


A learning algorithm for optimizing continuous integration development and testing practice

Dusica Marijan¹  | Arnaud Gotlieb¹ | Marius Liaaen²

¹Simula Research Laboratory, Lysaker, Norway

²Cisco Systems, Lysaker, Norway

Correspondence

Dusica Marijan, Simula Research Laboratory, PO Box 134, 1325 Lysaker, Norway.
Email: dusica@simula.no

Funding information

The Research Council of Norway, Grant/Award Number: Certus SFI

Summary

Continuous integration, at its core, includes a set of practices that aim to prevent and reduce the cost of software integration issues by merging working software copies often. Regression testing is considered a good practice in software development with continuous integration, which ensures that code changes are not negatively affecting software functionality. As, nowadays, software development is carried out iteratively, with small code increments continuously developed and regression tested, it is of critical importance that continuous regression testing is time efficient. However, in practice, regression testing is often long lasting and faces scalability problems as software grows larger or as software changes are made more frequently. One contributing factor to these issues is test redundancy, which causes the same software functionality being tested multiple times across a test suite. In large-scale software, especially highly configurable software, redundancy in continuous regression testing can significantly grow the size of test suites and negatively affect the cost effectiveness of continuous integration. This paper presents a practical learning algorithm for optimizing continuous integration testing by reducing ineffective test redundancy in regression suites. The novelty of the algorithm lies in learning and predicting the fault-detection effectiveness of continuous integration tests using historical test records and combining this information with coverage-based redundancy metrics. The goal is to identify ineffective redundancy, which is maximally reduced in the resulting regression test suite, thus reducing test time and improving the performance of continuous integration. We apply and evaluate the algorithm in two industrial projects of continuous integration. The results show that the proposed algorithm can improve the efficiency of continuous integration practice in terms of decreasing test execution time by 38% on average compared to the industry practice of our case study and by 40% on average compared to the retest-all approach. The results further demonstrate no significant reduction in fault-detection effectiveness of continuous regression testing. This suggests that the proposed algorithm contributes to the state of the practice in the continuous integration development and testing of highly configurable systems.

KEYWORDS

continuous integration, continuous integration testing, highly configurable software, highly interleaved test predictive algorithm, regression testing, regression trees, test optimization, test redundancy

1 | INTRODUCTION

Continuous integration (CI) practice promotes releasing software into production rapidly and reliably, for which an efficient and systematic software testing practice is needed. Automated regression testing makes part of such a practice, typically running frequently to ensure that changes made to the software in frequent code commits and bug fixing have not introduced new bugs. With changes continuously made to software, test suites for checking the correctness of software often grow in size, covering new functionality or new scenarios for the existing functionality. Since large test suites make it difficult to perform time-efficient regression testing in CI, methods have been proposed to alleviate the problem, such as test selection and prioritization^{1,2} or reduction³ (discussed later). However, one specific case challenges these existing methods in practice. That is the case of testing highly configurable software (HCS) in CI. HCS represents the class of software systems that contain many configuration parameters (features) for configuring the software functionality at design time or runtime. The unique challenge that exists in the testing of HCS stems from a large configuration space where features interact at various interaction levels for different software variants. Consequently, tests overlap in terms of feature coverage, which causes that specific features or feature combinations are disproportionately tested with respect to other features. Therefore, the unique challenge of HCS testing in CI is to effectively select and test the combinations of software features, ensuring high distinct configuration coverage while maximizing fault-detection effectiveness.

We have observed the described challenge of improving the effectiveness of HCS testing in practice, during a long-standing collaboration with Cisco Systems Norway. The project is on the CI and testing of video conferencing HCS (VCS). On the one hand, we have observed the difficulty of cost effectively selecting an adequate set of test cases that ensure high distinct configuration coverage, given the rising software complexity and size of evolving HCS. The test cases overlap in terms of feature coverage, which we refer to as *interleaved tests*. Interleaved tests cause *test redundancy*, which increases test effort and test maintenance costs, if not handled systematically. On the other hand, through the project, we have observed the need to improve the time efficiency of regression testing of industrial HCS in CI, given the nature of CI requiring short and dynamic test budgets.

A common approach to CI regression testing in practice is a retest-all (RA) approach, which runs all available test cases, or a selective RA approach, which runs all test cases covering the parts of software affected by changes. However, for real software systems, RA can lead to scalability problems, if systems are large and software modifications (triggering regression testing) are made frequently.⁴ To make continuous regression testing efficient, only the relevant minimal set of tests that adequately cover the changes should be selected and run. While the RA approach is easily automated, selective regression testing with test optimization and redundancy analysis in practice is, in some companies (eg, Cisco video conferencing motivating this work), a manual process done by developers or testers. This makes the process often unsystematic and liable to subjectivity. Because test suites grow in size over time, introducing redundancy, manually selecting an adequate nonredundant set of regression tests soon becomes inefficient and impractical. This is especially evident in CI development, where regression testing runs as part of a timeboxed development iteration restricted to a specific duration. As these iterations are short, less time is available for testing, making it difficult to select regression tests using a manual approach. Because regression test feedback needs to be provided rapidly, efficient regression test selection needs to be automated and further optimized for minimal test redundancy.

Related approaches to improving regression testing have been mainly focused on test minimization, without specific focus on redundancy analysis for highly interleaved tests in HCS in CI. We remind that the challenge of HCS testing lies in effectively handling a large configuration space (and, thus, a large test space) with feature overlap, causing inherent test redundancy. Moreover, existing regression testing approaches sometimes do not focus on the specific needs of CI environments, such as high time efficiency, which is especially important due to the extensive test suites typically found in testing industrial HCS. Unlike these approaches, we focus on reducing the redundancy of highly interleaved regression tests for HCS in CI. In our previous work,⁵ we proposed an algorithm for minimizing the test redundancy of interleaved tests in HCS using feature overlap across a test suite and historical fault-detection effectiveness of test cases. This algorithm helped classify regression test cases as unique, totally redundant, and partially redundant tests. In this work, we extend the algorithm by introducing a more accurate classification of partial redundancy. Specifically, we use regression trees to predict the fault-detection effectiveness of tests based on historical test execution records, which provide a more accurate classification of effective and ineffective partially redundant regression tests and, thus, more efficient regression test suites.

To evaluate our algorithm, we conducted an empirical study on a large industrial VCS and one industrial HCS mobile application, both developed in CI. We use historical test execution data in the evaluation in both cases. We compare the algorithm with current industry practice (CIP), with an advanced RA approach, with random test selection (RS), and with some existing test reduction and prioritization algorithms, in terms of fault-detection effectiveness, time efficiency, and

size efficiency. The results of our study show that our algorithm can improve the cost effectiveness of the CI regression testing of HCS compared to industry practice, the existing test reduction and prioritization algorithms, and often used practical strategies such as random and RA. To evaluate the performance of our prediction model, we use metrics such as precision, recall, accuracy, and the F-score, which demonstrated a good performance of the prediction model. Overall, the experimental results show that our proposed algorithm contributes to an improved continuous regression testing practice in terms of fault-detection effectiveness and time effectiveness.

