The Journal of Systems and Software 217 (2024) 112157

Contents lists available at ScienceDirect

The Journal of Systems & Software

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

Feature-oriented test case selection and prioritization during the evolution of highly-configurable systems☆

Willian D.F. Mendonça [a,b,*], Wesley K.G. Assunção [c], Silvia R. Vergilio [a]

[a] Computer Science Department, Federal University of Paraná (UFPR), CP: 19081, CEP: 81.531-980, Curitiba, Brazil
[b] Faculdade Biopark, Toledo, Brazil
[c] Department of Computer Science, North Carolina State University (NCSU), Raleigh, USA

ARTICLE INFO

Dataset link: https://osf.io/wgk7c

Keywords:
Software evolution
Software product line
Regression testing
Test case prioritization

ABSTRACT

Testing *Highly Configurable Systems (HCSs)* is a challenging task, especially in an evolution scenario where features are added, changed, or removed, which hampers test case selection and prioritization. Existing work is usually based on the variability model, which is not always available or updated. Yet, the few existing approaches rely on links between test cases and changed files (or lines of code), not considering how features are implemented, usually spread over several and unchanged files. To overcome these limitations, we introduce `FeaTestSelPrio`, a feature-oriented test case selection and prioritization approach for HCSs. The approach links test cases to feature implementations, using HCS pre-processor directives, to select test cases based on features affected by changes in each commit. After, the test cases are prioritized according to the number of features they cover. Our approach selects a greater number of tests and takes longer to execute than a changed-file-oriented approach, used as baseline, but `FeaTestSelPrio` performs better regarding detected failures. By adding the approach execution time to the execution time of the selected test cases, we reached a reduction of ≈50%, in comparison with retest-all. The prioritization step allows reducing the average test budget in 86% of the failed commits.

1. Introduction

Highly Configurable Systems (HCSs) are pieces of software designed to promote customization and flexibility while leveraging systematic reuse (Michelon et al., 2021). The systematic reuse is achieved by creating a common core of assets that is shared among different products that can be derived from an HCS (Clements and Northrop, 2002). The customization and flexibility comes from a set of configuration options, enabling different functionalities to be selected to a given context or needs (Von Rhein et al., 2015). For the design and implementation of HCSs, the configuration options are captured as *features* of the target domain or market segment (Kang et al., 1990).

The main benefit of HCSs is to reduce the time to market of new products, by systematically reusing the core of assets (Clements and Northrop, 2002), as mentioned above. However, due to this extensive reuse, the testing of HCSs is a critical activity (do C. Machado et al., 2014). A bug in a common feature (i.e., configuration option) can lead to failures in a potentially large number of software products (Ferreira et al., 2019). To make matters worse, HCSs usually have a substantial number of features and interrelated dependencies that increases the complexity of testing (Medeiros et al., 2018).

Software testing for HCSs is even more challenging when we consider an evolution scenario, such as in the cases in which *Continuous Integration (CI)* practices are adopted (Lima et al., 2020). HCSs can be updated, integrated, and tested several times a day, and each cycle needs to be fast (Zhao et al., 2017). CI environments automatically support tasks such as the build process, test execution, and test results reporting, allowing engineers to merge code that is under development or maintenance with the mainline code base at frequent time intervals (Duvall et al., 2007). The results are used to solve problems and find faults. Re-executing all test cases (i.e., *retest-all* technique) during each evolution cycle of the HCS may be impracticable, as the test activity in an industrial environment is extremely costly (Garousi and Zhi, 2013). Then, providing quick feedback is essential to reduce development costs (Jiang and Chan, 2016). To this end, the application of *Regression Testing (RT)* techniques in a very cost-effective way is fundamental (Garousi and Zhi, 2013). The best known and used RT techniques are Yoo and Harman (2012): (i) Test Case Minimization (TCM) usually removes redundant test cases, minimizing the test set according to some criteria; (ii) Test Case Selection (TCS) selects a

☆ Editor: Laurence Duchien.

* Corresponding author.
E-mail addresses: willianmendonca@ufpr.br (W.D.F. Mendonça), wguezas@ncsu.edu (W.K.G. Assunção), silvia@inf.ufpr.br (S.R. Vergilio).

https://doi.org/10.1016/j.jss.2024.112157
Received 31 January 2024; Received in revised form 25 May 2024; Accepted 8 July 2024
Available online 10 July 2024

subset of test cases, the most important for testing the software; and (iii) Test Case Prioritization (TCP) attempts to reorder a set of tests to identify an ideal order that, ideally, maximizes early fault detection.

Existing TCS approaches for HCSs are usually based on *Feature Model (FM)* or other artifacts representing variability (Lity et al., 2019; Lachmann et al., 2015; Al-Hajjaji et al., 2017; Lachmann et al., 2017; Silveira Neto et al., 2010). However, none of them consider that HCSs are usually developed by adopting CI practices, in a scenario where the FMs are rarely updated (Vierhauser et al., 2010; Ghanam and Maurer, 2010; Heider et al., 2012), making the use of those approaches difficult. Other HCS testing approaches rely on dynamic analysis based on the test failure-history, execution, or coverage (Marijan et al., 2017; Marijan and Liaaen, 2018; Marijan et al., 2019). In some pieces of work, the approaches are only evaluated with systems well-modularized (Jung et al., 2019), having test cases separated by feature and without overlap among features, which is different from real scenarios. Testing approaches in CI are also based on code changes. For instance, TCS approaches select test cases related to the files changed in the current commit (Gligoric et al., 2015; Bertolino et al., 2020; Romano et al., 2018). But those existing approaches and tools do not consider HCS particularities (i.e., features as building blocks), and they are mostly based on Java language only (Jung et al., 2019, 2020, 2023). However, C and C++ are the preferred language, largely adopted for HCS developers (Medeiros et al., 2018; Michelon et al., 2022). Yet, the few existing approaches rely on links between test cases and files or lines of code, limiting the selection to test cases related to file changes, not considering the whole implementation of features, which can be spread over many files other than the changed ones.

Motivated by these facts, in a previous work, we introduced `FeaTestSel` (Mendonça et al., 2023), a feature-oriented approach for test case selection that links test cases to features using HCS pre-processor directives. Given the source code of an HCS and a set of test cases available for a given commit, `FeaTestSel` selects the best test cases to be executed in order to cover the features changed in the corresponding evolution cycle. `FeaTestSel` also produces different traceability reports linking, namely (i) test cases to code lines of the system, (ii) features to code lines of the system, and (iii) test cases to features. Differently from related work, the implementation of `FeaTestSel` works for systems in the C/C++ language. Our approach is feature-oriented and needs only static analysis of the source code. Results from our previous work with the system `Libssh` show that `FeaTestSel` reduces the testing runtime to ≈50% compared with the retest-all technique. Furthermore, the approach was able to maintain the test quality by selecting 100% of test cases that revealed failures. The main advantage of a feature-oriented approach is to provide a natural way to think about the units to be tested, since features are the building blocks of HCSs, used as main units for HCS design and communication among stakeholders.

Leveraging contributions of our previous study, in the present work we propose an extension for `FeaTestSel`, namely `FeaTestSelPrio` (**Fea**ture-oriented **Test** Case **Sel**ection and **Pri**oritization for Highly Configurable Systems). This extension includes an additional step that performs the test case prioritization considering the reports generated by `FeaTestSel`. Furthermore, new evaluation results and analysis are added, not only for `Libssh`, but also for a new HCS, namely `Libsoup`.

The prioritization strategy adopted is based on the number of features associated to a test case, assuming that the greater the number of changed features a test case covers in the system, the more fault-prone such a test case is. The idea is, after test selection, to rank the test cases in order to early execute those with a high probability of revealing faults. Considering the constraints of CI test budgets, early fault detection is essential because when a test case fails, test execution can be ended, and fewer resources are spent (Lima and Vergilio, 2020).

The study we conducted to evaluate `FeaTestSelPrio` has two main objectives. First, we evaluate the steps of `FeaTestSel` by conducting the same analysis as our previous work, but now by adding a new system. `FeaTestSel` results are compared with the results of a changed-file-oriented approach, regarding the percentage of test reduction and quality with respect to detected failures, using retest-all as baseline. We observe that the changed-file-oriented approach reaches a greater percentage but sacrifices failure-detection. `FeaTestSel` reaches a better trade-off considering these both factors and a cost-effective TCS. In the analysis of the set of commits with logs, our feature-oriented selection reaches an average reduction in the number of test cases of ≈42%. Also, on average, the execution time of `FeaTestSel` summed to execution time of the selected test cases fits well in a budget of 50% of the average time required to execute all the tests available in the commit.

For the second objective of our evaluation, we study the applicability of `FeaTestSelPrio` by comparing the time it takes to select and prioritize test cases summed to the time to execute them. This total time is then compared with the time in between CI cycles. The average time to perform the selection and prioritization is 26.40 s. This time, when summed to the time to execute the selected test cases (worst case), is 119.75 s. The prioritization step of `FeaTestSelPrio` on average takes only 0.42 s. The prioritization step is also evaluated considering early-fault detection, leading to a reduction of 44% in the budget required to detect a failure in comparison with a reduction of 23% produced by the order generated by applying only `FeaTestSel`. In 86% of the failed commits, the prioritization requires a lower budget.

In summary, this paper introduces a feature-oriented test case selection and prioritization approach to test HCSs, which has the main following contributions:

1. `FeaTestSelPrio` allows the selection and prioritization of test cases that are related to not only changed parts of the code, but to the entire feature changed in a commit. This provides a natural way to test HCS, considering their main building blocks;
2. Differently from other approaches in the literature, our approach does not require historical data or training phase, requiring only the source code and test cases of the HCS to be applied;
3. `FeaTestSelPrio` contributes to reduce the number of test cases, without reducing efficacy in the number of revealed bugs. The prioritization results allow an early failure-detection and rapid feedback; and
4. Our approach produces several intermediate outputs, including the traceability of test cases to source code, traceability of test cases to features, and system features and location of features in source code. These outputs can be used by software engineers for improving development and reducing testing costs.

The paper is structured as follows. Section 2 reviews related work. Section 3 contains an example that serves as motivation for a feature-oriented approach. Section 4 introduces the proposed approach and presents its implementation aspects. Section 5 describes the methodology adopted in the approach evaluation. Section 6 presents and analyzes the obtained results. Section 7 discusses some implications for research and practice, as well as limitations our approach and threats to the validity of our results. Section 8 concludes the paper.

2. Related work

In the literature, we can find three mapping studies that focus on reporting existing pieces of work on RT for HCSs (Mendonça et al., 2022a; Kumar and Rajkumar, 2016; Runeson and Engström, 2012). The proposed methods, approaches, or tools mainly focus on selecting a representative set of HCS variants to be tested, identify differences among products, and using the FM to derive and select test cases. More details of related work and observed limitations are presented in the following.

Some pieces of work are devoted to the selection/prioritization of the best product configurations to be tested, having as focus the FM (Kumar and Rajkumar, 2016; Parejo et al., 2016; Ensan et al., 2011)

and considering different goals such as: combinatorial testing, product similarity, coverage of variability and important features. These works do not directly deal with test cases, instead they focus on selecting a representative set of products. With a similar goal, some studies apply model-based test considering the delta concept. For instance, the work of Lity et al. (2019) captures commonality and variability of an evolving product line by means of differences between variants and versions of variants to select the test cases to be retested. Lachmann et al. (2015) introduce an incremental delta-oriented approach for improving Software Product Line (SPL) integration testing efficiency by prioritizing test cases for product variants. Al-Hajjaji et al. (2017) selected the most dissimilar product to the previously tested ones, in terms of deltas. Lachmann et al. (2017) present a TCP approach based on risk-based testing, which can automatically compute component failure impact and component failure probabilities for each product variant under test.

In the context of TCS, the approach proposed by Wang et al. (2017, 2013) applies a classification of annotated test cases. The idea is to ensure that all test cases associated with a specific functionality provided by the user are executed. Hajri et al. (2020) present an automated test case classification and prioritization approach that supports use case-driven testing in product lines. These two approaches have a limited applicability, because they need the FM or other artifacts that represent variability as a source of information. This is a disadvantage, because they are not always available, and in the HCS evolution process may be outdated (Vierhauser et al., 2010; Ghanam and Maurer, 2010; Heider et al., 2012). Other TCS approaches select a subset of existing test cases to be reused for testing a new product (Lochau et al., 2012; Wang et al., 2016; Xu et al., 2013), also focusing more on the variants than in the test of the HCS as a whole.

