An Automated Approach for Selecting Bugs in Component-Based Software Projects

G. Lima¹, and G. Elias¹
¹Informatics Center, Federal University of Paraíba, João Pessoa, PB, Brazil

Abstract - Bug-fixing tasks represent a critical obstacle for maintaining software systems, in which project managers are faced with the challenge of manually selecting bugs to be fixed in the next release, usually concerning priority and severity criteria indicated in bug reports. For large complex systems, in which a considerable number of bugs exists, ad-hoc manual approaches are unsuitable, representing a timing consuming, labor-intensive and error prone strategy. In such a context, exploring search-based software engineering techniques, this paper proposes an automated approach for selecting interdependent bugs in component-based software projects, adopting a multi-customer model for choosing the more relevant bugs from the viewpoint of the software producer and its customers, but with total cost limited by the project budget.

Keywords: Bug Selection, Software Components, Search-Based Software Engineering, Genetic Algorithms.

1 Introduction

In software development processes, software bugs represent one of the more critical obstacles for maintaining and evolving software systems. As a complicating factor, the intrinsic complexity related to bug dependencies has the potential to increase maintenance costs up to 70% of the project budget [1]. In such a scenario, due to budget or time-to-market constraints, project managers may be faced with the challenge of manually selecting a lot of bugs to be fixed in the next release of the software product, in general based on priority and severity criteria indicated in bug reports.

Unfortunately, in large and complex software systems, it is common the existence of a considerable number of bug reports. For instance, Avik, Hiew and Murphy [2] reveal the case of the Eclipse Project in which contributing developers reported 3,246 bugs over a four-month period, averaging 29 bug reports per day. Accordingly, in large and complex software systems, ad-hoc manual approaches for selecting bugs are not efficient and effective, representing a timing consuming, labor-intensive and error prone task, mainly when considering bug dependencies, which establish precedence and concurrency relationships between their bug fixing tasks.

Taking into account the large and complex problem space, according Xiao and Afzal [3], there is a need to support efficient and effective bug-assignment policies that can schedule different bug-fixing tasks by considering the available resource constraints and bug requirements. Consequently, in this paper, the hypothesis is that an automated approach for selecting bugs to be fixed has the potential to be more efficient and effective. An automated approach can simultaneously evaluate and balance multiple competing and conflicting factors, such as: (i) bug dependencies; (ii) bug priority and severity from the viewpoint of the software producer and its customers; (iii) cost associated with each bug-fixing task; and (iv) software project budget. By evaluating all of them, an automated approach can find a reasonable good solution that represents a tradeoff between constraints, relevance, cost and budget, providing maximum business value for the software producer and its customers.

In such a direction, Search-Based Software Engineering (SBSE) techniques seem to be a natural choice for investigating such an automated approach for selecting bugs. As coined in [4], SBSE is a recent area that adopts search-based optimization algorithms to address problems in software engineering. SBSE offers a suite of adaptive automated solutions in situations typified by large complex problem spaces with multiple competing and conflicting objectives [5].

Thus, exploring SBSE techniques, this paper proposes an automated approach for selecting bugs in component-based software projects, adopting a multi-customer model in order to select the more relevant bugs from the viewpoint of the software producer and its customers, but with total cost limited by the project budget. The proposal evaluates constraints related to bug dependencies, establishing precedence and concurrency relationships between their respective bug-fixing tasks. As an additional contribution, by assuming that dependencies between software components imply dependencies between their associated bugs, the proposal explores the component-based architecture of the software project as a mean to track bug dependencies.

The remainder of this paper is organized as follows. Section II introduces important concepts and fundaments. Then, Section III presents the automated approach for selecting bugs in component-based software projects, detailing its structure, phases, parameters, mathematical model, representation and fitness function. Thereafter, in Section IV, an experiment is presented based on a real software project, evincing the practical applicability of the proposed approach. In sequence, Section V discusses related work, pointing out the contribution of the proposed approach. In conclusion, Section VI presents final remarks and delineates future work.

2 Concepts and fundaments

Bug fixing is a process adopted by software development teams to keep track and manage bug reports throughout the lifecycle of software bugs, covering from their initial reporting to their final resolution. As a mean to automate and support...
such a process, a Bug Tracking System (BTS) is a software tool that stores bug reports, providing a clear, centralized view of development requests and their status, and making available information about the software team's productivity [6].