Contributions. This paper makes the following contributions with respect to our previous work.⁵

1. We introduce a novel classification algorithm for effective and ineffective test cases in CI based on regression trees.
2. We perform the evaluation of the proposed classification algorithm using precision, recall, accuracy, and F-score metrics, demonstrating improvement in the performance of test redundancy reduction, compared to the previous work.
3. We provide new experimental results for the existing test data, showing the improved performance of the algorithm in terms of fault-detection effectiveness and time efficiency, due to the novel algorithmic improvements in predicting ineffective partially redundant test cases.
4. We perform a more comprehensive evaluation of the algorithm and the overall approach to test redundancy reduction by introducing a novel case study of continuous regression testing, which reduces the threat to the external validity of our results.
5. We provide the comparison of the algorithm with two state-of-the-art techniques for test prioritization² and reduction.³
6. We provide the evaluation of the algorithm in terms of gradual performance in fault-detection effectiveness for different time budgets.
7. We complement the results of time efficiency reported previously with the results of size efficiency (test suite size decrease).

In the remainder of this paper, we review the background work relevant for the understanding of our algorithmic approach in Section 2. We describe the industry case study of CI and the testing of industrial video conferencing software that motivated our work in Section 3. We give the problem statement in Section 4. We describe our solution to reducing ineffective test redundancy in practical continuous regression testing in Section 5. We present the experimental study performed to evaluate the algorithm in Section 6 and discuss the results in Section 7. We discuss threats to validity in Section 8 and review related work in Section 9. Finally, we give the conclusion and highlight further research in Section 10.

2 | BACKGROUND

This section gives an overview of the background concepts relevant for our work, such as CI, testing of HCS, test redundancy, regression test selection in HCS testing, and regression trees.

2.1 | Continuous integration

In CI development, teams work in short development cycles, continuously integrating working code copies to the central repository. As CI aims at minimizing software integration time, one key aspect of efficient CI is a testing process able to provide rapid feedback on software failures. To enable rapid test feedback in CI, test cycles are restricted to a specific (short) duration. We refer to this time duration of each test cycle as a *time budget*. Time budgets can vary from a cycle to a cycle, and they include time to select relevant tests for running, run the tests, and report test results to developers.

2.2 | Testing of HCS

HCS consists of a common code base and a set of configuration options (features) used for customizing main software functionality. For example, features in a configurable video communication software include video resolution or networking protocol type. We consider that a test case for the testing of HCS is an integration test aimed at checking the correctness of interaction between involved features. Given an HCS with a set of features $FS = \{f_1, f_2, \dots, f_n\}$ and a test suite $TS = \{T_1, T_2, \dots, T_m\}$ for the testing of HCS, we can define an association function $\text{Cov}: TS \rightarrow FS$, associating a

set of covering features to each test case. $\text{Cov}(T_i) = \{f_1, f_2, \dots, f_k\}, k \leq n$, represents a set of features tested by T_i . We assume that the relation between $T_i \in TS$ and $f_i \in FS$ is “many to many,” which means that a feature can be covered by multiple test cases, and a test case can cover multiple features. We further assume that $\forall T_i \in TS, \text{Cov}\{T_i\} \neq \emptyset$, ie, each test covers at least one feature.

Configurable software typically requires sophisticated testing approaches due to the large number of features found in realistic HCS, which normally causes redundancy in tests. The problem becomes especially evident in continuous regression testing, which runs frequently and is highly time constrained. Test feedback on pass/fail regression tests needs to be provided rapidly, which necessitates that regression test suites are optimized for less redundancy. A common approach for the testing of HCS has been combinatorial interaction testing,^{6,7} where tests are developed to cover combinations of features depending on target coverage criteria.⁸ However, existing approaches predominantly consider a fixed-strength interaction coverage (for example, pairwise), whereas we noticed that in practice, realistic systems often require testing the combinations of features with various degrees of interaction.

2.3 | Test case redundancy

Test redundancy is often defined in terms of coverage metrics, such that for a given coverage criterion (for example, pairwise feature coverage), if two tests T_i and T_j execute the same pair of features, one of the tests in a suite $TS = \{T_i, T_j\}$ contributes to test suite redundancy.

There are numerous causes of test redundancy, for example, test reuse in manual test specification, when existing tests are modified for testing new similar functionality, unintentionally leaving parts of already tested functionality. Other causes include incomplete-requirement specification, redundancy of requirements, legacy, static test suites, parallel testing, or distributed testing.⁹ In this work, we focus on the redundancy of integration tests, particularly redundancy that is introduced during test case design, where test cases are developed to test a varying number of feature interactions. Redundant combinations of feature interactions in a test suite cause the same functionality being executed in multiple instances.

Formally, if we use $FSet$ to denote a feature set covered by a test suite TS and $\text{Cov}(T)$ to denote the set of features covered by the test case $T \in TS$, then

1. a test case T is considered *redundant* in TS if $\text{Cov}(TS \setminus \{T\}) = FSet$, and
2. a test case T_i is considered *redundant* with respect to T_j if $\text{Cov}(T_i) \subseteq \text{Cov}(T_j)$.

2.4 | Regression test selection

Regression test selection is a technique used to improve the cost efficiency of software testing after a change has been made, by selecting an effective set of test cases that will check whether the change (eg, defect fix) introduced new faults. For a given set of changes $Ch = \{Ch_1, Ch_2, \dots, Ch_n\}$ made to a software system S and an existing test suite $TS = \{T_1, T_2, \dots, T_n\}$ used to test S , regression test selection determines $TS' \subseteq TS$, consisting of tests that are relevant for Ch , to be used for the testing of a modified system S' . Selection is performed according to the selection function g_{sel} that uses information about Ch_i and T_i to find those T_i that are relevant for Ch_i . Regression test selection may additionally use a set of objectives to optimize the execution of TS' . Some typical objectives include high fault detection, low execution time/cost, etc. Regression test selection may also use historical test execution data to find an optimal TS' based on past test performance. Although various approaches have been proposed for regression test selection, the challenge remains to efficiently identify and reduce ineffective test redundancy for highly interleaved tests.