Jung et al. (2023) propose ActSPL, an automated method for reusing the existing test cases for a new product of a product family. The basic assumption is that, when a test case covers only the pieces of code commonly shared by two or more products, the test case is sharable for the products. By using this assumption, ActSPL examines if a test case of an existing product covers pieces of code that belong to a new product. By doing so, ActSPL determines whether an existing test case is reusable for the new product. The work of Silveira Neto et al. (2010) describes an RT framework for HCSs at the integration level. The idea is to reduce testing effort by selecting and prioritizing test cases based on architectural similarities between products. However, it requires several input artifacts, which are often unavailable, such as test scripts and integration level test suites. In addition, the intervention of testing experts is necessary. These approaches consider differences between two products, and not the whole product family. Another limitation is that they do not consider an evolution scenario.

Tufail et al. (2017) present a systematic review on traceability techniques and tools that link test cases to requirements based on static information, without requiring the program execution or test coverage. But the pieces of works mentioned in the review do not deal with the concept of feature, the main HCS element. Some pieces of work introduce methods based on changed files or versions (Gligoric et al., 2015; Bertolino et al., 2020; Romano et al., 2018). An example is the tool Ekstazi (Gligoric et al., 2015) that calculates the checksum of the new file versions and considers that the file has changed if the checksums are different from the old file version ones. To select test cases, Ekstazi calculates the file dependencies of the test units. Bertolino et al. (2020) presents a TCS approach, applying a criterion based on static dependency analysis at the class level. These approaches can be used in the HCSs, but they work only for Java code and do not consider HCSs particularities. Some studies for HCSs that are also based on source code (Meinicke et al., 2016; Kim et al., 2012, 2011), do not generate traceability for features, but only link test cases to lines of code.

The work of Tuglular and Şensülün (2019) generates a traceability between the feature and test cases, but using a specific language through annotations. The code-based method of Jung et al. (2019) considers the similarity and variability of a product family, leaving out test cases unaffected by source code changes. But the evaluation considers only well-developed HCSs (i.e., toy systems), and test cases were developed specifically for the evaluation, what hampers the use of the approach in practice. In another work, Jung et al. (2020) propose a TCS method to avoid repeating equivalent test runs that cover exactly the same source code sequence and produce the same test result on two or more variants. To identify equivalent test runs, test case execution traces and source code checksum values are used. The several steps involved may be quite costly if applied to large systems that are constantly evolving. Yet, it is specific for Java and both pieces of work do not consider traceability to features, but only to source code files.

Another limitation of existing studies is that some of them do not directly select or prioritize test cases and do not consider the SPL evolution, that is, the different versions of variants in an RT scenario. For instance, the great majority does not consider particularities of the CI environment. TITAN (Marijan et al., 2017) is a data-history approach used to determine an optimal test order to ensure feature coverage, early fault detection, and reduced execution time. The approach considers that the test cases have macros indicating which HCS features they exercise. However, we can observe that in most open-source HCSs these macros do not exist. This hampers its applicability for general scenarios. Other studies of Marijan and Liaaen (2018), Marijan et al. (2019) identify redundancy by analyzing the overlap of configurations options in a test set. Afterward, the tests are classified as unique, fully redundant, or partially redundant. Then, historical test information is used to determine which configurations have demonstrated high failure in previous test runs. Based on this information, the approach classifies partially redundant tests into effective and ineffective tests. The analysis uses code coverage per test case, and this dynamical strategy can degrade the performance of algorithms. Approaches that perform only static analysis are more suitable.

Lima et al. (2020), Prado Lima et al. (2022) introduce two strategies to apply a TCP learning-based approach called COLEMAN in the CI of HCSs: the Variant Test Set Strategy (VTS) that relies on the test set specific for each variant; and the Whole Test Set Strategy (WST) that prioritizes the test set composed by the union of the test cases of all variants. COLEMAN is an approach that learns from the test case failure-history, guided by a reward function. The main idea of these approaches is to deal with volatility of variants, that is, a new variant can be added in the cycle and new test cases can be added or removed. In the evaluation, WTS provides better results in the less restrictive budgets, and VTS the opposite. WTS seems to better mitigate the problem of beginning without knowledge, and is more suitable when a new variant to be tested is added. Our work differs from those because we first select a test set based on the evolution of features, which contributes to reduce costs and bypass the variant volatility problem.

In summary, existing work present the following main limitations: (i) are dependent on models or other artifacts (e.g., FM or HCS architecture) that may not exist or be outdated (Lity et al., 2019; Lachmann et al., 2015; Al-Hajjaji et al., 2017; Lachmann et al., 2017; Silveira Neto et al., 2010; Wang et al., 2017, 2013; Hajri et al., 2020); (ii) require a failure-history or dynamic analysis (Marijan et al., 2017; Marijan and Liaaen, 2018; Marijan et al., 2019; Prado Lima et al., 2022; Lima et al., 2020), what is costly and may be not suitable for a CI scenario; (iii) do not consider HCS particularities and/or languages such as C and C++, largely adopted for the HCS development (Gligoric et al., 2015; Bertolino et al., 2020; Romano et al., 2018; Tuglular and Şensülün, 2019); (iv) consider that there is a kind of mapping from code to the test cases or that the HCSs are developed following a specific format (Marijan et al., 2017; Jung et al., 2019); and (v) do not work with the concept of features, which are fundamental units of design and communication in the HCS context (Meinicke et al., 2016; Kim et al., 2012, 2011; Jung et al., 2020). Our approach, presented in Section 4,

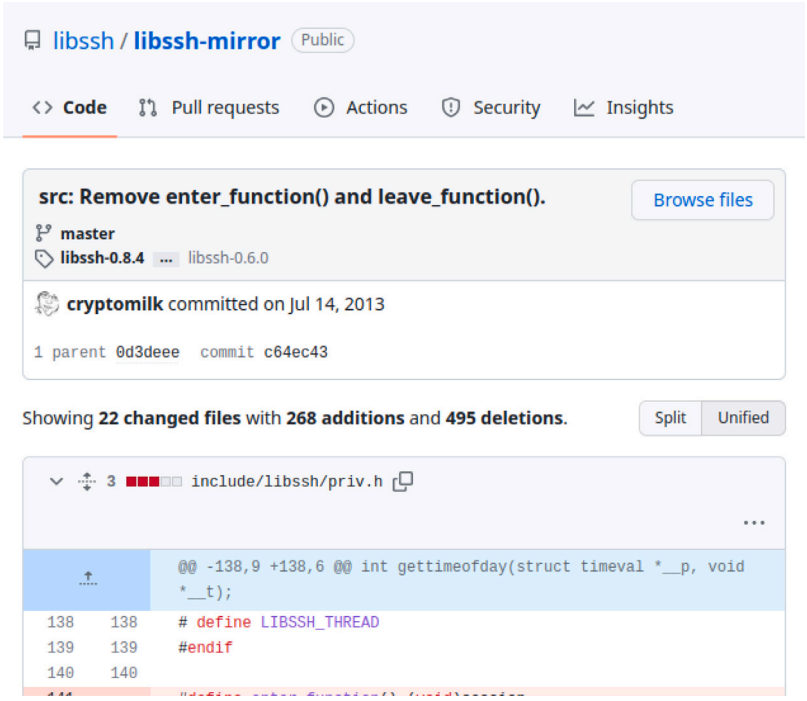


Fig. 1. Number of modifications made in commit `c64ec43`.

addresses these limitations. We aim for an approach that has automated TCS and TCP based on features. It relies only on the static analysis of the source code, and works for HCSs written in C/C++ language. Moreover, we introduce a prioritization step to deal properly with the test budgets.

3. Motivating example

In this section, we present a motivating example showing the importance of considering the relation between features and test cases for RT during evolution of HCSs. To this end, we use the system `Libssh` (presented in details in Section 5.2), which is in constant evolution, updated by several developers, sometimes more than once a day. This is the common case of open-source HCSs, leading to the issues discussed below.

By analyzing the CI cycles and the test case evolution in `Libssh`, we observe that, in the great majority of the commits, the retest-all technique is executed after each cycle. In this scenario, suppose that a developer needs to make a small change in the system related to a feature of this HCS, and after this change the retest-all technique is adopted. This is an expensive and time-consuming approach that uses additional computation resources for every change, even the simple ones. This retest may be necessary several times due to the number of developers involved.[1]

For illustration, we take as example the commit `c64ec43`,[2] which performs the removal of the functions `enter_function()` and `leave_function()`. This modification is related to 22 changed files with 268 additions and 495 deletions, as shown in Fig. 1. In this commit, 117 test cases are available, making the retest-all technique to take up to five minutes to run per variant.[3] Notice here that the HCS can receive several updates a day, as can be seen in the repository, and many variants must be tested.

Instead of always executing all test cases (i.e., retest-all), we can adopt a smarter strategy. An alternative is to trace test cases to the 22 files changed and select only the test cases associated to them. But this technique may not select the test cases related to all the features changed in the commit, and that can cause failures in unchanged files because of features interactions (Meinicke et al., 2016) or the Ripple effect (Yau et al., 1978). An example is presented in Fig. 2, in which the feature `WITH_SSH1` was changed in the commit referred above. In this specific change, the call to `leave_function()` was excluded. However, the exclusion of this call can change all the functionality of `WITH_SSH1` and impact other unchanged files that also implement this feature, such as `options.c` and `channels1.c`.

While analyzing the execution of test cases for `Libssh`, we also observed that the test cases are always executed in the same order. Currently, CI cycles do not have any strategy or method to prioritize the test cases based on the information about `Libssh` evolution. Thus, test cases are executed without considering they have different importance. For instance, some test cases have higher probability of failing than others. Then, to produce a rapid feedback for the test, it is important these test cases are executed first. A solution to this is to rank the test cases. To address this issue, we introduce a feature-oriented selection and prioritization approach (presented in the next section). This approach is capable of capturing the relationship between features and test cases, including all impacted test cases in the selected test set.

From the `Libssh` repository, we obtained insights that motivated our work. For instance, we can observe that in the commit `d7477dc7`[4] there are 179 test cases, from which 4 failed during the test. In a more detailed analysis, we see that these test cases cover on average 6.5 features, while the test cases that not failed cover 3.9. This serves as motivation to our approach, which generates a rank based on feature coverage, considering test cases that are related to a greater number of features are more fault-prone.

[1] `Libssh` is developed by the collaboration among 110 developers, according to https://github.com/libssh/libssh-mirror/graphs/contributors.

[2] https://gitlab.com/libssh/libssh-mirror/-/commit/c64ec43

[3] Time per variant is found at https://gitlab.com/libssh/libssh-mirror/-/pipelines.

[4] https://gitlab.com/libssh/libssh-mirror/-/commit/d7477dc7

```
1698        -   enter_function();
1699  1655      #ifdef WITH_SSH1
1700  1656        if (channel->version==1) {
1701  1657          rc = channel_request_pty_size1(channel,terminal, col, row);
1702        -     leave_function();
      1658  +
1703  1659        return rc;
1704  1660        }
1705  1661      #endif
```

Fig. 2. Example of modification in commit `c64ec43`.

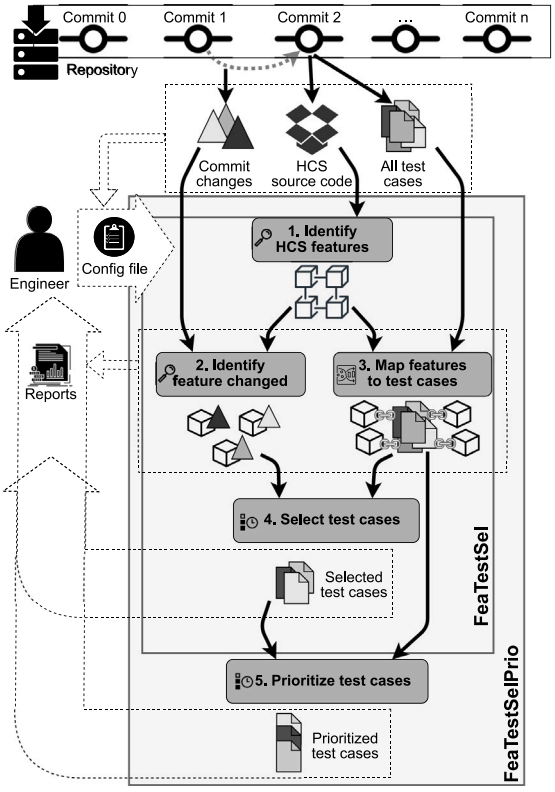


Fig. 3. Overview of our approach.

