# Feature-oriented Test Case Selection during Evolution of Highly-Configurable Systems

Willian D. F. Mendonça
DInf, Federal University of Paraná
Curitiba, Brazil

Wesley K. G. Assunção
CSC, North Carolina State University
Raleigh, USA

Silvia R. Vergilio
DInf, Federal University of Paraná
Curitiba, Brazil

## ABSTRACT

Ensuring the quality of *Highly Configurable Systems (HCSs)* during its evolution and maintenance is challenging. As an HCS evolves, new features are added, changed, or removed, which makes the test case selection for regression testing a difficult task. The use of test traceability can help in this task, but there is a lack of studies exploring the use of trace links for HCS testing. 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 files/lines of code, limiting the selection to test cases related to file changes, not considering the whole implementation of features, which can be spread in many files other than the changed ones. Considering this limitation, this work presents a test case selection approach, namely `FeaTestSel`, that links test cases to features using HCS pre-processor directives. Then, the selection of test cases is based on features affected by changes in each commit. In addition to the selected test cases, the approach also produces the following reports to support the test activity: the lines of code that correspond to each feature, the lines exercised by each test case, and the test cases linked to each feature. To validate the approach, we rely on `Libssh`, a real open-source HCS in constant evolution. By adding the execution time of the approach to the execution time of the selected test cases, we achieved a reduction of approximately ≈50%, in comparison with the retest-all technique. Furthermore, the approach was able to maintain quality by selecting 100% of failed test files. The traceability and reports produced by our approach can also be used for further work by researchers, analysis of the test quality by engineers, or as a source of information for tool builders.

## CCS CONCEPTS

• **Software and its engineering → Software product lines**; **Software testing and debugging**; **Traceability**; **Software maintenance tools**.

## KEYWORDS

Software Evolution, Software Product Line, Regression Testing

*SPLC'23, August 28 – September 1, 2023, Tokyo, Japan*
© 2023 Association for Computing Machinery.
ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

**ACM Reference Format:**
Willian D. F. Mendonça, Wesley K. G. Assunção, and Silvia R. Vergilio. 2023. Feature-oriented Test Case Selection during Evolution of Highly-Configurable Systems. In *Proceedings of 27th ACM International Systems and Software Product Lines Conference (SPLC'23).* ACM, New York, NY, USA, 11 pages. https://doi.org/XXXXXXX.XXXXXXX

## 1 INTRODUCTION

*Highly Configurable Systems (HCSs)* provide adaptable and flexible solutions to complex and real-world problems. HCSs are complex and predominantly configurable software, which means that they incorporate a series of options (a.k.a., features) used for software customization, allowing different functionalities and configurations to be selected to a given user context [42]. Quality assurance of HCSs is very important, but is a hard task [4]. Most HCSs incorporate a substantial number of configuration options, causing high complexity for the test [25]. Typically, the configuration options are interrelated through either inclusive or exclusive relationships, which further increases the testing effort. This is even more challenging in the *Continuous Integration (CI)* context, where the system is constantly evolving. HCSs can be updated, integrated, and tested several times a day, and each cycle needs to be fast [46].

In this scenario, 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 [8]. Thus, the use of a *Test Case Selection (TCS)* technique is fundamental. TCS techniques have as goal the selection of the best test cases from the test set available for a commit according to some criteria, such as code coverage, execution time, fault detection, and so on. Ideally, in the case of HCSs, all possible configurations should be tested, but this is often unfeasible because the number of configurations grows exponentially with the number of features, and several test cases overlap [33]. TCS requires specific approaches to consider the HCS evolution, where features are constantly added, modified, or even removed [31]. Furthermore, if a CI environment is adopted, there are strict time constraints imposed to run the tests, the *test budget* [18]. To efficiently deal with these challenges with a balance among time, cost, and test quality, practitioners need efficient TCS approaches [23].

We can find many TCS approaches based on *Feature Model (FM)* or other artifacts [1, 15, 16, 19, 38]. However, none of them consider that HCSs are usually developed by adopting CI practices. In this agile context, the models may be not updated accordingly, making the use of those approaches difficult or even impossible. Other HCS testing approaches need some kind of dynamic analysis based on the test failure-history, execution, or coverage [21–23]. In some pieces of work, the approaches are only evaluated with systems well-modularized [10], in a context where the test cases are separated by feature and do not overlap. This is different from what is found in practice. Approaches that take into account the code changes are more suitable and used. For instance, approaches that select test cases related to the files changed in the current commit [3, 9, 35]. But those existing approaches and tools do not consider HCS particularities, and they are mostly based on Java language only [10, 11]. Yet, those approaches also have limitations, since it only focuses on changed parts of the code without considering the change impact on features - building blocks of HCSs.

Motivated by these facts, in this paper, we present `FeaTestSel` (**Fea**ture-oriented **Test** Case **Sel**ection for Highly Configurable Systems). Given the source code of an annotated 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` produces different traceability reports linking (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 our approach works for systems in C/C++ language. Our approach is feature-oriented and needs only static analysis of the source code.

`FeaTestSel` proved to be highly efficient in reducing the time spent executing test cases. By adding the execution time of the approach to the execution time of the test cases, we reached a reduction of approximately ≈50% compared with the retest-all technique. Furthermore, the approach was able to maintain the test quality by selecting 100% of failed test files. Compared with our approach, the changed-file-oriented approach reaches a greater reduction in the time and number of selected test cases (≈97%) but the quality regarding the number of detected failures is very low. In addition to the reduction of testing effort, our approach also produces complementary reports that can be used by software engineers for analysis and improvement of the test case regression process as a whole.

The paper is structured as follows. Section 2 reviews related work. Section 3 contains an example that serves as motivation to our work. 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 analyses the obtained results. Section 7 discusses threats to the validity of our results. Section 8 concludes the paper and points research directions.

## 2 RELATED WORK

The literature on the topic of regression testing is vast [45], and it also includes reviews related to the HCS context [5, 6, 14, 27, 36]. The studies more related to ours are the ones that describe approaches and tools for TCS and test traceability, described below.

Some pieces of work on regression test of HCSs introduce approaches based on models and on the delta-oriented concept. For instance, the work of Lity et al. [19] captures commonality and variability of an evolving product line by means of differences between variants and versions of variants for TCS. Lachmann et al. [16] show an incremental delta-oriented approach for improving the efficiency of integration testing of HCSs by prioritizing test cases for product variants. The approach of Al-Hajjaji et al. [1] selects products that are the most dissimilar, in terms of deltas, to products tested previously. They also study the impact of adding delta modeling feature selection on product prioritization. The approach of Lachmann et al. [15] is risk-based, and computes component failure impact and component failure probabilities for each product variant under test. In addition to the FM, the approach of Wang et al. [43, 44] uses component feature models, in which an annotated classification of test cases are used for test case selection. The idea is to ensure that all test cases associated with a specific functionality provided by the user are executed. These approaches select the best configurations of products to be tested and have the FM as a starting point. This is a disadvantage, because sometimes this model and other ones required are not always available, and in the HCS evolution process they may be outdated.

