UNIVERSIDADE FEDERAL DO PARANÁ

WILLIAN DOUGLAS FERRARI MENDONÇA

TEST CASE SELECTION AND PRIORITIZATION DURING THE EVOLUTION OF HIGHLY CONFIGURABLE SYSTEMS

CURITIBA PR

2023

WILLIAN DOUGLAS FERRARI MENDONÇA

TEST CASE SELECTION AND PRIORITIZATION DURING THE EVOLUTION OF HIGHLY CONFIGURABLE SYSTEMS

Tese apresentada como requisito parcial à obtenção do grau de Doutor em Ciência da Computação no Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, da Universidade Federal do Paraná.

Área de concentração: *Ciência da Computação*.

Orientador: Silvia Regina Vergilio.

Coorientador: Wesley Klewerton Guez Assunção.

CURITIBA PR

2023

DADOS INTERNACIONAIS DE CATALOGAÇÃO NA PUBLICAÇÃO (CIP)
UNIVERSIDADE FEDERAL DO PARANÁ
SISTEMA DE BIBLIOTECAS – BIBLIOTECA DE CIÊNCIA E TECNOLOGIA

Mendonça, Willian Douglas Ferrari
Test case selection and prioritization during the evolution of highly configurable systems / Willian Douglas Ferrari Mendonça. – Curitiba, 2023.
1 recurso on-line : PDF.

Tese (Doutorado) - Universidade Federal do Paraná, Setor de Ciências Exatas, Programa de Pós-Graduação em Informática.

Orientador: Silvia Regina Vergilio
Coorientador: Wesley Klewerton Guez Assunção

1. Sistemas de recuperação da informação. 2. Software – Testes. 3. Integração de dados (Computação). I. Universidade Federal do Paraná. II. Programa de Pós-Graduação em Informática. III. Vergilio, Silvia Regina. IV. Assunção, Wesley Klewerton Guez. V. Título.

Bibliotecário: Elias Barbosa da Silva CRB-9/1894

MINISTÉRIO DA EDUCAÇÃO
SETOR DE CIÊNCIAS EXATAS
UNIVERSIDADE FEDERAL DO PARANÁ
PRÓ-REITORIA DE PESQUISA E PÓS-GRADUAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO INFORMÁTICA -
40001016034P5

TERMO DE APROVAÇÃO

Os membros da Banca Examinadora designada pelo Colegiado do Programa de Pós-Graduação INFORMÁTICA da Universidade Federal do Paraná foram convocados para realizar a arguição da tese de Doutorado de **WILLIAN DOUGLAS FERRARI MENDONÇA** intitulada: **TEST CASE SELECTION AND PRIORITIZATION DURING THE EVOLUTION OF HIGHLY CONFIGURABLE SYSTEMS**, sob orientação da Profa. Dra. SILVIA REGINA VERGILIO, que após terem inquirido o aluno e realizada a avaliação do trabalho, são de parecer pela sua APROVAÇÃO no rito de defesa.
A outorga do título de doutor está sujeita à homologação pelo colegiado, ao atendimento de todas as indicações e correções solicitadas pela banca e ao pleno atendimento das demandas regimentais do Programa de Pós-Graduação.

CURITIBA, 20 de Dezembro de 2023.

Assinatura Eletrônica
09/01/2024 15:42:06.0
SILVIA REGINA VERGILIO
Presidente da Banca Examinadora

Assinatura Eletrônica
09/01/2024 11:03:34.0
THELMA ELITA COLANZI LOPES
Avaliador Externo (UNIVERSIDADE ESTADUAL DE MARINGÁ)

Assinatura Eletrônica
02/01/2024 15:47:21.0
ANDREY RICARDO PIMENTEL
Avaliador Interno (UNIVERSIDADE FEDERAL DO PARANÁ)

Assinatura Eletrônica
05/01/2024 13:18:26.0
MARIA CLAUDIA FIGUEIREDO PEREIRA EMER
Avaliador Externo (UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ)

Assinatura Eletrônica
31/12/2023 10:28:50.0
WESLEY KLEWERTON GUEZ ASSUNÇÃO
Coorientador(a) (NORTH-CAROLINA UNIVERSITY)

Documento assinado eletronicamente de acordo com o disposto na legislação federal Decreto 8539 de 08 de outubro de 2015.
Gerado e autenticado pelo SIGA-UFPR, com a seguinte identificação única: 333861
Para autenticar este documento/assinatura, acesse https://siga.ufpr.br/siga/visitante/autenticacaoassinaturas.jsp e insira o codigo 333861

"Sempre e nunca são palavras que você sempre deve lembrar e nunca usá-las."

À minha esposa, Cláudia, e aos meus filhos, Benício e Joaquim, bem como aos meus pais, Gisneida e Juarez...

ACKNOWLEDGEMENTS

I would like to express my profound gratitude, first and foremost, to God.

A special thank you to my advisors, Dr. Silvia Regina Vergilio and Dr. Wesley Assunção, for the incredible opportunity to collaborate in such an enriching work environment. Their inexhaustible patience, constant guidance, and generosity in sharing knowledge were crucial to my academic growth. From the beginning of this process, they believed in me, understood my challenges, and guided me through overcoming obstacles.

I cannot fail to express my deep gratitude to my parents, Gisneida Ferrari de Mendonça and Juarez Alves de Mendonça, whose unwavering support was the foundation of this journey.

To my wife, Cláudia Vanessa Robl, I am grateful for her unwavering support and remarkable patience throughout this journey. Her presence has been a pillar of strength, and her dedication as an exemplary mother to our children, born during this doctoral process, is priceless.

To my fellow doctoral colleagues, Paulo Farah and Jackson Lima, who have become great friends on this journey. Without their support and collaboration, reaching this point would not have been possible. A special thanks to all colleagues in the GrES Research Group.

Finally, I want to express my gratitude to the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) for their valuable contribution and financial support to this work.

RESUMO

Garantir a qualidade de Sistemas Altamente Configuráveis (do inglês *Highly Configurable Systems (HCSs)* durante sua evolução e manutenção é um desafio. À medida que um HCS evolui, novas funcionalidades (Do inglês *features*) são adicionadas, alteradas ou removidas, o que torna o Teste de Regressão (do inglês *Regression Testing*, RT) uma tarefa difícil, ainda mais em um cenário de evolução em que são adotadas práticas de Integração Contínua (do inglês *Continuous Integration*, CI), o teste está normalmente sujeito a algumas restrições de tempo, o chamado orçamento de teste, e é necessário um *feedback* rápido sobre a execução do teste. Assim, as técnicas que demoram demasiado tempo a executar não são adequadas. Para enfrentar estes desafios, existem na literatura algumas abordagens que aplicam diferentes técnicas de RT, tais como Seleção de Casos de Teste (do inglês *Test Case Selection*, TCS) e Priorização de Casos de Teste (do inglês *Test Case Prioritization*, TCP) no contexto de CI. No entanto, a grande maioria dessas abordagens não considera as particularidades dos HCSs. As abordagens existentes dependem normalmente de artefatos como o *Feature Model*, que na maioria dos casos não estão disponíveis ou não são atualizados. As abordagens que podem ser utilizadas são baseadas em alterações de arquivos e funcionam apenas para a linguagem Java. Dadas estas limitações, este trabalho tem como objetivo investigar as vantagens da aplicação de abordagens de RT orientadas a *feature* durante o processo de evolução de HCSs. A *feature* é um conceito chave para os HCSs, e a hipótese deste trabalho é que uma abordagem orientada a *features* pode selecionar e priorizar casos de teste de uma forma rentável. Para investigar essa hipótese, propomos a `FeaTCSP` (**Feat**ure-oriented **T**est **C**ase **S**election and **P**rioritization for Highly Configurable Systems), uma abordagem orientada a *feature* que permite o uso das técnicas de TCS e TCP considerando a evolução das *feature* do HCS no espaço e no tempo e as restrições de um ambiente de CI. Esta abordagem recebe como entrada um arquivo de configuração contendo os caminhos para o código fonte da HCS e a pasta de casos de teste, e engloba passos diferentes e independentes, que podem ser executados pelo testador de acordo com os seus objetivos. A `FeaTCSP` produz diferentes relatórios: as linhas de código que correspondem a cada *feature* , as linhas exercitadas por cada caso de teste e os casos de teste ligados a cada *feature*. Orientado por estes relatórios, o testador pode utilizar a abordagem para selecionar um número reduzido de casos de teste, para produzir um *rank* de prioridades para a execução dos casos de teste, ou para produzir um *rank* para o conjunto de testes reduzido selecionado anteriormente. Todas as formas de utilização da `FeaTCSP` são avaliadas utilizando o `Libssh`, um HCS indústrial de código aberto em constante evolução. Os resultados mostram a aplicabilidade da abordagem e que esta contribui para o RT em HCSs. Em todas as utilizações, `FeaTCSP` apresenta melhor desempenho do que uma *baseline* orientada para alterações de arquivos. Neste sentido, `FeaTCSP` contribui para escolher um conjunto reduzido de casos de teste sem comprometer a qualidade do teste em termos de falhas. Além disso, a utilização da técnica TCP tem como objetivo a detecção precoce de falhas, reduzindo o tempo gasto na execução dos testes.

Palavras-chave: Sistemas Altamente Configuráveis, Teste de Software, Integração Contínua

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 *Regression Testing (RT)* a hard task, even more in an evolution scenario where *Continuous Integration (CI)* practices are adopted and updates are frequent in different cycles of integration, built and testing. In CI environments, the test is usually subject to the some time restrictions, the so-called test budget, and rapid feedback on the test execution is required. Then techniques that take too long to execute are not suitable. To address these challenges, in the literature there are some approaches that apply different RT techniques, such as *Test Case Selection (TCS)* and *Test Case Prioritization (TCP)* in CI context. However, the great majority of these approaches do not consider HCS particularities. Existing approaches usually rely on the artifacts, such as Feature Model, which are not in most cases available or updated. Approaches that can be used are based on file changes and work only for Java language. Given these limitations, this work aims to investigate the advantages of applying feature-oriented RT approaches during the evolution process of HCSs. Feature is a key concept for HCSs, and the hypothesis of this work is that a feature-oriented approach can select and prioritize test cases in an cost-effective way. To investigate this hypothesis, we propose `FeaTCSP` (**Feat**ure-oriented **T**est **C**ase **S**election and **P**rioritization for Highly Configurable Systems), a feature-oriented approach that allows the use of TCS and TCP techniques considering the evolution of the HCS features in space and time and the restrictions of a CI environment. This approach takes as input a configuration file containing the paths to the source code of the HCS and the test case folder, and encompasses different and independent steps, which can be executed by the tester according to her(him) objectives. `FeaTCSP` produces different reports: the lines of code that correspond to each feature, the lines exercised by each test case, and the test cases linked to each feature. Oriented by these reports, the tester can use the approach for selecting a reduced number of test cases, to produce a prioritization rank for executing the test cases, or to produce a rank for the selected reduced test set. All the ways of using `FeaTCSP` are evaluated by using `Libssh`, a real open-source HCS in constant evolution. The results shows the applicability of the approach and that it contributes to the RT of HCSs. In all uses, `FeaTCSP` presents better performance than a baseline oriented to file changes. In this sense, `FeaTCSP` contributes to choose a reduced test case set without compromising the testing quality in terms of failures. Moreover, the use of TCP technique aims to provide early failures detection, reducing the time spent for test execution.

Keywords: Highly Configurable Systems, Software Testing, Continuous Integration.

LIST OF ACRONYMS

DINF	Departamento de Informática
PPGINF	Programa de Pós-Graduação em Informática
UFPR	Universidade Federal do Paraná
HCSs	Highly Configurable Systems
RT	Regression Testing
CI	Continuous Integration
TCS	Test Case Selection
TCP	Test Case Prioritization
FeaTCSP	Feature-oriented Test Case Selection and Prioritization for Highly Configurable Systems
SPL	Software Product Line
TCM	Test Case Minimization
FM	Feature Model
Test2Feature	Feature-based Test Traceability Tool for Highly Configurable Software
FeatTestSel	Feature-oriented Test Case Selection during Evolution of Highly-Configurable Systems
VCSs	Version Control Systems
OSF	Open Science Framework

CONTENTS

1 INTRODUCTION . . . 9
1.1 MOTIVATIONS . . . 10
1.2 OBJECTIVES . . . 11
1.3 METHODOLOGY . . . 11
1.4 PUBLICATIONS . . . 13
1.5 TEXT OUTLINE . . . 14
2 BACKGROUD . . . 15
2.1 REGRESSION TESTING TECHNIQUES . . . 15
2.2 REGRESSION TESTING OF SOFTWARE PRODUCT LINES . . . 15
2.3 CONTINUOUS INTEGRATION ENVIRONMENTS . . . 17
2.4 SOFTWARE PRODUCT LINE REGRESSION TESTING: A RESEARCH ROADMAP . . . 17
2.4.1 Motivations . . . 18
2.4.2 Objectives . . . 18
2.5 CONCLUDING REMARKS . . . 18
3 TEST2FEATURE: FEATURE-BASED TEST TRACEABILITY TOOL FOR HIGHLY CONFIGURABLE SOFTWARE . . . 28
3.1 MOTIVATIONS . . . 28
3.2 OBJECTIVES . . . 28
3.3 CONCLUDING REMARKS . . . 28
4 FEATURE-ORIENTED TEST CASE SELECTION DURING EVOLUTION OF HIGHLY-CONFIGURABLE SYSTEMS . . . 33
4.1 MOTIVATIONS . . . 33
4.2 OBJECTIVES . . . 33
4.3 CONCLUDING REMARKS . . . 33
5 FEATURE-ORIENTED TEST CASE SELECTION AND PRIORITIZATION FOR HIGHLY-CONFIGURABLE SYSTEMS . . . 46
5.1 MOTIVATIONS . . . 46
5.2 OBJECTIVES . . . 46
5.3 CONCLUDING REMARKS . . . 46
6 CONCLUSIONS . . . 60
6.1 CONTRIBUTIONS . . . 61
6.2 LIMITATIONS . . . 62
6.3 FUTURE WORK . . . 62
REFERENCES . . . 63

1 INTRODUCTION

Software systems must be designed to meet the demands and requirements of users and customers. However, by nature, users/customers have different needs, mainly related to their business domain, organizational process, environment restriction, and specialized hardware devices. Currently, software systems must take into account these differences in needs and operate on them, in order to be successful systems. *Software Product Line (SPL)* is an approach for developing and managing family of software products that can be customized with different configurations (variabilities), while common software assets can be reused in a systematic and disciplined way (Van der Linden et al., 2007). SPLs are generally implemented as *Highly Configurable Systems (HCSs)*, using different configuration options and applying strategies such as conditional compilation, conditional execution or, build systems-to create custom system products, also known as variants (Mukelabai et al., 2018; Von Rhein et al., 2015).

With the widespread adoption of the agile paradigm, most software organizations adopt *Continuous Integration (CI)* practices in the software development (Lima and Vergilio, 2020b). A CI environment allows more frequent integration of software changes, making software evolution faster and more cost-effective (Zhao et al., 2017). This is because 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. Then, to provide quick feedback is essential to reduce development costs (Jiang and Chan, 2016). During the software lifecycle, continuous changes occur, whether in the system itself or in its environment. The numerous changes made to the software can make it more complex and different from the original design, decreasing the quality of the software. After the changes, software engineers must perform *Regression Testing (RT)* to confirm that the changes have not adversely affected existing system features. However, RT is an expensive task (Yoo and Harman, 2012).

A common approach to RT in practice is retest-all, which is to rerun all tests, sometimes including those that do not specifically cover the parts of the software affected by the changes. However, to make regression testing more efficient, only the relevant minimum set of tests that adequately cover the changes should be selected and run. Although the retest-all approach is easily automated, selective regression testing is mostly a manual process done by developers/testers and, therefore, often unsystematic and subject to subjectivity. As test suites grow in size over time, introducing redundancy, manually selecting a suitable non-redundant set of regression tests soon becomes inefficient and impractical (Marijan et al., 2019). This is especially evident in CI development, where regression testing is performed as part of a development iteration limited to a specific duration, the test budget. Because these interactions are short, less time is available for testing, making it difficult to select regression tests using a manual approach.

To automate RT tasks, we found in the literature three basic RT techniques are used(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 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.

These RT techniques have been explored and adapted for the CI context (Lima and Vergilio, 2020b). However, few of these works consider HCS particularities. HCSs are complex

and predominantly configurable software, which means that they incorporate a series of options (a.k.a., features) used for software customization. This allows different functionalities and configurations to be selected to a given user context (Von Rhein et al., 2015), what makes software testing a hard activity (Medeiros et al., 2018; Machado et al., 2014). Software Testing of HCSs is challenging, even more when we consider an evolution scenario, i.e. where (CI) practices are adopted. HCSs can be updated, integrated, and tested several times a day, and each cycle needs to be fast (Zhao et al., 2017).

RT techniques have already been applied to SPL/HCS as can be seen in reviews published in literature (Engström, 2010a,b; Mendonça et al., 2022a). But we observe that few works adopt the RT techniques in a complementary. When employing combined RT techniques, we can leverage the best of each technique individually. The goal is always to obtain the smallest possible set of test cases without compromising quality (TCS) and to order this set with the best possible execution sequence (TCP). Existing TCS approaches are usually based on *Feature Model (FM)* or other artifacts (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 models are rarely updated, making the use of those approaches difficult. Other HCS testing approaches need some kind of 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), in a context where the test cases are separated by feature and do not overlap. This is different from real scenarios. Approaches based on code changes are more suitable and used. For instance, approaches that 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.

Some pieces of work introduce methods based on changed files or versions (Gligoric et al., 2015; Bertolino et al., 2020; Romano et al., 2018). Bertolino et al. (2020) 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 (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 traceability of test cases to features is extremely important, as it can be used for various analyses in the evolutionary process of an HCS: the number of test cases linked to features, features without linked test cases, feature coverage, using this data as input for other approaches, among other applications.

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 application 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. Existing works are mostly based on Java language (Jung et al., 2019, 2020, 2023). 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. So none of these works consider traceability for features but for source code. Furthermore, they do not consider combined RT applications, such as TCS and TCP.

1.1 MOTIVATIONS

Testing in an agile evolutionary environment or in a CI environment is extremely expensive and it becomes unfeasible to use the retest-everything technique for each evolution of the system. This evolution happens continuously and needs to be quick (Zhao et al., 2017). Test activity is already a challenge for single systems, but when we think in the evolution context of HCSs, this challenge is even greater. They incorporate a substantial number of configuration parameters, causing high complexity of HCS tests (Medeiros et al., 2017). Typically, most configuration parameters are interrelated through inclusive or exclusive relationships, which further increases the effort of the HCS test. Also, if the software is constantly released following a continuous delivery model, there are strict time restrictions imposed on test runs. To efficiently address these challenges with an optimal trade-off between time, cost and test quality, professionals need efficient test approaches (Marijan et al., 2017).

The combined techniques of RT can be highly effective in this context. However, the presented approaches that employ this strategy are specific and require various adaptations to be applied in the evolution process of HCS or CI. Given that evolution in these contexts occurs very rapidly, any approach involving manual processes or relying on dynamic source code analysis may take a considerable amount of time to execute, becoming unfeasible due to testing budget constraints. Even approaches that utilize learning models or search algorithms requiring long execution times may exceed the stipulated budget in CI environments.

Therefore, investigating or adapting RT techniques for HCS is fundamental. The works found addressing this issue (Marijan and Liaaen, 2018; Marijan et al., 2017, 2019; Marijan and Sen, 2017; Lima et al., 2020; Pett et al., 2021) present some generalization problems or do not apply a selection technique that considers the evolution of features over space and time. Additionally, they do not use source code as a basis for analysis.

The approaches proposed by Jung et al. (Jung et al., 2019, 2020, 2023) use source code as a basis for analysis and apply a test case selection process. However, these approaches focus exclusively on Java source code, following the trend of the majority of works. These works only consider test traceability for source code, disregarding the features of the systems and their evolution. Additionally, they assume that HCS/SPL are well-developed, using simplified (toy) systems for validation. Our work aims to fill this gap by proposing a TCS and TCP approach that is applicable to the evolution process of HCS or to HCS in a CI environment. The core of our approach is the features of HCSs, as the feature represents the core of HCS development. When considering features, we can cover several branching points of the HCS, going beyond the simple changed file or the specific function modified in that commit.

1.2 OBJECTIVES

The goal main is investigate the advantages of applying feature-oriented RT approaches during the evolution process of HCSs. The specific objectives of the approach are:

- Reduce the high cost of the RT step;
- Reduce the set of test cases, and maintain quality;
- Prioritize the set of test cases, in order to reveal defects early;
- Develop an approach that can be used in the CI environment.

1.3 METHODOLOGY

This work aims to investigate the advantages of applying feature-oriented RT approaches during the evolution process of HCSs. The hypothesis of this work is that a feature-oriented approach can select and prioritize test cases in the context of HCS in an cost-effective way. In this sense, it contributes to choose a reduced test case set without compromising the testing quality in terms of detected fault. Moreover, the use of TCP technique aims to provide early fault detection, reducing the time spent for test execution.

To investigate the above-mentioned hypothesis, we propose `FeaTCSP` (**Feat**ure-oriented **T**est **C**ase **S**election and **P**rioritization for Highly Configurable Systems), a feature-oriented approach that allows the use of TCS and TCP techniques considering the evolution of the HCS features in space and time and the restrictions of a CI environment. This approach takes as input a configuration file containing the paths to the source code of the HCS and the test case folder, and encompasses different steps as described below.

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.

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).

3. *Map features to test cases*: the lines exercised by each test case are identified and traceability links between feature and test cases are created.

4. *Select test cases*: the output of the previous steps 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.

5. *Apply prioritization strategies*: by using the reports generated in the previous steps, a TCP prioritization strategy is used and a rank order of test cases is produced. The list of test cases to be prioritized can be composed by all test cases available or by the test cases selected in the previous step. To perform prioritization, three strategies are proposed, based on the mapping of features to test cases generated and uses three information sources: 1) `FeatChgHist`: the history of feature changes; 2) `FeatChgCommit`: features that changed in the current commit; and 3) `FeatCov`: the number of features covered by the test case.

To implement the first three steps, we implement a tool, namely `Test2Feature` (Mendonça et al., 2022b). This tool given the source code of an annotated HCS and a test case set, produces test traceability reports linking test cases to the HCS features. The tool encompasses three modules that generate three main outputs: traceability from features to code lines, traceability from test cases to code lines, and traceability from features to tests.

The steps can be executed by the tester according to her(him) objectives, which can be only to follow the feature evolution along the commits, or using the reports for RT. S(he) can adopt the approach to 1) only to reduce the number of test cases, that is, as a TCS approach, which we named `FeatTestSel`; 2) only for TCP, approach we named `FeatTCP`, to reduce the test execution time searching early fault detection; or 3) to use TCS and TCP in a combined way, approach we named `FeatTCSP`.

We conducted different investigations on the use of these different ways of our approach, considering a real-world HCS (Mendonça et al., 2023b,a). The results shows that the approach

contributes to perform RT of HCSs considering the evolutions of these features in space in time in a very cost-effective way, compared with a baseline, a file-based approach. This work contributes to the area of RT by proposing a feature-oriented approach applicable in the evolutionary process of HCSs and in CI environments. Such an approach is lightweight and model-free, capable of automating Test Case Selection and Prioritization (TCSP). It is independent of any learning or optimization model. Additionally, instead of dynamic analysis, which would require execution, we employ static analysis that relies solely on the source code of the HCS as input. `FeatTCSP` has been designed to seamlessly integrate into the evolutionary process of HCSs, aiming to substantially reduce the cost associated with test case execution.