4. Proposed approach

In our previous work, we introduced `FeaTestSel` (**Fea**ture-oriented **Test** Case **Sel**ection for Highly Configuration Systems) (Mendonça et al., 2023), which consists of four steps that are presented in Fig. 3. The tester needs to only provide a configuration file (*Config file*) containing the paths to the source code of the HCS and the test case folder. In Step 1, *Identify HCS features*, the source code corresponding to each feature of the HCS is determined. The lines of code that implement each feature of the system are identified automatically based on pre-processor directives. After this, two independent steps are performed. In Step 2, *Identify features changed*, the source code of the current commit is compared with the previous one to identify feature changes (i.e., features modified, added, or removed). In Step 3, *Map features to test cases*, the lines of code exercised by each test case are identified and trace links between feature and test cases are created. In the fourth step, *Select test cases*, the output of Steps 2 and 3 are used to select test cases related to feature changes in a given commit. The main *output* consists of the selected test cases and reports with traceability information between test cases and features.

In the present work, we devise an extension to `FeaTestSel`, called `FeaTestSelPrio` (**Fea**ture-oriented **Test** Case **Sel**ection and **Pri**oritization for Highly Configurable Systems). This extension includes an additional step, *Prioritize test cases*, that performs the test case prioritization considering the reports generated by `FeaTestSel`. We can observe in Fig. 3 that this step is applied considering as input the test cases selected in Step 4 of `FeaTestSel`, combining selection and prioritization techniques. In addition to the test cases, the prioritization step also uses as input the traceability of features to test cases generated in Step 3. At the end, a set of prioritized test cases is produced.

`FeaTestSelPrio` is designed to be lightweight; it does not need any learning process or any long history of changes or test failures to perform the test case selection and prioritization. Thus, our approach can be executed after each commit, identifying feature implementations, feature changes, and feature to test traceability using the latest version of the HCS. Our approach was implemented in Python, version 3.9.10. The outputs are saved as CSV files, for which we adopt PANDAS.[5] In the next subsections, we describe each step of our approach in details and present more implementations aspects.

4.1. Input

As input, `FeaTestSelPrio` receives a configuration file (*Config file*) containing paths to some folders used as source of information, as illustrated in Fig. 4 for the system `Libssh`. They are: (i) `repository_URL`: contains the URL to the repository of the HCS (e.g., the URL of the HCS on GitHub); (ii) `system_path`: indicates the path of the source code folder with the implementation of the features; (iii) `test_name`: defines a pattern in the nomenclature of test cases that allows the approach to identify which source code files are related to test cases. Alternatively, the software engineer can provide the folder name, `test_folder`, used to store test cases, when this is a practice in the project. In this case, the test case selection will perform faster. However, using the test names brings the benefit that all system files will be checked, since by using a good search string, hardly any test case file will be forgotten; and (iv) `prioritization`: contains 1 whether the prioritization will be performed after the selection, or 0 otherwise.

4.2. Identify HCS features

To mine features and their changes, our implementation abstracts the source code of a commit snapshot at the level of pre-processor directives, which are distinguished in conditional blocks (i.e., `#if`, `#ifdef`, `#elif`, `#else`, and `#ifndef`), definition lines (i.e., `#define` and `#undef` directives), or import file lines containing `#include` directives, similarly to related work (Michelon et al., 2021). This abstraction is less computationally expensive, as we do not need to analyze the abstract syntax tree to obtain, for each file, the lines of code containing pre-processor directives. The approach uses a Constraint Satisfaction Problem Solver (Schiex and de Givry, 2019) to reliably identify features interacting/depending on the execution of other features (Benavides

[5] https://pandas.pydata.org/

Fig. 4. Configuration file used as input.

et al., 2006), and thus which features belong to which conditional blocks to obtain the features lines.

The code excerpt in Fig. 5 is used to illustrate the strategy adopted for mining features lines. The figure contains four variation points (i.e., conditional blocks) wrapped by conditional directives. For each conditional block, the approach creates constraints related to each particular conditional block of code. For instance, lines 1–3 belong to feature A. This is the simplest case, where there are no interactions or nested features. But the block of code of lines 6–8 belongs to a feature internally defined, inside feature A. This block of code of lines 6–8 is activated when feature A is selected, and thus when B is greater than 10. However, this is not the only constraint to take into account to determine which features belong to the block of code of lines 6–8 because there is an outermost block wrapping lines 6–8 with the conditional expression `#if C`. In this case, feature C also has to be selected so that this block of code can be executed. Therefore, in such cases, where multiple features imply executing a block of code, a heuristic is adopted to consider the lines as part of the closest feature (not defined internally via `#define` directive) to the block of code. In this way, the block of code of lines 6–8 is assigned to the feature C. Therefore, the lines of code of feature C begins at line 5 and ends at line 9.

We also have a corner case example (Ludwig et al., 2019) at lines 11–13, where there is a negated conditional expression, namely the block of code is executed when feature D is not selected. In this case, there are no nested features or feature interactions, and this block of code is thus considered part of the system core (BASE feature), as there is no feature responsible for executing this block. After getting the features responsible to execute each block of code, we obtain the line numbers of each file that belongs to a feature. Therefore, lines 1–3 are related to feature A, lines 5–9 to feature C, lines 11–13 to feature D, and to BASE, as well as line 15, which is outside any variation point.

4.3. Identify feature changed

The outputs of previous steps are used to identify the features and their locations. When observing differences between commits, a feature change is identified in two cases: (i) a new feature is found, i.e., a new pre-processor directive with a feature annotation is added; or (ii) a change in an existing feature occurs, i.e., a change in between pre-processor directives that delimit blocks of code belonging to a feature. To obtain this information, we first collect all conditional block macros and all `#defines` present in all files from each release commit. Then, we look for macros never defined within the source code, i.e., that can only be defined externally by the user, from the command line. In this way, we obtain the macros that can be considered as features of the system.

For each Git commit n, we generate the differences between the actual commit and the previous commit $n-1$. In the case of the initial Git commit in the project, we deal with all inserted files as the point of difference. From these differences, we can obtain the tree node that reflects the changes. If there are modifications to any external features or discrepancies in non-code files, such as binary files, BASE is considered as the altered feature. In other words, for any code additions or removals within the project body that do not pertain to an external feature, the root feature BASE is identified as the modified node.

```
1   #ifdef A
2       #define B 15
3   #endif
4
5   #if C
6       #if B > 10
7           <code>
8       #endif
9   #endif
10
11  #ifndef D
12          <code>
13  #endif
14
15  <code>
```

Fig. 5. Conditional blocks of feature implementations.

In summary, after knowing all blocks of code that belong to the features, our approach uses Git diff[6] to collect code fragments that differ for the same file from one commit to another. This process obtains the differences of fragments with patches. These patches represent the differences between two text files in a line-oriented manner, as calculated by a diff utils library.[7]

4.4. Map features to test cases

To identify dependencies between test cases and blocks of code belonging to features, this step uses static analysis and the tool Test2Feature (Mendonça et al., 2022b). First, a dependency graph using all code available in the repository is created. Then, the dependencies between test cases and source code implementing features are collected. Finally, using the output of Step 1, namely the lines of code implementing each feature, the approach creates trace links between the test cases and the features they are related to.

Basically, a merge between the output of Step 1 and the test cases found in the HCS (i.e., links between the location of the features along with the location of the test cases) is performed. In this way, it is possible to know exactly the location of the tests and features per line of the files. Initially, a merge is performed considering the localization files, then, a filter is applied considering the location of the code lines. The output of this step is stored in a CSV file.

[6] https://git-scm.com/docs/git-diff
[7] https://java-diff-utils.github.io/java-diff-utils/

This step of our approach is implemented based on Doxygen.[8] Doxygen supports visualization of the relationships between various elements through dependency graphs, inheritance, and collaboration diagrams, generated automatically, in different formats. Our implementation uses Doxygen for C/C++ and the function dependency graph in XML format. This tool generates the dependency graph as XML files, performing static analysis of the source code. Doxygen for C/C++ properly parses pre-processor directives by default, considering this is a common construct of C/C++. We defined the minimal set of parameters in order to generate all possible dependencies available in Doxygen and to make the tool executes faster.

4.5. Select test cases

This step selects test cases that cover the features changed in the commit (output of Step 2: *Identify feature changed*). When a change in a feature is identified, all test cases that are linked to this specific feature are selected. We consider a test case covers a feature when it exercises the lines within the blocks of code belonging to a feature, leveraging the trace links created in Step 3 (*Map features to test cases*).

Despite its simplicity, this selection has an advantage when compared to existing approaches. Approaches that are changed-oriented mostly select test cases that are direct related to the changed files, without considering features. In our case, even if the changed file is not touched by the test case, but the test case touches other parts of the features being changed in the given commit, that test case is selected. Since features are building blocks of HCSs, it is important to verify the behavior of the changed features, and it is also important to consider the coverage of the features.

4.6. Prioritize test cases

To perform this step, the parameter `prioritization` in the configuration file must be set to 1. In such a case, the test case prioritization based on features is performed using the trace links between features and test cases (Step 3). The main idea is that test cases associated to a greater number of features are ranked first, because they have a higher probability to reveal faults, as shown in our motivating example. Also, these test cases that are linked to different features may exercise feature interactions, which a common HCS characteristic (Meinicke et al., 2016).

To implement this prioritization strategy to maximize the number of features tested, we create a list for each test case, indicating the features it is associated. We then generate an overall coverage list that includes the names of the test cases, the list of features they cover, and a count of the number of features covered by each test case. The greater the number of features a test case covers in the system, the higher its priority. Test cases covering more features are ranked first. In case of a tie in the number of covered features, the tiebreaker is random.

4.7. Illustrative example

To illustrate the required input and produced outputs of `FeaTeRPrio`, we use again `Libssh` and focus on the commit `c64ec43`.[9] This commit was chosen because it represents well the evolution of an HCS, with changes occurring in five different features: `WITH_ZLIB`, `WITH_SSH1`, `WITH_SFTP`, `WITH_SERVER`, and `BASE`. The feature `BASE` encompasses the implementation of all parts of the HCS that do not belong to a feature (i.e., blocks of code that are not wrapped in pre-processor directives). `BASE` corresponds to a large portion of code, and mostly changes in all commits.

[8] https://doxygen.nl/index.html
[9] https://github.com/libssh/libssh-mirror/commit/c64ec43

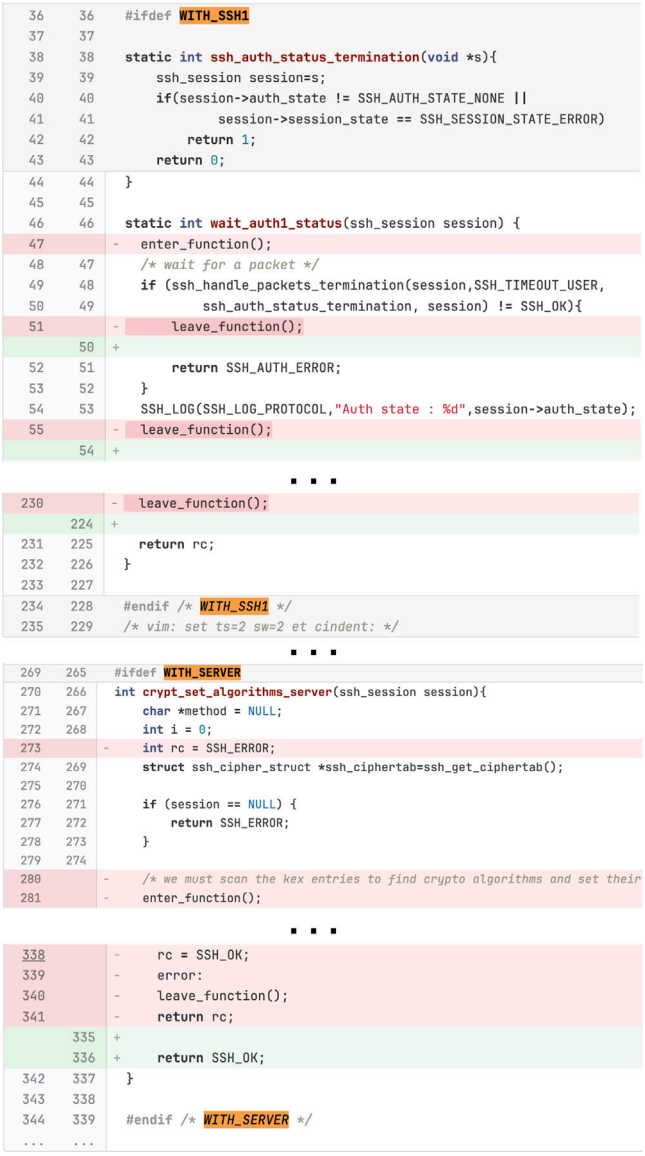


Fig. 6. Excerpt of source code for `Libssh$` features changed as part of commit `c64ec43`.