The work of Silveira Neto et al. [38] describes a regression testing 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 many input artifacts, which are often unavailable, such as test scripts and integration level test suites. In addition, the intervention of testing experts is necessary. For the approach validation, different versions of a system were created to simulate an HCS.

An approach, called TITAN, which can be used in an evolution scenario, was proposed by Marijan et al. [23]. The test data history is used to determine an optimal test order to ensure feature coverage, early fault detection, and reduced execution time. TITAN implements test prioritization and minimization techniques, and provides test traceability and visualization. The approach considers that the test cases have tags for HCS features, but we can observe that in most open-source HCS systems these macros do not exist. This hampers its applicability for general scenarios. Other studies of Marijan et al. [21, 22] 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 when inserted in industrial environments that undergo constant updates. In this way, approaches that perform only static analysis are more suitable.

Tufail et al. [40] 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 [3, 9, 35]. An example is the tool Ekstazi [9] 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. [3] presents a TCS approach, applying a criterion based on static dependency analysis at the class level. These approaches above 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 [12, 13, 26], do not generate traceability for features, but only link test cases to lines of code. The work of Tuglular and Şensülün [41] generates a traceability between the feature and test cases, but using a specific language through annotations.

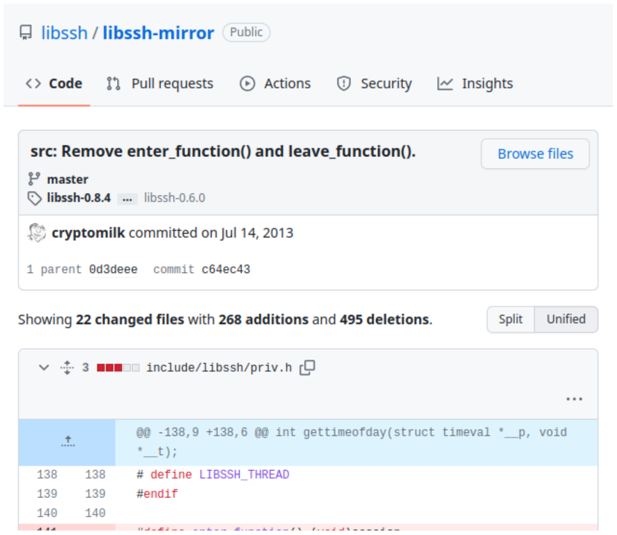
The code-based method of Jung et al. [10] considers the similarity and variability of a product family, leaving out test cases unaffected by source code changes. But the application considers only well-developed HCSs (i.e., toy systems), and test cases were developed specifically for the evaluation. In practice, however, for most of the industrial HCSs there are no links between test cases and features/products. Thus, it is hard the application of the approach in a real context, even more when the system is constantly evolving. Jung et al. [11] 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 products of a product family. To identify equivalent test runs, test case execution traces and source code checksum values are used. The several steps of the approach may be quite costly if applied to large systems that are constantly evolving. In addition, it is specific for Java.

In summary, existing pieces of work present the following main limitations: (i) are dependent on models that can be outdatet [1, 15, 16, 19, 38]; (ii) require a failure-history or dynamic analysis [21–23], 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 [3, 9, 35, 41]; (iv) consider that there is a kind of mapping for the test cases or that the HCSs are developed following a specific format [10, 23]; and (v) do not work with the concept of features, fundamental in the HCS context [12, 13, 26]. To address these limitations, we present an approach (in Section 4) that works for systems written in C/C++ language. Our approach is changed-based and feature-oriented, relying only on the static analysis of the source code. In the next section, we present a motivating example showing the importance of considering the changed features in comparison with a changed-file-oriented approach.

## 3 MOTIVATING EXAMPLE

To illustrate the importance of using a feature-oriented approach, we consider the system `Libssh` that is also used in our evaluation (described in Section 5.2). The system has 110 developers[1] and it is constantly evolving. `Libssh` is commonly updated more than once a day. 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 retest may be necessary several times due to the number of developers involved. This is an expensive approach that requires a lot of time and resources for every change, even the simple ones. For instance, consider 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 Figure 1. In this commit, 117 test cases are available. The retest-all technique, for example, considering `Libssh`, can 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.

[1]According to https://github.com/libssh/libssh-mirror/graphs/contributors
[2]https://github.com/libssh/libssh-mirror/commit/c64ec43
[3]Examples of time per variant: https://gitlab.com/libssh/libssh-mirror/-/pipelines


libssh / libssh-mirror Public

Code Pull requests Actions Security Insights

src: Remove enter_function() and leave_function(). Browse files
master
libssh-0.8.4 ... libssh-0.6.0
cryptomilk committed on Jul 14, 2013
1 parent 0d3deee commit c64ec43

Showing 22 changed files with 268 additions and 495 deletions. Split Unified

3 include/libssh/priv.h

```
@@ -138,9 +138,6 @@ int gettimeofday(struct timeval *__p, void
*__t);
# define LIBSSH_THREAD
#endif

```

Figure 1: Number of modifications made in commit `c64ec43`

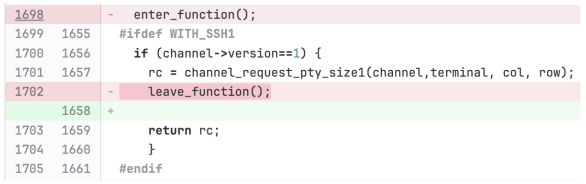

```
-    enter_function();
 #ifdef WITH_SSH1
     if (channel->version==1) {
       rc = channel_request_pty_size1(channel,terminal, col, row);
-      leave_function();
+
       return rc;
     }
 #endif
```

Figure 2: Example of modification in commit `c64ec43`

An alternative to this technique is to trace test cases to the 22 files changed and select only the test cases associated to them. But this technique does not select the test cases related to the features that were changed in the commit, which can cause failures in other files not changed. An example is presented in Figure 2, in which the feature `WITH_SSH1` was changed in the referred commit. In the 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 files not changed that also implements this feature, such as `options.c` and `channels1.c`. To address this issue, we introduce in the next section a feature-oriented selection approach, which is capable of capturing these relationships and including all the impacted test cases in the selected test set.

## 4 PROPOSED APPROACH

Our approach, called `FeaTestSel` (**Fea**ture-oriented **Test** Case **Sel**ection for Highly Configuration Systems), consists of four steps that are presented in Figure 3. The tester needs only to 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 exercised by each test case are identified and trace links between feature and test cases are created. In the last 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.

FeaTestSel 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. Thus, our approach can be executed after each commit, identifying feature implementations, feature changes, and feature to test traceability using the most updated version of the HCS. In the following, we present the procedure and implementation aspects of each step of our approach, and how the information of the configuration file is used as input. Our implementation, adopts Python programming language, version 3.9.10. To deal with the CSV files, we adopted the PANDAS[4] library.