2.5 | Regression trees for test classification

Regression trees¹⁰ are a variant of decision trees, models that can predict values of a target dependent variable, which is represented by leaves in a tree, based on the value of input independent variables. Dependent variables are called response variables, and independent variables are called predictors. In regression trees, the target variable is continuous. In our case of testing HCS, which consists of a large number of features, regression trees can be used to predict whether a test case will detect failures in execution based on its previous fault-detection effectiveness. In particular, the tree nodes represent software features. These features are building blocks for test cases. Traversing the tree from a node to a leaf constructs a set of features that represent a test case. Dependent variables represent the predicted fault-detection effectiveness of

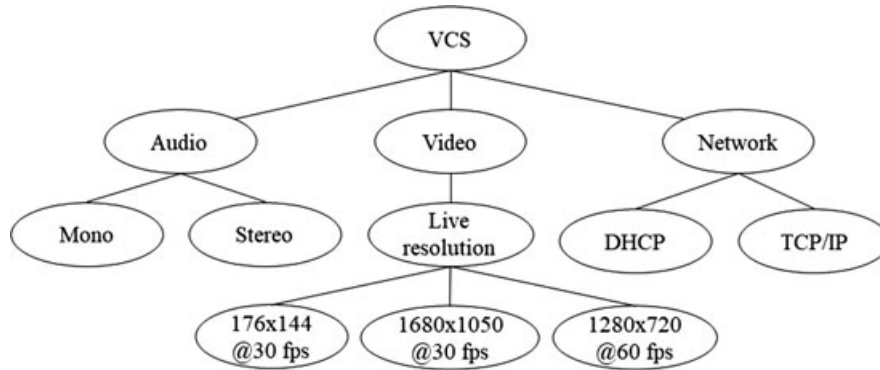


FIGURE 1 Simplified regression tree for the classification of effective and ineffective partially redundant test cases for highly configurable video conferencing software in continuous integration. DHCP, Dynamic Host Configuration Protocol; TCP/IP, Transmission Control Protocol/Internet Protocol; VCS, video conferencing highly configurable software

a test case, and independent variables represent the historical fault-detection effectiveness of a test case. Historical test execution data indicate whether features covered by a test case caused failures in previous test executions. A decision based on a predictor is represented by internal nodes in a tree, where each edge gives a next decision. Traversing the tree from a node to a leaf gives the prediction for the effectiveness of a test case, based on the values of predictors. Finally, the response variable is assigned a binary value, which means that a test case is effective or not in the coming runs. A simplified example of a regression tree is given in Figure 1.

A variety of prediction models exist, such as support vector machines, random forests, and neural networks, to name a few. However, the reason why we chose regression trees is twofold. First, a graphical representation of regression trees resembles feature models¹¹ commonly used to represent the variability in HCS.¹² Engineers in our industrial case study, which motivates this work, were already familiar with feature models; therefore, they considered decision trees simple to understand and interpret. Since the proposed work is aimed at improving industry practice, we believe that choosing the right technology plays an important role in the success of adopting the solution by industry. Second, according to the studies of Wu et al¹³ and Li et al,¹⁴ classification and regression trees are considered among the top 10 algorithms in data mining. They are also successfully used for predicting software quality.¹⁵

3 | INDUSTRIAL CASE STUDY

We present a case study describing the industrial practice of CI testing of highly configurable VCS at Cisco. This case study motivated our research, and the proposed solution is aimed at improving the existing practice in this domain. In this section, we describe one class of video conferencing systems, ie, C90 codec. We describe the CIP of testing C90. Finally, we introduce notations and assumptions used throughout this paper.

Cisco VCS is developed as a product line, with the core conferencing functionality that is common to all product variants and a set of features that can be used to configure the variants according to user requirements. C90 is an HCS consisting of a wide range of features, including multisite features, audio and video features, and security features, as illustrated in Figure 2, which makes the testing of C90 complex. In the CIP of testing C90, a QA engineer selects and combines features creating valid test cases, which are continuously executed as part of CI. This gives the motivation to define a set of tests such that the test execution time is as minimized as possible.

Consider a video conferencing system *CBC* (Cisco Business Conferencing) that consists of a set of features referred to as $FS = \{f_1, f_2, \dots, f_n\}$. FS are used to build a set of products $CBCProd = \{P_1, \dots, P_z\}$, which are software solutions representing various configurations of desktop or boardroom conferencing. There is a test suite $TS = \{T_1, T_2, \dots, T_m\}$ used for the testing of *CBC*.

$Cov(T_i) = \{f_1, f_2, \dots, f_k\}, k \leq n$ denotes a set of features tested by T_i . When TS is developed, association tags are established between pairs T_i/f_i , denoting which tests are relevant for which features. Building association tags between tests and features is a manual process, because a similar process of building tags exists for another purpose in the software development stage of VCS. Therefore, when implementing the proposed approach, we inherited the tagging system for

Multisite features

- 4-way 1080p30 High Definition SIP/H.323 MultiSite
- Full individual audio and video transcoding
- Individual layouts in MultiSite CP (takes out self view)
- H.323/SIP/VoIP in the same conference
- Support for Presentation (H.239/BFCP)
- Best Impression (Automatic CP Layouts)
- H.264, Encryption, Dual Stream from any site
- IP Downspeeding
- Dial in/Dial out
- Additional telephone call (no license required)
- Conference rates up to 10 Mbps

Audio features

- CD-Quality 20KHz Mono and Stereo
- Eight separate acoustic echo canceller
- 8-port Audio mixer
- Automatic Gain Control (AGC)
- Automatic Noise Reduction
- Active lip synchronization

Audio standards

- G.711, G.722, G.722.1, 64 kbps & 128 kbps MPEG4 AAC-LD, AAC-LD Stereo



Security features

- Management via HTTPS and SSH
- IP Administration Password
- Menu Administration Password
- Disable IP services
- Network Settings protection

Bandwidth

- H.323/SIP up to 6 Mbps point-to-point
- Up to 10 Mbps total MultiSite bandwidth

Protocols

- H.323
- SIP

IP network features

- DNS lookup for service configuration
- Differentiated Services (QoS)
- IP adaptive bandwidth management (in)
- Auto gatekeeper discovery
- Dynamic playout and lip-sync buffering
- H.245 DTMF tones in H.323
- Date and Time support via NTP
- Packet Loss based Downspeeding
- URI Dialing
- TCP/IP
- DHCP
- 802.1x Network authentication
- ClearPath



Video features

- Native 16:9 Widescreen
- Advanced Screen Layouts
- Intelligent Video Management
- Local Auto Layout
- 9 embedded individual video compositors

Live video res.

- 176 x 144@30 fps (QCIF)
- 352 x 288@30 fps (CIF)
- 512 x 288@30 fps (w288p)
- 576 x 448@30 fps (448p)
- 768 x 448@30 fps (w448p)
- 704 x 576@30 fps (4CIF)
- 1024 x 576@30 fps (w576p)
- 1280 x 720@30 fps (720p30)
- 1920 x 1080@30 fps (1080p30)
- 640 x 480@30 fps (VGA)
- 800 x 600@30 fps (SVGA)
- 1024 x 768@30 fps (XGA)
- 1280 x 1024@30 fps (SXGA)
- 1280 x 768@30 fps (WXGA)
- 1440 x 900@30 fps (WXGA+)
- 1680 x 1050@30 fps (WSXGA+)
- 1600 x 1200@30 fps (UXGA)
- 1920 x 1200@25 fps (WUXGA)
- 512 x 288@60 fps (w288p60)
- 768 x 448@60 fps (w448p60)
- 1024 x 576@60 fps (w576p60)
- 1280 x 720@60 fps (720p60)
- 720p30 from 768kbps
- 720p60 from 1152kbps
- 1080p30 from 1472kbps

FIGURE 2 Highly configurable Cisco video conferencing system C90 [Colour figure can be viewed at wileyonlinelibrary.com]

the purpose of test reduction. When software changes, for example, features are modified or new features are added, developers can see which tests are affected by changes, through tags. Then, developers can change the affected tests and the mapping between features and tests accordingly. In this industrial practice of testing highly configurable conferencing systems at Cisco, feature changes that require test modifications do not happen frequently, and therefore, updating the association tags and tests manually does not create scalability problems.