A bug report is a document or record explaining the details of a bug found in a software package or component, which is part of the respective software system. In currently available BTSs, bug reports usually include a wide variety of information. Among such information, the proposed approach has special interest in the following types: (i) bug priority and severity; (ii) bug dependencies; (iii) target asset related to the bug; and (iv) bug cost required by the bug-fixing task.

On the one hand, bug priority represents a measure of importance level of the reported bug from the perspective of the interested party. Thus, simple terms or values such as low/medium/high or 1/2/3 can indicate the bug priority. It is important to emphasize that the proposed approach adopts the priority values defined in Bugzilla, ranging from P1 to P5, where P1 is the highest priority and P5 is the lowest priority.

On the other hand, bug severity represents a measure of criticalness level of the reported bug, indicating the impact of the bug from the perspective of the interested party. In general, a descriptive list such as critical/usability/minor can indicate the bug severity. For instance, a system crash would be critical, while a string typo would not. Note that the proposed approach adopts Bugzilla values, including blocker, critical, major, minor and trivial, where blocker is the highest severity and trivial is the lowest severity.

When reporting a bug, it is also important to indicate bug dependencies, which define precedence and concurrency relationships among bugs. For instance, in Bugzilla, a bug that depends on other one can only be set as fixed after fixing the other one. But two or more bugs that do not depend on each other can be fixed in any sequence or even simultaneously.

As a mean to detect bug dependencies automatically, the proposed approach explores the component-based architecture of the software project, assuming that dependencies between software components imply dependencies between their respective bugs. To do that, for each reported bug, the approach requires the identification of the target asset, representing the software package, module or component where the bug occurs.

As another important information, in general, a bug report also includes the bug cost, which represents the financial cost or human effort required for fixing the respective bug. Project managers ought to regard such a bug cost in order to select bugs in a cost-effective manner, considering budget constraints for providing maximum business value for the organization. Note that the proposed approach does not define any method for estimating cost or effort, but an interesting reader can find in [7] an approach for predicting bug fixing effort.

## 3 Proposed approach

The proposed approach deals with the Bug Selection Problem (BSP), a term coined herein, which shares some similarities with the Next Release Problem (NRP) [8]. Both, BSP and NRP, have been proposed to model the decision for balancing the software producer profits and the customer satisfaction, under the constraint of a predefined budget of the software project. On the one hand, NRP focuses on the need to determine the lot of requirements that should be added to a software product as part of its next version in requirements engineering. On the other hand, BSP focuses on the need to determine the lot of bugs that should be fixed in a software product as part of the next version in software maintenance, more precisely during bug fixing processes. Usually, NRP has been investigated in SBSE as NP-hard. Thus, due to representational similarities, BSP should also be studied as NP-hard, and, as such, be solved using SBSE techniques.

Thus, in the BSP context, this paper proposes an automated approach for selecting bugs in component-based software projects. As shown in Fig. 1, the proposed approach adopts a layered architecture, structured in three phases.

### 3.1 Recommending bug dependencies

In order to identify bug dependencies, it is assumed that dependencies between software components imply dependencies between their associated bugs. Thus, the approach explores the component-based architecture of the software project for tracking and identifying bug dependencies. As shown in Fig. 2, this phase has three steps.
The first step Map Bugs to Assets has the purpose of identifying the software assets in which each bug occurs. To do that, this step takes as input: (i) the bug reports available in a BTS; and (ii) the component-based architecture available in a Software Documentation Repository (SDR).

Initially, as the basic element of the proposed approach, this step produces as output the set of candidate bugs reported in the BTS, represented in the proposed approach by the set \( B = \{ b_1, b_2, \ldots, b_m \} \). Besides, as a basic architectural information, this step adopts the software architecture as input for characterizing the component-based software project as a set of software components or assets, represented in the proposed approach by the set \( A = \{ a_1, a_2, \ldots, a_k \} \).

Now, the first step ought to extract the mapping from bugs to assets. As mentioned, each bug report usually indicates the software asset in which the bug occurs. Thus, it is possible from the set of bug reports in a BTS to extract the bugs-to-assets mapping. Here, this mapping is represented by a binary matrix \( BA_{B \times A} = \{ ba_{ij} \mid \exists b_i \in B \land \exists a_j \in A \land ba_{ij} \in \{0, 1\} \} \), in which each term \( ba_{ij} \) is defined by (1), where rows and columns represent bugs and assets, respectively. Note that, as defined, each bug occurs in only one asset, but each asset can be related to zero or more bugs.

\[
ba_{ij} = \begin{cases} 
1 & \text{if bug } b_i \text{ occurs in asset } a_j \\ 
0 & \text{otherwise} 
\end{cases} \tag{1}
\]

As an additional output, based on information in bug reports, this step also extracts as output the bug cost or effort associated with each bug-fixing task, represented in the proposed approach by the set \( V = \{ v_i \mid \exists b_i \in B \land v_i > 0 \} \). As said before, the cost or effort for fixing a bug is an information indicated by developers in bug reports.