## 4.1 Input

The *input* provided is a configuration file (*Config file*), as illustrated in Figure 4 for the system Libssh. In the file, the software engineer defines the paths to folders that the approach uses as a source of information, containing at least three pieces of information: (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; and (iii) test_names: 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 string 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.

## 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 preprocessor 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 [30, 32]. 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 preprocessor directives. The approach uses a Constraint Satisfaction Problem Solver [37] to reliably identify features interacting/depending on the execution of other features [2], and thus which features belong to which conditional blocks to obtain the features lines.

Figure 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 [20] at lines 11-13, where there is a negated conditional expression, i.e., 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.

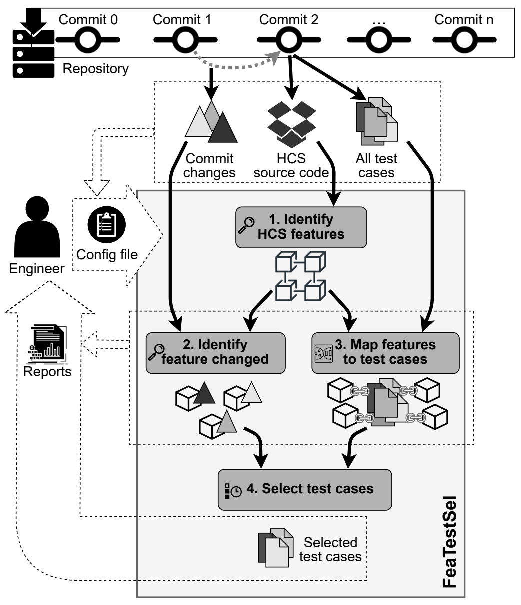
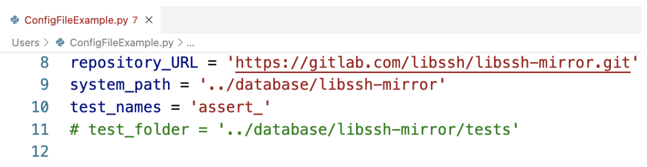


**Figure 3: Input, four steps, and output for FeaTestSel**



```
repository_URL = 'https://gitlab.com/libssh/libssh-mirror.git'
system_path = '../database/libssh-mirror'
test_names = 'assert_'
# test_folder = '../database/libssh-mirror/tests'
```

**Figure 4: Configuration file used as input for the approach**

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

```
#ifdef A
    #define B 15
#endif

#if C
    #if B > 10
        <code>
    #endif
#endif

#ifndef D
        <code>
#endif

<code>
```

**Figure 5: Conditional blocks of feature implementations.**

## 4.3 Identify Feature Changed

The mining process performed in this step uses the outputs from the previous steps to identify changed features. First, we collect all conditional block macros and all `#defines` present in all files from each release commit. Next, we looked for macros that were 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. After the blocks are identified, for each block the approach uses Git diff[5] to collect code fragments that differ for the same file from one commit to another, thus, the differences of fragments with patches from Git are obtained. These patches represent the differences between two text files in a line-oriented manner, as calculated by a diff utils library.[6] In summary, we use the previous step to identify the features and their locations. When observing a diff between commits, a feature change is identified if a new feature is found or a change in the feature location occurs.

## 4.4 Map Features to Test Cases

In this step, our approach uses static analysis to identify dependencies between test cases and source code implementing features. To do this, we use the tool Test2Feature [28] that employs static analysis to identify dependencies between test cases and source code implementing features. First, we create a dependency graph using all code available in the repository. 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. In summary, 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.

This step was implemented based on `Doxygen`,[7] a tool that generates the dependency graph as XML files, performing static analysis of the source code. We defined the minimal set of parameters in order to generate all possible dependencies available in Doxygen and to make the tool execution faster. This tool, which has the GNU General Public License, was initially developed with a view to keeping the source code of C++ systems documented, but also has support for other languages like C, C#, Python, Java, and others. For our approach, we used Doxygen for C/C++ code. Doxygen supports visualization of the relationships between various elements through dependency graphs, inheritance, and collaboration diagrams, generated automatically, in different formats. Our implementation uses the function dependency graph in XML format.

## 4.5 Select Test Cases

After executing Steps 2 and 3, the required data (i.e., traceability of tests for the feature) to perform the last step of the approach is available. The implementation of the last step of the approach is simple. For each commit in the HCS repository, knowing the features that changed based on the output of Step *2. Identify feature changed*, the approach selects the test cases covering such features.

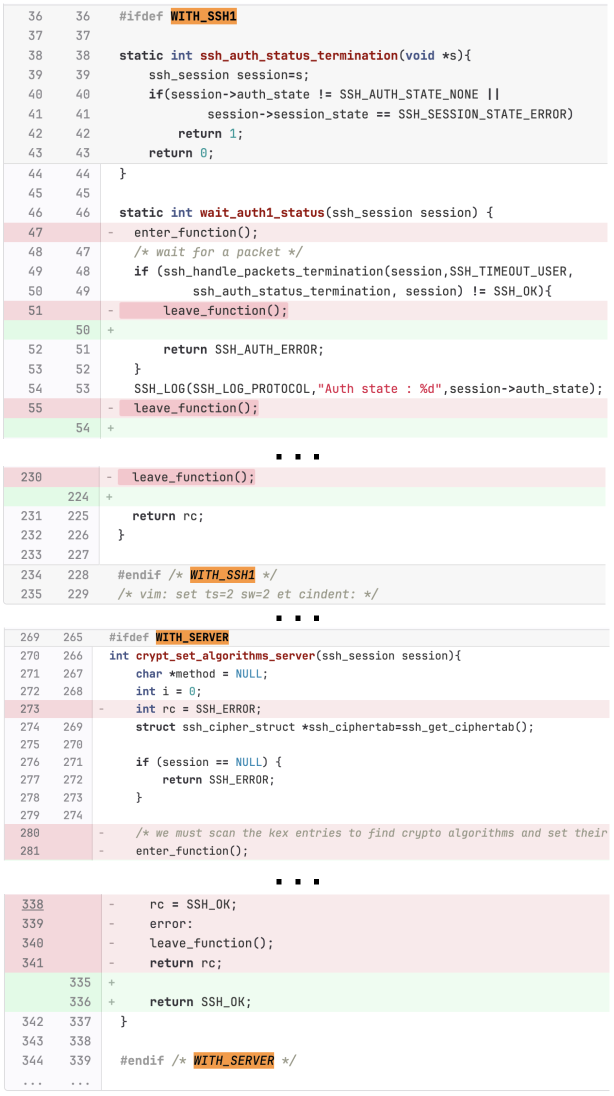
Despite its simplicity, this selection has an advantage when compared to existing approaches. Approaches that also are change-oriented, mostly select test cases that are direct related to the changed files, without considering features. In our case, even if the changed file is not touched by the test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

## 4.6 Illustrative Example

To illustrate required input and produced outputs of `FeaTestSel`, we use again `Libssh` and focus on the commit c64ec43.[8] This commit was chosen because it represents well the evolution of an HCS, with changes occurring in four different features: `WITH_ZLIB`, `WITH_SSH1`, `WITH_SFTP` (connect.c file), `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., are not wrapped in pre-processor directives). `BASE` corresponds to a large portion of code, and changes in all commits.