1.4 PUBLICATIONS

During the doctorate period, two awards were received: Best Paper of the main track of SPLC2023 and Best Paper of Industry track of ERES2018. The following papers were produced:

1. (Mendonça et al., 2024) Mendonça, W. D. F., Assunção, W. K. G., and Vergilio, S. R. (2024). Feature-oriented test case selection and prioritization for highly-configurable system. This paper will be submitted to a journal or conference in the area.

2. (Mendonça et al., 2023b) MENDONÇA, W. F.; ASSUNCAO, W. K. G.; VERGILIO, S. R. Feature-oriented Test Case Selection during Evolution of Highly-Configurable Systems. Software Product Line Conference 2023 (SPLC), Tokyo. ACM, New York. pg 76-86. August/September 2023.

3. (Mendonça et al., 2022b) MENDONÇA, W. F.; MICHELON, G.K.; VERGILIO, S. R.; ASSUNCAO, W. K. G.; EGYED, A. (2022) Test2Feature: Feature-based Test Traceability Tool for Highly Configurable Software, SPLC2022.

4. (Prado Lima et al., 2022) LIMA, J. A. P.; MENDONÇA, WILLIAN D. F.; ASSUNCAO, W. K. G.; VERGILIO, S. R. Cost-effective learning-based strategies for test case prioritization in Continuous Integration of Highly-Configurable Software. EMPIRICAL SOFTWARE ENGINEERING, 2022.

5. (Mendonça et al., 2022a) MENDONÇA, W. F.; ASSUNCAO, W. K. G.; VERGILIO, S. R. Software Product Line Regression Testing: a Research Roadmap. International Conference on Enterprise Information Systems (ICEIS), 2022, Prague.

6. (Lima et al., 2020) LIMA, J. A. P.; MENDONÇA, W. D.F.; VERGILIO, SILVIA R., ASSUNÇÃO, W. K. G. Learning-based Prioritization of Test Cases in Continuous Integration of Highly-Configurable Software. Sofware Product Line Conference (SPLC2020)

7. (Mendonça et al., 2020) MENDONÇA, W. D.F.; ASSUNÇÃO, W.K.G.; ESTALISLAU, L. V. ; VERGILIO, S. R. ; GARCIA, A. Towards a Microservices-Based Product Line with Multi-Objective Evolutionary Algorithms. IEEE Congress on Evolutionary Computation (CEC2020)

8. (Mendonça et al., 2019) MENDONÇA, W. D. F. ; ASSUNÇÃO, W. K. G. ; VERGILIO, S. R. Reusing Test Cases on Graph Product Line Variants. In: the IV Brazilian Symposium, 2019, Salvador. Proceedings of the IV Brazilian Symposium on Systematic and Automated Software Testing - SAST 2019. New York: ACM Press, 2019. p. 52.

9. (da Silva et al., 2019) DA SILVA, H. N. ; FARAH, P. R. ; MENDONÇA, W. D. F. ; VERGILIO, S. R. Assessing Android Test Data Generation Tools via Mutation Testing. In: the IV Brazilian Symposium, 2019, Salvador. Proceedings of the IV Brazilian Symposium on Systematic and Automated Software Testing - SAST 2019, 2019. p. 32.

10. (Assunção et al., 2018) ASSUNCAO, W. K. G. ; MENDONÇA, W. D. F. ; VERGILIO, S. R. .Reúso de Software: Do Oportunista ao Sistemático. In: Escola Regional de Engenharia de Software (ERES 2018), 2018, Dois Vizinhos.

1.5 TEXT OUTLINE

This thesis is a compilation of the main related papers produced during the doctorate period and mentioned above. The document is organized as follows:

Chapter 2 - Research on SPL Regression Testing: this chapter describes the background for the understanding of this work. It presents concepts related to RT, SPL/HCS, and Continuous Integration environments. Moreover it overviews the current knowledge on SPL regression testing and presents a research roadmap for the upcoming years. This chapter relies on Paper 5.

Chapter 3 - Test2Feature: Feature-based Test Traceability Tool for Highly Configurable Software: this chapter introduces `Test2Feature`, the tool that implements Steps 1 to 3 of our approach. Given the source code of an annotated HCS and a test case set, Test2Feature produces test traceability reports linking test cases to the HCS features. This chapter relies on Paper 3.

Chapter 4 -FeatTestSel- Feature-oriented Test Case Selection during Evolution of Highly-Configurable Systems: this chapter presents the `FeaTestSel` approach. This approach selects the best test cases to be executed in order to cover the features changed in the corresponding evolution cycle. This chapter relies on Paper 2.

Chapter 5 -FeatTCSP- Feature-oriented Test Case Selection and Prioritization for Highly-Configurable Systems: this chapter introduces the `FeatTCSP` approach. The approach that allows TCS and TCP approach. This chapter describes the prioritization strategies implemented and the results of different evaluations conducted. This chapter relies on Paper 1.

Chapter 7 - Conclusion: this chapter concludes this thesis, by presenting a summary of the thesis, main contributions, limitations, and future work.

2 BACKGROUD

In this chapter, we review background related to our work. In Section 2.1 we present the main Regression Testing techniques (TCP and TCS). In Section 2.2 we present concepts related to SPL and HCS. In Section 2.3 we show some characteristics of the continuous integration environments. Finally, in Section 2.4 we present a systematic review of the current knowledge on regression testing in SPL and provide a roadmap for the coming years.

2.1 REGRESSION TESTING TECHNIQUES

Changes are frequent during the software life-cycle. In order to check if any change has not adversely affected existing features of the system, software engineers perform RT. The most straightforward RT approach, known as re-test all, executes all the current test cases in the test suite. However, such an approach presents high costs (Yoo and Harman, 2012).

Many techniques have been studied as alternatives to aid the RT process, seeking to reduce the effort required in various ways (Yoo and Harman, 2012). Among these techniques, the main ones are TCM, TCS, and TCP. TCM seeks to maintain the most important test cases in the test suite, removing obsolete or redundant test cases. TCS aims to select a subset of test cases, the most important ones to test the software. TCP attempts to re-order a test suite to identify an "ideal" ordering of test cases that maximizes specific goals, such as early fault detection.

While test case selection techniques also seek to reduce the size of a test suite, the majority of the selection techniques are modification-aware. That is, the selection is not only temporary (i.e. specific to the current version of the program), but also focused on the identification of the modified parts of the program. Test cases are selected because they are relevant to the changed parts of the SUT, which typically involves a white-box static analysis of the program code. More formally, following Yoo and Harman (2012), the selection problem is defined as follows. Given the program, P, the modified version of P , P', and a test suite, T. The TCS problem concerns the selection of a test set $T' \subseteq T$ to be executed on P'.

TCP concerns ordering test cases for early maximization of some desirable properties, such as the rate of fault detection. It seeks to find the optimal permutation of the sequence of test cases. It does not involve selection of test cases, and assumes that all the test cases may be executed in the order of the permutation it produces, but that testing may be terminated at some arbitrary point during the testing process. More formally, given a test suite T, the set PT of all possible permutations of T, and a function f that determines the performance of a given prioritization T'' from PT to real numbers, the TCP problem aims at finding the best T' to achieve certain specific criteria measured by f. The TCP problem is defined as follows (Rothermel et al., 2001): Find $T' \in PT$ such that $(\forall T'')$ $(T'' \in PT)$ $(T'' \neq T')$ $[f(T') \geq f(T'')]$.

2.2 REGRESSION TESTING OF SOFTWARE PRODUCT LINES

A software system usually must be delivered with different configurations of features, with each feature representing a functionality of the system accessible to developers and users. To remain competitive, companies have to satisfy different customer needs of the market segment they serve. SPL engineering is a systematic approach to deal with the development of customized system products (Pohl et al., 2005; Assunção et al., 2017). An SPL is a set of software-intensive systems that share a common set of artifacts developed in a prescribed way to facilitate their systematic

reuse (Clements and Northrop, 2001). The customized software products, a.k.a. variants, result from the derivation of SPL artifacts, i.e., the selection of a different set of features that are of interest to a customer. To allow customization, the features of an SPL are implemented using variability mechanisms (Apel et al., 2013).

A widely used variability mechanism in SPLs is based on annotations (Queiroz et al., 2017). Annotations rely on preprocessor directives such as #ifdef and #endif, that enclose blocks of variable code and enable to tailor system variants to different hardware platforms, operating systems, and application scenarios (Medeiros et al., 2017). Annotation-based SPLs are often implemented as HCSs (Jin et al., 2014). HCSs use techniques such as feature flags, feature toggles, or feature switches, to turn on configuration options/features needed to be included in a product (Dintzner et al., 2018; Kästner et al., 2011). However, features also need to evolve over time. For instance, when a specific feature is adapted to a new hardware platform, then a new version of a variant is created. This evolution in time (Seidl et al., 2014) is aided by some tools such as Version Control Systems (VCSs) (Ruparelia, 2010).

Regarding the test activity of SPLs/HCSs, engineers face challenges related to the number of versions and variants of software products to be tested, and consequently the risk for redundant testing. Also, the testing challenges resemble those of regression testing for single-product software, although adding the complexity of parallel variants (Runeson and Engström, 2012; Engström, 2010b).

Figure 2.1 overviews the main elements of the SPL regression testing process. The main difference between regression testing of SPLs and single-product software is the existence of a set of product variants to be tested (at the top of Figure 2.1). Then, in addition to the management of test suites with several test cases (at the bottom of Figure 2.1), in the SPL context there is also a set of products to be taken into account. But differently from the test case suite, this set of products may include a huge number of product variants, potentially exponential and even infeasible of being reached. Thus, strategies to test only the most relevant ones are needed.

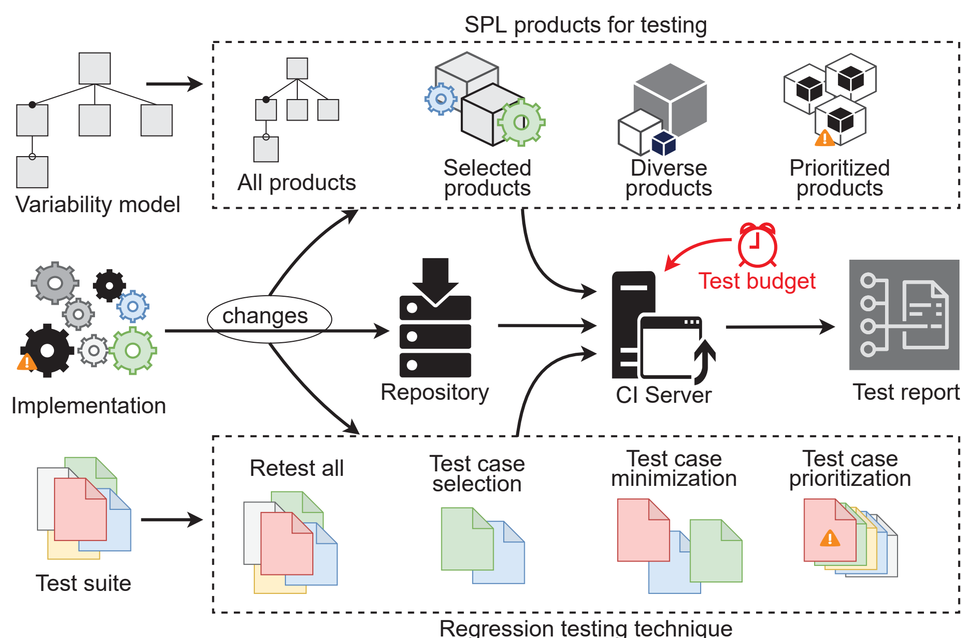


Figure 2.1: SPL regression testing process

To perform SPL regression testing we can use different techniques for both, test suites and product variants. For example, for small SPLs, retesting all test cases on all products can be feasible, even though retest-all is a technique known to be highly redundant and costly (Qu et al.,

2012). However, real-world systems usually deal with huge test suites (bin Ali et al., 2019). This leads to the need of techniques to efficiently conduct SPL regression testing. The main regression testing techniques are presented in the bottom of Figure 2.1 (presented in Section 2.1). Similar techniques are used for the SPL products, but in this case, minimization is usually concerned with a set of diverse products that represent features combinations, e.g., pair-wise criterion (Runeson and Engström, 2012; Engström, 2010b).

2.3 CONTINUOUS INTEGRATION ENVIRONMENTS

Continuous Integration plays an important role in agile development, allowing reduced integration effort, lower number of uncorrected errors for long periods, and delivering a product version at any moment. In CI development, teams work continuously integrating code and make smaller code commits every day, usually monitored by a CI server. When a change occurs, the CI server clones this code, builds it, and runs the testing processes. When the entire process is finished, a report (feedback) is generated by the CI server, and the developers are informed. Figure 2.2 illustrates the process.

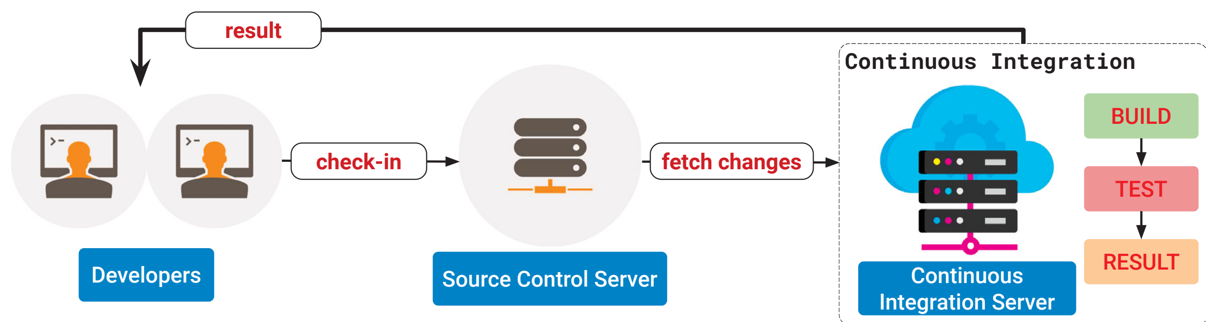


Figure 2.2: Overview of a Continuous Integration environment adapted from Lima and Vergilio (2020a)

CI automated support is very important to CI in practice. Zhao et al. (2017) present some results about the impact of adopting the framework Travis CI on development practices in a collection of GitHub projects. They observed an increase in the number of daily commits (frequency of 78 commits every day) and, after some initial adjustments, an increase in the number of automated tests.

We can see that in this scenario presented by Zhao et al. (2017) to re-execute all test cases is unfeasible, and it is fundamental to perform RT activities in a very cost-effective way to ensure that recent changes have not negatively impacted functionality previously tested, and, considering CI goals, to provide a rapid test feedback on software failures. Besides, in a company, multiple projects may share the same CI workflow, and regression testing usually runs a time restricted to a specific duration, the test budget. This makes difficult the use of traditional RT techniques for minimization and selection of test cases that rely on code analysis and instrumentation. They are time-consuming and produce results that quickly become inaccurate due to the frequent changes (Elbaum et al., 2014). But in the context of HCSs, the application of a TCS technique becomes indispensable, considering the large number of different test cases for different variants. This code and instrumentation analysis must happen incrementally, following frequent changes, and still be able to keep a low testing budget. In this scenario TCP techniques improve the cost-effectiveness of regression testing by ordering test cases to allow early execution of the most important ones, usually those test cases with a high probability of revealing failures (Elbaum et al., 2014).

2.4 SOFTWARE PRODUCT LINE REGRESSION TESTING: A RESEARCH ROADMAP

In this section, we present a comprehensive exploration of the current landscape and challenges in the realm of regression testing for SPLs. The roadmap the increasing complexity of SPLs and the critical necessity for effective regression testing strategies to ensure the continual quality of these product lines.

2.4.1 Motivations

Our motivation is in the absence of a consolidated research roadmap to guide the development of specific regression testing techniques for SPLs. The intricate nature of SPLs, with their multiple variants and interdependencies, highlights the need for a strategic vision to direct research efforts in addressing the unique challenges in this domain.

2.4.2 Objectives

The primary objective is to establish a research roadmap that guides both the academic and industrial communities in identifying and developing advanced regression testing methods tailored to SPLs. The paper proposes a framework to address issues such as efficient test case selection, managing changes in variants, and minimizing the execution cost of tests in SPLs.

2.5 CONCLUDING REMARKS

This chapter introduced topics related to this work, including main concepts regarding RT techniques, CI environment, and SPL regression testing process. We observe that the application of RT techniques in the CI of SPL/HCS is fundamental to reduce efforts and costs. But RT approaches with such a goal have faced some additional challenges related to the number of variants and the features evolving in space and time. Furthermore, we emphasize the strategic importance of addressing specific challenges related to regression testing in Software Product Lines. The proposed roadmap provides a valuable guide for future research, encouraging innovation in creating more effective and efficient testing techniques to ensure continuous quality in dynamic and ever-evolving SPL environments. Furthermore, we consider this work as the initial basis for work related to the thesis, but in the course of the next articles/chapters we will present new work and updates in the related work sections in papers.

Software Product Line Regression Testing: a Research Roadmap

Willian D. F. Mendonça[1], Wesley K. G. Assunção[2] and Silvia R. Vergilio[1]
[1] *DInf, Federal University of Paraná, Curitiba, Brazil*
[2] *DI, Pontifical Catholic University of Rio de Janeiro, Rio de Janeiro, Brazil*
willianmendonca@ufpr.br, silvia@inf.ufpr.br, wassuncao@inf.puc-rio.br

Keywords: Software testing, Software reuse, Software evolution, Research opportunities, Systematic mapping

Abstract: Similarly to traditional single-product software, Software Product Lines (SPLs) are constantly maintained and evolved. However, an unrevealed bug in an SPL can be propagated to a wide set of products and impact customers differently, depending on the set of features they are using. In such scenarios, SPL regression testing is paramount to avoid undesired problems and guarantee that the SPL maintenance and evolution are performed accordingly. Although there are several studies on SPL regression testing, the research community lacks a clear set of research opportunities to be addressed in a short and medium term. To fulfill this gap, the goal of this work is to overview the current body of knowledge of SPL regression testing and present a research roadmap for the following years. For this, we conducted a systematic mapping study that found 27 primary studies. We identified techniques used by the approaches, and applied strategies. Test case selection and prioritization techniques are prevalent, as well as fault and coverage based criteria. Furthermore, based on gaps and limitations reported in the studies we distilled a set of future work opportunities that serve as a guide for new research in the field.

1 Introduction

Software Product Line Engineering is a reuse-oriented approach to systematically develop families of software systems. A Software Product Line (SPL) allows cost-efficiently derivation of tailored products to specific markets, utilizing common and variable assets in a planned manner (Linden et al., 2007). We have seen several pieces of work describing adoption of SPLs in industry in the last years (Grüner et al., 2020; Abbas et al., 2020).

Similarly to traditional single-product software development, SPLs are constantly maintained and evolved (Marques et al., 2019). However, an unrevealed bug in an SPL can be propagated to a wide set of products and impact customers differently, depending on the set of features in the products they use. In such a scenario, SPL testing is paramount to avoid undesired problems (do Carmo Machado et al., 2014; Engström and Runeson, 2011). More specifically, regression testing has the role of guaranteeing that maintenance and evolution of SPLs are performed accordingly (Runeson and Engström, 2012; Engström, 2010b).

Although there are several recent primary studies on SPL regression testing, the research community lacks a clear set of research opportunities to be addressed in a short and medium term. The last secondary pieces of work on this topic were published in 2010 (Engström, 2010b; Engström, 2010a). Therefore, there is a need for an updated and comprehensive study to fulfill this gap (bin Ali et al., 2019; Marques et al., 2019). Based on this, the goal of this work is to overview the current scenario and existing body of knowledge of SPL regression testing to present a research roadmap for the following years. For this, we conducted a systematic mapping study to collect existing studies on SPL regression testing (Petersen et al., 2015). Guided by four research questions that aim to identify existing SPL regression testing approaches and their main characteristics, we identified 27 primary studies published in the period of 2005 to 2020. More than 85% of them were published after 2010, therefore, not discussed in the last literature review on the topic (Engström, 2010a).

The approaches proposed in the collected studies are analyzed considering regression testing technique supported, input and output artifacts used, strategy to apply the technique, and testing criteria adopted. As a result, and main contribution of our work, we distilled a roadmap with research opportunities for future work. This roadmap spans through the whole

process of regression testing. It serves as a guide to motivate new research in the field and make existing approaches adopted in practice.

2 Related Work

In the literature, we can find pieces of work that report secondary studies for general regression testing (Yoo and Harman, 2012; Minhas et al., 2017). Also, studies that carry out mapping and surveys for SPL testing in general (Lee et al., 2012; do Carmo Machado et al., 2014), and other ones, more related to ours, addressing specifically the topic of SPL regression testing (Engström, 2010b; Engström, 2010a). However, these pieces of work were published in 2010. As a consequence, they do not encompass more than 10 years of research and practice in the field. The analysis of more recent studies, published in the last decade, allows us to derive new opportunities, and to discuss new trends not presented in related work.

Besides the studies on testing and regression testing, some papers published in the last two years have described studies on diverse topics of SPLs. For example, a mapping study on SPL evolution (Marques et al., 2019) reinforces the need for regression testing. We can also mention secondary studies on SPL and variability management in emerging technologies, such as IoT (Geraldi et al., 2020), microservices (Mendonça et al., 2020; Assunção et al., 2020), and composition of ML/AI products with SPLs (Nomme, 2020). Based on that, we can argue the need of focusing on regression testing.

3 Study Design and Execution

The goal of our study is *to overview the current scenario and body of knowledge of SPL regression testing to present a research roadmap for the following years*. Considering this goal, our study was guided by the following Research Question (RQ): "***Which are the existing SPL regression testing approaches and what are their characteristics?***". This question aims to characterize the regression testing approaches that are specific for SPLs. We identify and discuss applied techniques, strategies, input and output artifacts and testing criteria. From this general question, we derived four sub-RQs, as follows: **RQ1.** What are the addressed SPL regression testing techniques? **RQ2.** What kind of strategies are used to apply SPL regression testing techniques? **RQ3.** What are the input and output artifacts used? **RQ4.** Which are the testing criteria adopted?

Next, we present in detail the methodology adopted to conduct our systematic mapping study in order to achieve our goal and answer the posed RQs.

3.1 Primary Sources Selection

For the selection of primary sources, we followed a methodology based on the systematic mapping method, according to the process proposed by Petersen et al. (Petersen et al., 2015). From the goal and RQs of our study, we derived two main keywords, namely "*regression testing*" and "*software product line*". To define the search string[1], we composed these keywords with their lexical and syntactic alternatives (synonym, plural, gerund, etc.).

The string was used for searching studies in five digital libraries, as presented in Table 1. We did not define an initial publication date for the studies, then all the returned papers were considered. Table 1 presents the data sources used and the period covered. As we can see, a total of 2508 studies were found, including the period from 1985 to 2021. The search on these libraries was completed on Jan. 26th, 2021.

Table 1: Number of studies retrieved by each digital library.

Digital Library	Studies	Covered Period
ACM	300	1985-2021
IEEEXplore	24	2013-2021
ScienceDirect	307	1996-2021
Scopus	1106	1996-2021
Springer Link	771	1993-2021
Total	2508	1985-2021

For screening the studies retrieved from the digital libraries, we followed five steps, which are presented in Figure 1. In the first step we applied a filter to keep only studies on the area of computer science, remaining 1428 studies. For managing this set of studies, we used Parsifal[2]. This tool automatically identified 143 duplicated studies (second step). In the third step we read the title, abstract, and keywords of 1272 studies and removed 1064 of them that were out of the scope of our work. The remaining 208 papers went to a full reading, in which we considered one inclusion criteria (IC) and four exclusion criteria (EC), presented in Table 2. Finally, we composed a set of 27 primary

[1]The final string was: *("regression testing"* ***OR*** *"regression test")* ***AND*** *("product line"* ***OR*** *"SPL"* ***OR*** *"product-family"* ***OR*** *"product family"* ***OR****"highly-configurable"* ***OR*** *"highly configurable"* ***OR*** *"feature model"* ***OR*** *"feature-model"* ***OR*** *"FM"* ***OR*** *"variability analysis")*

[2]A web-based tool for planning, conducting and reporting the systematic reviews: https://parsif.al/

sources (see Table 3). The process was conducted by the two first authors of this paper and validated by the third one. From the 27 primary studies, we extracted pieces of information that are related to five dimensions in order to answer our RQs, shown in Table 3. After this, for each dimension we interactively identified relevant categories used to classify, discuss, analyze the studies, and we provide the complete extraction in a spreadsheet[3].