Each test $T_i \in TS$ is associated with a set of historical records $Fde(T_i) = \{Fde_{T_{i1}}, Fde_{T_{i2}}, \dots, Fde_{T_{ij}}\}$ that correspond to the execution statuses of j past test case executions (fail/pass status). $Fde_{T_{ij}} = \{status_{ij}, C_{fail}(Fde_{T_{ij}})\}$ is associated with one or more configurations responsible for failure $C_{fail}(Fde_{T_{ij}}) = \{C_1, C_2, \dots, C_m\}$, where $C_i = \{f_1, f_2, \dots, f_k\}$.

CBC is developed and evolves incrementally and iteratively, following a CI practice, where much of the functionality is shared between different products. *TS* is developed to test all *CBCProd*, and because of shared functionality between products, the same parts of *CBC* are executed multiple times. These conditions are illustrated in Figure 3. Five products in the Figure consist of different functionality modules $FS = \{A, B, C, D, E, \dots, N\}$. In the context of *CBC*, features include an audio protocol or a video resolution of a conferencing call, or network type, and they are reused across *CBC*. Tests in *TS* are highly interleaved and specified with a varying degree of feature coverage, such as single-feature coverage, ie,

$$\text{Cov}(T5) = \{I\}, \text{Cov}(T6) = \{N\}, \text{Cov}(T7) = \{N\},$$

or multiple-feature coverage, ie,

$$\text{Cov}(T1) = \{B, D\}, \text{Cov}(T2) = \{C, D\}, \text{Cov}(T3) = \{A, K\}, \text{Cov}(T4) = \{A, K, L, M\}.$$

TS evolves and grows, accumulating tests with different coverage criteria, which, over time, increases the risk of redundant testing. When using a single-feature coverage criterion, tests $T1$ and $T2$ overlap, as $\text{Cov}(T1) = \{B, D\}$ and $\text{Cov}(T2) = \{C, D\}$, contributing to *partial redundancy* in *TS*. We say that $T1$ is partially redundant with respect to $T2$, and vice versa. Assuming that $TS = \{T1, T2\}$, eliminating either $T1$ or $T2$ would leave a feature in *CBC* (B or C) untested. However, in a test suite containing multiple instances of partial redundancy for the same set of features, partially redundant tests can be eliminated. Contrarily, $\{T3, T4\}$ contributes to *total redundancy* in *TS*, as $\text{Cov}(T3) = \{A, K\}$ is a proper subset of $\text{Cov}(T4) = \{A, K, L, M\}$. We say that $T3$ is totally redundant with respect to $T4$.

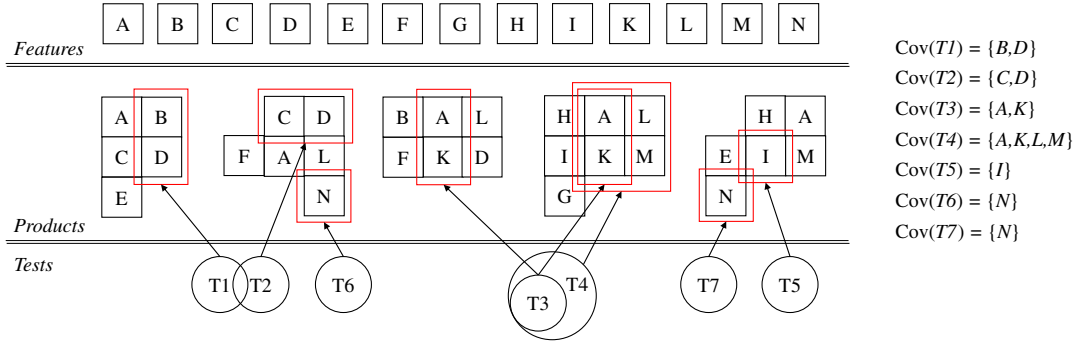


FIGURE 3 Test redundancy in highly configurable software: test cases T1-T7 cover a set of features A, ..., N in different combinations across different products, causing partial or total test overlap [Colour figure can be viewed at wileyonlinelibrary.com]

In the current industry testing practice, CIP, whenever *CBC* is modified, introducing feature changes $Ch = \{Ch_1, Ch_2, \dots, Ch_n\}$, affected products in *CBCProd* are regression tested to ensure that new modifications do not negatively affect existing functionality. Regression tests are selected mainly manually from *TS* based on change impact analysis and intuitive assessment of test relevance for a given Ch_i . For example, when preparing a test suite for running, developers can see the information about the tests affected by the given changes. After that, developers manually select a subset of the automatically selected test suite, based on their assessment of the test relevance/importance for the given run. The motivation for this approach lies in the need to reduce total test execution time as much as possible. To be able to benefit from CI, test feedback needs to be produced as quickly as possible. One way to improve the agility of a test cycle is to automate the process of regression test selection (time savings compared to manual selection). Another way is to reduce the size of a regression test suite by automatically identifying and removing redundancy (time savings in test execution). In this paper, we focus on the latter, enabling a more cost-effective regression testing of HCS in CI, by identifying and eliminating redundancy caused by overlapped feature combinations across tests.

4 | PROBLEM STATEMENT

Let us consider the target test coverage to be a single-feature coverage. Then, a test $T_i \in TS$ can be considered.

Definition 1. T_i is totally redundant of *TS*, iff $\exists T_j \in TS, i \neq j. Cov(T_i) \subseteq Cov(T_j)$.

Definition 2. T_i is partially redundant of *TS*, iff $\exists T_j \in TS, i \neq j. Cov(T_i) \neq Cov(T_j)$ and $Cov(T_i) \cap Cov(T_j) \neq \emptyset$.

Definition 3. T_i is unique of *TS*, iff $\neg \exists T_j \in TS, i \neq j. Cov(T_j) \cap Cov(T_i) \neq \emptyset$.

From the example given in Figure 3, T3 is totally redundant of T4 because all feature combinations covered by T3 are also covered by T4. T5 is unique, because it uniquely covers feature I. T1 and T2 are mutually partially redundant, with respect to single-feature coverage, as they both cover feature D, but also other different features.

As observed previously,¹⁶ using coverage-based metrics in test reduction solely does not ensure the increased fault-detection effectiveness of the resulting reduced test suite. However, supplementing coverage metrics with the information about past fault-detection effectiveness of tests can help achieve this goal. We refer to this past fault-detection effectiveness information of a test suite as *test history Fde*. Test history can be collected during test suite execution, and it typically includes data about pass/fail/inconclusive test results, which can be used for fault localization and debugging purposes. Furthermore, assuming a fixed feature-combination coverage as a target test coverage criterion is allegedly an oversimplification for a vast majority of realistic and complex HCS systems. Features are typically integrated within different products in a varying degree of interaction. Thus, effective regression testing needs to be able to detect the cases when the same functionality (features) is executed as part of different feature combinations across different tests and products.

On the basis of the presented concepts, we formulate the problem of test redundancy reduction for highly interleaved tests in CI as follows.