An excerpt of the source code corresponding to the `Libssh` features is present in Figure 6. 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 Figure 7. From this test case, Table 1 presents an excerpt of the CSV document generated by our approach, with the test case `set_ops` having traceability to the files `sample.c`, `error.c` and `options.c`. Moreover, it is possible to trace specific functions and lines. Figure 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`.

[5]https://git-scm.com/docs/git-diff
[6]https://java-diff-utils.github.io/java-diff-utils/
[7]https://doxygen.nl/index.html
[8]https://github.com/libssh/libssh-mirror/commit/c64ec43



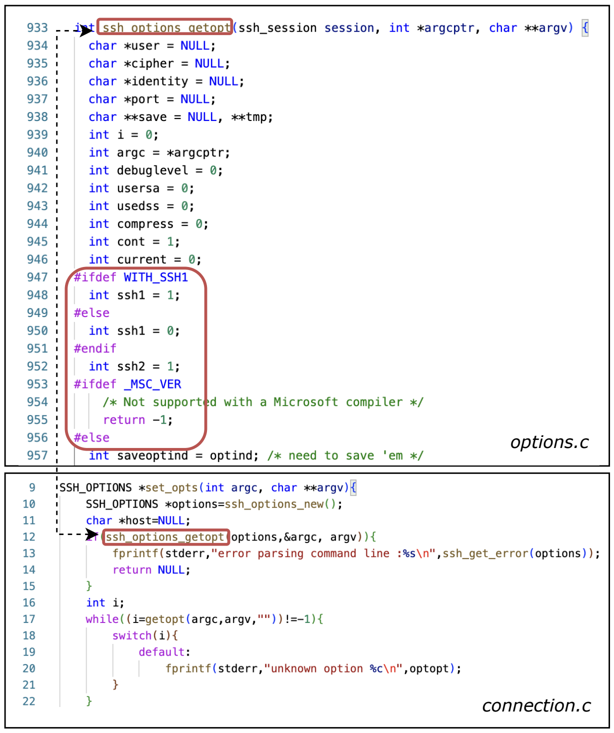
```
   #ifdef WITH_SSH1

   static int ssh_auth_status_termination(void *s){
       ssh_session session=s;
       if(session->auth_state != SSH_AUTH_STATE_NONE ||
               session->session_state == SSH_SESSION_STATE_ERROR)
           return 1;
       return 0;
   }

   static int wait_auth1_status(ssh_session session) {
-    enter_function();
     /* wait for a packet */
     if (ssh_handle_packets_termination(session,SSH_TIMEOUT_USER,
             ssh_auth_status_termination, session) != SSH_OK){
-        leave_function();
+
         return SSH_AUTH_ERROR;
     }
     SSH_LOG(SSH_LOG_PROTOCOL,"Auth state : %d",session->auth_state);
-    leave_function();
+

. . .

-    leave_function();
+
     return rc;
   }

   #endif /* WITH_SSH1 */
   /* vim: set ts=2 sw=2 et cindent: */

. . .

   #ifdef WITH_SERVER
   int crypt_set_algorithms_server(ssh_session session){
       char *method = NULL;
       int i = 0;
-      int rc = SSH_ERROR;
       struct ssh_cipher_struct *ssh_ciphertab=ssh_get_ciphertab();

       if (session == NULL) {
           return SSH_ERROR;
       }

-      /* we must scan the kex entries to find crypto algorithms and set their
-      enter_function();

. . .

-      rc = SSH_OK;
-      error:
-      leave_function();
-      return rc;
+
+      return SSH_OK;
   }

   #endif /* WITH_SERVER */
   ...
```

**Figure 6: Excerpt of source code corresponding to the Libssh features changed as part of commit c64ec43**

To illustrate feature traceability, we also use the granularity of lines of code and the feature WITH_SSH1 as example. Table 2 presents an excerpt of the CSV file generated by our approach. We can see 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 two CSV files, performing a merge of the related lines of code. We can see in Figure 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 Figure 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.

**Table 1: Traceability of Test Cases to Source Code Lines**

| TestFile | TestCase | TargetFile | TargetFunction | LineFrom | LineTo |
|---|---|---|---|---|---|
| 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 |



```
933  int ssh_options_getopt(ssh_session session, int *argcptr, char **argv) {
934    char *user = NULL;
935    char *cipher = NULL;
936    char *identity = NULL;
937    char *port = NULL;
938    char **save = NULL, **tmp;
939    int i = 0;
940    int argc = *argcptr;
941    int debuglevel = 0;
942    int usersa = 0;
943    int usedss = 0;
944    int compress = 0;
945    int cont = 1;
946    int current = 0;
947  #ifdef WITH_SSH1
948    int ssh1 = 1;
949  #else
950    int ssh1 = 0;
951  #endif
952    int ssh2 = 1;
953  #ifdef _MSC_VER
954      /* Not supported with a Microsoft compiler */
955      return -1;
956  #else
957    int saveoptind = optind; /* need to save 'em */
```

```
 9  SSH_OPTIONS *set_opts(int argc, char **argv){
10      SSH_OPTIONS *options=ssh_options_new();
11      char *host=NULL;
12      if(ssh_options_getopt(options,&argc, argv)){
13          fprintf(stderr,"error parsing command line :%s\n",ssh_get_error(options));
14          return NULL;
15      }
16      int i;
17      while((i=getopt(argc,argv,""))!=-1){
18          switch(i){
19              default:
20                  fprintf(stderr,"unknown option %c\n",optopt);
21          }
22      }
```

**Figure 7: Traceability between test cases and feature**

**Table 2: Traceability of Test Cases to Features**

| TargetFile | FeatureName | FeatFrom | FeatTo |
|---|---|---|---|
| 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 |

## 5 EVALUATION SETUP

This section describes details of the study we conducted to evaluate FeaTestSel's applicability and performance. For that, we compare its results with the retest-all technique and a changed-file-oriented approach. All experiments were performed on an Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz 32-Core server, 64GB RAM, running on Linux Ubuntu 18.04.1 LTS.

### 5.1 Research Questions

The study was guided by the following *Research Questions (RQ)*:

**RQ1:** *Is FeaTestSel applicable considering budget constraints of industrial HCSs?* This question aims to assess the applicability of

**Table 3: Partial Set of Test Cases Selected by FeaTestSel**

| TestFile | TestCase | TargetFile | TargetFunction | LineFrom | LineTo | FeatureName | FeatFrom | FeatTo |
|---|---|---|---|---|---|---|---|---|
| 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 |

our approach by comparing the execution time of the selected test cases summed to the time spent to perform the selection with the time between commits. The idea is to check if 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.