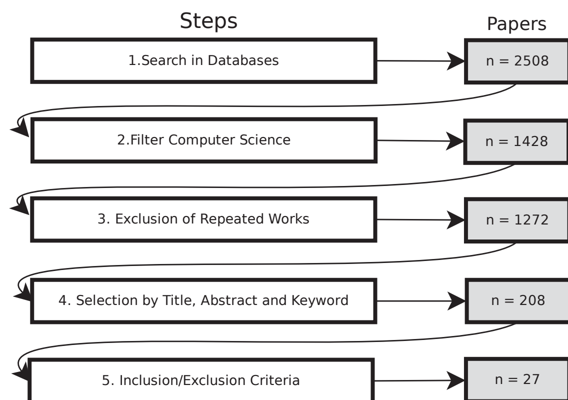


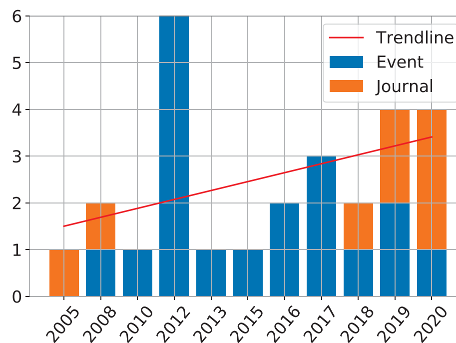
Figure 1: SPL regression testing process

Table 2: Inclusion and exclusion criteria.

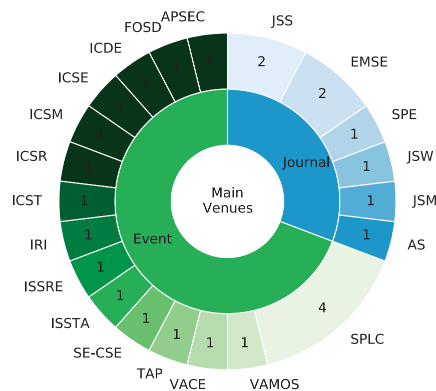
Inclusion Criteria	
IC1	Clear reference to regression testing. The paper refers to one of the regression testing techniques, making clear how is its adoption/use in the regression testing activity.
Exclusion Criteria	
EC1	Out of scope. The paper does not satisfy IC1. It is not clear or presented how the technique is applied in the SPL regression testing.;
EC2	Not available online;
EC3	Not in English;
EC4	Abstracts, posters, reviews, conference reviews, chapters, thesis, keynotes, shorts paper and doctoral symposiums.

4 Results

For an overview of the primary studies demographics, Figure 2 depicts the number of publications over the years and the publication venues. Studies on SPL regression testing have been published since 2005, mainly on conferences, symposiums, and workshops (19 studies, ≈ 70%). Only five studies have been published in journals (≈ 30%). Although the year with most publications was 2012, the trend line shows an increase in the number of publications in the last years (Figure 2(a)). The studies come from 21 different venues, as presented in Figure 2(b). This means that research on SPL regression testing is disseminated in the wide range of venues (events and journals).



(a) Publications per years



(b) Publication venues

Figure 2: Primary sources overview

To answer the RQs of this study, we refer to Table 3 that chronologically presents each primary study according to the dimensions and categories presented in the previous section. Answers to the RQs are presented in the following subsections.

4.1 RQ1. Regression testing techniques

The techniques used in the studies are presented in the 2nd to 5th columns of Table 3. Most of the primary studies apply the *selection* technique (18 out of 27, 67%). This was expected, since the tester would prefer to select test cases and/or products related to changed features. Among the studies on this category,

[3]Primary Studies Dataset: https://docs.google.com/spreadsheets/d/1LdrD1h76pTRfIdo4auwtEloca48sZLJnxTUDWuV76Uc/edit?usp=sharing

Table 3: Details of the SPL regression testing approaches found in the primary sources.

Paper	Technique				Input					Output			Strategy			Testing criterion			
	Prioritization	Minimization	Selection	Retest all	Variability model	Test suite	Source code	State machine	Other	List of test cases	List of products	Other	Expert-based	AI-based	Comparison-based	Coverage-based	Combinatorial	Fault-based	Other
(Al Dallal and Sorenson, 2005)				✓				✓	✓	✓			✓			✓		✓	
(Qu et al., 2008)	✓						✓		✓		✓			✓		✓	✓	✓	
(Al-Dallal and Sorenson, 2008)			✓			✓	✓			✓			✓						✓
(Neto et al., 2010)	✓		✓		✓	✓	✓		✓	✓					✓			✓	
(Lochau et al., 2012)			✓		✓			✓		✓	✓				✓	✓			
(Heider et al., 2012)				✓	✓							✓			✓			✓	
(Remmel et al., 2011)			✓		✓							✓			✓			✓	
(Robinson and White, 2012)			✓				✓			✓		✓			✓			✓	
(Qu et al., 2012)			✓				✓				✓				✓	✓		✓	
(Neto et al., 2012)	✓		✓		✓	✓	✓		✓	✓					✓			✓	
(Remmel et al., 2013)			✓		✓					✓		✓			✓		✓		
(Lachmann et al., 2015)	✓							✓	✓	✓					✓	✓		✓	
(Lity et al., 2016)			✓			✓		✓	✓	✓					✓				✓
(Lachmann et al., 2016)	✓					✓		✓		✓					✓	✓			✓
(Al-Hajjaji et al., 2017)	✓							✓	✓		✓				✓			✓	✓
(Marijan et al., 2017)	✓	✓				✓				✓				✓		✓		✓	
(Lachmann et al., 2017)	✓							✓	✓			✓			✓			✓	✓
(Marijan and Liaaen, 2018)			✓			✓			✓	✓					✓	✓	✓	✓	✓
(Souto and d'Amorim, 2018)			✓		✓	✓			✓	✓	✓			✓			✓	✓	
(Jung et al., 2019)			✓			✓	✓			✓		✓			✓	✓			
(Fischer et al., 2019)			✓				✓			✓					✓	✓			
(Marijan et al., 2019)			✓			✓			✓	✓				✓				✓	✓
(Lity et al., 2019)			✓					✓		✓					✓	✓		✓	✓
(Jung et al., 2020)			✓				✓			✓					✓			✓	
(Fischer et al., 2020)			✓				✓			✓					✓	✓			
(Lima et al., 2020a)	✓								✓	✓				✓				✓	✓
(Hajri et al., 2020)	✓		✓						✓	✓					✓	✓		✓	
Total	10	1	18	2	7	10	10	8	13	21	5	6	2	5	20	13	4	19	8

one deals with the selection of SPL products (Souto and d'Amorim, 2018) and 17 focus on the selection of test cases (Neto et al., 2010; Al-Dallal and Sorenson, 2008; Lochau et al., 2012; Remmel et al., 2011; Robinson and White, 2012; Neto et al., 2012; Remmel et al., 2013; Lity et al., 2016; Marijan and Liaaen, 2018; Souto and d'Amorim, 2018; Jung et al., 2019; Fischer et al., 2019; Marijan et al., 2019; Lity et al., 2019).

Prioritization is the second most common technique. We found 10 studies (37%) applying this regression testing technique, which aims at establishing an order of test cases or products that must be executed firstly. These test cases or products are usually

those with high probability of failing. The goal is detecting faults as early as possible, making SPL regression testing more effective and efficient. Three primary sources prioritize SPL products (Qu et al., 2008; Qu et al., 2012; Al-Hajjaji et al., 2017) and six ones the test cases (Neto et al., 2010; Neto et al., 2012; Lachmann et al., 2015; Lachmann et al., 2016; Marijan et al., 2017; Lachmann et al., 2017; Lima et al., 2020a; Hajri et al., 2020).

Retest all and minimization are the techniques less investigated. Two studies apply the *retest all* technique; both focusing on test cases. The first one focuses on the creation of test cases that can be reused as much as possible in different configurations (Al Dallal and Sorenson, 2005). The second one uses the test suite to identify changes in products derived with the same configuration, but from before and after SPL evolution (Heider et al., 2012). Regarding *minimization*, only one paper uses this technique, together with prioritization, to reduce testing execution of continuous integration cycles (Marijan et al., 2017). This study focuses on minimization of test cases. Prioritization and selection are combined in three studies (Neto et al., 2010; Neto et al., 2012; Hajri et al., 2020).

Answer to RQ1: *Most of the primary studies apply the selection technique (67%), followed by prioritization (37%). Minimization is applied only by one study. Only four studies combine techniques.*

4.2 RQ2. Strategies

Most approaches (20 out of 27, 74%) use a *comparison-based* strategy to apply the regression testing techniques. A possible reason for this is the structure of SPL. This strategy relies on comparison of whole SPLs or products versions before and after modifications (change impact analysis) (Neto et al., 2010; Remmel et al., 2011; Qu et al., 2012; Neto et al., 2012; Remmel et al., 2013; Marijan and Liaaen, 2018; Jung et al., 2020; Hajri et al., 2020), comparison of similarities and differences among test cases (overlap analysis) (Jung et al., 2019; Fischer et al., 2019; Fischer et al., 2020), and delta-oriented analysis of products differences (Lochau et al., 2012; Lachmann et al., 2015; Lity et al., 2016; Lachmann et al., 2016; Al-Hajjaji et al., 2017; Lachmann et al., 2017; Lity et al., 2019).

AI-based strategies use optimization or machine learning algorithms. The algorithms are used for selection of SPL products to conduct time-space efficiently regression testing (Souto and d'Amorim, 2018). This allows prioritization and minimization of test cases due to limited time budget in continuous integration (Marijan et al., 2017; Marijan et al., 2019; Lima et al., 2020a), and for test case prioritization in order to maximize coverage of a set of SPL products (Qu et al., 2008).

Two studies apply an *expert-based* strategy in which experts manually design reusable test cases to be used in a greater number of products (Al Dallal and Sorenson, 2005; Al-Dallal and Sorenson, 2008).

Answer to RQ2: *Most strategies (74%) belong to the comparison-based category. AI-based strategies are poorly explored. This category includes only five studies ($\approx$ 18%). The expert-based strategy is explored only in two works that use manual approaches.*

4.3 RQ3. Input and output artifacts

There is no predominant type of artifact used as input. Test suite was expected to be widely used as input, as well as source code (10 studies found in each category, 37%). Taking into account the focus of the primary sources on SPLs, the variability model is also a common artifact. Interestingly, state machines are used very often (8 studies, 30%). State machines are used to represent products and analyze differences, allowing the application of regression testing techniques. In the category *other*, we also observed as input: UML models (Al Dallal and Sorenson, 2005), case models (Hajri et al., 2020), configuration options (Qu et al., 2008), feature dependencies (Neto et al., 2012; Neto et al., 2010), software architecture (Lachmann et al., 2015; Lity et al., 2016; Al-Hajjaji et al., 2017; Lachmann et al., 2017; Neto et al., 2012; Neto et al., 2010), history of fault-detection (Marijan and Liaaen, 2018; Lima et al., 2020a), and list of changes (Marijan et al., 2019; Souto and d'Amorim, 2018).

Regarding output, a list of prioritized, minimized, or selected test cases is by far the most common artifact (21 studies, 74%). A list of products is generated in five approaches. Both lists are output in only two studies (Lochau et al., 2012; Souto and d'Amorim, 2018). In the category *other*, the output artifacts are: change impact reports (Heider et al., 2012), failure report (Remmel et al., 2011; Robinson and White, 2012; Lachmann et al., 2017; Remmel et al., 2011), and test cases partition table (Jung et al., 2019).

Answer to RQ3: *A large variety of artifacts are used; 13 studies ($\approx$ 48%) belong to the category others, followed by the category source code with 10 studies ($\approx$ 37%) and the category test cases with 10 studies ($\approx$ 37%). On the other hand, two main categories of outputs were identified, namely lists of test cases (the prevalent) and lists of products.*

4.4 RQ4. Testing Criteria

Fault-based is the most common criterion adopted in nine studies (70%). This criterion uses practical knowledge to select, minimize or prioritize test cases or SPL products. In other words, products or test cases more likely to fail, or that have a history of fails, deserve more attention.

Coverage-based criteria, which is also widely used for single product systems, is the focus of 13 studies (48%). Examples of elements to be covered in this category are code elements (Qu et al., 2008), all-transitions in state machines (Lity et al., 2019), and architecture changes (Lachmann et al., 2017).

Due to the nature of SPLs, i.e., based on combination of features, criteria derived from the *combinatorial* testing is also observed in primary studies, such as pair-wise. This criterion aims to cover variant interaction to select a subset of all possible variant combinations (Remmel et al., 2013; Qu et al., 2008; Marijan and Liaaen, 2018). In the category *other* we observe criteria based on test execution time (Al-Dallal and Sorenson, 2008; Marijan et al., 2019), risk (Lachmann et al., 2017), change impact (Lity et al., 2016; Lity et al., 2019), signals (Lachmann et al., 2016; Al-Hajjaji et al., 2017), dissimilarity (Lachmann et al., 2016), and historical information (Marijan and Liaaen, 2018).

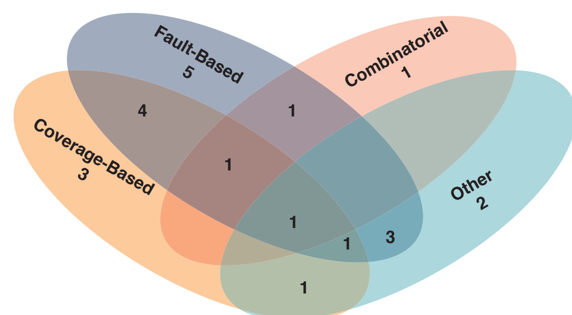


Figure 3: Combination of testing criteria.

Figure 3 presents the number of primary sources per criteria and their intersection. As traditional in software testing, a combination of criteria can be used for evaluating test cases (Melo et al., 2019). For example, fault-based and coverage-based are commonly applied with each other. In three primary sources, fault-based is also applied together with at least two additional criteria (Lity et al., 2019; Qu et al., 2008; Marijan and Liaaen, 2018). One study uses the four types of criteria (Marijan and Liaaen, 2018), two studies use three types (Qu et al., 2008; Lity et al., 2019), and nine a pair of criteria (Al Dallal and Sorenson, 2005; Qu et al., 2012; Lachmann et al., 2015; Lachmann et al., 2016; Al-Hajjaji et al., 2017; Marijan et al., 2017; Lachmann et al., 2017; Souto and d'Amorim, 2018; Marijan et al., 2019). This shows a trend on combining different criteria in the proposed approaches.

Answer to RQ4: *Fault-based and coverage-based criteria are the most applied in, respectively, 70% and 48% of the studies. Many studies (58%) combine more than one criterion.*

5 A Roadmap for Future Research on SPL Regression Testing

As a result of the reading, analysis, and discussion of the primary studies, we defined a roadmap to serve as a guide for researchers and practitioners to contribute to the body of knowledge, both in theory and practice, on SPL regression testing. This roadmap consists of six research opportunities and future directions related to gaps identified in the literature, trends concerned to emerging technologies, and limitations of existing approaches to their application in practice. Each opportunity of the roadmap is described next.

1. To explore intelligent and learning approaches: As with many other activities of software engineering, software testing is influenced by several factors, criteria, and constraints. This requires proper approaches to satisfactorily aid SPL testing, such as multi-criteria optimization strategies from the field of Search-Based Software Engineering (SBSE) (Colanzi et al., 2020). SBSE uses artificial intelligence to solve software engineering problems. In RQ2 we observed the opportunity of exploring artificial intelligence and machine learning techniques for SPL regression testing. A promising opportunity in this direction is the use of multi/many-objective search techniques, as for example, assessing the performance and scalability (Wang et al., 2014). These algorithms can also be used to deal with the trade-off between test case prioritization compared to prioritization of products (Al-Hajjaji et al., 2017).

2. To explore hybrid approaches: Combining different techniques and strategies to take advantage of the strength of each one can lead to better results of SPL regression testing approaches. As we discussed in the answer of RQ1, minimization techniques are poorly investigated, which could be combined with selection or prioritization techniques. Also, adding semantic impact analysis techniques in combination with syntactic techniques can enable immediate feedback on the change impact (Robinson and White, 2012). Also, the combination of product selection,

configuration augmentation, or reduction techniques may further improve the testing effectiveness and efficiency, compared to either approach alone (Qu et al., 2012). Finally, considering both fine and coarse granularity of artifacts might allow more comprehensive decisions during test activity (Lity et al., 2016).

3. To derive test cases automatically based on SPL modifications: Ideally, test cases might be derived in the same way products variants are (Fischer et al., 2019; Fischer et al., 2020). However, quality of initial product tests is particularly crucial for the subsequent iterations (Lochau et al., 2012). Another challenge to achieve automatic derivation of test cases is that building and maintaining traceability links between features and test cases can be complex and time-consuming. For SPLs that evolve at a moderate rate, building a feature model and traceability links requires high upfront costs (Marijan and Sen, 2017).

4. To properly test feature interactions: Two studies highlight the need of investigating feature interactions in depth. One study suggests considering feature interaction failures across historical executions (Marijan and Liaaen, 2018). Another study mentions the use of results from testing different configuration combinations to discover interactions among configuration options (Fischer et al., 2019).

5. To support SPL regression testing in continuous integration: Continuous Integration has become a *de facto* practice in software development, even in the context of SPLs (Lima et al., 2020b). This also affects regression testing, as there usually exist time budgets (Marijan and Liaaen, 2018). For example, one study suggests splitting test activity in two phases, namely one for the nightly runs and another more complex one for release-acceptance testing (Remmel et al., 2011). And finally, two primary sources mention investigating trade-off between early fault detection and efficient regression testing (Lity et al., 2019) and experimenting variable time budgets with the test prioritization and reduction approaches (Marijan et al., 2019).

6. To manage regression testing for SPL evolution in space and time: It has been acknowledged that SPLs evolve in space (introduction or exclusion of features) and in time (version of a same feature) (Berger et al., 2019; Michelon et al., 2020b; Michelon et al., 2020a). However, in the literature of SPL regression testing, we have not found approaches with such characteristics. For example, SPLs evolving in space and time require lifting of traditional analyses to consider both dimensions of variation (Berger et al., 2019). However, we argue that there is a potential for reusing existing regression analyses for testing purposes, some of which has been already exploited (Heider et al., 2012; Lochau et al., 2012).

6 Concluding Remarks

This work overviews existing literature on SPL regression testing, a fundamental activity to reveal problems and guarantee that SPL maintenance and evolution were performed accordingly. Test case selection and prioritization are the most addressed techniques. A wide range of input artifacts are used, and the main output is the list of selected/prioritized/ minimized test cases. Comparison-based strategy is by far the most applied strategy, together with fault-based and coverage-based criteria.

Based on the results, we defined a research roadmap for short and medium term. This roadmap describes some research opportunities. Among them, we can mention to introduce multi-criteria approaches, to explore the advantages of hybrid approaches, automatic derivation of test cases based on SPL modifications, proper testing of feature interactions, to support the SPL regression testing in continuous integration environments, and management of regression testing for SPLs evolving in space and time. In addition to this, we observe some opportunities based on limitations and trends related to the investigation of SPL regression testing in the context of emerging technologies and practical needs, such as to properly test feature interactions.

REFERENCES

Abbas, M., Jongeling, R., Lindskog, C., Enoiu, E. P., Saadatmand, M., and Sundmark, D. (2020). Product line adoption in industry: An experience report from the railway domain. In *24th ACM Conference on Systems and Software Product Line-Volume A*, SPLC '20, New York, NY, USA. ACM.

Al Dallal, J. and Sorenson, P. (2005). Reusing class-based test cases for testing object-oriented framework interface classes. *Journal of Software Maintenance and Evolution: Research and Practice*, 17(3):169–196.

Al-Dallal, J. and Sorenson, P. G. (2008). Testing software assets of framework-based product families during application engineering stage. *JSW*, 3(5):11–25.

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

Assunção, W. K. G., Krüger, J., and Mendonça, W. D. F. (2020). Variability management meets microservices: Six challenges of re-engineering microservice-based webshops. In *24th ACM Conference on Systems and Software Product Line-Volume A*. ACM.

Berger, T., Chechik, M., Kehrer, T., and Wimmer, M. (2019). Software Evolution in Time and Space: Unifying Version and Variability Management (Seminar 19191). *Dagstuhl Reports*, 9(5):1–30.

bin Ali, N., Engström, E., Taromirad, M., Mousavi, M. R., Minhas, N. M., Helgesson, D., Kunze, S., and Varshosaz, M. (2019). On the search for industry-relevant regression testing research. *Empirical Software Engineering*, 24(4):2020–2055.

Colanzi, T. E., Assunção, W. K., Vergilio, S. R., Farah, P. R., and Guizzo, G. (2020). The symposium on search-based software engineering: Past, present and future. *Information and Software Technology*, 127:106372.

do Carmo Machado, I., McGregor, J. D., Cavalcanti, Y. C., and de Almeida, E. S. (2014). On strategies for testing software product lines: A systematic literature review. *Information and Software Technology*, 56(10):1183–1199.

Engström, E. (2010a). Exploring regression testing and software product line testing: Research and state of practice. Licentiate Thesis, Department of Computer Science, Lund University.

Engström, E. (2010b). Regression test selection and product line system testing. In *3rd International Conference on Software Testing, Verification and Validation*, pages 512–515. IEEE.

Engström, E. and Runeson, P. (2011). Software product line testing–a systematic mapping study. *Information and Software Technology*, 53(1):2–13.

Fischer, S., Michelon, G. K., Ramler, R., Linsbauer, L., and Egyed, A. (2020). Automated test reuse for highly configurable software. *Empirical Software Engineering*, 25(6):5295–5332.

Fischer, S., Ramler, R., Linsbauer, L., and Egyed, A. (2019). Automating test reuse for highly configurable software. In *Proceedings of the 23rd International Systems and Software Product Line Conference-Volume A*, pages 1–11.

Geraldi, R. T., Reinehr, S., and Malucelli, A. (2020). Software product line applied to the internet of things: A systematic literature review. *Information and Software Technology*, 124:106293.

Grüner, S., Burger, A., Kantonen, T., and Rückert, J. (2020). Incremental migration to software product line engineering. In *24th ACM Conference on Systems and Software Product Line-Volume A*. ACM.

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

Heider, W., Rabiser, R., Grünbacher, P., and Lettner, D. (2012). Using regression testing to analyze the impact of changes to variability models on products. In *Proceedings of the 16th International Software Product Line Conference-Volume 1*, pages 196–205.

Jung, P., Kang, S., and Lee, J. (2019). Automated code-based test selection for software product line regression testing. *Journal of Systems and Software*, 158:110419.

Jung, P., Kang, S., and Lee, J. (2020). Efficient regression testing of software product lines by reducing redundant test executions. *Applied Sciences*, 10(23):8686.

Lachmann, R., Beddig, S., Lity, S., Schulze, S., and Schaefer, I. (2017). Risk-based integration testing of software product lines. In *Proceedings of the Eleventh International Workshop on Variability Modelling of Software-intensive Systems*, pages 52–59.

Lachmann, R., Lity, S., Al-Hajjaji, M., Fürchtegott, F., and Schaefer, I. (2016). Fine-grained test case prioritization for integration testing of delta-oriented software product lines. In *Proceedings of the 7th International Workshop on Feature-Oriented Software Development*, pages 1–10.