The second step, named Identify Asset Dependencies, has the purpose of identifying assets dependencies. As proposed herein, an asset \( a_j \) depends on another asset \( a_k \) if and only if \( a_j \) requires an operation provided by \( a_k \). For extracting asset dependencies, this step takes as input the component-based architecture together with the set of related software assets \( A \). First, the component-based architecture is transformed into a directed graph, in which the vertices are defined by assets and the directed edges are defined by required and provided interfaces, pointing from the calling asset with the required interface to the called asset with the provided interface.

The proposed approach represents assets dependencies by an adjacency matrix, which is a binary square matrix \( AD_{A \times A} = \{ ad_{jk} \mid \exists a_j \in A \land \exists a_k \in A \land ad_{jk} \in \{0, 1\} \} \), in which rows and columns represent assets. The term \( ad_{jk} \) is defined in (2), assuming a value of one when there is an edge from asset \( a_j \) to asset \( a_k \), and zero when there is no edge. Note that, as defined, each asset can depend on zero or more assets.

\[
ad_{jk} = \begin{cases} 
1 & \text{if asset } a_j \text{ depends on asset } a_k \\ 
0 & \text{otherwise} 
\end{cases} \tag{2}
\]

Now, in the third step Identify Bug Dependencies, the bugs-to-assets mapping \( BA_{B \times A} \) together with the assets dependencies \( AD_{A \times A} \) are mathematically handled using boolean matrix operations for identifying bugs dependencies, which is represented by the binary square matrix \( BD_{B \times B} \). Note that a given bug \( b_i \) related to an asset \( a_j \) depends on another bug \( b_j \) related to an asset \( a_k \) if and only if \( a_j \) depends on \( a_k \).

For extracting bugs dependencies \( BD_{B \times B} \), as defined in (3), the third step captures all the direct and indirect dependencies a bug possesses with other bugs, which is similar to the concept known as visibility-dependence, proposed in [9] for evaluating complex software architectures.

\[
BD_{B \times B} = (BA_{B \times A}^T A^T AD_{A \times A}^T A^T BA_{B \times A}) \tag{3}
\]

To calculate all bugs dependencies \( BD_{B \times B} \), the proposed approach explores the concept of transitive closure of the directed graph defined by the component-based architecture, which defines another directed graph with the same set of vertices, but with an edge from asset \( a_j \) to asset \( a_k \) if and only if \( a_k \) is reachable from \( a_j \) in the directed graph defined by the component-based architecture.

To do that, the assets dependencies \( AD_{A \times A} \) is taken as a Design Structure Matrix (DSM) [10]. First, each \( n \)-th power of the reachability matrix [11], represented as \( AD_{A \times A}^n \), is calculated for finding the direct and indirect dependencies each asset possesses with other assets for all successive path lengths. Summing these reachability matrices yields a visibility-dependence matrix \( AD_{A \times A}^V \), which shows all direct and indirect dependencies between assets for all possible path lengths up to the maximum, defined by the size of the set \( A \). Remember that the \( 0 \)-th power of a matrix is the identity matrix.

Now, the boolean multiplication of the direct bugs-to-assets dependencies, represented by the bugs-to-assets mapping \( BA_{B \times A} \), and the resulting visibility-dependence matrix \( AD_{A \times A}^V \) generates all direct and indirect bugs-to-assets dependencies matrix \( BA_{B \times A}^V \), evincing all assets affected by each bug. Lastly, the boolean multiplication of the resulting direct and indirect bugs-to-assets dependencies matrix \( BA_{B \times A}^V \) and the transpose of the bugs-to-assets mapping, represented by \( BA_{B \times A}^T \), yields all direct and indirect dependencies among bugs \( BD_{B \times B} \), revealing all bugs affected by each bug.