**RQ2:** *How is the performance of our approach when compared to the performance of the retest-all and changed-file-oriented approaches?* As mentioned in Section 2, some tools that map test cases to files are available, then they can be adopted to select the test cases associated to the files changed in the commit. This question compares the performance of this approach with the performance of FeaTestSel. The retest-all technique, which executes all the test cases available for the commit, is used as baseline to evaluate the percentage of reduction in the number of test cases, as well as in the test execution time obtained by both approaches.

**RQ3:** *Is fault-detection quality of the test cases maintained?* This question investigates whether by reducing the number of test cases, it is still possible to maintain quality in terms of detected failures. To this end, the retest-all approach is considered. We analyze the number of failures detected by the test cases selected by FeaTestSel and by the changed-file-oriented approach in comparing with the number of failures detected if the whole test set were executed.

## 5.2 System

Our assessment is based on the SSH library (Libssh),[9] which is an open source cross-platform C library 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, and so on. Libssh is statically configurable with the C preprocessor, being an HCS. Libssh is hosted on GitLab,[10] which provides an environment with version control system and CI pipelines. The logs of the CI pipeline tasks are the source of information that can be used for evaluation of the approaches. Libssh has been used in the literature by other studies on the subject of HCSs [7, 17, 24, 25, 34].

## 5.3 Applying the approaches

To mine the commit history of the HCS, we used PyDriller [39] library from the initial to the last available commit. For the changed-file-oriented approach, the source code is scanned, looking for the test files. Next, the traceability of test cases to source code is created. Then, we used PyDriller to find the files that were changed in that commit. Finally, we selected the test cases that have traceability for these files.

The repository of Libssh contains around five thousand commits, from which we used 4,388. We discarded commits not associated with test cases. This set will be referred as the *whole set of commits*. To evaluate the quality of the selected test cases, we need logs with failure information, as well as the test case execution time. For this end, we used a repository containing 303 commits from a related work [17], referred as *set of commits with logs*.

To calculate the average execution time of each test case, an approximation was necessary, since the evaluated approaches use function granularity (i.e., a test case corresponds to a function), while the available system logs use file granularity. Thus, for each CI cycle, there is an execution time per test file per job of that cycle. To perform this calculation, 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 calculate the average time each test case takes to run. For example, if a file takes an average of 100 seconds to execute in a commit and has 10 functions, each function takes an average of 10 seconds 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 seconds 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. For this end, we mined the logs to identify failed files in the failed commits. For instance, the commit 10728f85[11] has three failed 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 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 a failure is detected.

## 6 ANALYSIS OF RESULTS

This section presents and discusses the results to answer the three RQs or our study. The dataset and results are available online [29], as a supplementary material.

[9]https://www.libssh.org/
[10]https://gitlab.com/libssh/libssh-mirror
[11]See Log Line 1130 at https://gitlab.com/libssh/libssh-mirror/-/jobs/78303050

## 6.1 RQ1: `FeatTestSel` applicability

To answer RQ1, we collected the time the approach takes to perform the selection for the whole set of commits (a total of 4,388). The approach takes an average time of 20.82 seconds, taking the maximum time of 35.72 seconds in the last commit evaluated, and a minimum time of 9.68 seconds in the first one. An explanation for this, is that in the initial commits the system size is smaller.

The average time our approach takes to execute considering the 303 commits with logs is 25.97 seconds. For these 303 commits with logs, we performed an analysis considering the time available between the CI cycles and the time of our approach takes to execute, summed to the time to execute the selected test cases. Using the procedure described in Section 5.3, the average time to execute all the test cases was 239.22 seconds (≈4min). The test cases selected by `FeatTestSel` take on average 92.78 seconds to run. By adding to this time the average time our approach takes to perform the selection (25.97 seconds), the runtime is 118.75 seconds (≈2min). 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 minutes (standard deviation equals to 459.28), what shows the applicability of our approach.

**RQ1**: *We can conclude that our approach is applicable in practice. It takes an average time of 20.82 seconds to execute, and 35.72 seconds in the worst case. When the smaller set of commits with logs is considered, the average time to perform the selection is 25.97 seconds. This time, when summed to the time spent to execute the selected test cases, is 118.75 seconds, what represents a reduction of ≈50% compared to retest-all.*

**Implications.** We can observe that our approach is lightweight. Differently to other TCS approaches, it does not require a failure-history nor the application of search-based/learning techniques, what leads to a reduced time to perform the selection. It is worth to observe that the approach does not only produce the set of selected test case, but an entire static analysis of the system for each commit. The approach delivers results that can be analyzed qualitatively and/or quantitatively by developers for possible improvements in the system. As an example, developers can use the test case traceability to feature to identify features that need more test. The approach fits in a budget of 50% of time that would be spent by executing all the available test cases for the commit. We observe by analyzing the whole set of commits, the time the approach takes to execute is dependent on the size of the system and number of test cases, this relationship should be better explored in future works.

## 6.2 RQ2: Performance of `FeatTestSel` and changed-file-oriented approaches

In this section, we compare the `FeatTestSel` and changed-file-oriented approaches against retest-all, regarding the number of selected test cases and the time the selection takes to execute.

We first analyze the percentage of reduction in the number of test cases of both approaches, considering the whole set of 4,388 commits. As both 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 is given by the number of functions existing inside all the files in the logs. Then, each function is considered a test case. The average reduction for our approach is of 24.22%, and for the changed-file-oriented approach is 94.23%. When the set of 303 commits with logs is analyzed, these percentages are 41.98% and 97.27%, respectively. The reduction of test cases is smaller when we analyze many commits. This happens because in the initial commits there are few test cases, and in these cases usually 100% of the test cases are selected. 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.

To better compare both approaches, we use Figures 8 and 9, considering the 303 commits with logs. Figure 8 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 approach); the green, the test cases selected by our approach; and the orange, the number of test cases selected by the changed-file-oriented approach. We can observe that the changed-file-oriented approach always selects fewer test cases than `FeatTestSel`, but, sometimes, it does dot select any test case, as indicated by the orange line in the figure. This happens in 199 commits (out of 303, ≈65.67%). In other commits few test cases are selected, in fact the number of test cases selected depends on the performed changes. For example, in commit 12284b75,[12] 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.[13] In this commit seven files are changed (245 additions and 13 deletions), and this approach selected 334 test cases out of 719. 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. Figure 9 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. On the other hand, we can observe that this time reduction is quite drastic, spending an average of 4.15 seconds. As mentioned, in the last subsection, our approach the selected test sets generated by our approach take on average 92.78 seconds to execute.