Fig. 6 presents an excerpt of the source code corresponding to some `Libssh` features. We can observe changes in the feature `WITH_SSH1`, on lines 47, 51, and 55; and in the feature `WITH_SERVER`, on lines 273 and 281. In our example, consider the test case `set_ops` from the file `connection.c`, presented in Fig. 7. From this test case, Table 1 presents an excerpt of the CSV document generated by our approach. The test case `set_ops` has traceability to the files `sample.c`, `error.c` and `options.c`. Moreover, it is possible to trace specific functions and lines. Fig. 7 presents an example of the traceability between the test file `connection.c` and the file `options.c`, showing the traceability at the function and feature levels. In Table 1 we can see that the function `ssh_options_getopt` corresponds to the lines 933 to 1102 in the test file `connection.c`, where there is a call to this function on line 12. Thus, there is a traceability between the test case `set_opts` and the function `ssh_options_getopt`.

To illustrate feature traceability, we also use the granularity of lines of code and the feature `WITH_SSH1` as an example. Table 2 presents an excerpt of the CSV file generated by our approach. We can see that `WITH_SSH1` is found between lines 947 to 949 in the file `options.c`. Thus, to know exactly which test cases touch which feature, we use the

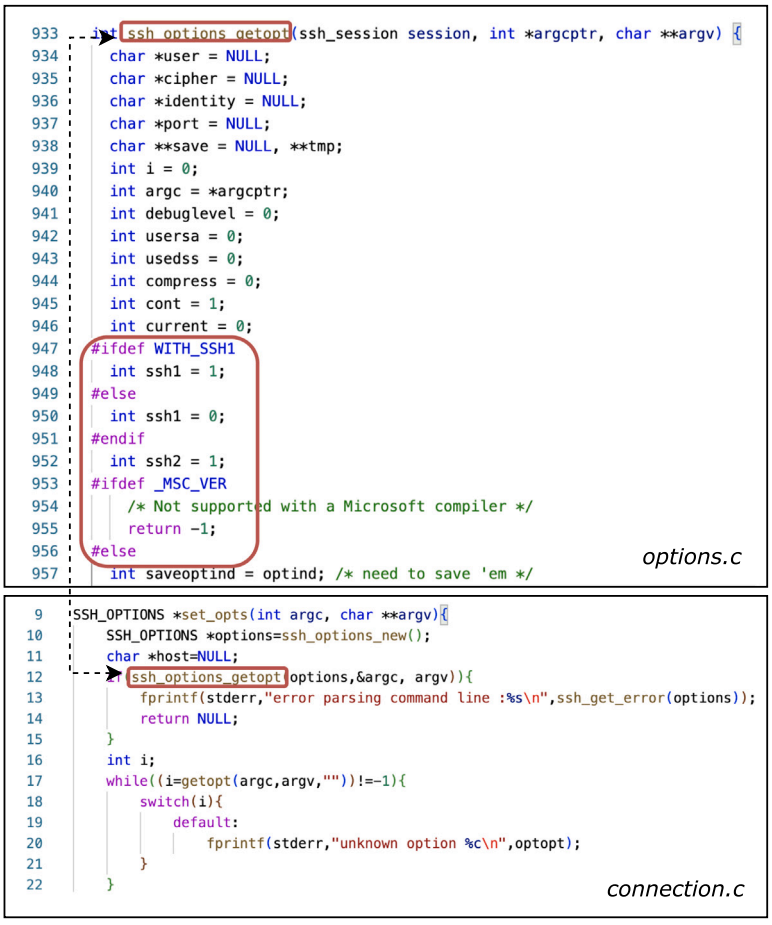


Fig. 7. Traceability between test cases and feature.

Table 1
Traceability of Test Cases to Source Code Lines.

Test file	Test case	Target file	Target function	Line from	Line to
connection.c	set_opts	sample.c	host	39	39
connection.c	set_opts	error.c	ssh_get_error	109	113
connection.c	set_opts	options.c	ssh_options_getopt	933	1102

Table 2
Traceability of Test Cases to Features.

Target file	Feature name	Feat from	Feat to
src/client.c	WITH_SSH1	340	343
src/channels.c	WITH_SSH1	1292	1298
src/channels.c	WITH_SSH1	1655	1661
src/options.c	WITH_SSH1	947	949
src/channels1.c	WITH_SSH1	41	389
src/server.c	WITH_SSH1	344	348

two CSV files, performing a merge of the related lines of code. We can see in Fig. 7 the `#ifdef` corresponding to the feature `WITH_SSH1` and check this feature is inside the function `ssh_options_getopt` and the test case `set_opts` touches this function. Then, in the case of a change in this feature, this test case must be selected. In addition to this, we can observe in Fig. 7 the feature is between the range of the function `ssh_options_getopt` that covers lines 933 to 1102. Table 3 presents the CSV file generated at the end of Step 4 where we can see in the test case `set_opts`.

In Step 5, *Prioritize Test Case*, we count the number of features covered by the test cases. Using Step 3 of `FeaTestSel`, which maps features to test cases, we can calculate this coverage, updating it with each evolution of the HCS, i.e., commit by commit. Table 4 provides a partial record of the results of this strategy. In this case, it can be observed that, in the first three rows, the test cases cover exactly eight features. In the range from line four to ten, test cases cover three features. The next test cases in the rank are not presented in the table, but cover a lower number of features. The test cases ranked in the end of the list do not cover any feature.

5. Evaluation setup

This section presents details of the evaluation of `FeaTestSel-Prio`. The main goal is to evaluate the applicability of our approach and compare their results with a changed-file-oriented approach, having the retest-all technique as baseline.

5.1. Research questions

Based on the evaluation goal, we formulated four *Research Questions (RQ)*:

RQ1: *How is the performance of `FeaTestSel` when compared to the performance of the retest-all and changed-file-oriented approaches?* This RQ focuses on the test case selection. We compare `FeaTestSel` with a changed-file-oriented approach, which selects test cases associated to the files changed in the commit (see Section 2). The analysis also considers the time taken to perform the selection and the percentage of reduction in the number of test cases in comparison to executing all the test cases available for the commit (retest-all technique).

RQ2: *What is the failure-detection quality of the test cases when `FeaTestSel` is used?* This question investigates whether by reducing the

Table 3
Partial Set of Test Cases Selected by FeaTestSel.

Test File	Test Case	Target file	Target function	Line from	Line to	Feature name	Feat from	Feat to
connection.c	set_opts	options.c	ssh_options_getopt	933	1102	WITH_SSH1	947	949
test_exec.c	do_connect	channels.c	ssh_channel_open_session	920	936	WITH_SSH1	925	929
test_exec.c	do_connect	channels.c	ssh_channel_request_exec	2382	2428	WITH_SSH1	2395	2399
bench_raw.c	upload_script	channels.c	ssh_channel_open_session	920	936	WITH_SSH1	925	929
bench_raw.c	upload_script	channels.c	ssh_channel_request_exec	2382	2428	WITH_SSH1	2395	2399
bench_raw.c	benchmarks_raw_up	channels.c	ssh_channel_open_session	920	936	WITH_SSH1	925	929
bench_raw.c	benchmarks_raw_up	channels.c	ssh_channel_request_exec	2382	2428	WITH_SSH1	2395	2399
bench_raw.c	benchmarks_raw_down	channels.c	ssh_channel_open_session	920	936	WITH_SSH1	925	929

Table 4
Example — Feature Coverage.

Index	TestFile	TestCase	FeatureNameList	Count
1	torture_pki.c	torture_pki_generate_key_dsa	['WITH_SSH1', 'WITH_PCAP', 'HAVE_ECC', 'HAVE_OPENSSL_ECC', 'HAVE_LIBCRYPTO', 'DEBUG_CRYPTO', 'BASE', 'HAVE_LIBGCRYPT']	8
2	torture_pki.c	torture_pki_generate_key_rsa	['HAVE_ECC', 'WITH_SSH1', 'HAVE_LIBGCRYPT', 'WITH_PCAP', 'HAVE_LIBCRYPTO', 'BASE', 'DEBUG_CRYPTO', 'HAVE_OPENSSL_ECC']	8
3	torture_pki.c	torture_pki_generate_key_rsa1	['WITH_PCAP', 'BASE', 'HAVE_LIBCRYPTO', 'DEBUG_CRYPTO', 'WITH_SSH1', 'HAVE_ECC', 'HAVE_LIBGCRYPT', 'HAVE_OPENSSL_ECC']	8
4	connection.c	set_opts	['WITH_SSH1', 'BASE', '_MSC_VER']	3
5	test_socket.c	main	['WITH_SSH1', '_WIN32', 'BASE']	3
6	torture.c	torture_ssh_session	['WITH_SSH1', 'WITH_PCAP', 'BASE']	3
7	torture_keyfiles.c	torture_privatekey_from_file	['HAVE_LIBGCRYPT', 'HAVE_LIBCRYPTO', 'BASE']	3
8	torture_keyfiles.c	torture_privatekey_from_file_passphrase	['BASE', 'HAVE_LIBCRYPTO', 'HAVE_LIBGCRYPT']	3
9	torture_keyfiles.c	torture_pubkey_generate_from_privkey	['HAVE_LIBCRYPTO', 'HAVE_LIBGCRYPT', 'BASE']	3
10	torture_knownhosts.c	torture_knownhosts_port	['WITH_PCAP', 'BASE', 'WITH_SSH1']	3
...	...	...	...	...

number of test cases, it is still possible to maintain the test quality in terms of detected failures. We then analyze the number of failures detected by the test cases selected by `FeaTestSel` and by the changed-file-oriented approach in comparison with the number of failures detected when all test cases available are executed (retest-all technique).

RQ3: *Is `FeaTestSel` applicable considering budget constraints of industrial HCSs?* This question aims to assess the applicability of our test case selection approach by analyzing the time between CI cycles, with the execution time of the selected test cases summed to the time spent to perform the selection. We want to check whether the execution of the selected test set generated by our approach implies in a reduced time to execute the tests. Moreover, we are interested in the time our approach takes to execute. If such time is too long, the use of `FeaTestSel` is impracticable in a CI environment.

RQ4: *What is the performance of `FeaTestSelPrio` for early fault detection?* In this RQ, we evaluate the performance of the prioritization step of our approach. Using an indicator for early fault detection, we compared the test case order produced by `FeaTestSelPrio` with the order produced by `FeaTestSel`. In fact, `FeaTestSel` does not produce any order, but the result of our implementation is given according to the order of the test cases appear in the directory. The order in which files are returned depends on the specific implementation of the underlying file system. However, the order is generally alphanumeric, that is, the files are returned in lexicographic order. Moreover, we also evaluated the execution time for the prioritization step of `FeaTestSelPrio`.

5.2. Target systems and data collection

Our evaluation relies on two systems, namely `Libssh` and `Libsoup`, which further details are presented in Table 5. These systems are also used in the HCS literature (Medeiros et al., 2018; Oliveira et al., 2019). `Libssh`[10] is an open source cross-platform SSH library in C that implements the SSHv2 protocol on the client and server side. This library is designed to remotely run programs, transfer files, use a secure and transparent tunnel, manage public keys, to cite some of its features. `Libssh` is an HCS statically configurable with the C pre-processor directives. `Libsoup`[11] is a client/server HTTP library developed for the GNOME environment. Utilizing GObjects and a simplified main

[10] https://www.libssh.org/
[11] https://wiki.gnome.org/Projects/libsoup

Table 5
Target Systems Information.