Note that, herein, bugs dependencies \( BD_{B \times B} \) is also called basic solutions \( BX_{B \times B} \), that is \( BX_{B \times B} = BD_{B \times B} \). Such a terminology is adopted due to the following rationale. Let \( b_i \) be a bug. Then, all bugs marked with one in row \( i \) of the basic solutions \( BX_{B \times B} \), defined by the row matrix \( [bx_{i,1}, bx_{i,2}, \ldots, bx_{i,m}] \), represents the bugs that must be fixed together with or before bug \( b_i \). Indeed, if bug \( b_i \) is selected to be fixed, then all bugs it depends on must be also selected.

### 3.2 Recommending bug priority and severity

As a mean to estimate bug priority and severity from the producer’s and customers’ point of view, the approach assumes that the producer and each customer provide information about priority and severity for each reported bug. To do that, as shown in Fig. 3, the second phase is structured in three steps.

On the producer side, the first step Identify Priority and Severity for Producer assigns priority and severity values for reported bugs, generating as output the producer priority and producer severity, denoted by \( PP = \{ pp_i \mid \exists b_i \in B \land pp_i \in PV \} \) and \( PS = \{ ps_i \mid \exists b_i \in B \land ps_i \in SV \} \), respectively.
3.3 Recommending bug-fixing tasks

Based on bugs dependencies, costs, priorities and severities, estimated in the first and second phases, the third phase of the proposed approach adopts SBSE concepts and techniques for representing the Bug Selection Problem (BSP) as an optimization problem in order to recommend a set of good enough or even optimal solutions.

In such a direction, the third phase is structured based on the well-known search strategy of genetic algorithms [12], which is inspired by the process of biological evolution and natural selection. Once the genetic representation and fitness function are defined, the genetic algorithm proceeds by initializing a population of candidate solutions, and then improves the population through a generational process, based on repetitive application of classical genetic operators such as selection, crossover, mutation and replacement, which concludes when a termination condition is reached.

Due to space limit, the following discussion in this section has the focus on two aspects of the proposed approach, which are illustrated in Fig. 4 on the left and right frames, respectively: (i) generation of a valid candidate solution, constrained by bugs dependencies; and (ii) evaluation of a valid candidate solution, based on the proposed fitness function and constrained by the software project budget.

3.3.1 Generation of a valid candidate solution

As can be seen on the left frame in Fig. 4, the first step Select Bugs without Dependencies is responsible for randomly selecting bugs to be fixed without considering dependencies among them. In the proposed approach, this initial solution is called an independent solution, represented by the chromosome $IX = \{i_{x_1} | \exists b_1 \in B \land i_{x_1} \in [0,1]\}$, where each gene $i_{x_1}$ is one or zero when bug $b_1$ is selected or not, respectively.

Thereafter, the second step Derive Bugs with Dependencies has the purpose of transforming the independent solution $IX$ in a valid candidate solution constrained by bug dependencies. In the proposed approach, the valid solution is called a dependent solution, represented by the chromosome $X = \{x_{x_1} | \exists b_1 \in B \land x_{x_1} \in [0,1]\}$, such that each gene $x_{x_1}$ is one or zero when bug $b_1$ is selected or not, respectively. For instance, consider a bug $b_1$ that depends on another bug $b_2$. In such a case, if bug $b_1$ is selected in the independent solution $IX$, then $b_2$ must be also present in the dependent solution $X$.

In order to derive the dependent solution $X$, the proposed approach handles as boolean matrices the set of basic solutions $BX_{B \times B}$ and the independent solution $IX$. As such, as defined in (6), the dependent solution $X$ can be directly derived by the boolean multiplication among $IX$ and $BX_{B \times B}$.

$$X = IX \cdot BX_{B \times B}$$
For concluding the generation of a valid solution, by summing the specific cost $v_i$ for each selected bug $x_i$, the third step Estimating Bug-Fixing Cost calculates the total value for all bug-fixing tasks related to all selected bugs. Here, such a value is called bug-fixing cost, denoted in (7) by the term $VS_X$.

$$VS_X = \sum_{b \in E} v_i \cdot x_i$$  \hspace{1cm} (7)