Lachmann, R., Lity, S., Lischke, S., Beddig, S., Schulze, S., and Schaefer, I. (2015). Delta-oriented test case prioritization for integration testing of software product lines. In *Proceedings of the 19th International Conference on Software Product Line*, pages 81–90.

Lee, J., Kang, S., and Lee, D. (2012). A survey on software product line testing. In *16th International Software Product Line Conference-Volume 1*, pages 31–40.

Lima, J. A. P., Mendonça, W. D., Vergilio, S. R., and Assunção, W. K. (2020a). Learning-based prioritization of test cases in continuous integration of highly-configurable software. In *Proceedings of the 24th ACM Conference on Systems and Software Product Line: Volume A-Volume A*, pages 1–11.

Lima, J. A. P., Mendonça, W. D. F., Vergilio, S. R., and Assunção, W. K. G. (2020b). Learning-based prioritization of test cases in continuous integration of highly-configurable software. In *24th ACM Conference on Systems and Software Product Line-Volume A*. ACM.

Linden, F. J. v. d., Schmid, K., and Rommes, E. (2007). *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*. Springer.

Lity, S., Morbach, T., Thüm, T., and Schaefer, I. (2016). Applying incremental model slicing to product-line regression testing. In *International Conference on Software Reuse*, pages 3–19. Springer.

Lity, S., Nieke, M., Thüm, T., and Schaefer, I. (2019). Retest test selection for product-line regression testing of variants and versions of variants. *Journal of Systems and Software*, 147:46–63.

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

Marijan, D., Gotlieb, A., and Liaaen, M. (2019). A learning algorithm for optimizing continuous integration development and testing practice. *Software: Practice and Experience*, 49(2):192–213.

Marijan, D. and Liaaen, M. (2018). Practical selective regression testing with effective redundancy in interleaved tests. In *Proceedings of the 40th International*

Conference on Software Engineering: Software Engineering in Practice, pages 153–162.

Marijan, D., Liaaen, M., Gotlieb, A., Sen, S., and Ieva, C. (2017). Titan: Test suite optimization for highly configurable software. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 524–531. IEEE.

Marijan, D. and Sen, S. (2017). Detecting and reducing redundancy in software testing for highly configurable systems. In *2017 IEEE 18th International Symposium on High Assurance Systems Engineering (HASE)*, pages 96–99. IEEE.

Marques, M., Simmonds, J., Rossel, P. O., and Bastarrica, M. C. (2019). Software product line evolution: A systematic literature review. *Information and Software Technology*, 105:190 – 208.

Melo, S. M., Carver, J. C., Souza, P. S., and Souza, S. R. (2019). Empirical research on concurrent software testing: A systematic mapping study. *Information and Software Technology*, 105:226–251.

Mendonça, W. D. F., Assunção, W. K. G., Estanislau, L. V., Vergilio, S. R., and Garcia, A. (2020). Towards a microservices-based product line with multi-objective evolutionary algorithms. In *IEEE Congress on Evolutionary Computation*. IEEE.

Michelon, G. K., Obermann, D., Assunção, W. K. G., Linsbauer, L., Grünbacher, P., and Egyed, A. (2020a). Mining feature revisions in highly-configurable software systems. In *24th ACM International Systems and Software Product Line Conference - Volume B*, page 74–78. ACM.

Michelon, G. K., Obermann, D., Linsbauer, L., Assunção, W. K. G., Grünbacher, P., and Egyed, A. (2020b). Locating feature revisions in software systems evolving in space and time. In *24th ACM Conference on Systems and Software Product Line-Volume A*. ACM.

Minhas, N. M., Petersen, K., Ali, N. B., and Wnuk, K. (2017). Regression testing goals-view of practitioners and researchers. In *2017 24th Asia-Pacific Software Engineering Conference Workshops (APSECW)*, pages 25–31. IEEE.

Neto, P. A. d. M. S., do Carmo Machado, I., Cavalcanti, Y. C., De Almeida, E. S., Garcia, V. C., and de Lemos Meira, S. R. (2010). A regression testing approach for software product lines architectures. In *2010 Fourth Brazilian Symposium on Software Components, Architectures and Reuse*, pages 41–50. IEEE.

Neto, P. A. d. M. S., do Carmo Machado, I., Cavalcanti, Y. C., de Almeida, E. S., Garcia, V. C., and de Lemos Meira, S. R. (2012). An experimental study to evaluate a spl architecture regression testing approach. In *2012 IEEE 13th International Conference on Information Reuse & Integration (IRI)*, pages 608–615. IEEE.

Nomme, S. S. (2020). Composing software product lines with machine learning components. Master's thesis, University of Oslo.

Petersen, K., Vakkalanka, S., and Kuzniarz, L. (2015). Guidelines for conducting systematic mapping studies in software engineering: An update. *Information and Software Technology*, 64:1–18.

Qu, X., Acharya, M., and Robinson, B. (2012). Configuration selection using code change impact analysis for regression testing. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pages 129–138. IEEE.

Qu, X., Cohen, M. B., and Rothermel, G. (2008). Configuration-aware regression testing: an empirical study of sampling and prioritization. In *Proceedings of the 2008 international symposium on Software testing and analysis*, pages 75–86.

Remmel, H., Paech, B., Bastian, P., and Engwer, C. (2011). System testing a scientific framework using a regression-test environment. *Computing in Science & Engineering*, 14(2):38–45.

Remmel, H., Paech, B., Engwer, C., and Bastian, P. (2013). Design and rationale of a quality assurance process for a scientific framework. In *2013 5th International Workshop on Software Engineering for Computational Science and Engineering (SE-CSE)*, pages 58–67. IEEE.

Robinson, B. and White, L. (2012). On the testing of user-configurable software systems using firewalls. *Software Testing, Verification and Reliability*, 22(1):3–31.

Runeson, P. and Engström, E. (2012). Regression testing in software product line engineering. In *Advances in computers*, volume 86, pages 223–263. Elsevier.

Souto, S. and d'Amorim, M. (2018). Time-space efficient regression testing for configurable systems. *Journal of Systems and Software*, 137:733–746.

Wang, S., Buchmann, D., Ali, S., Gotlieb, A., Pradhan, D., and Liaaen, M. (2014). Multi-objective test prioritization in software product line testing: an industrial case study. In *18th International Software Product Line Conference-Volume 1*, pages 32–41.

Yoo, S. and Harman, M. (2012). Regression testing minimization, selection and prioritization: a survey. *Software testing, verification and reliability*, 22(2):67–120.

3 TEST2FEATURE: FEATURE-BASED TEST TRACEABILITY TOOL FOR HIGHLY CONFIGURABLE SOFTWARE

Based on the mapping presented in the previous chapter, we identified the need for a tool for test case traceability to features. As HCSs evolve, new features are added, changed, or removed, which hampers the selection and evolution of test cases. The use of test traceability reports can help in this task, but there is a lack of studies addressing HCS test-to-feature traceability. Existing work usually are based on the variability model, which is not always available or updated. Some tools only link test cases to code lines. Considering this gap, this chapter introduces our tool called `Test2Feature`, a tool that traces test cases to features using the source code of annotated HCSs, written in C/C++.

3.1 MOTIVATIONS

Our motivation is the observed lack of tools explicitly designed for tracing tests in HCS based on their features. The particularities of these systems, with their diverse configurations, require an approach that explicitly links test cases to the specific features of the software. This gap in the available tools motivated our research to address this need.

3.2 OBJECTIVES

The primary objective of our work is to introduce and showcase the `Test2Feature`, tool a feature-based test traceability solution tailored for HCS. The tool aims to establish direct links between test cases and the unique features of the software, facilitating a nuanced understanding of test coverage across various configurations. We delve into the implementation details and highlight the practical advantages of `Test2Feature` in real-world scenarios.

3.3 CONCLUDING REMARKS

In conclusion, we underscore the significant contribution of `Test2Feature` as a practical tool to enhance test traceability in HCS environments. The tool provides an innovative approach to tying test cases directly to specific features, easing the adaptation of tests to changes in software configurations. The outcomes indicate that `Test2Feature` fills a crucial void, offering an effective solution to improve traceability and reliability in testing HCS. We also noticed that this tool is quite powerful and can be integrated into parts of other tools or approaches.

Test2Feature: Feature-based Test Traceability Tool for Highly Configurable Software

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

Gabriela K. Michelon
Alexander Egyed
ISSE, Johannes Kepler University Linz
Linz, Austria

Wesley K. G. Assunção
ISSE, Johannes Kepler University Linz
Linz, Austria
Opus, Pontifical Catholic University of Rio de Janeiro
Rio de Janeiro, Brazil

ABSTRACT

To ensure the quality of Highly Configurable Software (HCS) in an evolution and maintenance scenario is a challenging task. As HCSs evolve, new features are added, changed, or removed, which hampers the selection and evolution of test cases. The use of test traceability reports can help in this task, but there is a lack of studies addressing HCS test-to-feature traceability. Existing work usually are based on the variability model, which is not always available or updated. Some tools only link test cases to code lines. Considering this gap, this paper introduces `Test2Feature`, a tool that traces test cases to features using the source code of annotated HCSs, written in C/C++. The tool produces the following outputs: the code lines that correspond to each feature, the lines that correspond to each test case, and the test cases that are linked to each feature. `Test2Feature` is based only on the static analysis of the code. The traceability report produced can be used to ease different tasks related, for instance, to regression testing, feature management, and HCS evolution and maintenance.

CCS CONCEPTS

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

KEYWORDS

Highly Configurable Software, Regression Test, Feature-based, Test Traceability

ACM Reference Format:
Willian D. F. Mendonça, Silvia R. Vergilio, Gabriela K. Michelon, Alexander Egyed, and Wesley K. G. Assunção. 2022. Test2Feature: Feature-based Test Traceability Tool for Highly Configurable Software. In *26th ACM International Systems and Software Product Line Conference - Volume B (SPLC '22), September 12–16, 2022, Graz, Austria.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3503229.3547031

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SPLC '22, September 12–16, 2022, Graz, Austria
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9206-8/22/09…$15.00
https://doi.org/10.1145/3503229.3547031

1 INTRODUCTION

Highly Configurable Software (HCS) provides adaptable and flexible solutions to complex and real-world problems. HCSs are usually implemented by using different configuration options to create custom system products, also known as variants [20]. Quality assurance of HCS is a fundamental issue, but this involves some challenges. Ideally, all possible configurations should be tested, but this is often unfeasible because the number of possible configurations grows exponentially with the number of features. This makes HCS testing more complex and expensive, as many features often need to be tested and several test cases overlap [16]. For instance, the test case selection in a regression testing scenario requires specific approaches to consider the HCS evolution in space and time, where features are added, modified, or removed [15].

For easing that and other tasks, approaches and tools based on test traceability could be used. Tufail et al. [19] developed a tool for test traceability, which links test cases to a set of requirements without requiring the program execution or test coverage, which would be expensive; but these tools do not consider HCSs. A recent mapping on testing tools for HCSs [5] reports few tools addressing test case traceability for features [1, 7, 11]. These tools are based on the *Feature Model (FM)*. This can be a limitation because, for many HCSs, the FM is unavailable or outdated. Existing tools for HCS that are based on source code [8, 9, 12] do not generate traceability for features, but only link test cases to the lines of code. Another limitation is that the great majority of these tools can be used only for Java code.

Motivated by these facts, this paper introduces `Test2Feature`, a tool that, given the source code of an annotated HCS and a test case set, produces test traceability reports linking test cases to the HCS features. The tool encompasses three modules that generate three main outputs: traceability from features to code lines, traceability from test cases to code lines, and traceability from features to tests. In this sense, the main contribution of this work is to describe a tool that traces test cases to features from the source code of HCSs. The tool is available online[1] and allows developers and testers to know the lines of the source code and features that correspond to a test case. Differently from existing tools, `Test2Feature` works with annotated HCSs written in C/C++ and is based only on the static analysis of the code. The traceability report produced can ease different tasks, as for example, regression testing, feature management, and HCS evolution and maintenance.

The paper is structured as follows. Section 2 overviews related work. Section 3 describes the tool architecture. Section 4 illustrates

[1] https://github.com/willianferrari/Test2Feature.git

the outputs of `Test2Feature` through a case study. Section 5 concludes the paper.

2 RELATED WORK

As mentioned before, there are many approaches and tools in the literature to generate test traceability to requirements using static analysis [19]. Different kinds of artefacts are used to specify these requirements, such as use cases and user stories. But they usually do not trace test cases for HCS code. We have found a few work that link test cases to the HCS code lines [8, 9, 12], but they do not generate traceability to features. Another limitation is that they generate traceability only for Java code.

We observed in the literature a lack of testing tools for HCSs written in C/C++. Out of 34 tools reported by a recent mapping on testing tools for HCS/SPL [5] only four can be applied to C/C++ [2, 4, 6, 18], and no one of these tools addresses test case traceability. The mapping also reports some pieces of work addressing test-to-feature traceability in the context of test case selection and prioritization [1, 7, 11]. But the problem of these work is that the approaches they implement are based on an FM.

Differently from the related work mentioned above, our work generates the test-to-feature traceability from the annotated code of HCS. This brings some advantages for an HCS evolution scenario because FMs are not always available and updated. We observe a lack of tools for C/C++ language that link features to the source code in the context of HCS. In addition to this, we have not found a tool to link features to test cases, which is the main goal of Feature2Test, described in the next section.

3 `TEST2FEATURE` MODULES

`Test2Feature`[2] encompasses three modules as depicted in Figure 1. The input provided by the software engineer is a configuration file containing the paths for the system repository and test folder, as well as the current commit to be analyzed. This information is passed to the module *MineFeaturesLines*, responsible for localizing features by code lines, and to the module *MineTestLines*, responsible for identifying the code lines that correspond to each test case of the test folder. *MineTestLines* uses the tool `Doxygen`[3] to generate XML files representing the dependency graph for the functions of the C/C++ code. The information of these both modules is used by the module *MergeTestFeaturesLines* to generate a CSV file that contains the traceability of features to the test cases.

Configuration File: in the configuration file the software engineer sets the names of the test and system folders. In addition, s(he) can choose any previous commit by using Commit Sha. This is illustrated in Figure 2. The engineer needs to set four variables: 1) `REPGIT`: which contains the system URL in GitHub; 2) `SYSFOLDER` with the name of the system folder; 3) `TESTFOLDER` with the name of the test folder of the system; and 4) `NCOMMIT` with the Sha Hash.

MineFeaturesLines: this module uses the implementation of the approach for mining features in space and time, described in our previous work [13, 14]. The approach abstracts the source code of

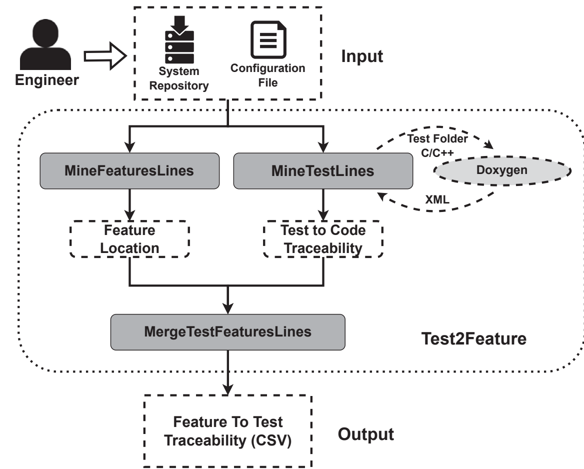


Figure 1: `Test2Feature` architecture

```
$ CofigFile.sh M ×
Users > willianmendonca > Documents > Tool-SPLC > Test2Feature > $ CofigFile.sh
1  #!/bin/bash
2
3  #repository URL
4  REPGIT="<URL REPOSITORY GitHub>"
5  #Name of System Folder
6  SYSFOLDER="<NAME FOLDER SYSTEM>"
7  #Name of Test Folder
8  TESTFOLDER="NAME FOLDER TEST"
9  #Commit Sha
10 NCOMMIT="<COMMIT SHA HASH>"
```

Figure 2: Configuration File Content

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. This abstraction is less computationally expensive and time-consuming, as we do not need to analyze the abstract syntax tree to obtain, for each file, all the lines of code containing preprocessor directives. The approach uses a Constraint Satisfaction Problem (CSP) [17] to reliably identify interacting features or features depending on the execution of other features [3], and thus which features belong to which conditional blocks to obtain the features lines.

Figure 3 illustrates how the approach for mining features lines works. The figure contains four variation points/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

[2]A video demonstrating the core features of the TEST2FEATURE tool is available at: https://www.youtube.com/watch?v=6RN6F1TNYr8
[3]https://doxygen.nl/index.html

```
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>
```

Figure 3: Conditional blocks of feature implementations.

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 [10] 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 belong to feature `A`; lines 5-9 to feature `D`; lines 11-13 belong to `BASE`, as well as line 15, which is outside of any variation point.

MineTestLines: this module generates as result the traceability of test cases to system functions by using the output of `Doxygen`. This tool has the GNU General Public License, initially developed with a view to keeping the source code of systems with annotated C++ code documented, but also has support for other languages like C, C#, Python, Java, and so on. Doxygen supports visualization of the relationships between various elements through dependency graphs, inheritance, and collaboration diagrams, generated automatically, in different formats. *MineTestLines* uses the function dependency graph in XML format. We defined the minimal set of parameters in order to generate all possible dependencies available in Doxygen and to make the tool execution faster.

MergeTestFeatureslines: this module produces a merge between the *MineTestLines* and *MineFeaturesLines* outputs, e.g., the location of the features along with the location of the test cases. 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.

Implementation Aspects: For the development of `Test2Feature` we used the Python programming language, in version 3.9.10. To deal with the CSV files the PANDAS data science library is used. The module *MineFeaturesLines* was developed in Java and uses as input a system folder already cloned and a Commit Sha. As output, it delivers a CSV file containing the location of the features per line.

4 ILLUSTRATIVE EXAMPLE

This section describes an example of our tool and the results generated by their modules. For this end, we use the `SQLite`[4] database system, available on GitHub. `SQLite` is an industrial case study, used on a large scale, and daily updated. After setting the variables in the configuration file, the modules *MineFeaturesLines* and *MineTestLines* can be run independently. Figure 4 shows an excerpt from the source code of the main.c file that belongs to `SQLite`, and Figure 5 presents the outputs produced by the modules *MineFeaturesLines* and *MineTestLines*. The `SQLITE_OMIT_WSD` feature of the code snippet in Figure 4 is responsible to execute the conditional block of lines 376-381.

```
375   int sqlite3_shutdown(void){
376   #ifdef SQLITE_OMIT_WSD
377     int rc = sqlite3_wsd_init(4096, 24);
378     if( rc!=SQLITE_OK ){
379       return rc;
380     }
381   #endif
```

Figure 4: Features `SQLite`

MineFeaturesLines and *MineTestLines* are modules for feature location (traceability of features to code) and traceability of the test cases to the code, respectively. Figure 5 shows an example of test-to-function traceability generated by the module *MineTestLines*. Test case traceability has the granularity of test cases to function considering the start line (column LineFrom) and end line (column LineTo); the test case `registerOomSimilator` (column TestCase) belonging to the file `fuzzcheck.c` (column TestFile) uses the function `sqllite3_shutdown` (column TargetFunction), located in the file main.c (column TargetFile) starting in line 375 and ending in line 418. The module *MineFeaturesLines* also generates a result that can be used by engineers for decision-making. Such a result can also be seen in Figure 5. We can observe the traceability between the output files, in this case, the feature `SQLITE_OMIT_WSD` (FeatureName) is located in the file `main.c` (TargetFile) between lines 376 and 381.

TestFile	TestCase	TargetFile	TargetFunction	LineFrom	LineTo
fuzzcheck.c	registerOomSimulator	main.c	sqlite3_shutdown	375	418
fuzzcheck.c	disableOom	fuzzcheck.c	oomCounter	627	
fuzzcheck.c	disableOom	fuzzcheck.c	oomRepeat	628	
fuzzcheck.c	isOffset	optfuzz-db01.txt	c	22	
fuzzcheck.c	isOffset	fuzzcheck.c	hexToInt	683	690
fuzzcheck.c	decodeDatabase	optfuzz-db01.txt	a	22	

TargetFile	FeatureName	FeatFrom	FeatTo
src/main.c	SQLITE_OMIT_WSD	376	381
src/sqliteInt.h	SQLITE_OMIT_WSD	1129	1135
src/ctime.c	SQLITE_OMIT_WSD	680	682
src/test_multiplex.c	SQLITE_OMIT_WSD	529	531
src/tclsqlite.c	SQLITE_TCL_DEFAULT_FULLMUTEX	3751	3753
src/test_config.c	SQLITE_OMIT_TCL_VARIABLE	646	648
src/ctime.c	SQLITE_OMIT_TCL_VARIABLE	645	647

Figure 5: Outputs from *MineTestLines* and *MineFeaturesLines*.

[4] https://github.com/sqlite/sqlite

Figures 6 and 7 show the outputs of the module *MergeTest-Featureslines*, which are, respectively, traceability from features to test cases and traceability from test cases to features. Figure 6 shows a file presenting traceability considering that the range corresponding to the feature belongs to the range that corresponds to the function on which the test case depends on. For example `registerOomSimilator` (TestCase) depends on function `sqllite3_shutdown` that corresponds to the lines 375 (LineFrom) to 418 (LineTo), and `SQLITE_OMIT_WSD` (FeatureName) corresponds to the lines 376 (FeatFrom) to 381 (FeatTo). We observe that lines 376 to 381 are within the range 375 to 418. Then, `SQLITE_OMIT_WSD` is within the range of the function on which the test case depends.

Figure 7 presents traceability considering that the function on which the test depends is inside the range that corresponds to the feature. For example, `vacuum1` (TestCase), which is located in the file `tt3_vacuum.c` (TestFile), depends on the function `sqlite3_enable_shared_cache` (TargetFunction), located in the file `btreee.c` (TargetFile), between lines 89 and 92. In this case, the feature `BASE` (FeatureName) is between lines 78 and 93. Thus, the test case is within the range that corresponds to the feature.

TestFile	TestCase	TargetFile	TargetFunction	LineFrom	LineTo	FeatureName	FeatFrom	FeatTo
fuzzcheck.c	registerOomSimulator	main.c	sqlite3_shutdown	375	418	SQLITE_OMIT_WSD	376	381
fuzzcheck.c	runCombinedDbSqlInput	main.c	sqlite3_file_control	3895	3940	BASE	3936	3938
fuzzcheck.c	runCombinedDbSqlInput	main.c	sqlite3_file_control	3895	3940	SQLITE_UNTESTABLE	3936	3938
fuzzcheck.c	runCombinedDbSqlInput	main.c	sqlite3_initialize	200	365	BASE	205	212
fuzzcheck.c	runCombinedDbSqlInput	main.c	sqlite3_initialize	200	365	BASE	212	295
fuzzcheck.c	runCombinedDbSqlInput	main.c	sqlite3_initialize	200	365	BASE	295	309
fuzzcheck.c	runCombinedDbSqlInput	main.c	sqlite3_initialize	200	365	BASE	309	317

Figure 6: Traceability of Features to Tests.

TestFile	TestCase	TargetFile	TargetFunction	LineFrom	LineTo	FeatureName	FeatFrom	FeatTo
tt3_vacuum.c	vacuum1	btree.c	sqlite3_enable_shared_cache	89	92	BASE	78	93
tt3_vacuum.c	vacuum1	btree.c	sqlite3_enable_shared_cache	89	92	BASE	81	93
tt3_shared.c	shared1	btree.c	sqlite3_enable_shared_cache	89	92	BASE	78	93
tt3_shared.c	shared1	btree.c	sqlite3_enable_shared_cache	89	92	BASE	81	93
tt3_index.c	create_drop_index_1	btree.c	sqlite3_enable_shared_cache	89	92	BASE	78	93
tt3_index.c	create_drop_index_1	btree.c	sqlite3_enable_shared_cache	89	92	BASE	81	93
threadtest3.c	dynamic_triggers	btree.c	sqlite3_enable_shared_cache	89	92	BASE	78	93

Figure 7: Traceability of Tests to Features.