**RQ2**: *We conclude that the changed-file-oriented approach leads to a greater reduction of test cases than our approach and consequently a greater reduction in execution time, with a very drastic reduction, reaching ≈97%. The number of selected test set depends on the number of changes performed in the commit. On the other hand, the percentage of reduction of our approach depends on the size of the system and number of tests available in the commit. In the set of 303 commits with logs, this percentage reaches ≈42%.*

**Implications.** Our feature-oriented TCS approach presents some advantages regarding the changed-file-oriented one, presenting a good balance in the reduction of the test sets, making sure that some important test cases are selected in the regression testing. However, the reduction provided by the file-oriented approach is

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

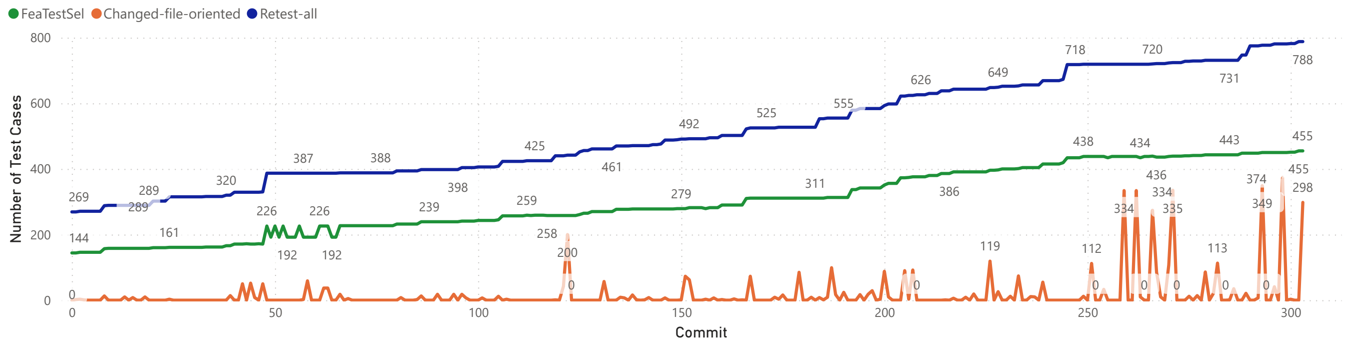


**Figure 8: Number of test cases selected by the approaches for the set of commits with logs**

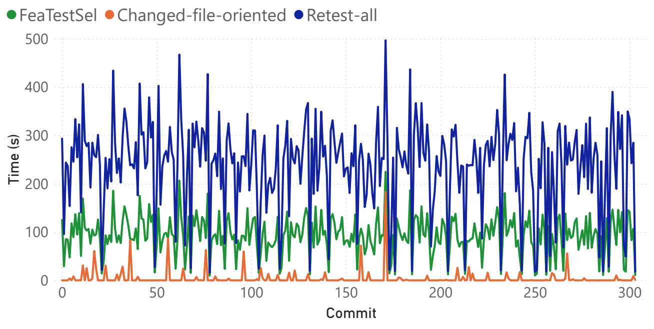


**Figure 9: Average time to execute the selection of test cases per commit and approach for the set of commits with logs**

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 context.

## 6.3 RQ3: Quality of test cases regarding the detected failures

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 detected failures. Figure 10 shows the number of failed files detected by each approach by commit with logs. 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, but this does not happen for the changed-file-oriented approach (orange line, in the middle). As we showed in the previous section, the number of test cases selected by the changed-file-oriented approach is very small in many cases, thus it does not maintain the quality.

To complement the analysis, we use a dynamic chart[14] considering the 303 commits with logs. Figure 11 shows a clipping of the

[14]Available at https://app.powerbi.com/view?r=eyJrIjoiYjk3MGUzNTgtOWE5MS00ZWNhLWI1MGMtZTYzMTcwZDVlODZlIiwidCI6ImRhZGFhOGQzLTIxYWEtNGRjNS05ODBlLTFiZjI0ZWY5Yzc0OCJ9&pageName=ReportSection.

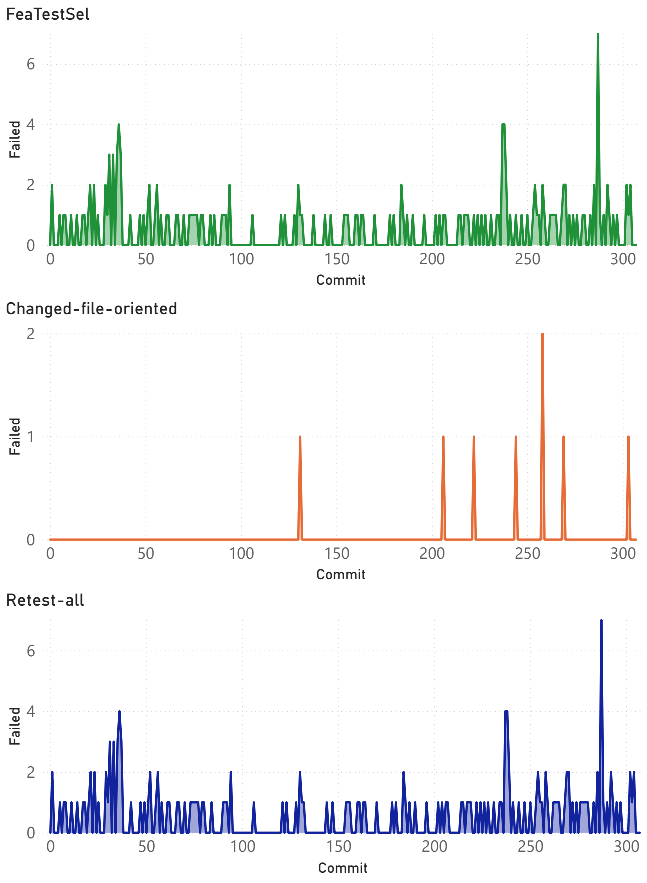


**Figure 10: Number of failed files per commit and approach for the set of commits with logs**

graph for commit d7477dc.[15] 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

[15]https://gitlab.com/libssh/libssh-mirror/-/commit/d7477dc

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.

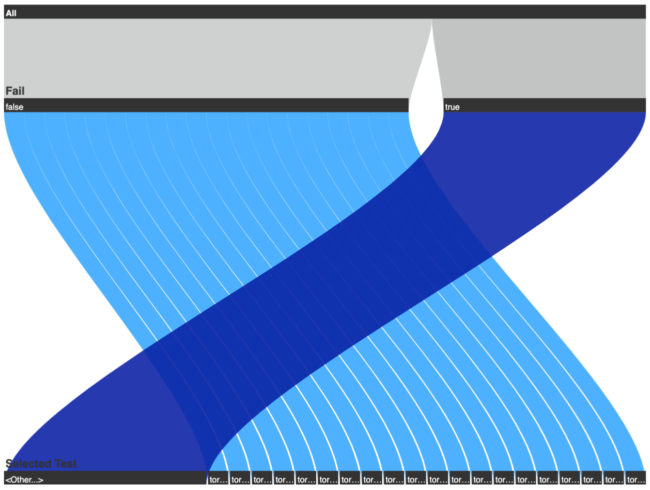


**Figure 11: Failed and selected test cases for commit `d7477dc`**

**RQ3**: *Based on the failure criterion and considering the retest-all technique baseline, our approach manages to select 100% of the times the test files that failed, maintaining the quality of the test case set. This does not happen for the changed-file-oriented approach.*

**Implications.** The main goal of testing is to detect faults. TCS approaches reduce the set of test cases to be executed, but the main testing goal must hold for making an approach used in industry. Our approach contributes to select a reduced number of test cases, and consequently reduces the time to execute them, and also keeps quality in terms of failure produced.