### 3.3.2 Evaluation of a valid candidate solution

After generating a valid dependent solution, as can be perceived on the right frame in Fig. 4, the upper steps Evaluate Bugs Coverage and Evaluate Bugs Interdependency can be performed concurrently. On the one hand, in the former step, the coverage degree, represented by the term $CD_X$ defined in (8), means a measure that describes the percentage of bugs $x_i$ covered by the dependent solution $X$ in relation to the whole set of candidate bugs $B$. On the other hand, in the latter step, interdependency degree, defined using the term $ID_X$ stated in (9), is a measure that indicates the proportion of paths covered by the dependent solution $X$ in relation to the whole set of paths in the transitive closure of the directed graph defined by direct and indirect dependent bugs in basic solutions $BX_{EB}$.

$$CD_X = \frac{\sum_{b \in E} x_i}{|B|}$$  \hspace{1cm} (8)

$$ID_X = \frac{\sum_{b \in E} \sum_{b' \in E} bx_{i,j} \cdot x_i}{\sum_{b \in E} \sum_{b' \in E} bx_{i,j}}$$  \hspace{1cm} (9)

At this point, as represented on the right frame in Fig. 4, the middle steps Evaluate Bugs Priority and Evaluate Bugs Severity estimate the priority and severity degrees associated to all bugs included in the dependent solution $X$, which are the main terms that compose de fitness function. As defined in (10), the priority degree $P_X$ is estimated based on the weighted-average value of the producer priority $pp_i$ and consumers priority $cp_i$ for each selected bug $b_i$, refactored by the coverage degree $CD_X$. Thus, if two dependent solutions have the same weighted-average priority, the one with higher coverage degree fixes more bugs and so is favored in the proposed approach.

$$P_X = CD_X \cdot (wp_p \cdot \sum_{b \in E} pp_i \cdot x_i + wc_p \cdot \sum_{b \in E} cp_i \cdot x_i) / |B|$$  \hspace{1cm} (10)

In a similar way, as defined in (11), the severity degree $S_X$ is estimated based on the weighted-average value of the producer severity $ps_i$ and consumers severity $cs_i$ for each selected bug $b_i$, refactored by the interdependency degree $ID_X$. Thus, if two dependent solutions have the same weighted-average severity, the one with higher interdependency degree fixes more critical bugs and so is preferred.

$$S_X = ID_X \cdot (ws_p \cdot \sum_{b \in E} ps_i \cdot x_i + ws_c \cdot \sum_{b \in E} cs_i \cdot x_i) / |B|$$  \hspace{1cm} (11)

Note that, as defined in (10) and (11) the priority and severity degrees can assume values in the interval $[0, 1] \in \mathbb{R}$. Besides, the terms $wp$ and $ws$ represent normalized weights, $wp_p + wc_p = 1$ and $ws_p + wc_c = 1$, applied for representing the effective contribution of producer and consumers in priority and severity degrees.

Finally, as indicated on the right frame in Fig. 4, the last step Evaluate Bugs Fitness has the goal of estimating the solution fitness, based on the proposed fitness function and constrained by the project budget $PB$. As defined in (12), the solution fitness $F_X$ is estimated based on the weighted-average value of the priority degree $P_X$ and the severity degree $S_X$, assuming values in the interval $[0, 1] \in \mathbb{R}$. Besides, the term $wf$ represents a normalized weight, $wf_p + wf_s = 1$, applied for representing the effective contribution of priority and severity degrees in the solution fitness.

$$F_X = \begin{cases} \frac{wf_p \cdot P_X + wf_s \cdot S_X}{VS_X} & \text{if } VS_X \leq PB \\ 0 & \text{otherwise} \end{cases}$$  \hspace{1cm} (12)

It is important to highlight that, if the bug-fixing cost $VS_X$ is greater than the project budget $PB$, the proposed approach penalizes the candidate dependent solution $X$, assigning zero to its solution fitness. Indeed, such a penalty drastically reduces the likelihood of more expensive solutions to influence in the population evolution, providing solutions that do not exceed the software project budget.

In conclusion, the formulation of the BSP problem in the proposed approach considers a fitness function whose value depends on bugs priority and severity, assigned by the software producer and its set of customers, through the inclusion of their reported bugs in the next release of the software product. As such, the aim of the BSP problem is to select the subset of reported bugs that maximizes the value of the aforementioned function with cost in budget.

### 4 Real-world case study

The proposed approach has been evaluated using a real-world dataset from a large academic management system, comprised of 226 software components. The dataset comes from a batch of 20 reported bugs, defining a search space of $2^{20}$ candidate solutions. Such reported bugs have been registered by both: (i) maintenance and testing teams on the producer side; and (ii) 3 course coordinators on the customers side.