Systems	Features	Total commits	Commits used	Commits with logs	LOC	Contributors
Libssh	144	5,958	4,388	303	85,817	110
LibSoup	29	3,896	3,420	–	64,265	247

loop, it seamlessly integrates with GNOME applications and provides a synchronous API designed for ease of use in command-line interface (CLI) tools. Its key features include asynchronous APIs, automatic connection reuse, TLS support, proxy functionality, compatibility with HTTP/1.1 and HTTP/2, and Client support for Digest, as well as Server support for Digest.

We used PyDriller (Spadini et al., 2018) to mine the commit history of the HCS and find the files changed in each commit. These files are the source of information to identify features changed (Step 2: Identify feature changed) and also to create the traceability between such files and test cases. Based on these trace links, we selected the test cases that are used by the changed-file-oriented approach. For that, the source code is scanned, looking for the test files.

For the data collection, `Libssh` repository contains almost six thousand commits, from which we used 4388. We discarded commits not associated with test cases and commits that are fork outside the repository. This set will be referred as the *whole set of commits*. To evaluate the quality of prioritized test cases, we need logs with failure and runtime information. To this end, we used a repository containing 303 commits from a related work (Lima et al., 2020), referred as *set of commits with logs*. The granularity of the logs is file-level, which gives us information about which test files reveal failures, called herein as failed files. For `Libsoup`, we could not obtain information about the logs, which did not allow us to know which test cases failed in each CI cycle. Thus, for `Libsoup` we used 3420 commits in our experiments, out of almost four thousand commits, equivalent to the *whole set of commits*. We also discard commits that do not have associated test cases and commits that are fork outside the repository. Due to the impossibility of obtaining CI logs for `Libsoup`, only `Libssh` was used in the analyses to answer RQ2 to RQ4, which require the execution time of each test case, as well as whether it failed, information included in the commits with logs.

5.3. Evaluation measures

To compute the average execution time of each test case, an approximation is necessary, since the evaluated approaches use function-level granularity (i.e., a test case corresponds to a function) and the system logs use file-level granularity. Thus, for each CI cycle, there is an execution time per test file for each job of that cycle. To perform this approximation, we first mined the number of functions each test file has and the average time this file takes to execute. We then divided the average time by the number of functions to compute the average time each test case (i.e., function) takes to run. For example, if a file takes an average of 100 s to execute in a commit and has 10 functions, each function takes an average of 10 s to execute. Then, we compared how many functions were selected by each approach for each test file. For example, if our approach selects 5 functions from these files, it would take 50 s on average to execute, which represents a 50% reduction in time compared to the execution time of the complete file.

To validate the quality of the approaches, we use the criterion based on detected failures. To this end, we mined the logs to identify test files failed in the failed commits. For instance, the commit `10728f85`[12] has three failed test files: `torture_packet`, `torture_algorithms`, and `pkd_hello`, thus, three failures are counted. We perform an analysis per-commit to verify whether the test cases selected by both approaches cover the failed test files. For example, considering the previous commit, if the approaches select test cases from the three files, the three failures are considered as detected. However, if only the `pdk_hello` file is affected, then only one failure is detected.

To answer RQ4, we adopted an early failure detection indicator. This indicator quantifies the average *test budget* required, that is, the percentage of test cases to be executed to find a failure in a commit. To identify each test case that fails in the commit, we rely on the system log. We compute the average budget required per commit because for each commit pipeline, there can be multiple jobs. Consequently, test cases run in parallel across these jobs, allowing different test cases to fail in the same commit.

Eq. (1) presents how we compute the early failure detection indicator. The average budget required to identify the failure is determined by γ, providing the average percentage needed to locate the test cases that fail in the commit. This equation is based on the sum of the position (P_i) of the test cases in the prioritized list, where nt is the total number of test cases for the commit, and nf is the total number of test cases that fail in the commit. γ represents the average budget per commit to identify the fault, considering all jobs.

$$\gamma = \frac{\sum_{i=1}^{nf} P_i \mathrm{nt}}{100 * nf} \tag{1}$$

For example, in the commit `17b518a`, there are 27 jobs, and failures occurred in two of these jobs. In the job `visualstudio/x86_64`,[13] the test case `torture_rekey` failed. Meanwhile, in the job `ubuntu/openssl_1.1.x/x86_64`,[14] the failure occurred in the test case `torture_misc`. To locate the `torture_misc` test file in this commit, it is necessary to execute 34 test cases from the total set of 638, which is approximately 5.3%. For the `torture_rekey` file, approximately 2.1% (13) of test cases are required. Therefore, to assess the commit as a whole, we compute the mean percentage, resulting that in average, 3.7% of test cases need to be executed to find both failures.

5.4. Platform for the experiments

All experiments were conducted on a server equipped with an Intel(R) Xeon(R) Gold 6230N CPU @ 2.30 GHz 48-Cores, 252 GB RAM, running on Linux Ubuntu 18.04.6 LTS.

6. Results and analysis

This section presents the results and discussions to answer the RQs of our study. The dataset and results are available online (Mendonça et al., 2024), as a supplementary material.

6.1. RQ1: Performance of `FeaTestSel` and changed-file-oriented approaches

To answer RQ1, we compare the `FeaTestSel` and changed-file-oriented approaches against retest-all, regarding the number of selected test cases. In this comparison, we analyze the percentage of reduction in the number of test cases and execution time. As both `FeaTestSel` and changed-file-oriented approaches consider function granularity, we count the number of test functions selected, as mentioned in Section 5.3. We consider the number of test cases to be executed by the retest-all strategy is given by the number of functions existing within all the files in the logs. Then, each function is considered a test case.

Table 6 presents, for both approaches, the percentage of reduction in the number of test cases in comparison to the retest-all technique,

[12] See Log Line 1130 at https://gitlab.com/libssh/libssh-mirror/-/jobs/78303050.

[13] https://gitlab.com/libssh/libssh-mirror/-/jobs/432862254

[14] https://gitlab.com/libssh/libssh-mirror/-/jobs/432862274

Table 6
Percentage of Reduction in the Number of Test Cases in Comparison to Retest-All and Time to Perform TCS — FeaTestSel and Changed-file-oriented Approach.

System	FeaTestSel						Changed-File Oriented Appr.					
	% Test reduction			Time (seconds)			% Test reduction			Time (seconds)		
	min	max	avg	min	max	avg	min	max	avg	min	max	avg
*Libssh**	0	45.29	24.22	9.68	35.72	20.82	27.5	100	94.23	2.01	21.04	8.73
Libssh$	37.74	45.29	41.98	18.61	35.72	25.97	51.11	100	97.27	6.77	18.41	11.98
LibSoup	0	39.06	18.62	4.15	37.68	11.47	0	100	91.68	0.03	8.40	4.83

* whole set of commits; $ set of commits with log

and time taken to perform the selection (in seconds). For Libssh, we consider both sets of commits (with or without logs). The minimum reduction percentage for both systems is 0, which occurred in the initial commits where the number of test cases is small, and in several commits, it is not possible to reduce the set because usually 100% of the test cases are selected. We observe that the percentage of test reduction reached by FeaTestSel vary according to the system and scenario considered. For Libsoup the percentage is lower. A possible explanation for this is on the number of features. Libsoup is smaller and has a reduced number of features, only 29, against 144 of Libssh (see Table 5). In this case, many test cases are associated with the feature BASE and are selected in the majority of commits.

The reduced number of features also implies a shorter time to perform the selection (11.47 s), making the feature location step less time-consuming. But this difference does not grow proportionally. Libssh has almost five times the number of features than LibSoup, but it requires only the double of time to execute (20.82 s).

Another point to be evaluated is the percentage of reduction considering the set of commits evaluated. We observe for Libssh that the percentage of reduction is greater when the set of commits with log is considered. In such scenario, there is no case where the percentage is zero. The 303 commits with log does not belong to the set of initial commits. These commits are in the middle of the evolution process of the system. When the system evolves and becomes more complex (in the last commits) there are a greater number of test cases, then the results of using a TCS approach are more significant. We also observe this difference regarding the set of commits considered for the changed-file-oriented approach.

The average percentage reduction in test cases for our approach, considering all three configurations (i.e., *Libssh**, *Libssh*$, and *LibSoup*), is equal to 21.41%, whereas for the changed-file-oriented approach, it is 92.95%. Notice here that at this point we are not considering the quality of test cases selected. The average runtime for our approach is 16.14 s, whereas for the changed-file-oriented approach, it is 6.78 s. As expected, our approach takes longer to execute due to feature to test case traceability step.

To analyze the behavior of the approaches over the commits, we present the absolute number of test cases selected for each commit in Fig. 8. The blue line represents the number of test cases available for the commit (retest-all technique); the green line represents the test cases selected by our approach; and the orange line represents the number of test cases selected by the changed-file-oriented approach. We can observe that for both systems, the changed-file-oriented approach selects fewer test cases than FeaTestSel in the great majority of commits. However, in some instances, it does not select any test case, as indicated by the orange line in the figure.

To better compare both approaches and consider the execution time of each test case, we use the system Libssh considering the 303 commits with logs. We first generate Fig. 9. that shows the number of test cases selected by each approach in each commit. The blue line shows the number of test cases available for the commit (retest-all); the green, the test cases selected by our approach; and the orange, the number of test cases selected by the changed-file-oriented approach. For this set, we observe that the file-oriented approach always selects fewer test cases than FeaTestSel. However, in several commits, it does not select any test cases, as indicated by the orange line in the figure. This happens in 199 out of 303 commits (≈65.67%). In other commits a few test cases are selected, in fact the number of test cases selected depends on the performed changes. For example, in commit 12284b75,[15] which involves changes in six files, only eight test cases were selected from a single test file, out of a total of 329 test cases available. But there are cases where a significant number of changes are made, and many test cases are selected, as it happens in commit 17b518a6.[16] In this commit seven files are changed (245 additions and 13 deletions), and this approach selected 334 out of 719 test cases. This does not happen for our approach, which selects test cases in all commits.

When we apply a TCS approach, we may be looking for a way to reduce the time spent executing the test cases. Fig. 10 shows the average time spent per commit with the execution of the test cases selected by each approach. We observe that the orange line is always lower, showing that the test set selected by the changed-file-oriented approach always takes less time to execute. The average time to execute all test cases (retest-all) is 239.22 s. The time to execute the test cases selected by FeaTestSel is 92.78 s, and the time to execute the ones selected by changed-file-oriented approach is 4.15 s. We can observe that this time reduction is quite drastic, what may affect the quality of test cases for failure detection, subject investigated in RQ2.

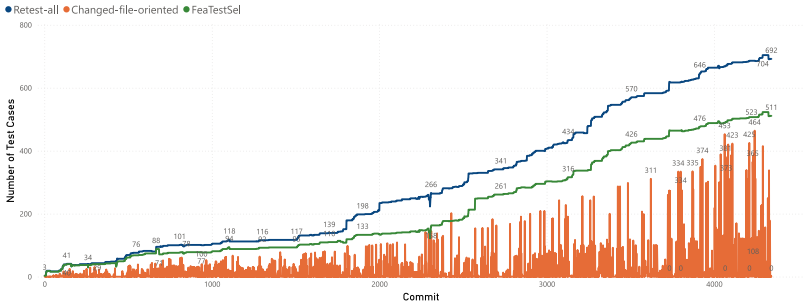
RQ1: *The changed-file-oriented approach leads to a expressive reduction in the number of test cases compared to our approach (average of 92%), as well as in the time to execute the selected test cases (4.15 s). This reduction depends on the changes in the commit, and the evolution process. The reduction percentage achieved by our approach (average of 21.41%) varies based on the system's size, number of tests available in the commit (evolution process), and number of features. In the set of commits with logs, this percentage reaches ≈42%, and the mean time to execute the selected test cases is 92.78 s.*

6.2. RQ2: Quality of selected test cases for failures detection

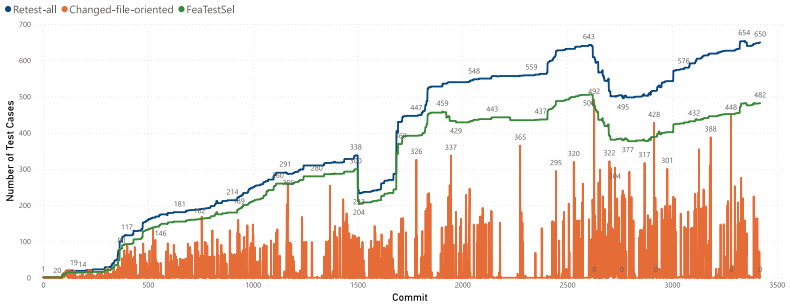
This RQ evaluates the quality of the test cases selected by FeaTestSel against the other approaches. The goal is to analyze if the reduction in the test cases impacts the number of failures detected. To this end, we use the set of Libssh commits with logs.