For a given feature change Ch , existing test suite TS , fault-detection records of each test based on test history Fde of TS , and a time budget Tb , select $TS' \subseteq TS$ as a regression test suite with the following properties.

Property 1. $\forall T \in TS'$. T includes features affected by Ch .

Property 2. $TS' \subset (TS \setminus \{\text{totally redundant tests}\})$. TS' excludes any totally redundant test in TS .

Property 3. $TS' \cap TS \supset \{\text{unique tests}\}$. TS' includes all unique tests in TS .

Property 4. $TS' \subset (TS \setminus \{\text{ineffective tests}\})$, where *ineffective* tests are tests in TS that are determined, by the test history Fde and the time budget Tb , as tests not contributing to the increased fault detection effectiveness of TS' , according to the process described in Section 5.

The major challenge in solving the above formulated problem lies in identifying *ineffective* test cases. The following section provides an explanation of *ineffective* test case, ie, when a partially redundant test case is considered *effective* and when it is considered *ineffective*.

5 | REDUNDANCY DETECTION AND REDUCTION

Given Definition 2 of the partially redundant test case, for $\{T_i, T_j \in TS | i \neq j, \text{Cov}(T_i) \neq \text{Cov}(T_j), \text{Cov}(T_i) \cap \text{Cov}(T_j) \neq \emptyset\}$, we want to determine if there exists $\{T_k \in TS | k \neq i, k \neq j, \text{Cov}(T_i) \setminus \text{Cov}(T_j) \subset \text{Cov}(T_k)\}$. Then, the following statements hold.

1. If such T_k exists, T_i is an ineffective partially redundant test that can be eliminated from TS without decreasing the fault-detection effectiveness of TS , as $\text{Cov}(T_j) \cap \text{Cov}(T_k) \supseteq \text{Cov}(T_i)$.
2. If no such T_k exists, T_i is a partially redundant test that executes a feature set not covered by any other test in TS .

This feature set could be a single feature or a combination of features. Since our work relates an HCS with a high degree of feature interaction, together with a test suite developed over time to cover these various interactions, it is expected that combinations with varying (high) degree of feature interactions will usually exist in a test suite.

The key step in our algorithm for optimizing CI testing by reducing redundancy in regression test suites is distinguishing between effective and ineffective partially redundant test cases. To determine whether T_i is an *effective* or an *ineffective* partially redundant test case, we make the following practical hypothesis: test cases covering combinations that have historically exhibited good fault-revealing performance can be classified as *effective*, and otherwise as *ineffective*. The hypothesis is based on the fact, suggested by previous studies,^{2,4,17,18} that test execution history can help improve the effectiveness of regression testing. Therefore, to optimally detect and minimize test redundancy, we combine coverage analysis with the information indicating the fault-detection capability of tests exhibited in past test executions for specific configurations.

In particular, $Fde(T_i) = \{Fde_T_{i1}, Fde_T_{i2}, \dots, Fde_T_{ij}\}$ denotes execution history for T_i over j test runs, where each failure is associated with one or more failing configurations $C_{\text{fail}}(Fde_T_{ij}) = \{C_1, C_2, \dots, C_m\}$, where $C_i = \{f_1, f_2, \dots, f_n\}$ is a set of features. Each feature is associated a *fault_index*, which denotes its historical fault proneness (the rate of being the part of failing test cases). In principle, when a test case fails, all features covered by the failed test case are assigned fault indexes. However, features are assigned fault indexes only when they contribute to distinct failures; otherwise, those features would become overemphasized. By distinct failures we mean the following: if a test case Ta detected a failure that has not been detected before, all features covered by Ta are assigned fault indexes. Then, if a test case Tb detected the same failure, features that are only covered by Tb and not by Ta are assigned fault indexes. If a single feature led to the failure, the fault index is 1, and if the failure occurred in a feature interaction, the fault index is assigned a value of $1/n$, where n is the number of the features contributing to the failure.

On the basis of fault indexes, we build statistical regression models that predict whether a test case T_i is likely to detect a failure in the upcoming execution. We use the C4.5 algorithm for building regression trees.¹⁹ The regression model predicts a value in the range of $[0, 1]$, where the values in the range of $[0, 0.5)$ mean that T_i is ineffective (not likely to detect a failure), and the values in the range of $[0.5, 1]$ mean that T_i is effective (likely to detect a failure). A simplified regression tree for highly configurable video conferencing software in CI is illustrated in Figure 1. Nodes of the tree represent software features. Features are building blocks for test cases. Therefore, a traversal from the root node to a leaf represents a test case.

5.1 | The algorithm

The following algorithm describes the overall approach to optimizing CI development and testing practice by reducing test redundancy based on coverage and history analysis.

The algorithm uses as input the existing test suite TS and feature changes Ch_i and produces as output a regression test suite that satisfies properties $P1$ - $P4$ defined above.

Algorithm 1. The algorithm

Input: The existing test suite TS

Feature changes Ch_i

Output: Regression test suite that satisfies the properties $P1$ - $P4$

Step 1:

Given the input, identify RTS' as a set of tests affected by Ch_i , based on the associations between features covered by a test and software modules implementing these features, similarly to the associations between features and test cases.

RTS' represents an initial regression test suite that needs to be processed further to satisfy the properties $P2$, $P3$, and $P4$.

Step 2:

Analyze total redundancy in RTS' to satisfy the Property $P2$.

Identify tests whose covering set of features is entirely covered by another test $Cov(T_i) \subseteq RTS'$ (Figure 4A) and remove such tests from RTS' .

Step 3:

Look for tests in RTS' that cover features not covered by other tests in RTS' (Figure 4B), extract these tests from RTS' , and construct RTS , satisfying the Property $P3$.

Step 4:

At this point, RTS' consists of partially redundant tests (Figure 4C).

To partition effective and ineffective tests, for each $T_i \in RTS'$, obtain the execution history $Fde(T_i) = \{Fde_{T_{i1}}, Fde_{T_{i2}}, \dots, Fde_{T_{ij}}\}$, with $j = 6$. This value has been experimentally evaluated as the optimal size of history window for test optimization for HCS.¹⁸ Longer windows capturing more of older failure information showed to provide less accurate indication of potentially failing test cases.

Build a regression tree that predicts the likelihood of a test case to detect failures in upcoming test executions, as a numerical value in the range $[0,1]$.

After obtaining predictions for fault-detection effectiveness of all partially redundant tests in RTS' , sort RTS' such that tests with higher likelihood of detecting failures come first in a sequence.

If one or more test cases are assigned equal weights, check their failure rate $Fr = total_num_of_failures / total_num_of_executions$ and give higher priority to those tests that have failed more frequently.

If one or more of these test cases have the same failure rate, assign them different weights at random.

Step 5:

Append to RTS the top n test cases from RTS' (with the highest priority) to fill the available time budget for RTS . The time budget is defined for every CI test run. n is determined by adding up test execution times of tests from RTS and RTS' as obtained from their historical execution records.

At this point, RTS is the selected regression test suite that satisfies all defined properties $P1$ - $P4$, as the solution of coverage- and history-based test redundancy reduction problem.