The project team working on the upcoming release has to select an appropriate set of bugs to cut-down the back-log of reported bugs. The project leader plans for fixing bugs without exceeding project budget and estimates bug-fixing cost using expert judgement, which is out of the scope of this paper.

Therefore, in such a context, considering 20 bugs with different priority, severity and costs, and besides, 3 customers with varying institutional importance derived based on the number of enrolled students in their respective courses, the approach proposed herein for selecting bugs with maximum possible institutional value and with cost in budget sounds to be very interesting. Due to space limit, the detailed data about bugs, assets and customers are not presented herein.

Further, instead of considering a given project budget, as a mean to deeply assess the performance of the proposed approach in relation to varying project budgets, the case study is based on 14 experiments, each one adopting a project budget level that represents 5, 10, 20, 25, 30, 40, 50, 60, 70, 75, 80, 90, 95 and 100% of the total project cost, which is estimated summing the cost of all reported bugs.

#### 4.1 Implementation and calibration

For evaluating the proposed approach, a fully compliant implementation has been developed in Java, providing three different search strategies: (i) genetic search; (ii) random search; and (iii) exhaustive search. In the genetic search strategy, the initial population is generated using a probabilistic-based scheme driven by bugs priority, severity and cost. Then, it improves the population through a generational process, based on repetitive application of the following genetic operators: tournament with roulette wheel selection; uniform crossover; binary mutation, elitist
replacement and generation-based termination. Again, due to space limit, a detailed rationale and description of the adopted genetic operators are not presented herein.

On the one hand, regarding the genetic search, the implementation allows to set the population size (40) and generation length (80). On the other hand, concerning the random search, the implementation allows to define the number of evaluated solutions (3,200). As calibrated, both strategies cover around 0.3% of the search space.

Concluding the calibration, all normalized weights for estimating priority \(w_p\), severity \(w_s\) and fitness \(w_f\) adopt the value 0.5, representing an equal influence for producer and customers. The calibration of all parameters has been taken after many experimentations, however, future work ought to tune them in a more systematic way.

4.2 Results and comparative evaluation

For each experiment, the genetic, random and exhaustive strategies were compared. Genetic and random searches were executed 100 times, and the mean values of the highest-ranking solutions were computed. Besides, the exhaustive search was executed just one time. Contrasting genetic and random strategies, as shown in Fig. 5, experimental results reveal that, in all cases, the genetic search is always better than the random search. Note that, regarding a budget level varying from 30% to 100%, the solution recommended by the genetic search is around 65% better that the one recommended by the random search. Notably, even in the worst case, which occurs in a budget level of 80%, the genetic search is around 24% better than the random search.

Fig. 5 - Experimental results

Contrasting the genetic and exhaustive strategies, the results reveal that, except in a budget level of 70%, the genetic search always found in all executions the optimal solutions, which were identified by the exhaustive search. Due to that, the results of the exhaustive search are not in Fig. 5.

Looking at the experiment with a budget level of 70%, the mean values in the genetic and exhaustive strategies are very similar, being 0.1568 and 0.1569, respectively. In a deeper inspection, it can be observed that the genetic search recommended the optimal solution in 96 of 100 executions. That is, in 96% of the executions, the genetic search also found the optimal solution.

As another interesting result, it must be highlighted the low processing cost of the proposal, perceived by its fast convergence (80 generations) and execution time (around 35 sec). Based on the execution time of the exhaustive search (near 35 min), it can be inferred that the genetic search is 60 times faster than the exhaustive search.

5 Related work

SBSE has been successfully applied to a number of activities throughout the software process lifecycle [13]. More recently, some studies related to bug-fixing tasks have been proposed, suggesting the use of search based optimization techniques on bug reports for automating some issues.

For instance, Xiao and Afzal [3] proposes a weighted single-objective approach for the problem of finding appropriate developer assignments for a set of bug-fixing tasks. In such a proposal, the value of selecting a given bug is estimated based on a weighted function that tries to maximize the number of high severity and high priority bugs resolved. Thereafter, conjointly, a single-objective fitness function combines the values of all selected bugs, trying to maximize the total value of resolved bugs from a scheduling perspective. Similarly, the proposed approach also tries to prioritize bugs assigned to high severity and high priority. However, while the Xiao-Afzal’s proposal does not infer or explore dependencies among bugs, the proposed approach deals with precedence and concurrency relationships defined by dependent bugs.