5 CONCLUSIONS

This paper introduces `Test2Feature`, a tool to generate test-to-feature traceability from the code of an annotated HCS to the test cases of its features. The tool has three modules that produce the following outputs: the code lines that correspond to each feature, the lines that correspond to each test case, and the test cases that are linked to each feature. In this way, `Test2Feature` allows developers and testers to know which lines of the source code correspond to a test case, and which features correspond to the lines of a test case.

Furthermore, the tool was developed to work at the script level. This makes it possible to insert the tool as part of larger processes for automatic decision-making. For instance, the results can be used by other approaches/tools for a detailed analysis of the test cases' behavior, and/or mapping the co-change of features and test cases in space in time, hence easing maintenance and evolution tasks.

As future work, we intend to develop a friendly interface and also organize the results in different ways. We intend to implement other mechanisms and modules to ease regression testing activities such as test case prioritization in continuous integration of HCSs, selection of test cases linked to some particular features, and generation of test cases based on the code modifications.

ACKNOWLEDGMENTS

This work is supported by CAPES (grant: 88887.464736/2019-00), CNPq (grant: 305968/2018-1), the LIT Secure and Correct Systems Lab, and FAPERJ, under the PDR-10 program (grant 202073/2020).

REFERENCES

[1] M. Al-Hajjaji, J. Meinicke, S. Krieter, R. Schröter, T. Thüm, T. Leich, and G. Saake. 2016. Tool demo: testing configurable systems with featureIDE. In *GPCE*. 173–177. https://doi.org/10.1145/2993236.2993254

[2] S. Apel, H. Speidel, P. Wendler, A. Von Rhein, and D. Beyer. 2011. Detection of feature interactions using feature-aware verification. In *ASE*. IEEE, 372–375. https://doi.org/10.1109/ASE.2011.6100075

[3] D. Benavides, S. Segura, P. Trinidad, and A. Ruiz-Cortés. 2006. *Using Java CSP Solvers in the Automated Analyses of Feature Models*. Springer, 399–408. https://doi.org/10.1007/11877028_16

[4] G. Cavarlé, A. Plantec, S. Costiou, and V. Ribaud. 2018. A feature-oriented model-driven engineering approach for the early validation of feature-based applications. *Science of Computer Programming* 161 (2018), 18–33.

[5] F. Ferreira, J. P. Diniz, C. Silva, and E. Figueiredo. 2019. Testing tools for configurable software systems: A review-based empirical study. In *VAMOS*. 1–10.

[6] C. Henard, M. Papadakis, and Y. L. Traon. 2014. Mutation-based generation of software product line test configurations. In *SSBSE*. Springer, 92–106.

[7] Martin Fagereng Johansen, Øystein Haugen, and Franck Fleurey. 2011. Properties of realistic feature models make combinatorial testing of product lines feasible. In *Model Driven Engineering Languages and Systems*. Springer, 638–652.

[8] C. H. P. Kim, D. S. Batory, and S. Khurshid. 2011. Reducing combinatorics in testing product lines. In *AOSD*. 57–68. https://doi.org/10.1145/1960275.1960284

[9] Chang Hwan Peter Kim, Sarfraz Khurshid, and Don Batory. 2012. Shared Execution for Efficiently Testing Product Lines. In *23rd SSRE*. 221–230. https://doi.org/10.1109/ISSRE.2012.23

[10] K. Ludwig, J. Krüger, and T. Leich. 2019. Covert and phantom features in annotations: do they impact variability analysis?. In *SPLC*. ACM, 31:1–31:13. https://doi.org/10.1145/3336294.3336296

[11] D. Marijan, A. Gotlieb, S. Sen, and A. Hervieu. 2013. Practical pairwise testing for software product lines. In *17th SPLC*. 227–235. https://doi.org/10.1145/2491627.2491646

[12] J. Meinicke, C. Wong, C. Kästner, T. Thüm, and G. Saake. 2016. On essential configuration complexity: Measuring interactions in highly-configurable systems. In *ASE*. 483–494. https://doi.org/10.1145/2970276.2970322

[13] G. K. Michelon, W. K. G. Assunção, D. Obermann, L. Linsbauer, P. Grünbacher, and A. Egyed. 2021. The life cycle of features in highly-configurable software systems evolving in space and time. In *GPCE*. ACM, 2–15. https://doi.org/10.1145/3486609.3487195

[14] G. K. Michelon, D Obermann, W. K. G. Assunção, L. Linsbauer, P. Grünbacher, and A. Egyed. 2020. Mining Feature Revisions in Highly-Configurable Software Systems. In *SPLC*. ACM, 74–78. https://doi.org/10.1145/3382026.3425776

[15] G. K. Michelon, D. Obermann, W. K. G. Assunção, L. Linsbauer, P. Grünbacher, S. Fischer, R. E. Lopez-Herrejon, and A. Egyed. 2022. Evolving software system families in space and time with feature revisions. *Empirical Software Engineering* 27, 5 (May 2022). https://doi.org/10.1007/s10664-021-10108-z

[16] M. Mukelabai, D. Nešiundefined, S. Maro, T. Berger, and J. Steghöfer. 2018. Tackling Combinatorial Explosion: A Study of Industrial Needs and Practices for Analyzing Highly Configurable Systems. In *ASE*. ACM, 155–166. https://doi.org/10.1145/3238147.3238201

[17] T. Schiex and S. de Givry (Eds.). 2019. *Principles and Practice of Constraint Programming*. LNCS '19, Vol. 11802. Springer. https://doi.org/10.1007/978-3-030-30048-7

[18] S. Souto, M. d'Amorim, and R. Gheyi. 2017. Balancing soundness and efficiency for practical testing of configurable systems. In *ICSE*. IEEE, 632–642. https://doi.org/10.1109/ICSE.2017.64

[19] 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 ICSRS*. 450–454. https://doi.org/10.1109/ICSRS.2017.8272863

[20] A. von Rhein, A. Grebhahn, S. Apel, N. Siegmund, D. Beyer, and T. Berger. 2015. Presence-condition Simplification in Highly Configurable Systems. In *ICSE*. IEEE, 178–188. https://doi.org/10.1109/ICSE.2015.39

4 FEATURE-ORIENTED TEST CASE SELECTION DURING EVOLUTION OF HIGHLY-CONFIGURABLE SYSTEMS

In this chapter, we explore the tool presented in the previous chapter as part of a test case selection approach, to face challenges found in the evolution of HCSs. Systems with a broad range of configurations pose unique challenges, especially concerning the efficient selection of test cases. Our research focuses on the critical need to adapt the testing strategy as these systems evolve.

4.1 MOTIVATIONS

Upon analyzing existing approaches applicable to the context of HCSs evolution, we observe a lack of specific approaches to deal with test cases. Most works have limitations, as they focus solely on the altered parts of the code without considering the impact of the change on the features – the building blocks of HCSs. The motivation for this study is to address this gap and emphasize the importance of adaptive test case selection to handle changes in system configurations and features.

4.2 OBJECTIVES

Our proposal aims to provide a practical and automated solution for test case selection, employing a feature-oriented approach during the system's evolution, `FeaTestSel` (**Fea**ture-oriented **Test** Case **Sel**ection for Highly Configurable Systems). We solely use the source code of the HCS as input for our approach, and all code analysis is static. The intention is to ensure comprehensive coverage, optimizing the detection of potential defects in a dynamic environment.

4.3 CONCLUDING REMARKS

By adding the execution time of the approach to the execution time of the selected test cases, we obtained a reduction of approximately $\approx 50\%$, compared to 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 future work by researchers, test quality analysis by engineers, or as a source of information for tool builders.

The results highlight the tangible effectiveness of our `FeaTestSel` approach in the context of HCS evolution. The automation of test case selection, considering changes in features, has proven to be a robust strategy for dealing with the complexities of continuous evolution in a continuous integration environment. We conclude by emphasizing the critical importance of adopting adaptive testing practices to ensure the ongoing quality of these systems throughout their evolutionary cycle. The data presented during our research supports the validity and practical usefulness of this innovative approach.

`FeaTestSel` is capable to reduce the number of test cases, by selecting those ones related to the features changed in the commit, and as a consequence to reduce costs. However, in a company, multiple projects may share the same CI workflow and regression testing usually runs a time restricted to a specific duration, the test budget. In this sense, the use of TCP techniques in combination with TCS can improve the RT cost-effectiveness. The idea is, after test selection, to rank the test cases in order to early execute those with a high probability of reveal faults.

Considering the constraints of test budgets, early fault detection is essential because when a test case fails, test execution can be ended, and fewer resources are spent.

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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SPLC '23, August 28-September 1, 2023, Tokyo, Japan
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0091-0/23/08…$15.00
https://doi.org/10.1145/3579027.3608979

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 *27th ACM International Systems and Software Product Line Conference - Volume A (SPLC '23), August 28-September 1, 2023, Tokyo, Japan.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3579027.3608979

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

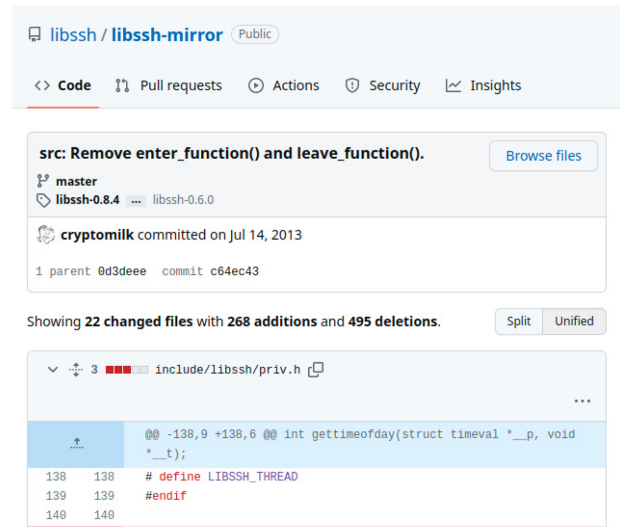


Figure 1: Number of modifications made in commit c64ec43

```
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
```

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

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

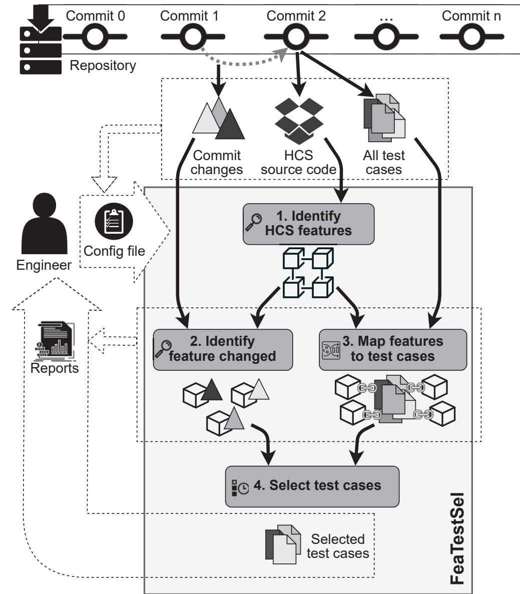


Figure 3: Input, four steps, and output for FeaTestSel

ConfigFileExample.py 7 ×
Users > ConfigFileExample.py > ...

```
 8  repository_URL = 'https://gitlab.com/libssh/libssh-mirror.git'
 9  system_path = '../database/libssh-mirror'
10  test_names = 'assert_'
11  # test_folder = '../database/libssh-mirror/tests'
12
```

Figure 4: Configuration file used as input for the approach

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.

```
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>
```

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

```
36   36      #ifdef WITH_SSH1
37   37
38   38      static int ssh_auth_status_termination(void *s){
39   39          ssh_session session=s;
40   40          if(session->auth_state != SSH_AUTH_STATE_NONE ||
41   41                  session->session_state == SSH_SESSION_STATE_ERROR)
42   42              return 1;
43   43          return 0;
44   44      }
45   45
46   46      static int wait_auth1_status(ssh_session session) {
47        -    enter_function();
48   47        /* wait for a packet */
49   48        if (ssh_handle_packets_termination(session,SSH_TIMEOUT_USER,
50   49                ssh_auth_status_termination, session) != SSH_OK){
51        -      leave_function();
     50   +
52   51          return SSH_AUTH_ERROR;
53   52        }
54   53        SSH_LOG(SSH_LOG_PROTOCOL,"Auth state : %d",session->auth_state);
55        -    leave_function();
     54   +
...
230       -    leave_function();
     224  +
231  225       return rc;
232  226     }
233  227
234  228     #endif /* WITH_SSH1 */
235  229     /* vim: set ts=2 sw=2 et cindent: */
...
269  265  #ifdef WITH_SERVER
270  266  int crypt_set_algorithms_server(ssh_session session){
271  267      char *method = NULL;
272  268      int i = 0;
273       -   int rc = SSH_ERROR;
274  269      struct ssh_cipher_struct *ssh_ciphertab=ssh_get_ciphertab();
275  270
276  271      if (session == NULL) {
277  272          return SSH_ERROR;
278  273      }
279  274
280       -   /* we must scan the kex entries to find crypto algorithms and set their
281       -   enter_function();
...
338       -     rc = SSH_OK;
339       -     error:
340       -     leave_function();
341       -     return rc;
     335  +
     336  +     return SSH_OK;
342  337    }
343  338
344  339    #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 */
                                                    options.c
```

```
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      }
                                                    connection.c
```

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

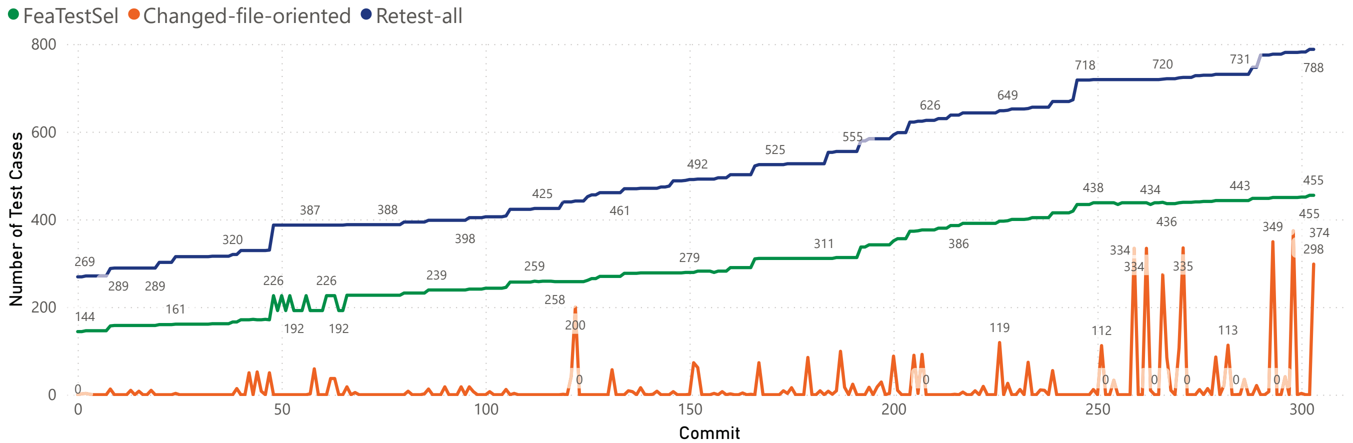


Figure 8: Number of test cases selected by the approaches for the set of commits with logs

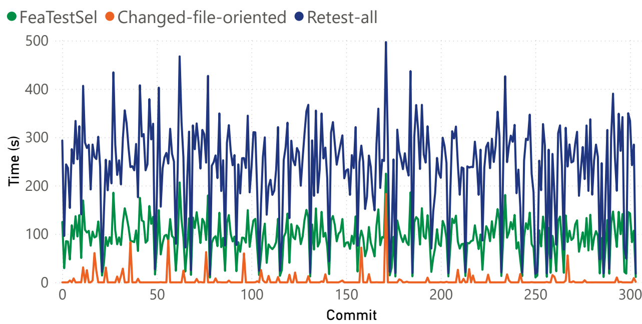


Figure 9: Average time to execute the selection of test cases per commit and approach for the set of commits with logs

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.

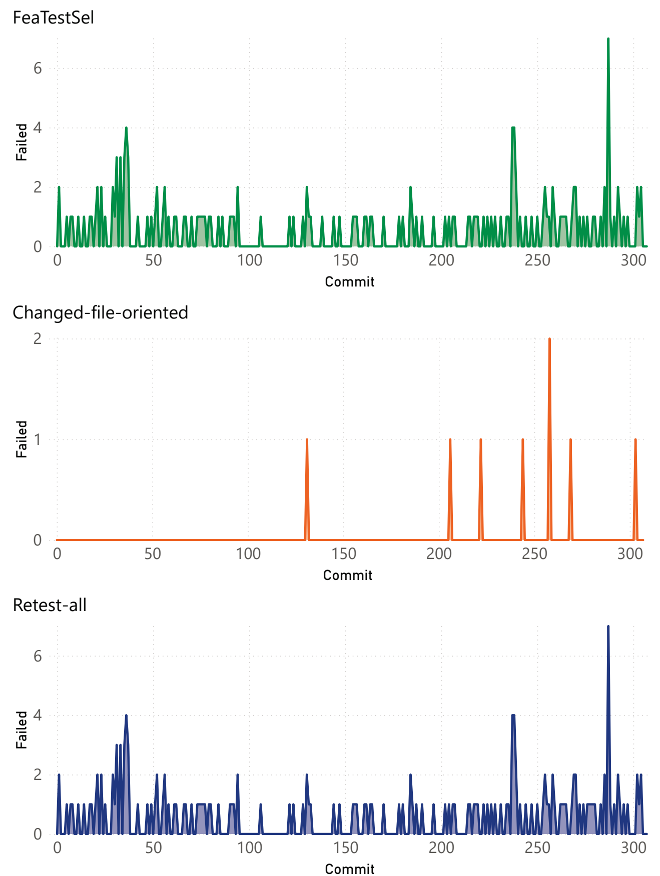


Figure 10: Number of failed files per commit and approach for the set of commits with logs

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.

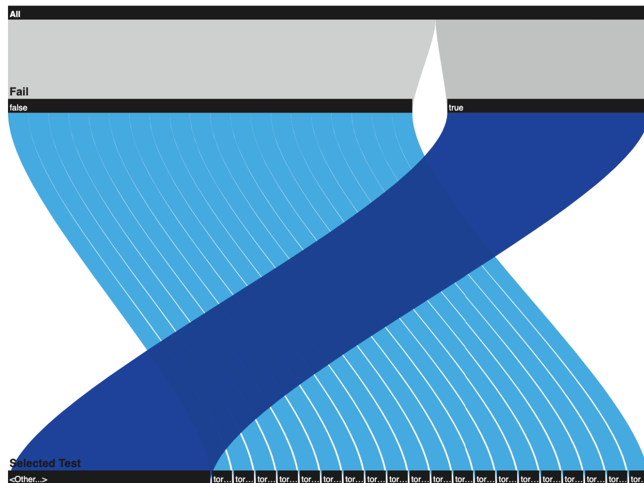


Figure 11: Failed and selected test cases for commit d7477dc

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 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.

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), FAPERJ PDR-10 Fellowship (grant: 202073/2020) and CAPES (grant: 88887.464736/2019-00). We would like to thank Gabriela K. Michelon and Stefan Fischer for the help in the implementation.

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.

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

[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 C. Machado, John D McGregor, Yguaratã Cerqueira Cavalcanti, and Eduardo S. 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] C. H. 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] C. H. 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 *Intl. Conference on Enterprise Information Systems (ICEIS).* SciTePress, 81–89.

[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 K. 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 K. Michelon, David Obermann, Wesley K. 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 C. 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.

5 FEATURE-ORIENTED TEST CASE SELECTION AND PRIORITIZATION FOR HIGHLY-CONFIGURABLE SYSTEMS

In the previous chapter, we introduced the `FeaTestSel` approach. In this chapter, we extend this approach, creating a combined Regression Test approach, first applying test case selection and then using the selected set as input for the prioritization step.

5.1 MOTIVATIONS

Testing in an agile evolutionary environment or in a CI environment is extremely expensive and it becomes unfeasible to use the retest-all technique for each evolution of the system. Test activity is already a challenge for single systems, but when we think in the evolution context of HCSs, this challenge is even greater. They incorporate a substantial number of configuration parameters, causing high complexity of HCS tests. Typically, most configuration parameters are interrelated through inclusive or exclusive relationships, which further increases the effort of the HCS test. Also, if the software is constantly released following a continuous delivery model, there are strict time restrictions imposed on test runs. To efficiently address these challenges with an optimal trade-off between time, cost and test quality, professionals need efficient test approaches.The combined techniques of RT can be highly effective in this context. However, the presented approaches that employ this strategy are highly specific and require various adaptations to be applied in the evolution process of HCS or CI. Our work aims to fill this gap by proposing a TCS and TCP approach that is applicable to the evolution process of HCS or to HCS in a CI environment.

5.2 OBJECTIVES

The research extends a previous approach, `FeaTestSel`, which serves as a feature-oriented test case selection strategy for HCSs. This approach aims to reduce the number of test cases while maintaining test quality in terms of fault detection. The new proposal, namely `FeaTCSP` (**Feat**ture-oriented **T**est **C**ase **S**election and **P**rioritization for Highly Configurable Systems), introduces an additional prioritization step. Three feature-oriented prioritization strategies are proposed: (i) `FeatChgHist`: the history of feature changes; (ii) `FeatChgCommit`: features that changed in the current commit; and (iii) `FeatCov`: the number of features covered by the test case.

5.3 CONCLUDING REMARKS

We conducted a comprehensive evaluation of these strategies using a real open-source HCS. It is noteworthy that `FeaTCSP` outperforms a file-based TCSP approach, achieving considerably lower average budgets compared to the baseline. Feature-oriented strategies, `FeatChgHist` and `FeatChgCommit`, exhibit comparable average budgets, around 13.5%, while `FeatCov` shows a slightly higher average budget, reaching 23.4%. Furthermore, in comparison to the exclusive use of `FeatTCP`, `FeatTCSP` utilizes a reduced number of test cases to identify faults in 57% of commits with failures.

We conclude that `FeatChgHist` and `FeatChgCommit` are effective and comparable in terms of average budget for fault identification. Additionally, considering the average execution

time of our `FeatTCP` approach, which is 15.18 seconds, with a maximum time of 31.38 seconds, highlights the feasibility of the approach. Emphasizing the applicability of `FeatTCP` in the evolution process of HCSs and CI environments.

Feature-oriented Test Case Selection and Prioritization for Highly-Configurable System

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

Testing *Highly Configurable Systems (HCSs)* is a challenging task, especially in an evolution scenario with Continuous Integration (CI) practices where frequent updates, integration, and testing cycles occur. In a previous work, we introduced `FeaTestSel`, a feature-oriented test case selection approach for HCSs. It allows a reduction in the number of test cases, maintaining test quality in terms of revealed faults. However, in a company, multiple projects may share the same CI workflow and regression testing usually runs a time restricted to a specific duration, the test budget. In this sense, the use of Test Case Prioritization (TCP) techniques in combination with Test Case Selection (TCS) can improve the regression test cost-effectiveness and produce a rapid feedback on the executed tests. Considering this as motivation, in this paper, we extend the approach `FeaTestSel` by proposing, implementing and evaluating and additional prioritization step. This extension is name `FeaTCSP` (**Feat**ture-oriented **T**est **C**ase **S**election and **P**rioritization for Highly Configurable Systems). To perform the TCP step we propose three feature-oriented prioritization strategies: (i) `FeatChgHist`: uses the history of feature changes; (ii) `FeatChgCommit`: uses the features that changed in the current commit; and (iii) `FeatCov`: the number of features covered by the test case. Our evaluation on a real open-source HCS shows that `FeaTCSP` outperforms a file-based TCSP approach, achieving significantly lower average budgets compared to the baseline. The Feature-oriented strategies, `FeatChgHist` and `FeatChgCommit`, exhibit comparable average budgets of around 13.5%, while `FeatCov` shows a slightly higher average budget of 23.4%. Moreover, compared to exclusively use of `FeatTCP`, `FeatTCSP` in 57% of failed commits(122), executes a reduced number of test cases required, to find the failure.