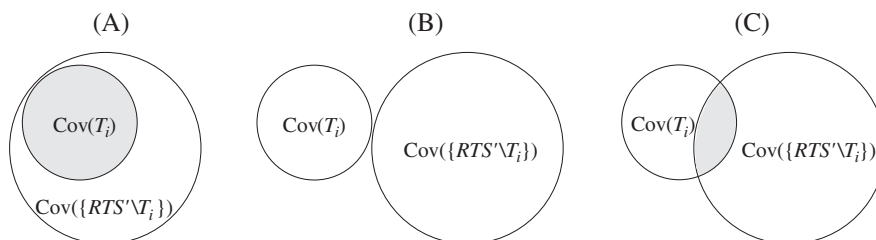


FIGURE 4 Test suite properties. A, T_i totally redundant; B, T_i unique; C, T_i partially redundant

6 | EMPIRICAL EVALUATION

To evaluate the proposed algorithm for optimizing CI testing practice named Redundancy and History based Selection (RHS), we performed extensive experiments using real-world test sets of two HCS systems developed and tested in CI.

In the evaluation, we measure the following properties: (1) *fault-detection effectiveness*, (2) *time efficiency*, and (3) *size efficiency* of the reduced regression test suites in comparison with (i) the *existing industry practice* of testing HCS in CI, (ii) the *RA approach*, and (iii) *RS*. As the proposed algorithm combines coverage-based test reduction for lower test redundancy and history-based test prioritization for increased fault-detection effectiveness, we compared it with two existing algorithms for test reduction³ and test prioritization.² To evaluate the performance of the classification model for predicting ineffective redundant test cases specifically, we use the following metrics: (a) *precision*, (b) *recall*, (c) *accuracy*, and (d) *F-score*.

The four experiments E1-E4 address the following research questions respectively.

- RQ1** What is the effect of test suite reduction using the proposed algorithm on the time efficiency of a test suite, compared to (a) the industry practice of testing HCS in CI, (b) the RA approach, and (c) some existing algorithms for test reduction and prioritization?
- RQ2** What is the effect of test suite reduction using the proposed algorithm on the size efficiency of a test suite, compared to (a) the industry practice of testing HCS in CI, (b) the RA approach, and (c) some existing algorithms for test reduction and prioritization?
- RQ3** What is the effect of test suite reduction using the proposed algorithm on the fault-detection effectiveness of a test suite, compared to (a) the industry practice of testing HCS in CI; (b) the RA approach; (c) RS, when controlling for the total test suite execution time; and (d) some existing algorithms for test reduction and prioritization?
- RQ4** What is the performance of the classification model for predicting ineffective partially redundant test cases?

The remainder of this section describes the test sets in more detail, presents the experiment methodology and measures, and finally presents and discusses the results of the experiments.

6.1 | Experiment test sets

We evaluated the proposed algorithm using real test suites of two industrial HCS from a video conferencing domain and a mobile application domain.

6.1.1 | Industrial test sets for video conferencing software

One industrial experiment subject is the video conferencing software system *CBC* introduced in Section 3. *CBC* contains a test suite *BCTS* consisting of 460 test cases, which cover the set of 75 features. Each test case contains historical execution records for the last 10 runs. The historical information includes test result status, execution duration, required resources, test identifiers, tested software version, and execution schedulers. In total, the execution history contains results of 4600 executions (460 tests \times 10 consecutive executions per test). Moreover, 19% of the test cases in the test suite are unique, and 22% are totally redundant. A minimum degree of feature interaction for tests in *BCTS* varies from 1 (single feature) to 3 (combination of three features), and a maximum degree of feature interaction varies from 4 to 7. *CBC* is tested according to the current industry testing practice, CIP, described in Section 3.

6.1.2 | Industrial test sets for mobile software

Another industrial experiment subject is a cross-platform mobile application software with a test suite *MATS* containing 500 test cases. The mobile software is highly configurable, to enable adaptation to a wide range of mobile platform variants with different screen sizes, resolutions, communication protocols, and CPUs. Test history is available for the test suite for the last 10 test executions, and it includes test result status, execution duration, required resources, and test identifiers. Test history contains 5000 execution records (500 tests \times 10 consecutive executions per test). Moreover, 24% of the test cases in the test suite are unique, and 28% are totally redundant. A minimum degree of feature interaction for tests in *MATS* varies from 2 (combination of two features) to 3 (combination of three features), and a maximum degree of feature interaction varies from 4 to 9.

Table 1 provides summary information about the industrial test suites.

TABLE 1 Industrial experiment subjects BCTS and MATS

	Video Conferencing Suite (BCTS)	Mobile Application Suite (MATS)
Number of test cases	460	500
*Percentage of unique test cases	19	24
*Percentage of totally redundant test cases	22	28
Number of test history records	4600	5000
Min feature interaction coverage	1-3	2-3
Max feature interaction coverage	4-7	4-9

6.2 | Measures and methodology

Measures. We evaluate the proposed algorithm using three metrics, namely, time efficiency, size efficiency, and fault-detection effectiveness, which are, in our context, defined as follows.

1. **Time efficiency** of a reduced test suite is the reduction of the execution time of the test suite compared to the execution time of its original (nonreduced) test suite.
2. **Size efficiency** of a reduced test suite is the reduction of the number of test cases of the test suite compared to the number of test cases of its original (nonreduced) test suite.
3. **Fault-detection effectiveness** of a reduced test suite is the ratio of the number of nonrepeated faults detected by the reduced test suite and the number of nonrepeated faults detected by its original (nonreduced) test suite. The closer the value to 1, the better the fault-detection effectiveness of a test suite. Nonrepeated faults are faults counted only once regardless of the number of test cases that detect that fault.

Methodology. At the first phase, we compare the proposed algorithm (RHS) with the industry testing practice (CIP). We retrieve modified features from history and automatically obtain a set of tests from *BCTS* affected by the modifications. The selection is based on the existing association tags between test cases and their covering features. The resulting test suite represents the initial regression test suite, which we refer to as *BCTS_Auto*. At the same time, we obtain a test suite that was selected, according to the previously described CIP practice, from *BCTS_Auto* by test engineers manually (for the given changes), which we refer to as *BCTS_Auto_Plus*. Then, we apply the RHS algorithm to *BCTS_Auto*, and using six historical execution records of *BCTS*, we analyze which feature combinations showed good fault-detection performance in the past. On the basis of this information, we select unique and effective partially redundant tests from *BCTS_Auto*, creating the final regression test suite *BCTS_RHS*. As we have available test execution history for 10 consecutive runs for *BCTS* and our algorithm uses the history window of size 6 (explained previously), we run the experiment five times. We start from the oldest six historical execution records and, in every experiment, replace the oldest record with a newer one, until we have used all the available records. This process produces five test suites *BCTS1-BCTS5*.

