John Wiley & Sons, 2009, vol. 74.

[54] W. B. Mock, "Pareto optimality," *Encyclopedia of Global Justice*, pp. 808–809, 2011.

[55] E. Ulungu, J. Teghem, P. Fortemps, and D. Tuyttens, "Mosa method: a tool for solving multiobjective combinatorial optimization problems," *Journal of multicriteria decision analysis*, vol. 8, no. 4, p. 221, 1999.

[56] P.-N. Tan *et al.*, *Introduction to data mining*. Pearson Education India, 2006.

[57] E. Enslen, E. Hill, L. Pollock, and K. Vijay-Shanker, "Mining source code to automatically split identifiers for software analysis," in *Mining Software Repositories, 2009. MSR'09. 6th IEEE International Working Conference on*. IEEE, 2009, pp. 71–80.

[58] A. A. Keller, *Multi-Objective Optimization in Theory and Practice II: Metaheuristic Algorithms*. Bentham Science Publishers, 2019.

[59] M. T. Emmerich and A. H. Deutz, "A tutorial on multiobjective optimization: fundamentals and evolutionary methods," *Natural computing*, vol. 17, no. 3, pp. 585–609, 2018.

[60] K. Deb and S. Gupta, "Understanding knee points in bicriteria problems and their implications as preferred solution principles," *Engineering optimization*, vol. 43, no. 11, pp. 1175–1204, 2011.

[61] F. Wilcoxon, S. Katti, and R. A. Wilcox, "Critical values and probability levels for the wilcoxon rank sum test and the wilcoxon signed rank test," *Selected tables in mathematical statistics*, vol. 1, pp. 171–259, 1970.