## 7 THREATS TO VALIDITY

*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.

*Construct validity:* A threat in this category is the approach used in the comparison. As we did not find approaches or tools available for C language, we used a simple but well-known changed-file selection approach that we developed internally as a basis for comparison.

*External validity:* We used only one system for evaluation, which can be considered a threat, since other systems may have different behaviors. However, the HCSs that are inserted into CI environments and have test logs are relatively few. Although our approach does not rely exclusively on logs for execution, in the case of `Libssh`, which has about five thousand commits, we managed to mine only 303 valid logs that were available to analyze execution time and failures. 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. Despite this, we have developed a robust approach that follows detailed steps in order to make it replicable in other systems. For future work, we are looking at other industrial-grade systems to be included in the validation.

*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 `FeatTestSel`, a feature-oriented TCS approach to be used during the evolution of HCSs. The approach produces several intermediate outputs, including: 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 analysis and improvement of the test case regression process as a whole.

`FeatTestSel` does not require a failure-history or dynamic analysis. The results with a set of commits with logs show an average reduction in the number of test cases of ≈42%, and that, on average, the time it takes to execute summed to the time of selected test cases fits well in a budget of 50% of the average time required to execute all the tests available in the commit. These percentages depend on the system size and number of test cases, being more significant in the last commits. This reduction does not imply in loosing important test cases because the approach manages to select 100% of the times the failed test files, maintaining the test quality.

As future work, we will search for other systems to improve the validation results. We intend to better evaluate the relationship between the system size and the reduction for the number of test cases and execution time. Furthermore, we intend to improve the selection by adding some criteria, such as changes in files along with changes and features, and adding a step of minimizing and/or prioritizing test cases.

## ACKNOWLEDGMENTS

This research is supported by CNPq (grant: 310034/2022-1) and FAPERJ PDR-10 Fellowship (202073/2020). We would like to thank Gabriela Karoline Michelon for the help with the implementation of step to identify feature changes.

## REFERENCES

[1] Mustafa Al-Hajjaji, Sascha Lity, Remo Lachmann, Thomas Thüm, Ina Schaefer, and Gunter Saake. 2017. Delta-oriented product prioritization for similarity-based product-line testing. In *2nd Intl. Workshop on Variability and Complexity in Software Design (VACE)*. IEEE, 34–40.
[2] David Benavides, Sergio Segura, Pablo Trinidad, and Antonio Ruiz-Cortés. 2006. Using Java CSP solvers in the automated analyses of feature models. *Intl. Summer School Generative and Transformational Techniques in Software Engineering* (2006), 399–408.
[3] Antonia Bertolino, Antonio Guerriero, Breno Miranda, Roberto Pietrantuono, and Stefano Russo. 2020. Learning-to-rank vs ranking-to-learn: strategies for

regression testing in continuous integration. In *ACM/IEEE 42nd Intl. Conference on Software Engineering.* 1–12.

[4] Ivan do Carmo Machado, John D McGregor, Yguaratã Cerqueira Cavalcanti, and Eduardo Santana De Almeida. 2014. On strategies for testing software product lines: A systematic literature review. *Information and Software Technology* 56, 10 (2014), 1183–1199.

[5] Emelie Engström. 2010. Regression Test Selection and Product Line System Testing. In *3rd Intl. Conference on Software Testing, Verification and Validation.* IEEE, 512–515.

[6] Fischer Ferreira, João P Diniz, Cleiton Silva, and Eduardo Figueiredo. 2019. Testing tools for configurable software systems: A review-based empirical study. In *13th Intl. Workshop on Variability Modelling of Software-Intensive Systems.* 1–10.

[7] Angelo Gargantini, Justyna Petke, Marco Radavelli, and Paolo Vavassori. 2016. Validation of Constraints Among Configuration Parameters Using Search-Based Combinatorial Interaction Testing. In *8th Intl. Symposium on Search Based Software Engineering.* Springer, Cham, 49–63.

[8] Vahid Garousi and Junji Zhi. 2013. A survey of software testing practices in Canada. *Journal of Systems and Software* 86, 5 (2013), 1354–1376.

[9] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. 2015. Practical regression test selection with dynamic file dependencies. In *Intl. Symposium on Software Testing and Analysis.* 211–222.

[10] Pilsu Jung, Sungwon Kang, and Jihyun Lee. 2019. Automated code-based test selection for software product line regression testing. *Journal of Systems and Software* 158 (2019), 110419.

[11] Pilsu Jung, Sungwon Kang, and Jihyun Lee. 2020. Efficient regression testing of software product lines by reducing redundant test executions. *Applied Sciences* 10, 23 (2020), 8686.

[12] Chang Hwan Peter Kim, Don S Batory, and Sarfraz Khurshid. 2011. Reducing combinatorics in testing product lines. In *10th Intl. conference on Aspect-oriented software development.* 57–68.

[13] Chang Hwan Peter Kim, Sarfraz Khurshid, and Don Batory. 2012. Shared execution for efficiently testing product lines. In *IEEE 23rd Intl. Symposium on Software Reliability Engineering.* IEEE, 221–230.

[14] Satendra Kumar and Rajkumar. 2016. Test case prioritization techniques for software product line: A survey. In *Intl. Conference on Computing, Communication and Automation (ICCCA).* IEEE, 884–889.

[15] Remo Lachmann, Simon Beddig, Sascha Lity, Sandro Schulze, and Ina Schaefer. 2017. Risk-based integration testing of software product lines. In *11th Intl. Workshop on Variability Modelling of Software-intensive Systems.* 52–59.

[16] Remo Lachmann, Sascha Lity, Sabrina Lischke, Simon Beddig, Sandro Schulze, and Ina Schaefer. 2015. Delta-oriented test case prioritization for integration testing of software product lines. In *19th Intl. Conference on Software Product Line.* 81–90.

[17] Jackson A Prado Lima, Willian DF Mendonça, Silvia R Vergilio, and Wesley KG Assunção. 2020. Learning-based prioritization of test cases in continuous integration of highly-configurable software. In *24th ACM conference on systems and software product line.* 1–11.

[18] Jackson A Prado Lima and Silvia R Vergilio. 2020. Test Case Prioritization in Continuous Integration environments: A systematic mapping study. *Information and Software Technology* 121 (2020), 106268.