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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
–, –
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN –…$00.00
https://doi.org/--

ACM Reference Format:
Willian D. F. Mendonça, Wesley K. G. Assunção, and Silvia R. Vergilio. 2023. Feature-oriented Test Case Selection and Prioritization for Highly-Configurable System. In ,. ACM, New York, NY, USA, 12 pages. https://doi.org/--

1 INTRODUCTION

Software Product Line (SPL) is an approach for developing and managing family of software products that can be customized with different configurations (variabilities), while common software assets can be reused in a systematic and disciplined way [22]. SPLs are generally implemented as *Highly Configurable Systems (HCSs)*, using different configuration options and applying strategies such as conditional compilation, conditional execution or, build systems- to create custom system products, also known as variants [36, 46]. HCSs are complex and predominantly configurable System, which means that they incorporate a series of options (a.k.a., features) used for software customization. This allows different functionalities and configurations to be selected to a given user context [46], what makes software testing a hard activity [3, 29]. Software Testing of HCSs are challenging, even more when we consider an evolution scenario, i.e., where *Continuous Integration (CI)* practices are adopted. HCSs can be updated, integrated, and tested several times a day, and each cycle needs to be fast [50].

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 [4]. The results are used to solve problems and find faults. Then, to provide quick feedback is essential to reduce development costs [10]. In this sense, the application of *Regression Testing (RT)* techniques in a very cost-effective way is fundamental [7]. The best known and used RT techniques are [49]: (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 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 are usually based on *Feature Model (FM)* or other artifacts [1, 17, 18, 23, 43]. However, none of them consider that HCSs are usually developed by adopting CI practices, in a scenario where the models are rarely updated, making the use of those approaches difficult. Other HCS testing approaches need some kind of dynamic analysis based on the test failure-history, execution, or coverage [25–27]. In some pieces of work, the approaches are only evaluated with systems well-modularized [11], in a context where the test cases are separated by feature and do not overlap. This is different from real scenarios. Approaches based on code changes are more suitable and used. For instance, approaches that

select test cases related to the files changed in the current commit [2, 8, 40]. But those existing approaches and tools do not consider HCS particularities, and they are mostly based on Java language only [11–13]. 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.

Motivated by these facts, in a previous work, we introduced `FeaTestSel` [33], 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 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 `FeaTestSel` works for systems in C/C++ language. Our approach is feature-oriented and needs only static analysis of the source code. The evaluation of the approach showed that 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.

`FeaTestSel` is capable to reduce the number of test cases, by selecting those ones related to the features changed in the commit, and as a consequence to reduce costs. However, in a company, multiple projects may share the same CI workflow and regression testing usually runs a time restricted to a specific duration, the test budget. In this sense, the use of TCP techniques in combination with TCS can improve the RT cost-effectiveness. The idea is, after test selection, to rank the test cases in order to early execute those with a high probability of reveal faults. Considering the constraints of test budgets, early fault detection is essential because when a test case fails, test execution can be ended, and fewer resources are spent [21]. To efficiently address these challenges with an optimal trade-off between time, cost and test quality, professionals need efficient test approaches [27].

Considering these points above, in this paper we extend the approach `FeaTestSel` by proposing, implementing and evaluating and additional prioritization step. This extension is namely `FeaTCSP` (**Feat**ture-oriented **T**est **C**ase **S**election and **P**rioritization for Highly Configurable Systems). To perform the TCP step we propose three prioritization strategies based on the mapping of features to test cases generated by `FeaTestSel`. Each strategy uses three information sources: (i) `FeatChgHist`: uses the history of feature changes; (ii) `FeatChgCommit`: uses the features that changed in the current commit; and (iii) `FeatCov`: the number of features covered by the test case.

To validate the `FeatTCSP` approach, we rely on (`Libssh`),[1], a real open-source HCS in constant evolution. The feature-oriented strategies, `FeatChgHist` and `FeatChgCommit`, show comparable average budgets, around 13.5%, while the FeatCov strategy presents a slightly higher average budget, reaching 23.4%. When compared with a file-based TCSP approach, `FeatTCSP` achieves significantly lower average budgets compared to those recorded by the baseline, which had an average budget of 96.8%. Additionally, `FeatTCSP` has shown that in 57% of commits with faults, this combined approach results in a reduced number of required test cases compared to the exclusive use of FeatTCP. These results highlight the adaptability and effectiveness of our innovative approach in addressing the challenges posed by the continuous evolution of HCS.

[1] https://www.libssh.org/

In summary, our contributions are as follows:

- We propose `FeatTCSP` a feature-oriented test case selection and prioritization approach.
- We implemente and evaluate three feature-oriented prioritization strategies: `FeatChgHist`, `FeatChgCommit`, and `FeatCov`.
- We evaluate the performance of `FeatTCSP` compared to a file-based TCSP approach, employing file-based selection and two file-based prioritization strategies.
- We have made available a repository [34] containing all datasets, code, and results from both approaches and baselines.

The paper is structured as follows. Section 2 contains related work. Section 3 reviews our previous work. Section 4 describes `FeaTCSP` and the proposed prioritization strategies, as well as its implementation aspects. Section 5 describes the methodology adopted in the evaluation. Section 6 presents and analyses the obtained results. Section 7 concludes the paper and points research directions.

2 RELATED WORK

As starting point in the search for related work, we reviewed different mapping studies on RT and SPL engineering [16, 31, 41]. Some pieces of work are devoted to the selection of the best product configurations to be tested, having as focus the Feature Model (FM) [16, 38]. Other model-based works consider the delta concept [23], and and others are based on code analysis [11]. For instance, the work of Lity et al. [23] 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. [18] show an incremental delta-oriented approach for improving SPL integration testing efficiency by prioritizing test cases for product variants. Al-Hajjaji et al. [1] selected the most dissimilar product to the previously tested ones, in terms of deltas. Lachmann et al. [17] 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. Some works select a set of products that should be tested considering different goals such as: combinatorial testing, product similarity, coverage of variability and important features [5].

Some TCS approaches select a subset of existing test cases to be reused for testing a new product [24, 47, 48]. Hajri et al. [9] present an automated test case classification and prioritization approach that supports use case-driven testing in product lines. Jung et al. [13] 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. Silveira Neto et al. [43] describe a RT framework 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, such as test scripts and integration level test suites. In addition, the intervention of testing experts is necessary. Most of these approaches consider differences between two products, and not the whole product family. They rely on models and other artifacts that are not always available or updated. The approach works only for Java and considers only toy systems. Some pieces of work introduce methods based on changed files or versions [2, 8, 40]. Bertolino et al. [2] 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 [14, 15, 30], do not generate traceability for features, but only link test cases to lines of code. The work of Tuglular and Şensülün [45] generates a traceability between the feature and test cases, but using a specific language through annotations. The code-based method of Jung et al. [11] 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, what hampers the use of the approach in practice. Jung et al. [12] 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. And none of their work considers traceability for features but for source code.

The problem of these works 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 a regression testing scenario. For instance, the great majority does not consider particularities of the CI environment. TITAN [27] 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 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. [25, 26] 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.

Prado Lima et al. [20, 39] introduce two strategies for applying a TCP learning-based approach called `COLEMAN` in the CI of HCS: 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 these works because we first select a test set based on the evolution of features, what contributes to reduce costs and bypass the variant volatility problem. Regarding differences with the works mentioned above, we aim for an approach that has an automated TCS based on features, the main key concept in HCS. Moreover, the approach should include a prioritization step to deal properly with the test budgets.

3 PREVIOUS WORK

In our previous work, we introduced `FeaTestSel` [33]. 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 functionalities changed in the corresponding evolution cycle. As additional output, the approach produces different traceability reports linking (i) test cases to system lines of code, (ii) features to system lines of code, and (iii) test cases to features.

`FeaTestSel` consists of four steps that are presented in Figure 1. The tester needs to only provide a configuration file (*Config file*). This file contains: (i) `repository_URL`: URL of the HCS repository (e.g., the GitHub URL); (ii) `system_path`: path to the source code folder; and (iii) `test_names`: naming pattern for test cases that allows the approach to identify which source code files are related to the test cases. Alternatively, the software engineer can provide the name of the folder, `test_folder`, used to store test cases, if that is a common practice in the project.

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. Utilizing Constraint Satisfaction Problem Solvers [42], features and their interactions are identified within conditional blocks. A heuristic approach resolves cases where multiple features imply executing a block of code.

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). This step leverages outputs from Step 1 to recognize changes in features. By collecting conditional block macros and `#defines` from release commits, macros exclusively defined externally (via the command line) are considered features. Using Git diff[2], differences in code fragments between commits are identified, signaling feature changes.

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 this step the tool Test2Feature [32] is used.

[2]https://git-scm.com/docs/git-diff

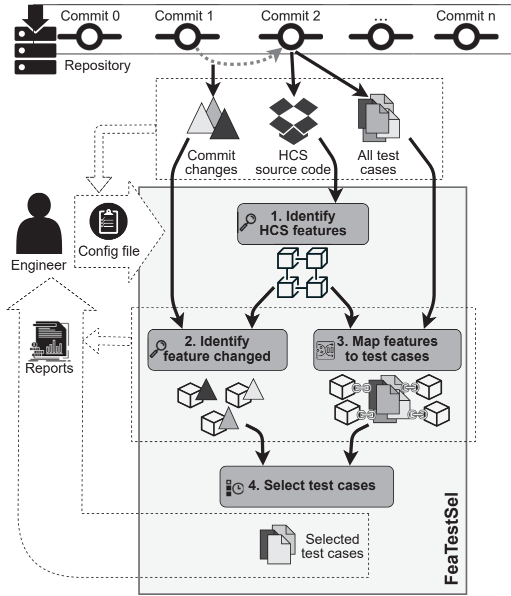


Figure 1: `FeaTestSel` **Overview [35].**

A dependency graph is created for all code in the repository, and dependencies between test cases and feature-implementing code are established. The identified features from the first step, are then linked to relevant test cases, creating a comprehensive mapping. This involves merging location files and applying filters based on code line locations, resulting in a CSV file that precisely outlines test and feature locations.

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.

4 TEST CASE PRIORITIZATION

Our approach, called `FeatTCSP` (**Feat**ure-oriented **T**est **C**ase **S**election and **P**rioritization for Highly Configurable Systems) is an extension of `FeaTestSel` described in the last section. This extension includes an additional step, *Apply prioritization strategies*, that performs the test case prioritization considering the reports generated by `FeaTestSel`, as illustrated in Figure 2. Observe that this step can be applied considering as input the test cases selected in Step 4 of `FeaTestSel`, combining selection and prioritization techniques. But, if desired, only the prioritization is performed. In such case, all test cases are used as input. In addition to the test cases, the general input for all strategies includes the traceability of features to test cases generated in Step 3 of `FeaTestSel`. Each step uses three information sources: (i) `FeatChgHist`: the history of feature changes; (ii) `FeatChgCommit`: features that changed in the current commit; and (iii) `FeatCov`: the number of features covered by the test case. When the prioritization step is applied, the configuration file needs to specify the prioritization strategy (parameter `strategy`), line 15. When the prioritization step is applied, the configuration file needs to specify the prioritization strategy (`strategy` parameter), line 15. For the `FeatChgHist` strategy we must assign the value 1, for

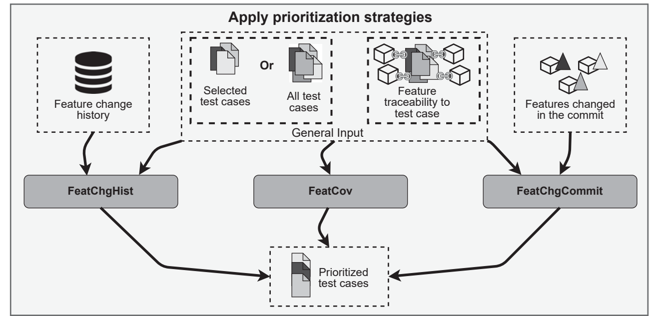


Figure 2: Apply prioritization strategies

`FeatChgCommit` the value 2 and for `FeatCov` the value 3, as illustrated in Figure 3 for the system `Libssh` choosing the `FeatChgHist` strategy. The three prioritization strategies are:

```
ConfigFileExample.py 2 ×
Users > ConfigFileExample.py > ...
 9  repository_URL = 'https://gitlab.com/libssh/libssh-mirror.git'
10  test_name = 'assert_'
11  system_path = '../database/libssh-mirror'
12  # test_folder = '../database/libssh-mirror/tests'
13
14  # 1=FeatChgHist 2=FeatChgCommit 3=FeatCov
15  strategy = 1
```

Figure 3: Configuration file used as input

Feature Change History Strategy (`FeatChgHist`**)**: the change history of each feature is built based on the changes occurred commit by commit. Additionally, we collect the date of the last modification to the feature to aid in the prioritization. As a result, a file is created including the feature name, the number n_c of commits in which the feature changed, and the date of the last modification. For each test case, we built the list of features for those the test cases have traceability. Then we collect the greatest value of n_c in the list, and use this number for prioritization. In case of a tie, the date of the last modification is considered. If the tie persists, a random prioritization is used.

Features Changed in the Commit (`FeatChgCommit`**)**: the prioritization is performed based on the feature changes in the current commit, using the result of the Step 2 *Identify feature changed* of `FeaTestSel`. Initially, we generate a list with the names of the features that have been changed in this commit. Subsequently, we use this list for test case prioritization. Thus, test cases with traceability to the modified features in this commit have priority and are placed at the top of the list, while the remainder of the list is randomly ordered.

Feature Coverage (`FeatCov`**)**: this strategy is based on the number of features covered by the test case. To implement this strategy, we create a list for each test case, indicating the features covered by that test case. We use Step 3 *Map Features to Test Cases* of `FeaTestSel`

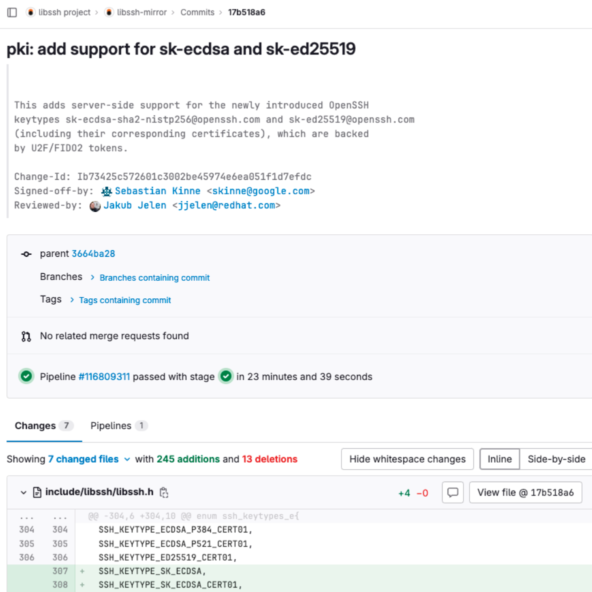


Figure 4: Summary of commit 17b518a, system Libssh.

to generate this list. Subsequently, we 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. We consider a test case covers a feature if it exercises the lines of files touched by that feature. The great the number of features a test case covers in the system, the higher its priority. Test cases covering more features are be ranked first. In case of a tie in the number of covered features, the tiebreaker is random.

4.1 Illustrative Example

To illustrate the outputs produced by FeatTCSP, we use the commit 17b518a[3] of the Libssh system. Figure 4 illustrates that in this commit seven different files changes, 245 lines of code are added and 13 lines are removed. This commit represents the HCS evolution, with changes occurring in four different features: HAVE_LIBGCRYPT, HAVE_OPENSSL_ECC, HAVE_GCRYPT_ECC, HAVE_ECC, and BASE. The BASE feature encompasses the implementation of all parts of the HCS that do not correspond to a specific feature (i.e., not involved in pre-processor directives). BASE accounts for a substantial portion of the code and changes in almost all commits.

Figure 5 depicts a snippet of the code corresponding to the feature HAVE_OPENSSL_ECC, where a conditional IF statement is added, between lines 1349 and 1353. Given the extensive changes in this commit, modifications have occurred in various other sections as well. By applying Step 3 of FeaTestSel, we obtained a CSV file containing the traceability between test cases and features. A portion of this file is presented in Table 1. In this case, we observe that there are three test cases in the file torture_pki_ed25519.c that have traceability to the feature HAVE_OPENSSL_ECC. Additionally, we note that this is the same file changed as shown in Figure 5,

[3]https://gitlab.com/libssh/libssh-mirror/-/commit/17b518a

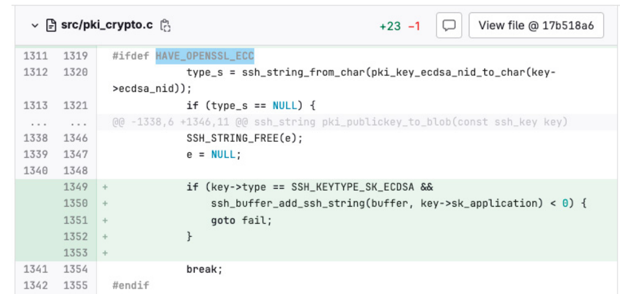


Figure 5: Excerpt of source code corresponding to the Libssh features changed as part of commit 17b518a

pki_crypto.c. In our previous work [33], we provided a detailed explanation of all these steps.

By using the traceability between features and test cases as well as features changed in the commits, the FeatChgCommit strategy utilizes, for prioritization, the features that changed in the commit, in addition to the BASE feature. For this specific commit, five features are considered. As an example of this strategy for the same commit, we can observe Table 2, which provides a record of the results. We notice that the BASE feature appears at the beginning, followed by the HAVE_GCRYPT_ECC and HAVE_LIBGCRYPT features. The next ones only appear at positions 146 and 250, HAVE_OPENSSL_ECC, and HAVE_ECC; this is because the tiebreaker is random, as they all have the same weight due to changes in this commit. Starting from line 597, random test cases begin, which do not have traceability to any feature or to features that did not undergo changes in this commit, as shown in the table.

Initially, for the application of the FeatChgHist strategy (Feature Changes History), it is crucial to have the change history, as shown in Table 3. This record is generated incrementally with each commit, depending on the altered features. We can observe in the table that, for this specific commit, the features BASE, HAVE_LIBGCRYPT, HAVE_ECC, HAVE_OPENSSL_ECC, and HAVE_GCRYPT_ECC were incremented. The result for this strategy already presents distinct characteristics, as evidenced in Table 4, where all the first lines, from 1 to 469, belong to the BASE feature. From line 470, the second feature with more changes begins to appear, and only from line 523 do the features that changed specifically in this commit appear.

To apply the FeatCov strategy, we count the number of features covered by the test cases. Using Step 3 of FeatTestSel, 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 5 provides a partial record of the results of this strategy. In this case, it can be observed that, in the first four rows, the test cases cover exactly nine features. In the five and six rows the test cases cover eight features. The next test cases are not presented in the table, but they cover a lower number of features. The last cases in the list do not cover any feature.

5 EVALUATION SETUP

This section describes details of the study we conducted to evaluate FeatTCSP applicability. The main goal is to investigate the

Table 1: Example of Test Case Traceability to Feature

TestFile	TestCase	TargetFile	TargetFunction	LineFrom	LineTo	FeatureName	FeatFrom	FeatTo
torture_pki_ed25519.c	torture_pki_ed25519_sign	pki_crypto.c	pki_signature_to_blob	1556	1585	HAVE_OPENSSL_ECC	1574	1577
torture_pki_ed25519.c	torture_pki_ed25519_sign	pki_crypto.c	pki_signature_to_blob	1556	1585	BASE	1574	1577
torture_pki_ed25519.c	torture_pki_ed25519_sign_openssh_privkey_passphrase	pki_crypto.c	pki_signature_to_blob	1556	1585	HAVE_OPENSSL_ECC	1574	1577
torture_pki_ed25519.c	torture_pki_ed25519_sign_openssh_privkey_passphrase	pki_crypto.c	pki_signature_to_blob	1556	1585	BASE	1574	1577
torture_pki_ed25519.c	torture_pki_ed25519_verify	pki_crypto.c	pki_signature_from_blob	1913	1985	HAVE_OPENSSL_ECC	1967	1973
torture_pki_ed25519.c	torture_pki_ed25519_verify	pki_crypto.c	pki_signature_from_blob	1913	1985	BASE	1967	1973
torture_pki_ed25519.c	torture_pki_ed25519_verify_bad	pki_crypto.c	pki_signature_from_blob	1913	1985	HAVE_OPENSSL_ECC	1967	1973

Table 2: Example - Features Changed in the Commit

Index	TestFile	TestCase	FeatureName	Date
1	torture_pki_ed25519.c	torture_pki_ed25519_cert_verify	BASE	2020-02-10
2	torture_options.c	torture_options_config_host	BASE	2020-02-10
3	torture_pki_ed25519.c	torture_pki_ed25519_generate_key	HAVE_GCRYPT_ECC	2020-02-10
4	torture_pki_ed25519.c	torture_pki_ed25519_cert_verify	HAVE_LIBGCRYPT	2020-02-10
...	...	...	...	...
246	torture_pki_ed25519.c	torture_pki_ed25519_sign_openssh_privkey_passphrase	HAVE_OPENSSL_ECC	2020-02-10
247	torture_misc.c	torture_path_expand_known_hosts	BASE	2020-02-10
248	torture_proxycommand.c	torture_options_set_proxycommand_notexist	BASE	2020-02-10
249	torture_sftp_dir.c	session_teardown	BASE	2020-02-10
250	unittests/torture_pki.c	torture_pki_verify_mismatch	HAVE_ECC	2020-02-10
...	...	...	...	...
597	torture_sftp_canonicalize_path.c	session_setup		
598	torture_knownhosts_parsing.c	torture_knownhosts_host_exists_global	WITH_PCAP	
599	torture_algorithms.c	torture_algorithms_aes128_cbc_hmac_sha1_etm		
...	...	...	...	

Table 3: Changes History

FeatureName	Count	Date
BASE	4960	2020-02-10
WITH_SERVER	161	2019-10-28
HAVE_LIBGCRYPT	160	2020-01-14
HAVE_LIBCRYPTO	142	2020-01-15
_WIN32	91	2019-12-10
DEBUG_CRYPTO	82	2020-01-15
WITH_SSH1	58	2018-06-28
HAVE_ECC	47	2020-02-10
HAVE_DSA	42	2019-11-05
WITH_SFTP	40	2019-11-21
_MSC_VER	40	2020-01-14
HAVE_OPENSSL_ECC	40	2020-02-10
HAVE_ARGP_H	36	2019-05-16
WITH_PCAP	35	2018-11-22
HAVE_SSH1	30	2009-07-25
HAVE_LIBMBEDCRYPTO	28	2020-01-15
HAVE_GCRYPT_ECC	20	2020-02-10
..	...	...