Fig. 11 presents the number of failures detected with the test cases selected. We can observe that the curves corresponding to our approach (green line, at the top of the figure) and retest-all (blue line, at the bottom) are exactly the same, which does not happen for the changed-file-oriented approach (orange line, in the middle). This means that the changed-file-oriented approach has a poor performance for selecting test cases that detect failures. This happens because the changed-file-oriented approach selects only a small set of test cases related to the files changed, as showed in the previous section. Despite leading to an expressive reduction of the number of test cases selected and execution time, these test cases do not maintain the quality of the testing activity.

[15] https://gitlab.com/libssh/libssh-mirror/-/commit/12284b75
[16] https://gitlab.com/libssh/libssh-mirror/-/commit/17b518a6



(a) *Libssh**



(b) LibSoup

Fig. 8. Number of test cases selected by the approaches for the whole set of commits.

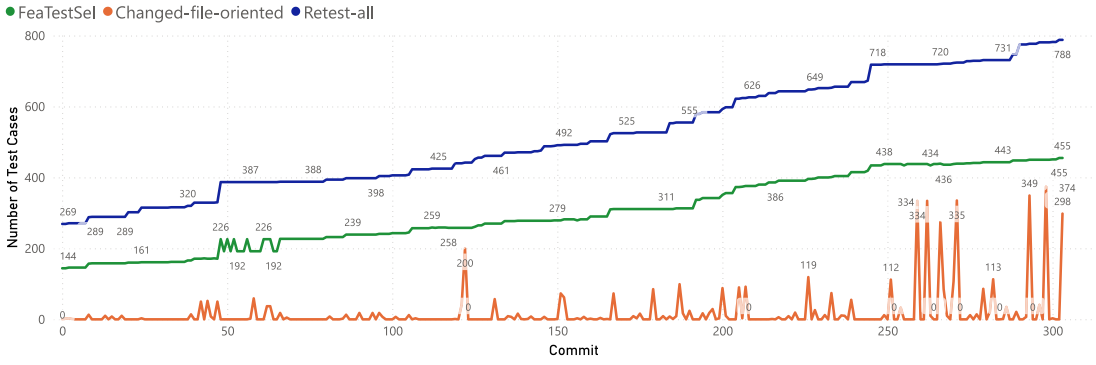


Fig. 9. Number of test cases selected by the approaches for the set of commits with logs — *Libssh*^{S}.

Fig. 10. Average time to execute the selected test cases per commit and approach for the set of commits with logs — *Libssh*^{S}.

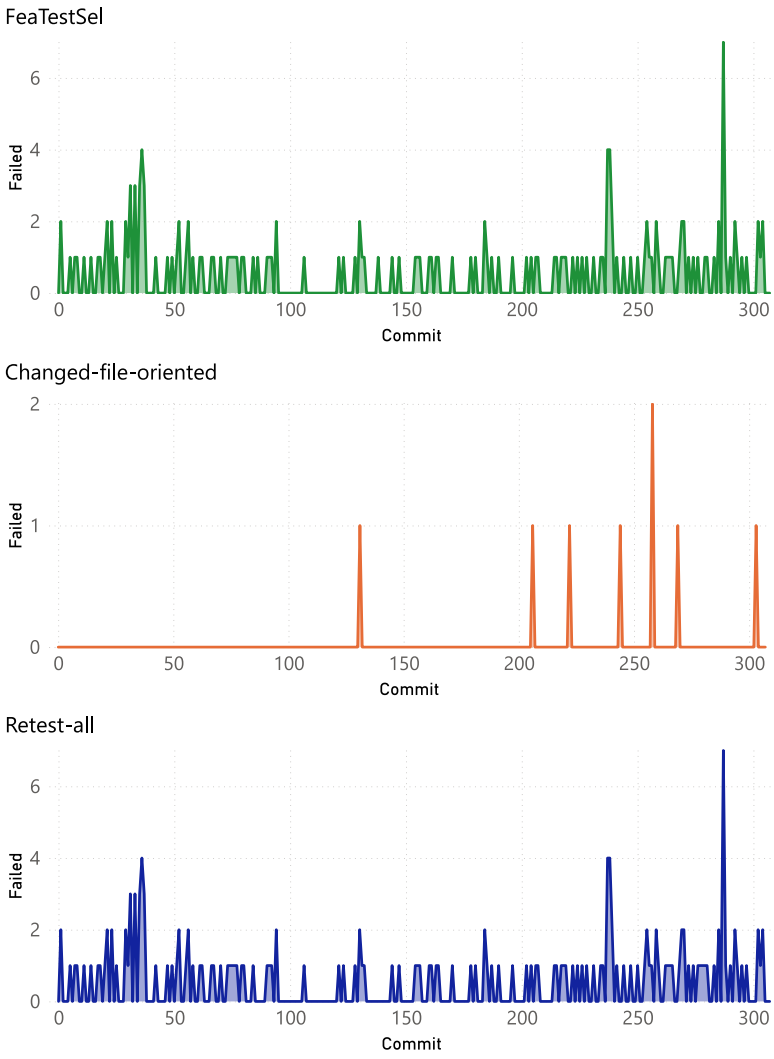


Fig. 11. Number of failed files per commit and approach for the set of commits with logs.

To complement the analysis, we created a dynamic chart[17] considering the 303 commits with logs. Fig. 12 shows a clipping of the graph for the commit `d7477dc`.[18] The graph is separated into three main lines: the first line represents all test cases; on the left part of the second line the failed test cases (false), and on the right of this line the test cases that passed (true); and the third line represents the test cases selected by our approach. In this commit, 162 test cases were selected by our approach from a total of 316 available. We can see for this commit that all the failed test cases were selected by our approach, so we are keeping the quality, considering the failure criterion. We can see similar behavior for the other commits in the dynamic chart made available.

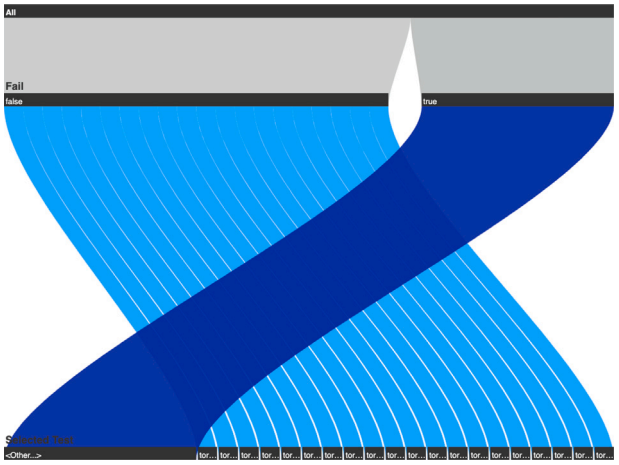


Fig. 12. Failed and selected test cases for commit `d7477dc`.

RQ2: *Based on the failure criterion and considering the retest-all technique baseline, our approach manages to always select (i.e., 100% of the times) the test files that reveal failures, maintaining the quality of the test case set. This does not happen for the changed-file-oriented approach.*

6.3. RQ3: `FeaTestSelPrio` applicability

This RQ investigates the applicability of our approach, considering the time available in the CI cycles. To answer RQ3, we need the

[17] Available at https://app.powerbi.com/view?....
[18] https://gitlab.com/libssh/libssh-mirror/-/commit/d7477dc

Table 7
Comparison between FeaTestSelPrio and FeaTestSel regarding the number of test cases required (budget) to detect known failures.

Strategies	Avg	Max	Min	SD
FeaTestSel	44.99%	94.31%	0.67%	34.68
FeaTestSelPrio	23.4%	71.82%	0.56%	16.37

time each test case takes to execute, thus, the results rely only on Libssh and its 303 commits with logs. For this analysis, we evaluate FeaTestSel applicability considering the time it takes to execute, summed to the time to execute the selected test cases.

As mentioned in the last section, the average time to execute all the test cases is 239.22 s (≈4 min). The test cases selected by FeaTestSel take on average 92.78 s to run. By adding to this time the average time our approach takes to perform the selection (25.97 s), the runtime is 118.75 s (≈2 min). This represents a reduction of ≈50% in the total runtime compared to the retest-all technique. We also observe that the interval between the CI cycles is on average 1142.52 min (standard deviation equals to 459.28 min), what shows the applicability of our approach.

RQ3: *When using the Libssh commits with logs, the average time to perform the selection is 25.97 s. Summing this time to the time spent to execute the selected test cases, the total is 118.75 s. This represents a reduction of ≈50% compared to retest-all. This time is compatible with the time between CI cycles, what shows the applicability of FeaTestSel.*

6.4. RQ4: Evaluating FeaTestSelPrio

To answer RQ4, we evaluate FeaTestSelPrio in comparison to FeaTestSel considering early failure detection. For this end, we use the set of commits with log of Libssh and the budget required to detect a failure (see Section 5.3). As mentioned before, FeaTestSel does not produce any order, then we execute the selected test cases according to the order they appear in the output of our implementation.

In Table 7, we can observe that the average budget value of FeaTestSelPrio is almost half of the budget value of FeaTestSel. The difference in the minimum values is very small, but it is significant in the maximum ones. Also, FeaTestSelPrio has a lower standard deviation (SD) than FeaTestSel, which represents that the set of ordered test cases improves the dispersion of the budgets necessary to the failure.

The results indicate that the use of FeaTestSelPrio leads to an early detection of failures with lower budgets on average. To corroborate our analysis, we compared the FeaTestSel and FeaTestSelPrio approaches according to the number of tests executed until a failure is produced, and generated Fig. 13, which displays a graph for the 122 failed commits from the total of 303 commits. The green line presents the number of test cases executed by FeaTestSel approach and the blue line by FeaTestSelPrio, which is mostly below the green line for all commits. It is possible to identify the failure, that is, to reduce budget, with a smaller number of test cases using FeaTestSelPrio in 86% of the commits.

The mean time to perform the prioritization step of FeaTestSelPrio is 0.42 s (minimum of 0.37 s, maximum of 0.74 s, and standard deviation equal to 0.03 s). This time summed to the time FeaTestSel takes to execute is equal to 26.40 s. Finally, we have to sum the time to actually execute the selected test cases, resulting in a total of 98.78 s, which is the worst case in the prioritization rank. This total time of 98.78 s is acceptable for practical application of our approach, considering the duration of the CI cycles.

RQ4: *The results of this RQ highlight the importance of the additional prioritization step in nearly every commit. The approaches FeaTestSel and FeaTestSelPrio identify a fault with a mean budget of, respectively, ≈44% and ≈23%. The incorporation of a prioritization step results in a reduction of ≈50% in the average budget required. Furthermore, FeaTestSelPrio allows fault detection earlier than FeaTestSel in ≈86% of the faulty commits, requiring an additional mean time of 0.42 s.*

7. Discussion

In this section, we discuss some implications as well as the threats to validity and limitations of our approach and evaluation.

7.1. Implications for the research and practice

FeaTestSelPrio is lightweight. Differently from other TCS approaches, our approach does not require neither a failure-history nor the application of search-based/learning techniques, what leads to a reduced time to perform the selection and prioritization. It is noteworthy to observe that the approach produces an entire static analysis of the system for each commit, delivering results that can be analyzed qualitatively and/or quantitatively by developers for possible improvements in the system. For example, developers can use the test case traceability to feature to identify features that need more test.

By analyzing the whole set of commits, we observe that the time the approach takes to execute is dependent on the size of the system and number of test cases, as well as the percentage in the test reduction. This implies that the approach is more beneficial as the system evolves and a greater number of test cases are in the repository. However, this relationship should be better explored in future works.

In comparison with a changed-file-oriented approach, FeaTestSel presents some advantages, obtaining a good balance in the test set reduction, making sure that some important test cases are selected in the regression testing, maintaining quality regarding failure detection. The reduction provided by the changed-file-oriented approach is very dependent on the number of changes in the commit. When a CI environment is adopted in the development, it is a very common practice to perform small changes and push them to the repository. This may be the reason why the changed-file-oriented approach usually selects a small test set, or even no test cases at all. This makes our approach more suitable in this scenario.