In the next phase of the experiment, we compare with the RA approach. We compare with RA because RA is a common approach to regression testing used in practice, instead of the tedious manual regression test selection/reduction. However, to allow for a fairer comparison, we use a modified retest-all (MRA) approach, which subsets the original test suite that would normally be executed by RA such that only the tests affected by a change are selected as regression tests. This was implemented using the association tags between features covered by a test and software modules implementing these features, similarly to the associations existing between test cases and their covering features. In this case, test suites *BCTS_Auto* obtained in the previous phase of the experiment are actually test suites produced by the MRA approach, which we refer to as *BCTS_MRA* in the comparison. We run all experiments for both test suites *BCTS* and *MATS*.

In the next phase of the experiment, we compare with the existing test prioritization algorithm ROCKET² and test reduction algorithm FLOWER.³ ROCKET is a test prioritization algorithm that orders test cases based on historical failure data, test execution time, and domain-specific heuristics. FLOWER is a test reduction algorithm that uses a test suite and the requirements covered by the suite to form a constrained flow network, which is traversed to find its maximum flows representing reduced test suites. Since ROCKET is a test prioritization algorithm that can produce prioritized test suites for a given time budget, after obtaining reduced test suites *BCTS_RHS* and *MATS_RHS*, we measure their total execution time and specify this parameter as input for ROCKET. This allows for a fairer comparison of RHS and ROCKET. We run ROCKET and FLOWER on both test suites *BCTS* and *MATS* and obtain reduced test suites *BCTS_ROCKET* and *BCTS_FLOWER* and *MATS_ROCKET* and *MATS_FLOWER*, respectively. We compare with these specific approaches because they are blackbox approaches not requiring access to source code. Developing a test reduction approach that

		Actual	
		Effective	Ineffective
Predicted	Effective	True Positives (TP)	False Positives (FP)
	Ineffective	False Negatives (FN)	True Negatives (TN)

FIGURE 5 Confusion matrix for assessing the quality of predictions of the effectiveness of partially redundant test cases

fulfills this requirement was imposed by our industry partners. Both these approaches have previously been applied to testing HCS.

Specifically, to answer RQ3, we compare the proposed algorithm with RS in terms of fault-detection effectiveness, while controlling for the total execution time of the regression test suite. The motivation to compare with RS comes from the observation that random selection is sometimes used as an alternative to automated regression test reduction, to reduce the cost of regression testing in cases where running the whole regression suite selected after code modifications would exceed time budget. First, we obtain *BCTS_RHS* as explained in the previous phase and measure its total test execution time $ET(BCTS_RHS)$. Then, we start randomly selecting tests from *BCTS*, and when each test is selected, we accumulate its execution time in $ET(BCTS_RS)$. We continue randomly selecting tests until $ET(BCTS_RS)$ is equal to $ET(BCTS_RHS)$. We refer to the resulting test suites as *BCTS_RS*. Then, for *BCTS_RS*, we measure the loss of fault detection compared to the fault-detection capability of *BCTS_RHS*. The loss is measured as the percentage of failures that were caused by configurations not covered by test cases in *BCTS_RS* and covered by *BCTS_RHS*. Finally, we compare RHS and RS in terms of the loss of fault-detection effectiveness. We have available test execution history for 10 consecutive runs for *BCTS*, and as our algorithm uses the history window of size 6 (explained previously), we run the experiment five times. We start from the oldest six historical execution records and, in every experiment, replace the oldest record with a newer one, until we have used all the available records. To account for randomness in RS, we repeat each experiment 100 times and determine the statistical significance of the results in nonparametric Mann-Whitney U tests, with the significance level of 0.01. We run the experiment also for the *MATS* test suite.

To answer RQ1-RQ3, we measure the percentage reduction of test suite execution time, percentage reduction of test suite size, and percentage reduction of fault-detection effectiveness of *BCTS_RHS* and *MATS_RHS* compared with *BCTS_Auto_Plus* and *MATS_Auto_Plus*, *BCTS_MRA* and *MATS_MRA*, *BCTS_ROCKET* and *MATS_ROCKET*, and *BCTS_FLOWER* and *MATS_FLOWER* and percentage reduction of fault-detection effectiveness of *BCTS_RHS* and *MATS_RHS* compared with *BCTS_RS* and *MATS_RS*.

The fourth experiment E4 addresses RQ4 by measuring the quality of predictions of the effectiveness of partially redundant test cases produced by our regression model. The quality is evaluated using a confusion matrix shown in Figure 5. *True positives (TP)* mean that the model predicted a test case as effective, and it actually is effective. *False positives (FP)* mean that the model predicted a test case as effective, while it actually is ineffective. *False negatives (FN)* mean that the model predicted a test case as ineffective, while it actually is effective. *True negatives (TN)* mean that the model predicted a test case as ineffective, and it actually is ineffective. We used the following four measures in the evaluation.²⁰

1. **Precision** is the ratio of the number of test cases correctly predicted as effective to the total number of test cases predicted as effective. Precision = 1 means that every test case that was predicted to be effective actually is effective.

$$\text{Precision} = \frac{TP}{TP + FP}$$

2. **Recall** is the ratio of the number of test cases correctly predicted as effective to the total number of test cases that are actually effective. Recall = 1 means that every test case that is effective was also predicted to be effective.

$$\text{Recall} = \frac{TP}{TP + FN}$$

3. **Accuracy** is the ratio of the number of correct predictions to the total number of test cases. Accuracy = 1 means that the classification model did not make any mistakes.

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

4. **F-score** is defined as the harmonic mean of *Precision* and *Recall*.

$$\text{F-score} = \frac{2 * \text{Precision} * \text{Recall}}{\text{Precision} + \text{Recall}}$$

In E4, we use the industrial test suites *BCTS* and *MATS* with fivefold cross-validation to evaluate the performance of prediction in the following manner. Each test suite is first randomly divided into five sets of tests. Then, four sets are selected for model training, and the fifth set is used for testing. The process is repeated five times, and each time, a different set is selected for testing. This whole process is run 10 times to account for randomness.

7 | RESULTS AND ANALYSIS

This section contains the results of the experiments addressing research questions RQ1, RQ2, RQ3, and RQ4, respectively.

7.1 | Time efficiency

In experiment E1, we evaluate what effect does test suite reduction have on the time efficiency of the reduced test suite, in comparison with CIP, RA, and FLOWER. The results show that RHS had improvements over CIP by 32% on average for *BCTS* and 38% on average for *MATS*. The results are shown in Figure 6. The x-axis represents five experiments corresponding to different historical execution data, whereas the y-axis denotes the percentage reduction of test suite execution time. Red bars showing negative values represent the analysis time required to run the proposed test reduction algorithm, as the percentage of the test suite execution time. The analysis time showed to be 3% on average (in the range of 2%-4% of the test suite execution time). The analysis time is shown as a negative value on the y-axis, because it increases the overall time required to produce and run the reduced test suite. Compared to MRA, the results show that RHS can reduce the