Table 4: Example - Feature Change History

Index	TestFile	TestCase	FeatureName	Date
1	torture_connect.c	torture_run_tests	BASE	2020-02-10
2	torture_knownhosts.c	torture_knownhosts_other_auto	BASE	2020-02-10
3	torture_pki_rsa.c	torture_run_tests	BASE	2020-02-10
...	...	...	...	...
470	pkd_daemon.c	pkd_exec_hello	WITH_SERVER	2019-10-28
471	torture_server_x11.c	test_ssh_channel_request_x11	WITH_SERVER	2019-10-28
472	torture_server_auth_kbdint.c	handle_kbdint_session_cb	WITH_SERVER	2019-10-28
...	...	...	...	...
523	torture_server.c	session_setup	HAVE_ECC	2020-02-10
524	torture_pki_dsa.c	torture_pki_dsa_cert_verify	HAVE_ECC	2020-02-10
..	...	...	...	...

performance obtained by applying only the test case prioritization step (approach called here `FeatTCP`) in comparison with the performance obtained by applying both steps of test case selection and prioritization `FeatTCSP`. In addition to this, the performance of the proposed prioritization strategies are evaluated and compared with changed-file-based strategies as a baseline.

5.1 Research Questions

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

RQ1: *How does the* `FeatTCP` *approach perform considering the three proposed prioritization strategies?* This question aims to understand the impact of the proposed strategies on the effectiveness of the `FeatTCP` approach: `FeatChgHist`, `FeatChgCommit` and `FeatCov`. The main goal is to identify the best strategy to be used in the prioritization, or to find guidelines for their application, considering an indicator of early failure detection.

RQ2: *How does the* `FeatTCSP` *approach perform considering the three proposed prioritization strategies?* This question aims to understand the impact of the proposed strategies on the effectiveness when test case selection and prioritization are employed in a combined way. The answer is obtained using the same indicator of RQ1.

RQ3: *What is the performance of the* `FeatTCP` *approach, adopting the strategy that presented the best performance in RQ1, in comparison with the changed-file based approach?* This question aims to investigate the performance of the best strategy point out by RQ1 in comparison with a baseline, a strategy that considers changes in files instead of features. This baseline was adopted in our previous work [33] and was chosen because as mentioned in Section 2 there are some tools that can map test cases to files and are usually adopted to test case selection. Then in this study, we implemented prioritization strategies based on changed files, similarly to the feature-oriented ones herein proposed.

RQ4: *What is the performance of the* `FeatTCSP` *approach, adopting the strategy that presented the best performance in RQ2, in comparison with the baseline?* This question has similar goal to RQ3, but now the same investigation is conducted to evaluate `FeatTCSP`, when test case selection and prioritization are employed in a combined way in comparison with a file-based TCSP approach

RQ5: *Which of the two approaches,* `FeatTCP` *or* `FeatTCSP` *is the best according to an indicator of early failure detection.* This question evaluates whether it is better to adopt a test prioritization approach alone or in a combination with a previous test case selection step. For this end, the prioritization strategy that presented the best performance in RQ1 and RQ2 are adopted.

5.2 System

As in our previous work [33] we use SSH library (`Libssh`),[4] which is an open source cross-platform C library that implements the

[4] https://www.libssh.org/

Table 5: Example - Feature Coverage

Index	TestFile	TestCase	FeatureNameList
1	torture_pki_ecdsa.c	torture_pki_fail_sign_with_incompatible_hash	['HAVE_OPENSSL_EVP_DIGESTSIGN', 'HAVE_LIBGCRYPT', 'DEBUG_CRYPTO', 'HAVE_OPENSSL_EVP_DIGESTVERIFY', 'HAVE_LIBMBEDCRYPTO', 'WITH_PKCS11_URI', 'HAVE_GCRYPT_ECC', 'HAVE_ECC', 'BASE']
2	torture_packet.c	torture_packet	['WITH_ZLIB', 'DEBUG_PACKET', 'HAVE_GCC_VOLATILE_MEMORY_PROTECTION', 'HAVE_SECURE_ZERO_MEMORY', 'BASE', 'HAVE_MEMSET_S', 'HAVE_ECC', 'WITH_PCAP', 'HAVE_DSA']
3	torture_pki_rsa.c	torture_pki_fail_sign_with_incompatible_hash	['HAVE_LIBMBEDCRYPTO', 'HAVE_LIBGCRYPT', 'HAVE_OPENSSL_EVP_DIGESTVERIFY', 'HAVE_GCRYPT_ECC', 'HAVE_OPENSSL_EVP_DIGESTSIGN', 'HAVE_ECC', 'WITH_PKCS11_URI', 'BASE', 'DEBUG_CRYPTO']
4	torture_pki_dsa.c	torture_pki_fail_sign_with_incompatible_hash	['HAVE_GCRYPT_ECC', 'BASE', 'HAVE_OPENSSL_EVP_DIGESTVERIFY', 'WITH_PKCS11_URI', 'DEBUG_CRYPTO', 'HAVE_LIBGCRYPT', 'HAVE_LIBMBEDCRYPTO', 'HAVE_ECC', 'HAVE_OPENSSL_EVP_DIGESTSIGN']
5	torture_pki_rsa_uri.c	torture_pki_sign_verify_uri	['HAVE_GCRYPT_ECC', 'WITH_PCAP', 'HAVE_DSA', 'HAVE_LIBMBEDCRYPTO', 'HAVE_ECC', 'WITH_PKCS11_URI', 'BASE', 'HAVE_LIBGCRYPT']
6	torture_pki_rsa.c	torture_pki_rsa_sha2	['BASE', 'HAVE_LIBMBEDCRYPTO', 'HAVE_ECC', 'WITH_PKCS11_URI', 'HAVE_DSA', 'HAVE_LIBGCRYPT', 'HAVE_GCRYPT_ECC', 'WITH_PCAP']
..	..	...	...

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,[5] 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 [6, 19, 28, 29, 37].

The Libssh repository contains around 5500 commits, and ≈ 144 features, of which we used 5726. We discarded commits not associated with test cases. This set will be referred as the *whole set of commits.* To evaluate the quality of prioritized test cases, we need logs with failure information. To this end, we used a repository containing 303 commits from a related work [19], referred as *set of commits with logs*

5.3 Indicator

To answer our RQs 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 produce a failure in a commit. To identify each test case fails in the commit, we use the system log. We use 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.

The indicator is defined by Equation 1. 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.

$$\gamma = \frac{\sum_{i=1}^{nf} P_i \text{nt}}{100 * nf} \quad (1)$$

[5]https://gitlab.com/libssh/libssh-mirror

For example, in the commit `17b518a`, there are 27 jobs, and in two of these jobs, failures occurred. In the job `visualstudio/x86_64`[6], the test case `torture_rekey` failed. Meanwhile, in the job `ubuntu/openssl_1.1.x/x86_64`[7], 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 calculate the mean percentage, resulting that in average, 3.7% of test cases need to be execute to find both failures.

5.4 Baseline implementation

To extract the HCS commit history, we use the PyDriller library [44] from the initial to the last available commit. To implement the baseline, a changed file-oriented approach, the source code is scanned, looking for the test files. Then, traceability of test cases to source code is created. We then use PyDriller to find the files that changed in that commit. Then we create a history of file changes. Finally, we select test cases that have traceability to these files. Furthermore, as a baseline, we adopt two file-based prioritization strategies: 1) File Change History (`FileChgHist`); 2) Files Changed in the Commit (`FileChgCommit`).

Analogously to the features-based strategies, to implement the `FileChgHist` strategy, we developed a list that maintains the incremental history of file changes commit by commit. For each file, we record the number of changes it underwent throughout the evolution of the system. Thus, prioritization is based on this list, ensuring that test cases with traceability to the most altered files are always at the top of the list. For the `FileChgCommit` strategy, we also consider file changes, but in this case, only files changed in the current commit. Thus, we generate a list of files that were changed in this commit and then create the prioritized list of test cases based on this list. Therefore, the test cases at the top of the list will be those with traceability to the changed files, while the rest of the list will be random.

[6]https://gitlab.com/libssh/libssh-mirror/-/jobs/432862254
[7]https://gitlab.com/libssh/libssh-mirror/-/jobs/432862274

Table 6: `FeatTCP` **- Budgets for Based Strategies**

Strategies	Avg	Max	Min
FeatChgHist	12.6%	56.5%	0.15%
FeatChgCommit	12.6%	50.6%	0.28%
FeatCov	20.0%	54.1%	0.48%

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

6 ANALYSIS OF RESULTS

This section presents and discusses the results to answer the RQs or our study. The dataset and results are available online [34], as a supplementary material.

6.1 RQ1: Feature-Oriented Strategies for FeatTCP

To answer RQ1, we calculated the early failure detection indicator for the prioritization list of test cases obtained by FeatTCP, commit to commit. As a result we obtained Figure 6 that presents a comparison of the values obtained for the three proposed strategies in the 122 commits that have failures. It is noticeable that the `FeatCov` strategy (in red) exhibits several high peaks in the budget percentage, surpassing 30%. This indicates that this strategy appears slightly less effective than the other two. When we examine `FeatChgCommit` (in blue) and `FeatChgHist` (in green), both strategies appear quite similar. In some specific commits, we observe higher peaks in the blue line, while in other points, peaks in the green line. This suggests that for some commits, the `FeatChgHist` strategy is more efficient, while for others, `FeatChgCommit` is preferable.

Furthermore, Table 6 presents the average budget for all commits, the minimum budget, and the maximum budget for each proposed strategy. Upon analyzing the table, we observe that the average budgets for the `FeatChgCommit` and `FeatChgHist` strategies are similar, with a variation between maximum and minimum values. `FeatChgCommit` has a lower maximum budget of 50.6%, while `FeatChgHist` has a lower minimum budget of 0.15%. Additionally, `FeatCov` shows maximum budget close to the other strategies, but a greater minimum value of 0.48%, and an average of 20%.

We collected the total time `FeatTCP` takes to execute for the whole set of commits (a total of 5726), running each strategy individually. Table 7 displays the average time in seconds for each strategy, along with the maximum and minimum times. As we can observe, the strategies have very similar execution times around 15s. The average maximum time is $\approx$ 31.38 seconds, while the average minimum time is $\approx$ 5.9 seconds. We also observe that the interval between the CI cycles is on average 1,142.52 minutes (standard deviation equals to 459.28), what shows the applicability of the `FeatTCP` approach.

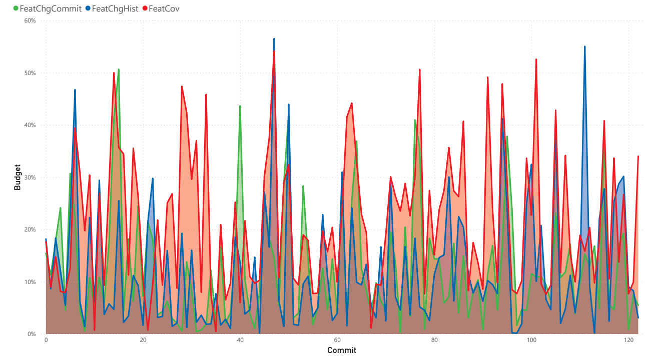


Figure 6: Comparing Features-Oriented Strategies - `FeatTCP` **approach**

Table 7: `FeatTCP` **- Execution Time for each strategy**

Strategies	Avg(s)	Max(s)	Min(s)
FeatChgHist	15.17	31.27	6.01
FeatChgCommit	15.18	31.11	6.02
FeatCov	15.21	31.77	5.89

RQ1: *We conclude that the strategies* `FeatChgHist` *and* `FeatChgCommit` *have similar budget values, with an average budget of 12.6%, indicating a low cost for fault identification. On the other hand, the* `FeatCov` *strategy has a slightly higher average budget, reaching 20%. We also observe that the runtime is not a decisive factor in choosing the strategy, as they are similar. Additionally, we find that* `FeatTCP` *can be applied in the evolution process of HCSs and in CI environments, with an average execution time of 15.18 seconds for all commits and a maximum time of 31.38 seconds.*

6.2 RQ2: Strategies for FeatTCSP

In this RQ analysis, we analyse the performance of the feature-oriented strategies used with `FeatTCSP`, approach that combines test case selection and prioritization. Similarly to RQ1, we generate Figure 7. We can observe that `FeatChgCommit` (in green) and `FeatChgHist` (in blue) are mostly equivalent in most commits, showing peaks in a few cases. `FeatCov` (in red) is slightly superior in several commits, requiring a higher budget to detect the fault.

To complement the analysis, we developed Table 8. In it, we can observe that the strategies `FeatChgHist` and `FeatChgCommit` have a similar average budget, 13.7% and 13.2%, respectively. Additionally, we noticed that the maximum and minimum budget times for the `FeatChgHist` strategy are lower compared to the other strategies. The `FeatCov` strategy showed a higher budget, with an average of 23.4%, and also a higher minimum budget, with 0.56%.

When combining the two selection and prioritization approaches, one might expect a significant increase in execution time. However, this is not the case due to the common steps between them: *Identify HCS resources*, *Identify feature changed*, and *Map features to test cases*. Together, these steps have an average time of approximately 14.7 seconds. The average time for the prioritization step, considering

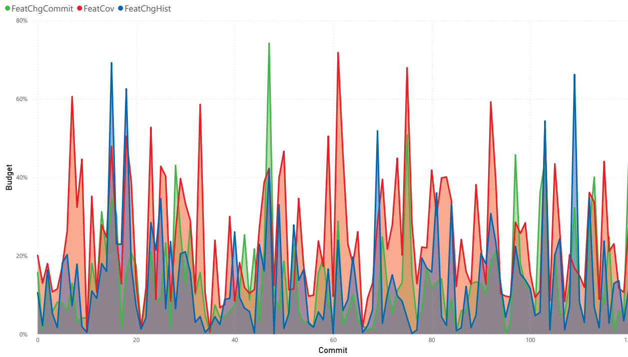


Figure 7: Comparing Features-Oriented Strategies - FeatTCSP approach

Table 8: FeatTCSP - Budgets for Feature-Based Strategies

Strategies	Avg	Max	Min
FeatChgHist	13.7%	69.2%	0.13%
FeatChgCommit	13.2%	74.2%	0.23%
FeatCov	23.4%	71.8%	0.56%

the FeatChgHist strategy, is ≈ 0.9 seconds, while for selection, it is ≈ 0.6 seconds. Therefore, we can conclude that, with a cost of less than 1 second, we can add the selection step to the approach.

RQ2: *We conclude* FeatChgCommit *has lower average budget, reaching 13.2%, while* FeatChgHist *has an average budget only 0.5% higher, with lower minimum value. On the other hand,* FeatCov *obtains a higher budget, with a difference of approximately 10%. Thus, we conclude that the* FeatChgCommit *strategy can identify faults with a lower budget. An additional execution time of less than one second is required to execute the TCS step in comparison with* FeatTCP*, then we can conclude that* FeatTSCP *is also applicable in the CI environment.*

6.3 RQ3: On the use of FeatTCP with file-based strategies

To answer this RQ, we employed the FeatTCP approach with file-based strategies and compared the performance of these strategies with the best feature-oriented strategy identified in RQ1 - FeatChgHist. To facilitate the analysis, we developed the graphs presented in Figures 8 and 9, comparing FeatChgHist with FileChgHist and FileChgCommit, respectively. The budget for the FileChgHist strategy is higher, reaching several peaks above 80%, as evidenced in Figure 8 by the gray line. Furthermore, it is observed that, in the majority of commits, this strategy has a higher budget, with an average budget of 31%, as seen in Table 9. On the other hand, in Figure 9, the FileChgCommit strategy (gray line) showed an average budget close to feature-oriented strategies, demonstrating a budget very similar to the FeatChgHist strategy, with only a few commits showing significantly higher budget peaks.

Additionally, Table 9 shows that the FileChgHist strategy has a maximum budget higher than all other strategies, reaching 87.6%,

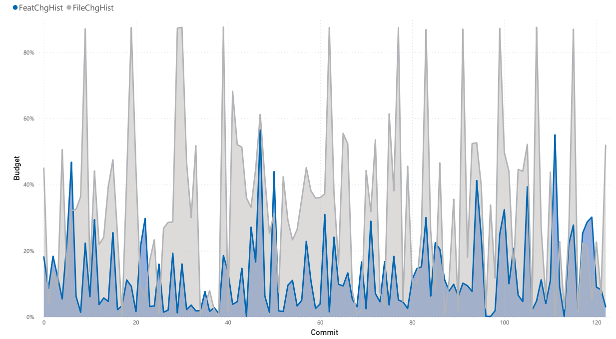


Figure 8: Comparing FeatChgHist with FileChgHist - FeatTCP approach.

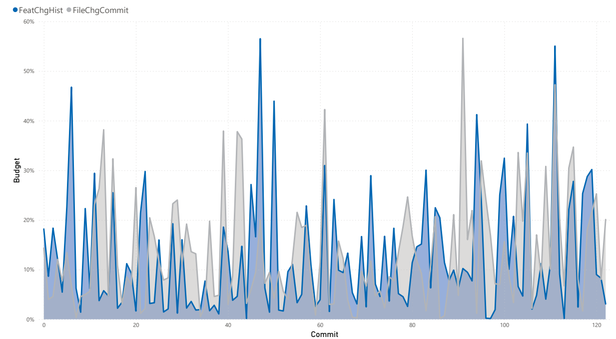


Figure 9: Comparing FeatChgHist with FileChgCommit - FeatTCP approach.

Table 9: FeatTCP - Budgets for File-Based Strategies

Strategies	Avg	Max	Min
FileChgHist	31.0%	87.6%	0.36%
FileChgCommit	13.7%	56.5%	0.17%

whereas the FileChgCommit strategy has a maximum budget of 56.5%, which is very close to the maximum budgets of all other strategies. This strategy has an average budget very close to the FeatChgHist strategy at 13.7%.

RQ3: *File-based strategies have a higher budget compared to the feature-oriented strategies. The* FileChgHist *strategy has an average budget ≈ 17% higher than the* FeatChgHist *strategy. On the other hand, the* FileChgCommit *strategy has a better performance, which is similar to the performance of* FeatChgHist.

6.4 RQ4: On the use of a file-based TCSP approach

To answer this RQ, we apply both file-based prioritization strategies to the test set selected with a file-based selection, adopted as a baseline in our previous work [33]. Then we compared the obtained results with the results of our approach FeatTCSP using the

best prioritization strategy pointed out by RQ2. Answering RQ2, we observed that the FeatChgHist and FeatChgCommit strategies can be considered equivalent. However, due to the fact that FeatChgCommit, on average, has a slightly lower budget in 0.5% of commits, it was chosen in the comparison.

Figure 10 compares the file-based strategy FileChgHist (in gray) with the feature-oriented strategy FeatChgCommit. It can be observed that the gray line in the graph often remains close to 100%. We consider it 100% when the input set of test cases needs to be executed in its entirety, or if the test case that fails is not in the set, meaning it was not selected. When comparing the FeatChgCommit strategy with FileChgCommit (Figure 11), a similar behavior is observed. The file-based approach is not able to select an ideal set of test cases for prioritization.

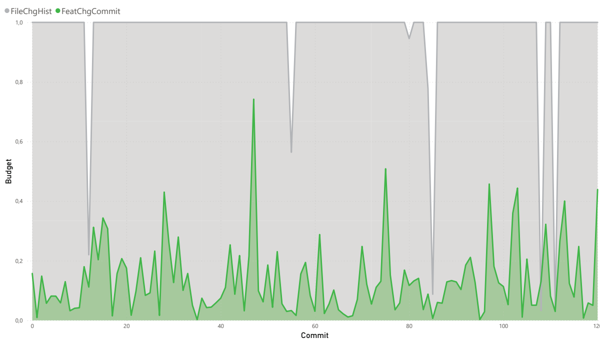


Figure 10: Comparing FeatChgCommit with FileChgHist - FeatTCSP and file-based TSCP approaches

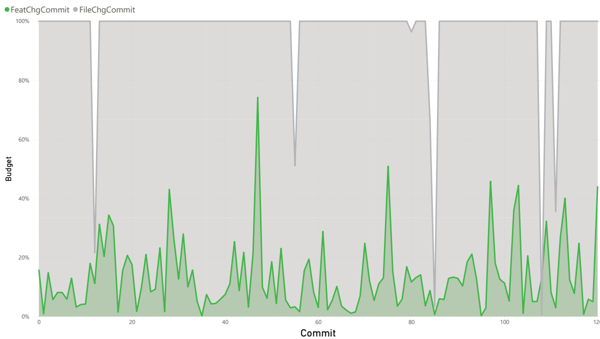


Figure 11: Comparing FeatChgCommit with FileChgCommit - FeatTCSP and file-based TSCP approaches

In Table 10, we can observe that file-based strategies have higher budgets than feature-oriented strategies, with average of 96.8%. Additionally, both strategies have maximum budgets of 100%, and minimum of 0.46% for the FileChgHist strategy and 3.2% for the FileChgCommit strategy.

> **RQ4**: *When a file-based selection of test cases is used in combination with file-based prioritization strategies, a poor performance is obtained. The budgets are considerable lower than the ones obtained by using FeatTCSP.*

Table 10: Budgets for File-based TCSP approach

Strategies	Avg	Max	Min
FileChgHist	96.8%	100.0%	0.46%
FileChgCommit	96.8%	100.0%	3.2%

6.5 RQ5: Comparing FeatTCP and FeatTCSP

To answer RQ5, we compared FeatTCP with FeatTCSP in terms of the number of test cases in the input sets and the budget required to produce a failure. By analysing Tables 6 and 8, we can see that for the strategies FeatChgHist and FeatChgCommit the impact in the average budget values of using as input selected test cases is not so great (around 1%). The minimum values when the selected test cases are used as input are lower. A great impact is observed when we analyse the maximum values. In some commits a budget of 10% greater is required. Then we can see that many times the use of TCSP can be advantageous because it allows the early fault detection with the same average budgets with a reduced number of test cases.

To corroborate this result, we compare FeatTCP and FeatTCSP approaches according to the number of test required and generate Figure 12, which displays a graph for the 122 commits with failures using the FeatChgHist strategy. It is possible to observe in orange the whole test set for the commit (equivalent to a Retest-All approach), in yellow the test cases selected by FeatTestSel in the commit. At this point, a significant difference in the number of test cases between these sets, that can be used as inputs for the prioritization process, is already noticeable.

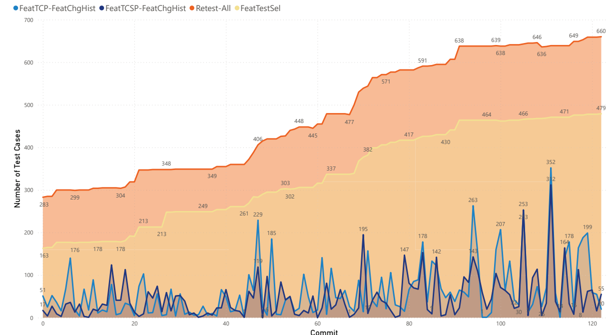


Figure 12: Comparing approaches according to number of test cases

In the light blue line, we represent the number of test cases needed to produce a failure considering the total set, produced by FeatTCP. A substantial difference is observed regarding Retest-All approach. In the dark blue line, we represent the test cases produced by FeatTCSP. In this case, we start with a smaller set to reduce the budget. Analyzing Figure 12, it is evident that the dark blue line is below the light blue line for most commits. We then can conclude that it is possible to identify the fault, that is, to maintain the budget, with a smaller number of test cases by using FeatTCSP.

RQ5: FeatTCP *and* FeatTCSP *approaches using the* FeatChgHist *strategy, can identify the fault on average with a budget of, respectively, only 12% and 13%. But* FeatTCSP *seems to be more advantageous because it allows you to run a reduced test suite on 57% of the measured faulty commits.*

7 CONCLUDING REMARKS

Our selection and prioritization approach (FeatTCSP) has advantages over existing approaches. Unlike other selection approaches that mainly focus on test cases directly related to altered files without considering features, our approach takes into account interactions with other parts of the features being modified in the provided commit. Even if the altered file is not directly affected by the test case, if the test case interacts with other parts of the modified features, it will be selected for regression testing. As features are building blocks of HCSs, it is important to verify the behavior of the change features.