We observe that our approach takes only few seconds to execute, and is suitable for the CI context. The time of the test cases selected take to execute summed to the time taken to perform the selection, or the selection and prioritization, is lower than the CI cycles, what shows the applicability of the approach. FeaTestSel contributes to select a reduced number of test cases, and consequently reduces the test execution time. We observe that it fits in a budget of 50% of time that would be spent by executing all the available test cases for the commit without loosing quality regarding possible failures produced. However, as RQ4 pointed out the use of FeaTestSelPrio is very important in the presence of a time budget as it happens in the CI scenarios, allowing an early fault-detection and a rapid feedback to the tester. In this work, we explored only a prioritization strategy based on the feature coverage. But other strategies could be explored in future work. Such strategies can explore other test characteristics such as test complexity, user, preferences, severity of the failures they reveal, feature-change history, and so on.

A limitation for the use of our approach is that our implementation requires as input HCSs developed using pre-processor directives (i.e., an annotative approach). Despite this being a limitation, the use of pre-processor directives is a very common practice, adopted by most HCSs (Medeiros et al., 2018). However, using other variability mechanisms such as component technologies (Szyperski et al., 2002),

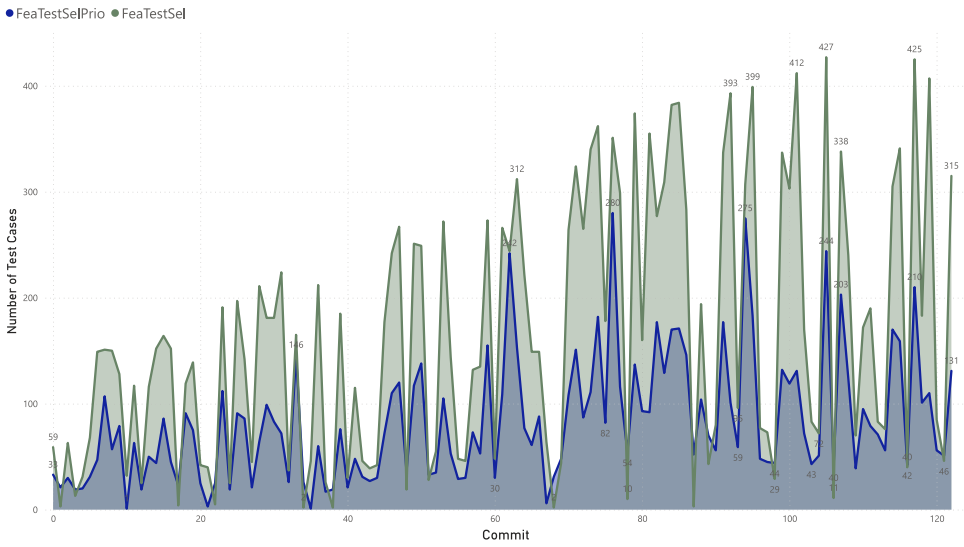


Fig. 13. Comparing approaches according to number of test cases executed to find a failure in the failed commits.

frameworks (Johnson and Foote, 1988), or feature-oriented programming with some form of feature modules (Prehofer, 1997) (i.e., a compositional approach) should be the focus of future work (Moreira et al., 2022).

7.2. Threats to validity

In this section, we discuss some treats to the validity of our study according to classification by Wohlin et al. (2000).

Internal Validity: We faced a problem when dealing with code scanning tools, which often use global variable names in their traceability. This lead to traceability false positives between global and local variables with the same name. To resolve this issue, we dropped all global variables during the *Map feature to Test Cases* step. To further improve this, we intend to completely eliminate the consideration of global variables in traceability during the code scan task. Furthermore, in the *Prioritize test cases* step, we consider the count of the number of features covered by the test case. However, in cases of a tie between test cases, the choice is currently made randomly. We plan to incorporate tie-breaking strategies, such as considering feature changes and the feature change history, to enhance this process. Moreover, it should be noted that the `Libssh` system logs use a file granularity, which led us to calculate an approximated value for the execution time of each test function in the test files. This may affect the accuracy of the test execution time calculation, but we still ensure that the total test suite is reduced while maintaining failure-based quality.

Construct validity: A threat in this category is the approach used in the comparison. As we did not find approaches or tools available for the C/C++ language, we used a simple but well-known changed-file selection approach that we developed internally as a basis for comparison. Possible errors in the implementations were minimizing by a manual validation of the results produced. Additionally, for the prioritization step, we performed a comparison with `FeatTestSel`, which produces a list of selected test cases in alphanumeric order. This could potentially impact the results. Furthermore, experiments should be conducted to evaluate alternative prioritization strategies.

External validity: We used only two systems in our evaluation, and only `Libssh` for answers the RQs that needed error log of the test cases. This is a threat, and our results may not be generalized. For future work, other industrial-grade systems should be included in the validation. Our implementation currently works for different annotated systems. But we have not found systems with logs. To minimize this threat, future work can execute other subject systems to report the test execution results, improving generalization.

Conclusion validity: a possible threat in the analysis conducted in our study is the indicators used, which may impact the results. To minimize this, we adopted indicators that are relevant for evaluating test case selection and prioritization techniques, and are related to reduction in the number of test cases and early-fault detection.

Reliability validity is concerned with reproducibility. We believe our study is replicable by following the steps outlined in Section 5, and we have made all raw results and logs available in our repository.

8. Concluding remarks

This work introduces `FeaTestSelPrio`, a feature-oriented test case selection and prioritization approach to be applied during the evolution of HCSs. Unlike other selection approaches that mainly focus on test cases directly related to changed files without considering features, our approach takes into account interactions with other parts of the features being modified in a given commit. The prioritization step uses the feature covered to rank test cases. Although we use system logs to validate our approach, we do not need access to any type of failure log or failure history for execution. Differently from several exiting prioritization approaches, the failure-history or dynamic analysis is not required. `FeaTestSelPrio` uses as input basically only the source code of the HCS and a configuration file as input. The approach produces several intermediate reports as output, which can be used by software engineers to make decisions for maintenance and/or upgrades for their HCSs. These reports allow an overview of various aspects of the systems, which are not possible to be analyzed only using the source code.

In the analysis of the set of commits with logs, our feature-oriented selection reaches an average reduction in the number of test cases of ≈42%. The time it takes to execute summed to the time to execute selected test cases fits well in a budget of 50%. These percentages depend on the system size, number of features and test cases, being more significant in the last commits, when the system evolved and has a greater number of test cases. This reduction does not imply loosing important test cases because the approach manages to select 100% of the failed test files, maintaining the test quality. `FeaTestSelPrio` is applicable considering CI cycles. When the set of commits with logs from `Libssh` is considered, the average time to perform the selection and prioritization is 26.40 s. This time, when summed to the time spent in the worst case to execute all the selected test cases, is 119.75 s. To execute the prioritization step of `FeaTestSelPrio` an average time of only 0.42 s is required.

Overall, our approach selects a greater number of test cases and takes longer to execute than a changed-file-oriented approach, but we observe a greater quality regarding the failures detected. We also observe the importance of the prioritization step that allows a reduction in the average budget in 86% of the failed commits.

Future work includes to obtain test logs of other systems and conduct other experiments. We intend to better evaluate the relationship between the system size and the reduction in the number of test cases, as well as execution time. Furthermore, we intend to improve the selection by adding some criteria, such as changes in files along with changes and features. Additionally, we should consider using Artificial Intelligence algorithms for a potential comparison and/or improvement of the results. Another research direction involves to propose and evaluate other prioritization strategies, which are also feature-oriented, but that consider detected and severity failures.

CRediT authorship contribution statement

Willian D.F. Mendonça: Visualization, Validation, Software, Methodology, Investigation, Data curation, Conceptualization. **Wesley K.G. Assunção:** Writing – review & editing, Writing – original draft, Visualization, Validation, Supervision, Methodology. **Silvia R. Vergilio:** Writing – review & editing, Writing – original draft, Visualization, Validation, Supervision.

Declaration of competing interest

The authors declare that they have no financial or non-financial conflict of interest that are directly or indirectly related to this work submitted for publication.

Data availability

The data is shared in the OSF repository: https://osf.io/wgk7c.

Acknowledgment

This research was funded by CNPq, Brazil (Grant 310034/2022-1), FAPERJ PDR-10 program, Brazil (Grant 202073/2020), and CAPES, Brazil (Grant 88887.464736/2019-00).

References

Al-Hajjaji, M., Lity, S., Lachmann, R., Thüm, T., Schaefer, I., Saake, G., 2017. Delta-oriented product prioritization for similarity-based product-line testing. In: 2nd Intl. Workshop on Variability and Complexity in Software Design. VACE, IEEE, pp. 34–40.

Benavides, D., Segura, S., Trinidad, P., Ruiz-Cortés, A., 2006. Using Java CSP solvers in the automated analyses of feature models. In: Intl. Summer School Generative and Transformational Techniques in Software Engineering. Springer, pp. 399–408.

Bertolino, A., Guerriero, A., Miranda, B., Pietrantuono, R., Russo, S., 2020. Learning-to-rank vs ranking-to-learn: strategies for regression testing in continuous integration. In: 42nd Intl. Conference on Software Engineering. pp. 1–12.

Clements, P., Northrop, L., 2002. Software Product Lines. Addison-Wesley Boston.

do C. Machado, I., McGregor, J.D., Cavalcanti, Y.C., De Almeida, E.S., 2014. On strategies for testing software product lines: A systematic literature review. Inf. Softw. Technol. 56 (10), 1183–1199.

Duvall, P.M., Matyas, S., Glover, A., 2007. Continuous Integration: Improving Software Quality and Reducing Risk. Pearson Education.

Ensan, A., Bagheri, E., Asadi, M., Gasevic, D., Biletskiy, Y., 2011. Goal-oriented test case selection and prioritization for product line feature models. In: 8th Intl. Conference on Information Technology: New Generations. IEEE, pp. 291–298.

Ferreira, F., Diniz, J.P., Silva, C., Figueiredo, E., 2019. Testing tools for configurable software systems: A review-based empirical study. In: 13th Intl. Workshop on Variability Modelling of Software-Intensive Systems. pp. 1–10.

Garousi, V., Zhi, J., 2013. A survey of software testing practices in Canada. J. Syst. Softw. 86 (5), 1354–1376.

Ghanam, Y., Maurer, F., 2010. Linking feature models to code artifacts using executable acceptance tests. In: International Conference on Software Product Lines. Springer, pp. 211–225.

Gligoric, M., Eloussi, L., Marinov, D., 2015. Practical regression test selection with dynamic file dependencies. In: Intl. Symposium on Software Testing and Analysis. pp. 211–222.

Hajri, I., Goknil, A., Pastore, F., Briand, L.C., 2020. Automating system test case classification and prioritization for use case-driven testing in product lines. Empir. Softw. Eng. 25 (5), 3711–3769.

Heider, W., Rabiser, R., Grünbacher, P., Lettner, D., 2012. Using regression testing to analyze the impact of changes to variability models on products. In: 16th Intl. Software Product Line Conference. SPLC, pp. 196–205.

Jiang, B., Chan, W.K., 2016. Testing and debugging in continuous integration with budget quotas on test executions. In: IEEE Intl. Conference on Software Quality, Reliability and Security. In: QRS, IEEE, pp. 439–447.

Johnson, R.E., Foote, B., 1988. Designing reusable classes. J. Object-Oriented Program. 1 (2), 22–35.

Jung, P., Kang, S., Lee, J., 2019. Automated code-based test selection for software product line regression testing. J. Syst. Softw. 158, 110419.

Jung, P., Kang, S., Lee, J., 2020. Efficient regression testing of software product lines by reducing redundant test executions. Appl. Sci. 10 (23), 8686.

Jung, P., Lee, S., Lee, U., 2023. Automated code-based test case reuse for software product line testing. Inf. Softw. Technol. 107372.

Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, A., 1990. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Tech. Rep. CMU/SEI-90-TR-021, Carnegie-Mellon University: SEI, (Accessed: 31 January 2024).

Kim, C.H.P., Batory, D.S., Khurshid, S., 2011. Reducing combinatorics in testing product lines. In: 10th Intl. Conference on Aspect-Oriented Software Development. pp. 57–68.