[19] Sascha Lity, Manuel Nieke, Thomas Thüm, and Ina Schaefer. 2019. Retest test selection for product-line regression testing of variants and versions of variants. *Journal of Systems and Software* 147 (2019), 46–63.

[20] Kai Ludwig, Jacob Krüger, and Thomas Leich. 2019. Covert and phantom features in annotations: Do they impact variability analysis?. In *23rd Intl. Systems and Software Product Line Conference-Volume A.* 218–230.

[21] Dusica Marijan, Arnaud Gotlieb, and Marius Liaaen. 2019. A learning algorithm for optimizing continuous integration development and testing practice. *Software: Practice and Experience* 49, 2 (2019), 192–213.

[22] Dusica Marijan and Marius Liaaen. 2018. Practical selective regression testing with effective redundancy in interleaved tests. In *40th Intl. Conference on Software Engineering: Software Engineering in Practice.* 153–162.

[23] Dusica Marijan, Marius Liaaen, Arnaud Gotlieb, Sagar Sen, and Carlo Ieva. 2017. TITAN: Test Suite Optimization for Highly Configurable Software. In *IEEE Intl. Conference on Software Testing, Verification and Validation (ICST).* IEEE, 524–531.

[24] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Rohit Gheyi, and Sven Apel. 2016. A Comparison of 10 Sampling Algorithms for Configurable Systems. In *38th Intl. Conference on Software Engineering (ICSE).* ACM, 643–54.

[25] Flávio Medeiros, Márcio Ribeiro, Rohit Gheyi, Sven Apel, Christian Kästner, Bruno Ferreira, Luiz Carvalho, and Baldoino Fonseca. 2018. Discipline Matters: Refactoring of Preprocessor Directives in the #ifdef Hell. *IEEE Transactions on Software Engineering* 44, 5 (2018), 453–469.

[26] Jens Meinicke, Chu-Pan Wong, Christian Kästner, Thomas Thüm, and Gunter Saake. 2016. On essential configuration complexity: Measuring interactions in highly-configurable systems. In *31st IEEE/ACM Intl. Conference on Automated Software Engineering.* 483–494.

[27] Willian DF Mendonça, Wesley KG Assunção, and Silvia R Vergilio. 2022. Software Product Line Regression Testing: A Research Roadmap. In *ICEIS.* SciTePress, 81–89. https://doi.org/10.5220/0010959700003179

[28] Willian DF Mendonça, Silvia R Vergilio, Gabriela K Michelon, Alexander Egyed, and Wesley KG Assunção. 2022. Test2Feature: feature-based test traceability tool for highly configurable software. In *26th ACM Intl. Systems and Software Product Line Conference-Volume B.* 62–65.

[29] Willian Mendonça, Silvia R Vergilio, and Wesley K G Assunção. 2023. Supplementary Material - Feature-oriented Test Case Selection during Evolution of High-Configurable Systems. https://doi.org/10.17605/OSF.IO/YD2WF

[30] Gabriela K Michelon, Wesley KG Assunção, David Obermann, Lukas Linsbauer, Paul Grünbacher, and Alexander Egyed. 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.* 2–15.

[31] Gabriela Karoline Michelon, David Obermann, Wesley KG Assunção, Lukas Linsbauer, Paul Grünbacher, Stefan Fischer, Roberto E Lopez-Herrejon, and Alexander Egyed. 2022. Evolving software system families in space and time with feature revisions. *Empirical Software Engineering* 27, 5 (2022), 112.

[32] Gabriela Karoline Michelon, David Obermann, Wesley Klewerton Guez Assunção, Lukas Linsbauer, Paul Grünbacher, and Alexander Egyed. 2020. Mining feature revisions in highly-configurable software systems. In *24th ACM Intl. Systems and Software Product Line Conference-Volume B.* 74–78.

[33] Mukelabai Mukelabai, Damir Nešić, Salome Maro, Thorsten Berger, and Jan-Philipp Steghöfer. 2018. Tackling combinatorial explosion: a study of industrial needs and practices for analyzing highly configurable systems. In *33rd ACM/IEEE Intl. Conference on Automated Software Engineering.* 155–166.

[34] Raiza Oliveira, Bruno Cafeo, and Andre Hora. 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, 1–9.

[35] Simone Romano, Giuseppe Scanniello, Giuliano Antoniol, and Alessandro Marchetto. 2018. SPIRITuS: A simple information retrieval regression test selection approach. *Information and Software Technology* 99 (2018), 62–80.

[36] Per Runeson and Emelie Engström. 2012. Chapter 7 - Regression Testing in Software Product Line Engineering. In *Advances in Computers*, Ali Hurson and Atif Memon (Eds.). Vol. 86. Elsevier, 223–263.

[37] T. Schiex and S. de Givry. 2019. Principles and Practice of Constraint Programming. In *25th Intl. Conference on Principles and Practice of Constraint Programming*, Vol. 11802. Springer.

[38] Paulo Anselmo da Mota Silveira Neto, Ivan do Carmo Machado, Yguarata Cerqueira Cavalcanti, Eduardo Santana De Almeida, Vinicius Cardoso Garcia, and Silvio Romero de Lemos Meira. 2010. A regression testing approach for software product lines architectures. In *2010 Fourth Brazilian Symposium on Software Components, Architectures and Reuse.* IEEE, 41–50.

[39] Davide Spadini, Maurício Aniche, and Alberto Bacchelli. 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, 908–911.

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

[41] Tugkan Tuglular and Sercan Şensülün. 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, 344–349.

[42] Alexander Von Rhein, Alexander Grebhahn, Sven Apel, Norbert Siegmund, Dirk Beyer, and Thorsten Berger. 2015. Presence-condition simplification in highly configurable systems. In *IEEE/ACM 37th IEEE Intl. Conference on Software Engineering*, Vol. 1. IEEE, 178–188.

[43] Shuai Wang, Shaukat Ali, Arnaud Gotlieb, and Marius Liaaen. 2017. Automated product line test case selection: industrial case study and controlled experiment. *Software & Systems Modeling* 16 (2017), 417–441.

[44] Shuai Wang, Arnaud Gotlieb, Shaukat Ali, and Marius Liaaen. 2013. Automated Test Case Selection Using Feature Model: An Industrial Case Study. In *Model-Driven Engineering Languages and Systems*, Ana Moreira, Bernhard Schätz, Jeff Gray, Antonio Vallecillo, and Peter Clarke (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 237–253.

[45] Shin Yoo and Mark Harman. 2012. Regression testing minimization, selection and prioritization: a survey. *Software testing, verification and reliability* 22, 2 (2012), 67–120.

[46] Yangyang Zhao, Alexander Serebrenik, Yuming Zhou, Vladimir Filkov, and Bogdan Vasilescu. 2017. The impact of continuous integration on other software development practices: a large-scale empirical study. In *32nd IEEE/ACM Intl. Conference on Automated Software Engineering (ASE).* IEEE, 60–71.