In the case of prioritization approaches, 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, unlike many prioritization approaches. Overall, our FeatTCSP approach uses only the source code of the HCS and a configuration file as input.

Furthermore, FeatTCSP is independent of any learning or optimization model. Furthermore, instead of dynamic analysis, which would require execution, we employ static analysis that relies solely on the HCS source code as input. FeatTCSP was designed to integrate perfectly into the HCSs evolution process, aiming to substantially reduce the cost associated with executing test cases.

In summary, our study provides valuable insights into the effectiveness of various strategies in the context of HCS evolution. For the FeatTCP approach, the comparison between feature-oriented strategies, FeatChgHist and FeatChgCommit, revealed similar results, showing an average budget of approximately 12.6%. The FeatCov strategy demonstrated a higher budget among feature-oriented strategies, ≈ 20%.

On the other hand, the file-based strategy, FileChgCommit, has a slightly higher average budget of approximately 13.7%, while the FileChgHist strategy presente the highest budget of approximately 31.1%.

Furthermore, our conclusions emphasize the superiority of feature-oriented strategies over their file-based counterparts, especially when using a combined approach of test case selection and prioritization. Our FeatTCSP approach achieves significantly lower average budgets of 96.8%, which is the baseline's average budget. Both FeatChgHist and FeatChgCommit, as feature-oriented strategies, demonstrate comparable average budgets of approximately 13.5%, while the FeatCov strategy shows a slightly higher average budget of 23.4%.

Therefore, we can conclude that leveraging a smaller set of inputs through the combined FeatTCSP approach further reduces the budget required for fault detection. In 57% of commits with faults, this combined approach results in fewer test cases needed to run to find the fault, compared to using FeatTCP alone. These results demonstrate the adaptability and effectiveness of our innovative approach to the challenges posed by the continuous evolution of HSCs.

After result analyzing our approach along with the proposed strategies, we recommend using the combined approach FeatTCSP. This is because the increase in execution time is small, but we manage to identify failures by running fewer test cases in more than 50% of the commits. As a strategy, we suggest adopting the FeatChgHist strategy, as it presents more consistent values for the two approaches FeatTCS and FeatTCSP. Additionally, its maximum and minimum budget values are lower than the other strategies.

As future work, we will search for other systems to improve the validation results. We intend to propose a feature-oriented indicator for a more robust validation of the approaches. Additionally, we should consider using Artificial Intelligence algorithms for a potential comparison and/or improvement of the results. Another research direction involves creating a unified prioritization strategy that incorporates all proposed strategies in a defined order, with established tie-breaking criteria.

ACKNOWLEDGMENTS

This research is supported by CNPq (grant: 310034/2022-1), FAPERJ PDR-10 Fellowship (grant: 202073/2020) and CAPES (grant: 88887.464736/2019-00).

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] 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.

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

[4] Paul M Duvall, Steve Matyas, and Andrew Glover. 2007. *Continuous integration: improving software quality and reducing risk.* Pearson Education.

[5] Alireza Ensan, Ebrahim Bagheri, Mohse Asadi, Dragan Gasevic, and Yevgen Biletskiy. 2011. Goal-Oriented Test Case Selection and Prioritization for Product Line Feature Models. In *8th Intl. Conference on Information Technology: New Generations*. IEEE, 291–298. https://doi.org/10.1109/ITNG.2011.58

[6] 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.

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

[8] 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.

[9] Ines Hajri, Arda Goknil, Fabrizio Pastore, and Lionel C Briand. 2020. Automating system test case classification and prioritization for use case-driven testing in product lines. *Empirical Software Engineering* 25, 5 (2020), 3711–3769.

[10] Bo Jiang and WK Chan. 2016. Testing and debugging in continuous integration with budget quotas on test executions. In *2016 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 439–447.

[11] 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.

[12] 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.

[13] Pilsu Jung, Seonah Lee, and Uicheon Lee. 2023. Automated Code-based Test Case Reuse for Software Product Line Testing. *Information and Software Technology* (2023), 107372.

[14] C. H. 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.

[15] C. H. 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] Jackson A. Prado Lima, Willian D. F. Mendonça, Silvia R. Vergilio, and Wesley K. G. 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: Volume A - Volume A* (Montreal, Quebec, Canada) *(SPLC '20)*. ACM, New York, NY, USA, Article 31, 11 pages. https://doi.org/10.1145/3382025.3414967

[21] 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.

[22] Frank J. van der Linden, Klaus Schmid, and Eelco Rommes. 2007. *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*. Springer-Verlag, Berlin, Heidelberg.

[23] 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.

[24] Malte Lochau, Ina Schaefer, Jochen Kamischke, and Sascha Lity. 2012. Incremental model-based testing of delta-oriented software product lines. In *Intl. Conference on Tests and Proofs*. Springer, 67–82.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] Willian DF Mendonça, Wesley KG Assunção, and Silvia R Vergilio. 2022. Software Product Line Regression Testing: A Research Roadmap. In *Intl. Conference on Enterprise Information Systems (ICEIS)*. SciTePress, 81–89.

[32] 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.

[33] 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 the 27th ACM International Systems and Software Product Line Conference - Volume A* (Tokyo, Japan) *(SPLC '23)*. Association for Computing Machinery, New York, NY, USA, 76–86. https://doi.org/10.1145/3579027.3608979

[34] Willian Mendonça, Silvia R Vergilio, and Wesley K G Assunção. 2023. Supplementary Material - Feature-oriented Test Case Selection and Prioritization for Highly-Configurable Software. osf.io/6wre9

[35] 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

[36] 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.

[37] 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.

[38] José A Parejo, Ana B Sánchez, Sergio Segura, Antonio Ruiz-Cortés, Roberto E Lopez-Herrejon, and Alexander Egyed. 2016. Multi-objective test case prioritization in highly configurable systems: A case study. *Journal of Systems and Software* 122 (2016), 287–310.

[39] Jackson A Prado Lima, Willian DF Mendonça, Silvia R Vergilio, and Wesley KG Assunção. 2022. Cost-effective learning-based strategies for test case prioritization in continuous integration of highly-configurable software. *Empirical Software Engineering* 27, 6 (2022), 133.

[40] 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.

[41] 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.

[42] 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.

[43] Paulo Anselmo da Mota Silveira Neto, Ivan do C. 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.

[44] 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.

[45] 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.

[46] 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.

[47] Shuai Wang, Shaukat Ali, Arnaud Gotlieb, and Marius Liaaen. 2016. A systematic test case selection methodology for product lines: results and insights from an industrial case study. *Empirical Software Engineering* 21 (2016), 1586–1622.

[48] Zhihong Xu, Myra B. Cohen, Wayne Motycka, and Gregg Rothermel. 2013. Continuous Test Suite Augmentation in Software Product Lines. In *17th Intl. Software Product Line Conference* (Tokyo, Japan) *(SPLC '13)*. Association for Computing Machinery, New York, NY, USA, 52–61. https://doi.org/10.1145/2491627.2491650

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

[50] 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.

6 CONCLUSIONS

The test of HCSs is a challenging task due to the great number of configurations involved, particularly in the CI context, where the software needs to be integrated and tested continuously. This makes RT an expensive task, and the application of TCS and TCP techniques are fundamental to reduce costs. However, the great majority of existing approaches do not consider HCS particularities. Motivated by these facts, this work introduced `FeaTCSP` (**Feat**ure-oriented **T**est **C**ase **S**election and **P**rioritization for Highly Configurable Systems), a feature-oriented approach that allows the use of TCS and TCP techniques considering the evolution of the HCS features in space and time, as well as the CI environment restrictions. This approach takes as input a configuration file, containing the paths to the source code of the HCS and the test case folder, and produces different reports: 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 automate these steps, we implemented `Test2Feature`, which allows developers and testers to know the lines of the source code and features that correspond to a test case. Differently from existing tools, `Test2Feature` works with annotated HCSs written in C/C++ and is based only on the static analysis of the code. The traceability report produced can ease different tasks, as for example, regression testing, feature management, and HCS evolution and maintenance.

Oriented by the reports produced by `Test2Feature`, the tester can use the approach in different ways according to the testing and organization goals. We can mention the visualization of features evolution in space an time, and the corresponding test cases, what can facilitate future maintenance and refactoring activities, as well as other tasks related to the ensurance of HCSs. A second use is just to reduce the number of test cases according to a feature-oriented approach namely `FeatTestSel`. Another use is the feature-oriented prioritization of test cases, that employs the prioritization strategies proposed having as input either all the test set available (`FeatTCP` approach), or the test sets selected (`FeatTSCP` approach). To perform prioritization, three strategies are proposed, based on the mapping of features to test cases generated and uses three information sources: 1) `FeatChgHist`: the history of feature changes; 2) `FeatChgCommit`: features that changed in the current commit; and 3) `FeatCov`: the number of features covered by the test case.

All the ways of using `FeaTCSP` were evaluated with `Libssh`, a real open-source HCS in constant evolution. The results shows the applicability of the approach considering time to execute the approaches and CI cycles. Moreover, it contributes to the RT of HCSs; in all uses, `FeaTCSP` presents better performance than a baseline, the file-based approach. In this sense, `FeaTCSP` contributes to choosing a reduced set of test cases without compromising the quality of the test in terms of detected faults. In addition, the use of the TCP technique aims to provide early fault detection, reducing the time spent running tests.

In summary, our study provides valuable insights into the effectiveness of TCS and TCP in the context of HCS evolution. Regarding the `FeatTCP` approach, the comparison between feature-oriented strategies, pointed out that `FeatChgHist` and `FeatChgCommit` reachs similar results, showing an average budget of approximately 12.6%. The `FeatCov` strategy exhibited a higher budget among the feature-oriented strategies, approximately 20%. Our conclusions emphasize the superiority of feature-oriented strategies over their file-based counterparts, especially when adopting a combined approach to test case selection and prioritization. `FeatTCSP` achieves average budgets of around 13.5%, while the `FeatCov` strategy shows a slightly higher average budget of 23.4%. On the other hand, considering file-based

approaches, using the `FileChgCommit` and `FileChgHist` strategies, the average budget is around 96.8%, with peaks reaching 100%. In addition to this, the results point out that by using a more restricted set of inputs through the combined `FeatTCSP` approach, we further reduce the budget needed for fault detection. In 57% of faulty commits, this combined approach results in fewer test cases that need to be executed to identify the fault, compared to using only `FeatTCP`. These results demonstrate the adaptability and effectiveness of our innovative approach in addressing the challenges posed by the continuous evolution of HCS.

6.1 CONTRIBUTIONS

This work contributes to the field of RT, offering a lightweight, model-free approach, applicable in the evolutionary process of HCSs and in CI environments, standing out for the automation of test case selection and prioritization. Summarizing we can mention the following main contributions:

- A systematic mapping about Software Product Line Regression Testing and research roadmap: we present a comprehensive overview of the current state of research in regression testing for SPL, while also providing a roadmap that can guide future research in the coming years;
- Tool for mapping test cases to lines of code and mapping test cases to Features - `Test2Feature`: we have developed a tool that generates traceability among test cases, source code lines, and features of HCSs. This tool can not only be used to assist engineers in decision-making but can also be integrated into future approaches and tools. The tool is available online[1].
- Feature-Oriented Selection Test Case Approach - `FeaTestSel`: this approach selects test cases for HCSs and the evaluation shows it contributes to reduce the set of test cases, and consequently test costs, without compromising quality.
- Feature-Oriented Selection and Prioritization Approach - `FeatTCSP`: we introduce a new feature-oriented test case prioritization approach that ranks test cases to anticipate failures. This approach can be employed by engineers to reduce the costs of the testing phase in CI environments. It first ranks test cases related to changed features that are more fault-proneness. This contributes to reveal failures early, stop the test execution more rapidly and reduce time between CI cycles.
- Test case selection and prioritization approaches based on file changes: we have implemented an approach based on file changes, which can be used as a baseline by the academic community for comparison and validation of their own approaches or tools.
- Replication packages for all studies conducted: Public repositories containing the dataset used in this work, implementations and results, which allow replication and can be used in future research, are available in the Open Science Framework (OSF).
  - Mendonça, W., Vergilio, S. R., and Assunção, W. K. G. (2023). Supplementary material - feature-oriented test case selection during evolution of high-configurable systems (Mendonça et al., 2023b) - available on OSF[2].

[1] https://github.com/willianferrari/Test2Feature.git
[2] https://osf.io/6wre9/

– Mendonça, W., Vergilio, S. R., and Assunção, W. K. G. (2023a). Supplementary material - feature-oriented test case selection and prioritization for highly-configurable software.(Mendonça et al., 2023a)- available on OSF[3].

6.2 LIMITATIONS

This section highlights the inherent limitations of the proposed approach, which constitute areas for improvement in future work. One of the main considerations is the restricted evaluation of the approach on a limited number of systems. Future investigations should broaden this analysis to include a more extensive range of systems, providing a more robust understanding of the applicability of the approach in diverse contexts. Another limitation is the use of only one indicator for the test budget.

While we have delved deeply into the evolution of HCSs, the lack of specific focus on the continuous integration environment is an area deserving additional attention. Given that continuous integration environments are constantly evolving, future studies can direct their efforts to better understand the specific nuances of this dynamic environment. Another relevant point concerns the budget limitation, both in the general evolution and in the specific context of continuous integration environments. In this regard, improving the step of the approach responsible for mapping and locating features is crucial to optimize resource allocation and ensure more efficient budget management.

6.3 FUTURE WORK

As a future work for our tool, we intend to develop a user-friendly interface and organize the results in different ways, improving the visualization for the user. Additionally, we plan to apply a validation process for the traceability quality, utilizing industrial-grade HCSs, and generation of test cases based on the code modifications.

For the `FeatTestSel` approach, we aim to 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 feature change.

We also intend to validate our approach with other systems and adopt other indicators for the prioritization strategies. Additionally, we will consider using Artificial Intelligence for a potential comparison and/or improvement of the results. Another research direction will involve creating a unified prioritization strategy that incorporates all proposed strategies in a defined order, with established tie-breaking criteria.

[3]https://osf.io/yd2wf/

REFERENCES

Al-Hajjaji, M., Lity, S., Lachmann, R., Thüm, T., Schaefer, I., and 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)*, pages 34–40. IEEE.

Apel, S., Batory, D., Kstner, C., and Saake, G. (2013). *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer.

Assunção, W., Mendonça, W., and Vergilio, S. (2018). Reúso de software: Do oportunista ao sistemático. In *Anais da II Escola Regional de Engenharia de Software*, pages 157–162. SBC.

Assunção, W. K. G., Lopez-Herrejon, R. E., Linsbauer, L., Vergilio, S. R., and Egyed, A. (2017). Reengineering legacy applications into software product lines: A systematic mapping. *Empirical Software Engineering*, pages 1–45.

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

bin Ali, N., Engström, E., Taromirad, M., Mousavi, M. R., Minhas, N. M., Helgesson, D., Kunze, S., and Varshosaz, M. (2019). On the search for industry-relevant regression testing research. *EMSE*, 24(4):2020–2055.

Clements, P. and Northrop, L. (2001). *Software Product Lines: Practices and Patterns*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

da Silva, H. N., Farah, P. R., Mendonça, W. D. F., and Vergilio, S. R. (2019). Assessing android test data generation tools via mutation testing. In *Proceedings of the IV Brazilian Symposium on Systematic and Automated Software Testing*, pages 32–41.

Dintzner, N., van Deursen, A., and Pinzger, M. (2018). Fever: An approach to analyze feature-oriented changes and artefact co-evolution in highly configurable systems. *Empirical Software Engineering*, 23(2):905–952.

Duvall, P. M., Matyas, S., and Glover, A. (2007). *Continuous integration: improving software quality and reducing risk*. Pearson Education.

Elbaum, S., Rothermel, G., and Penix, J. (2014). Techniques for improving regression testing in continuous integration development environments. In *22nd ACM SIGSOFT Intl. Symposium on Foundations of Software Engineering*, pages 235–245.

Engström, E. (2010a). Exploring regression testing and software product line testing: Research and state of practice. Licentiate Thesis, Department of Computer Science, Lund University.

Engström, E. (2010b). Regression test selection and product line system testing. In *3rd International Conference on Software Testing, Verification and Validation*, pages 512–515. IEEE.

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

Jiang, B. and 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*, QRS, page 439–447. IEEE.

Jin, D., Cohen, M. B., Qu, X., and Robinson, B. (2014). Preffinder: Getting the right preference in configurable software systems. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 151–162.

Jung, P., Kang, S., and Lee, J. (2019). Automated code-based test selection for software product line regression testing. *Journal of Systems and Software*, 158:110419.

Jung, P., Kang, S., and Lee, J. (2020). Efficient regression testing of software product lines by reducing redundant test executions. *Applied Sciences*, 10(23):8686.

Jung, P., Lee, S., and Lee, U. (2023). Automated code-based test case reuse for software product line testing. *Information and Software Technology*, page 107372.

Kästner, C., Giarrusso, P. G., Rendel, T., Erdweg, S., Ostermann, K., and Berger, T. (2011). Variability-aware parsing in the presence of lexical macros and conditional compilation. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, pages 805–824.

Kim, C. H. P., Batory, D. S., and Khurshid, S. (2011). Reducing combinatorics in testing product lines. In *10th Intl. conference on Aspect-oriented software development*, pages 57–68.

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

Lachmann, R., Beddig, S., Lity, S., Schulze, S., and Schaefer, I. (2017). Risk-based integration testing of software product lines. In *11th Intl. Workshop on Variability Modelling of Software-intensive Systems*, pages 52–59.

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

Lima, J. A. P., Mendonça, W. D., Vergilio, S. R., and Assunção, W. K. (2020). Learning-based prioritization of test cases in continuous integration of highly-configurable software. In *24th ACM conference on systems and software product line*, pages 1–11.

Lima, J. A. P. and Vergilio, S. R. (2020a). A multi-armed bandit approach for test case prioritization in continuous integration environments. *IEEE Transactions on Software Engineering*, 48(2):453–465.

Lima, J. A. P. and Vergilio, S. R. (2020b). Test case prioritization in continuous integration environments: A systematic mapping study. *Information and Software Technology*, 121:106268.

Lity, S., Nieke, M., Thüm, T., and Schaefer, I. (2019). Retest test selection for product-line regression testing of variants and versions of variants. *Journal of Systems and Software*, 147:46–63.

Machado, I. d. C., McGregor, J. D., Cavalcanti, Y. C., and De Almeida, E. S. (2014). On strategies for testing software product lines: A systematic literature review. *Information and Software Technology*, 56(10):1183–1199.

Marijan, D., Gotlieb, A., and Liaaen, M. (2019). A learning algorithm for optimizing continuous integration development and testing practice. *Software: Practice and Experience*, 49(2):192–213.

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

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

Marijan, D. and Sen, S. (2017). Detecting and reducing redundancy in software testing for highly configurable systems. In *2017 IEEE 18th International Symposium on High Assurance Systems Engineering (HASE)*, pages 96–99. IEEE.

Medeiros, F., Ribeiro, M., Gheyi, R., Apel, S., Kästner, C., Ferreira, B., Carvalho, L., and Fonseca, B. (2017). Discipline matters: Refactoring of preprocessor directives in the# ifdef hell. *IEEE Transactions on Software Engineering*, 44(5):453–469.

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

Meinicke, J., Wong, C.-P., Kästner, C., Thüm, T., and Saake, G. (2016). On essential configuration complexity: Measuring interactions in highly-configurable systems. In *31st IEEE/ACM Intl. Conference on Automated Software Engineering*, pages 483–494.

Mendonça, W. D., Assunção, W. K., Estanislau, L. V., Vergilio, S. R., and Garcia, A. (2020). Towards a microservices-based product line with multi-objective evolutionary algorithms. In *2020 IEEE Congress on Evolutionary Computation (CEC)*, pages 1–8. IEEE.

Mendonça, W. D., Assunção, W. K., and Vergilio, S. R. (2019). Reusing test cases on graph product line variants: Results from a state-of-the-practice test data generation tool. In *Proceedings of the IV Brazilian Symposium on Systematic and Automated Software Testing*, pages 52–61.

Mendonça, W. D., Assunção, W. K., and Vergilio, S. R. (2022a). Software product line regression testing: A research roadmap. In *Intl. Conference on Enterprise Information Systems (ICEIS)*, pages 81–89. SciTePress.

Mendonça, W. D., Vergilio, S. R., Michelon, G. K., Egyed, A., and Assunção, W. K. (2022b). Test2feature: feature-based test traceability tool for highly configurable software. In *26th ACM Intl. Systems and Software Product Line Conference-Volume B*, pages 62–65.

Mendonça, W. D. F., Assunção, W. K. G., and Vergilio, S. R. (2024). Feature-oriented test case selection and prioritization for highly-configurable system. In *To be submitted to special issue of SPLC2023, Journal of Systems and Software*. Elsevier.

Mendonça, W., Vergilio, S. R., and Assunção, W. K. G. (2023a). Supplementary material - feature-oriented test case selection and prioritization for highly-configurable software.

Mendonça, W., Vergilio, S. R., and Assunção, W. K. G. (2023b). Supplementary material - feature-oriented test case selection during evolution of high-configurable systems.

Mukelabai, M., Nešić, D., Maro, S., Berger, T., and Steghöfer, J.-P. (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*, pages 155–166.

Pett, T., Krieter, S., Runge, T., Thüm, T., Lochau, M., and Schaefer, I. (2021). Stability of Product-Line Samplingin Continuous Integration. In *15th international Conference on Variability Modelling of Software-Intensive Systems*, VaMoS'21. Association for Computing Machinery.

Pohl, K., Böckle, G., and Linden, F. J. v. d. (2005). *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

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

Qu, X., Acharya, M., and Robinson, B. (2012). Configuration selection using code change impact analysis for regression testing. In *28th IEEE Intl. Conference on Software Maintenance*, pages 129–138. IEEE.

Queiroz, R., Passos, L., Valente, M. T., Hunsen, C., Apel, S., and Czarnecki, K. (2017). The shape of feature code: an analysis of twenty c-preprocessor-based systems. *Software & Systems Modeling*, 16(1):77–96.

Romano, S., Scanniello, G., Antoniol, G., and Marchetto, A. (2018). Spiritus: A simple information retrieval regression test selection approach. *Information and Software Technology*, 99:62–80.

Rothermel, G., Untch, R. H., Chu, C., and Harrold, M. J. (2001). Prioritizing test cases for regression testing. *IEEE Transactions on software engineering*, 27(10):929–948.

Runeson, P. and Engström, E. (2012). Chapter 7 - regression testing in software product line engineering. In Hurson, A. and Memon, A., editors, *Advances in Computers*, volume 86, page 223–263. Elsevier.

Ruparelia, N. B. (2010). The history of version control. *ACM SIGSOFT Software Engineering Notes*, 35(1):5–9.

Seidl, C., Schaefer, I., and Aßmann, U. (2014). Integrated management of variability in space and time in software families. In *Proceedings of the 18th International Software Product Line Conference-Volume 1*, pages 22–31.

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

Tuglular, T. and 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*, volume 2, pages 344–349. IEEE.

Van der Linden, F. J., Schmid, K., and Rommes, E. (2007). *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*. Springer Science & Business Media, Berlin, Heidelberg.

Von Rhein, A., Grebhahn, A., Apel, S., Siegmund, N., Beyer, D., and Berger, T. (2015). Presence-condition simplification in highly configurable systems. In *IEEE/ACM 37th IEEE Intl. Conference on Software Engineering*, volume 1, pages 178–188. IEEE.

Yoo, S. and Harman, M. (2012). Regression testing minimization, selection and prioritization: a survey. *Software testing, verification and reliability*, 22(2):67–120.

Zhao, Y., Serebrenik, A., Zhou, Y., Filkov, V., and Vasilescu, B. (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)*, pages 60–71. IEEE.