Kim, C.H.P., Khurshid, S., Batory, D., 2012. Shared execution for efficiently testing product lines. In: IEEE 23rd Intl. Symposium on Software Reliability Engineering. IEEE, pp. 221–230.

Kumar, S., Rajkumar, 2016. Test case prioritization techniques for software product line: A survey. In: Intl. Conference on Computing, Communication and Automation. ICCCA, IEEE, pp. 884–889.

Lachmann, R., Beddig, S., Lity, S., Schulze, S., Schaefer, I., 2017. Risk-based integration testing of software product lines. In: 11th Intl. Workshop on Variability Modelling of Software-Intensive Systems. pp. 52–59.

Lachmann, R., Lity, S., Lischke, S., Beddig, S., Schulze, S., Schaefer, I., 2015. Delta-oriented test case prioritization for integration testing of software product lines. In: 19th Intl. Conference on Software Product Line. pp. 81–90.

Lima, J.A.P., Mendonça, W.D., Vergilio, S.R., Assunção, W.K., 2020. Learning-based prioritization of test cases in continuous integration of highly-configurable software. In: 24th ACM International Systems and Software Product Line Conference. SPLC, pp. 1–11.

Lima, J.A.P., Vergilio, S.R., 2020. Test case prioritization in continuous integration environments: A systematic mapping study. Inf. Softw. Technol. 121, 106268.

Lity, S., Nieke, M., Thüm, T., Schaefer, I., 2019. Retest test selection for product-line regression testing of variants and versions of variants. J. Syst. Softw. 147, 46–63.

Lochau, M., Schaefer, I., Kamischke, J., Lity, S., 2012. Incremental model-based testing of delta-oriented software product lines. In: Intl. Conference on Tests and Proofs. Springer, pp. 67–82.

Ludwig, K., Krüger, J., Leich, T., 2019. Covert and phantom features in annotations: Do they impact variability analysis? In: 23rd Intl. Systems and Software Product Line Conference-Volume a. pp. 218–230.

Marijan, D., Gotlieb, A., Liaaen, M., 2019. A learning algorithm for optimizing continuous integration development and testing practice. Softw. - Pract. Exp. 49 (2), 192–213.

Marijan, D., Liaaen, M., 2018. Practical selective regression testing with effective redundancy in interleaved tests. In: 40th Intl. Conference on Software Engineering: Software Engineering in Practice. pp. 153–162.

Marijan, D., Liaaen, M., Gotlieb, A., Sen, S., Ieva, C., 2017. TITAN: Test suite optimization for highly configurable software. In: IEEE Intl. Conference on Software Testing, Verification and Validation. In: ICST, IEEE, pp. 524–531.

Medeiros, F., Ribeiro, M., Gheyi, R., Apel, S., Kästner, C., Ferreira, B., Carvalho, L., Fonseca, B., 2018. Discipline matters: Refactoring of preprocessor directives in the #ifdef hell. IEEE Trans. Softw. Eng. 44 (5), 453–469.

Meinicke, J., Wong, C.-P., Kästner, C., Thüm, T., Saake, G., 2016. On essential configuration complexity: Measuring interactions in highly-configurable systems. In: 31st Intl. Conference on Automated Software Engineering. ASE, pp. 483–494.

Mendonça, W.D., Assunção, W.K., Vergilio, S.R., 2022a. Software product line regression testing: A research roadmap. In: Intl. Conference on Enterprise Information Systems. ICEIS, SciTePress, (ISSN: 2184-4992) ISBN: 978-989-758-569-2, pp. 81–89.

Mendonça, W.D.F., Assunção, W.K.G., Vergilio, S.R., 2023. Feature-oriented test case selection during evolution of highly-configurable systems. In: 27th Intl. Systems and Software Product Line Conference. SPLC, pp. 76–86.

Mendonça, W., Vergilio, S.R., Assunção, W.K.G., 2024. Supplementary Material - Feature-oriented Test Case Selection and Prioritization During the Evolution of Highly-Configurable System. OSF, http://dx.doi.org/10.17605/OSF.IO/WGK7C, URL https://osf.io/wgk7c/.

Mendonça, W.D., Vergilio, S.R., Michelon, G.K., Egyed, A., Assunção, W.K., 2022b. Test2Feature: feature-based test traceability tool for highly configurable software. In: 26th Intl. Systems and Software Product Line Conference. SPLC, pp. 62–65.

Michelon, G.K., Assunção, W.K., Obermann, D., Linsbauer, L., Grünbacher, P., Egyed, A., 2021. The life cycle of features in highly-configurable software systems evolving in space and time. In: 20th ACM SIGPLAN Intl. Conference on Generative Programming: Concepts and Experiences. pp. 2–15.

Michelon, G.K., Obermann, D., Assunção, W.K., Linsbauer, L., Grünbacher, P., Fischer, S., Lopez-Herrejon, R.E., Egyed, A., 2022. Evolving software system families in space and time with feature revisions. Empir. Softw. Eng. 27 (112), 1–54.

Moreira, R.A.F., Assunção, W.K., Martinez, J., Figueiredo, E., 2022. Open-source software product line extraction processes: the ArgoUML-SPL and Phaser cases. Empir. Softw. Eng. 27 (4), 85.

Oliveira, R., Cafeo, B., Hora, A., 2019. On the evolution of feature dependencies: An exploratory study of preprocessor-based systems. In: 13th Intl. Workshop on Variability Modelling of Software-Intensive Systems. ACM, ISBN: 9781450366489, pp. 1–9.

Parejo, J.A., Sánchez, A.B., Segura, S., Ruiz-Cortés, A., Lopez-Herrejon, R.E., Egyed, A., 2016. Multi-objective test case prioritization in highly configurable systems: A case study. J. Syst. Softw. 122, 287–310.

Prado Lima, J.A., Mendonça, W.D., Vergilio, S.R., Assunção, W.K., 2022. Cost-effective learning-based strategies for test case prioritization in continuous integration of highly-configurable software. Empir. Softw. Eng. 27 (6), 133.

Prehofer, C., 1997. Feature-oriented programming: A fresh look at objects. In: European Conference on Object-Oriented Programming. Springer, pp. 419–443.

Romano, S., Scanniello, G., Antoniol, G., Marchetto, A., 2018. SPIRITuS: A simple information retrieval regression test selection approach. Inf. Softw. Technol. 99, 62–80.

Runeson, P., Engström, E., 2012. Chapter 7 - Regression testing in software product line engineering. In: Hurson, A., Memon, A. (Eds.), In: Advances in Computers, vol. 86, Elsevier, (ISSN: 0065-2458) pp. 223–263.

Schiex, T., de Givry, S., 2019. Principles and practice of constraint programming. 25th Intl. Conference on Principles and Practice of Constraint Programming, vol. 11802, Springer.

Silveira Neto, P.A.d.M., do C. Machado, I., Cavalcanti, Y.C., de Almeida, E.S., Garcia, V.C., de Lemos Meira, S.R., 2010. A regression testing approach for software product lines architectures. In: 4th Brazilian Symposium on Software Components, Architectures and Reuse. IEEE, pp. 41–50.

Spadini, D., Aniche, M., Bacchelli, A., 2018. PyDriller: Python framework for mining software repositories. In: 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. ACM, ISBN: 9781450355735, pp. 908–911.

Szyperski, C., Gruntz, D., Murer, S., 2002. Component Software: Beyond Object-Oriented Programming. Pearson Education.

Tufail, H., Masood, M.F., Zeb, B., Azam, F., Anwar, M.W., 2017. A systematic review of requirement traceability techniques and tools. In: 2nd Intl. Conference on System Reliability and Science. pp. 450–454.

Tuglular, T., Şensülün, S., 2019. SPL-AT Gherkin: A gherkin extension for feature oriented testing of software product lines. In: 43rd Annual Computer Software and Applications Conference, vol. 2, IEEE, pp. 344–349.

Vierhauser, M., Grünbacher, P., Egyed, A., Rabiser, R., Heider, W., 2010. Flexible and scalable consistency checking on product line variability models. In: 25th Intl. Conference on Automated Software Engineering. ASE, pp. 63–72.

Von Rhein, A., Grebhahn, A., Apel, S., Siegmund, N., Beyer, D., Berger, T., 2015. Presence-condition simplification in highly configurable systems. In: 37th IEEE Intl. Conference on Software Engineering, vol. 1, IEEE, pp. 178–188.

Wang, S., Ali, S., Gotlieb, A., Liaaen, M., 2016. A systematic test case selection methodology for product lines: results and insights from an industrial case study. Empir. Softw. Eng. 21, 1586–1622.

Wang, S., Ali, S., Gotlieb, A., Liaaen, M., 2017. Automated product line test case selection: industrial case study and controlled experiment. Softw. Syst. Model. 16, 417–441.

Wang, S., Gotlieb, A., Ali, S., Liaaen, M., 2013. Automated test case selection using feature model: An industrial case study. In: Model-Driven Engineering Languages and Systems. Springer, ISBN: 978-3-642-41533-3, pp. 237–253.

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B., Wesslén, A., 2000. Experimentation in Software Engineering: An Introduction. Kluwer Academic Publishers, USA, ISBN: 0792386825.

Xu, Z., Cohen, M.B., Motycka, W., Rothermel, G., 2013. Continuous test suite augmentation in software product lines. In: 17th Intl. Software Product Line Conference. SPLC, ACM, ISBN: 9781450319683, pp. 52–61.

Yau, S.S., Collofello, J.S., MacGregor, T., 1978. Ripple effect analysis of software maintenance. In: 2nd International Computer Software and Applications Conference. COMPSAC, IEEE, pp. 60–65.

Yoo, S., Harman, M., 2012. Regression testing minimization, selection and prioritization: a survey. Softw. Test. Verif. Reliab. 22 (2), 67–120.

Zhao, Y., Serebrenik, A., Zhou, Y., Filkov, V., Vasilescu, B., 2017. The impact of continuous integration on other software development practices: a large-scale empirical study. In: 32nd Intl. Conference on Automated Software Engineering. ASE, IEEE, pp. 60–71.

Willian Douglas Ferrari Mendonça received his B.Sc. degree in Information Systems from Faculdade Sul Brasil in 2011, the M.Sc. degree in Electrical and Computer Engineering from State University of Western Paraná – UNIOESTE in 2014, and the Ph.D. in Computer Science from the Federal University of Paraná in 2023. He is currently an Assistant Professor at the Scientific and Technological Park of Biosciences — BIOPARK. His areas of interest are: Software Testing, Software Reuse (Product Lines and Highly-Configurable Systems), Search Based Software Engineering, and Microservices.

Wesley K. G. Assunção is an Associate Professor with the Department of Computer Science at North Carolina State University, USA. Previously, Dr. Assunção worked as a University Assistant in the Institute of Software Systems Engineering (ISSE) at Johannes Kepler University Linz (JKU), Austria (2021–2023); a Postdoctoral Researcher at Pontifical Catholic University of Rio de Janeiro (PUC-Rio), Brazil (2019–2023); and an Associate Professor at Federal University of Technology — Paraná, Brazil (2013–2020). He obtained his M.Sc. and Ph.D. in Computer Science from Federal University of Paraná (UFPR) also in Brazil. His research interests are: Software Modernization, Variability Management, Software Quality, Model-Driven Engineering, Collaboration in Systems Engineering, Software Testing, and Search-Based Software Engineering. Dr. Assunção' work has been published in several software engineering venues. He has also been serving as reviewers for many conferences and journal, and as organizer of conference, symposiums, workshops, competitions, and meetings. Further information: https://wesleyklewerton.github.io.

Silvia Regina Vergilio received M.Sc. and Ph.D. degrees from University of Campinas (UNICAMP), Brazil. She is currently a Full Professor of Software Engineering in the Computer Science Department at Federal University of Paraná (UFPR), Brazil, where she leads the Research Group on Software Engineering — GRES. She has involved in several projects and her research is mainly supported by CNPq (PQ Level 1D). Her research interests include Software Testing, Software Reliability, Software Reuse, and Search-based Software Engineering (SBSE). She serves as assistant editor of the Journal of Software Engineering: Research and Development, and acts as peer reviewer for diverse international journals. She also serves on the Program Committee of the main Brazilian Software Engineering conferences and other international ones, mainly related to Search-Based Software Engineering and Software Testing.