As another proposal, Dreyton et al. [14][15] investigates an approach to prioritize bugs in open source repositories. Like the approach proposed herein, Dreyton’s approach was initially proposed as a weighted single-objective formulation. It explores the concept of votes given by developers, and besides priority and severity assigned to reported bugs. The weighted fitness function evaluates relevance, importance and severity associated with a candidate solution, which represents a set of bugs to be prioritized. Note that, the number of prioritized bugs is directly specified as a pre-configured limit. Differently, in the proposed approach, the number of selected bugs is not pre-configured, but automatically elected and indirectly constrained by project budget. As such, the assumption is that it is more practical, effective and effortless to set a budget than to anticipate the number of bugs to be fixed.

With a similar goal, Netto et al. [16] proposes an approach for the problem of scheduling a set of bug-fixing tasks with different priorities to a set of available developers, with previous experience in a given set of bugs categories, according to a cost function modeled to measure the adherence of the proposed schedule, increasing the number of selected bugs, prioritizing most urgent bugs, and raising the occupation rate of the development team. On the one hand, like the approach proposed herein, the Netto’s approach was proposed as a weighted single-objective formulation. On the other hand, unlike the proposed approach, the Netto’s approach does not consider the concepts of bug severity and dependencies, which are well-known and widely adopted information models for bugs reports in current available BTS.

Consequently, as far as we know, the present approach is the first proposal that explores bug dependencies, dealing with precedence and concurrency relationships defined by them. Besides, as an additional contribution, instead of considering bug dependencies simply as an input parameter, the proposed approach adopts a novel tracking strategy in which bug dependencies can be directly inferred from relationships among assets at the architectural level of component-based software projects. Yet another innovation is the adoption of the concept of customers importance, and besides the concepts of bug priority and severity assigned by producer and customers.


6 Concluding remarks

In this paper, an automated approach is proposed for selecting bugs in component-based software projects. The proposal does not intend to select skilled developers to fix bugs, but to recommend the most suitable set of bugs, considering (i) bug priority and severity, informed by producer and customers; (ii) bugs dependencies, inferred from component-based software projects; (iii) multi-customers importance from the producer’s viewpoint; (iv) bugs coverage and interdependency, deduced from the transitional close defined by assets; but (v) constrained by the project budget.

By exploring a real-world dataset, 14 experiments were performed and the results were able to show that: (i) the genetic search outperforms the random search regarding the quality of recommended bug selections; (ii) the genetic search beats the exhaustive search regarding processing time and cost; (iii) the proposal is human competitive, focusing the bug-fixing activities on what is essential, decreasing the development team effort at selecting, among reported bugs, a set of highly important and impacting bugs within the project budget; and (iv) recommendations are sensitive to different calibrations, allowing to set options that better suits faced scenarios.

There can be four types of validity threats [17]: internal, external, construct and conclusion. Concerning internal threats, customers importance was based on the number of enrolled students, as such data does not exist in the BTS. So, this decision may impact on recommended bugs. Despite the case study calibration being empirically obtained, a fine-tuning parametrization would lead to better solutions in more complex scenarios. Applying other metaheuristics are pretty to contrast findings in terms of design suitability, quality of solutions, execution cost, and ease of understanding and usability.

Regarding external threats, the case study considered three customers and just one real-world dataset with 20 bugs only. Replications and adaptations on a wider range of datasets with a high number of customers and bug reports are desirable to achieve more generalized outcomes, making possible to identify biases. Indeed, the results as such should be applicable to the situations where similar assumptions are held. Otherwise, the formulation needs to be adapted accordingly.

In relation to construct threats, the proposed approach adopts concepts and information models which have been successfully applied in related work and BTS. The assumptions and rationale made sense for the type of case study discussed, however, the formulation might change in the case that different bug-fixing abstractions are perceived and adopted.

Referring to conclusion threats, as a mean to counter the stochastic nature of search techniques and ensure a fair comparison, genetic and random search strategies were performed multiple times (100) for each experiment, overcoming their inherent randomness. In complement, a more valuable statistical analysis needs to be conducted, measuring statistical similarities and differences in search strategies.

Now, regarding future work, it sounds interesting the formulation of the BSP problem as a multi-objective problem, in which solutions can be chosen from a Pareto Front. Besides, as another improvement, the intent is to include the concepts of bug scheduling and developer assignment to automate project planning and management. It is also imperative to deal with mentioned threats as sources for future work.

7 References