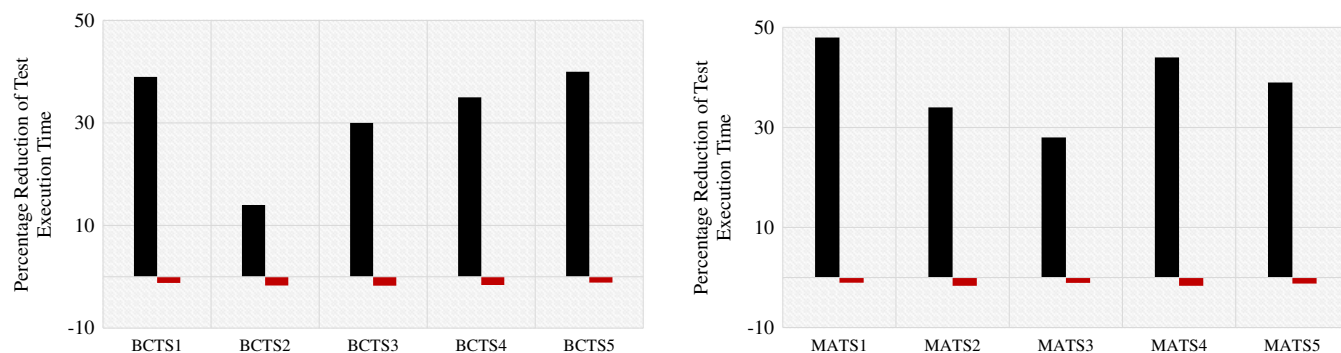


FIGURE 6 (Left) Percentage reduction of test execution time for RHS compared to current industry practice (CIP) for video conferencing software. (Right) Percentage reduction of test execution time for RHS compared to CIP for mobile application software [Colour figure can be viewed at wileyonlinelibrary.com]

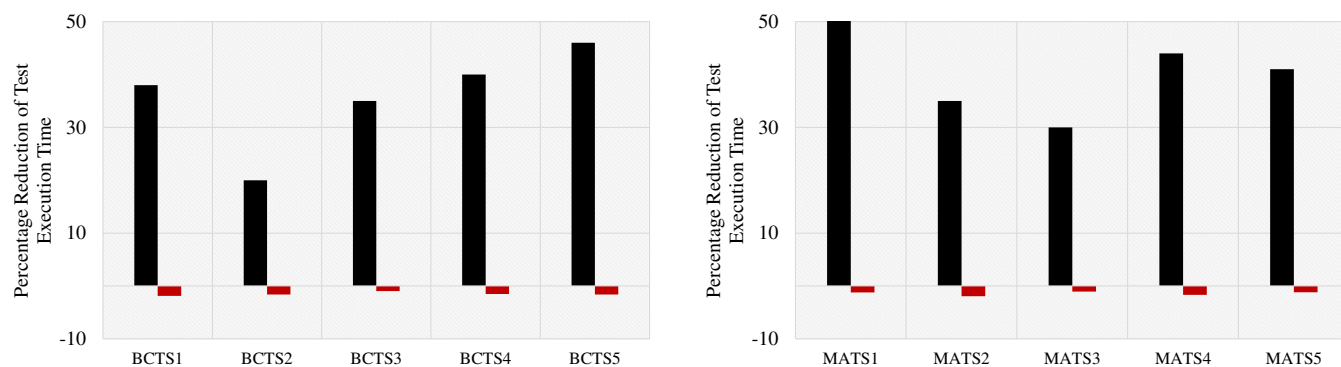


FIGURE 7 (Left) Percentage reduction of test execution time for RHS compared to the modified retest-all (MRA) approach for video conferencing software. (Right) Percentage reduction of test execution time for RHS compared to the MRA approach for mobile application software [Colour figure can be viewed at wileyonlinelibrary.com]

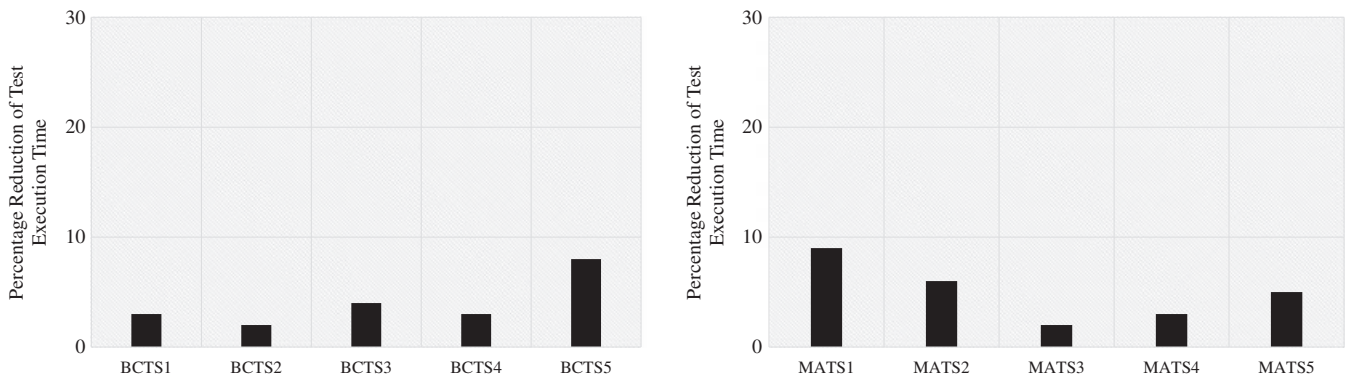


FIGURE 8 (Left) Percentage reduction of test execution time for RHS compared to FLOWER for video conferencing software. (Right) Percentage reduction of test execution time for RHS compared to FLOWER for mobile application software

total regression test execution time by 36% on average for *BCTS*. In the best case, using RHS led to reducing the execution time by 46% and, in the worst case, by 20%. For *MATS*, compared to the MRA approach, the results show an improvement of 40% on average. The results are shown in Figure 7. Compared to FLOWER, RHS showed to have reduced the test execution time by 4% on average for *BCTS* and 5% on average for *MATS*. The results are shown in Figure 8.

In summary, the results show that the proposed algorithm effectively reduces test execution time compared to industry practice, RA, and FLOWER.

7.2 | Size efficiency

In experiment E2, we measure what effect does test suite reduction have on the size efficiency of the reduced test suite, in comparison with CIP, RA, FLOWER, and ROCKET. The results indicate that RHS obtained test suites smaller by 27% on average compared to CIP for *BCTS* and smaller by 35% for *MATS*. The results are shown in Figure 9. The x-axis represents five experiments corresponding to different historical execution data, whereas the y-axis denotes the percentage reduction of test suite size. The results further show that RHS can reduce test suite size by 31% on average compared to the MRA approach for *BCTS* and by 37% on average for *MATS*. The results are shown in Figure 10. In comparison with FLOWER, RHS showed to have reduced test suite size by 3% on average for *BCTS* and 3% on average for *MATS*. The results are shown in Figure 11. In comparison with ROCKET, RHS increased test suite size by 7% on average for *BCTS* and by 6% on average for *MATS*. This is because ROCKET orders higher the test cases that have a shorter execution time. This means that in a given time budget, a ROCKET-prioritized test suite will have a higher number of faster (in execution) test cases compared to RHS. The results are shown in Figure 12.

In summary, the results show that the proposed algorithm reduces test suite size compared to industry practice, the RA approach, and the FLOWER algorithm.

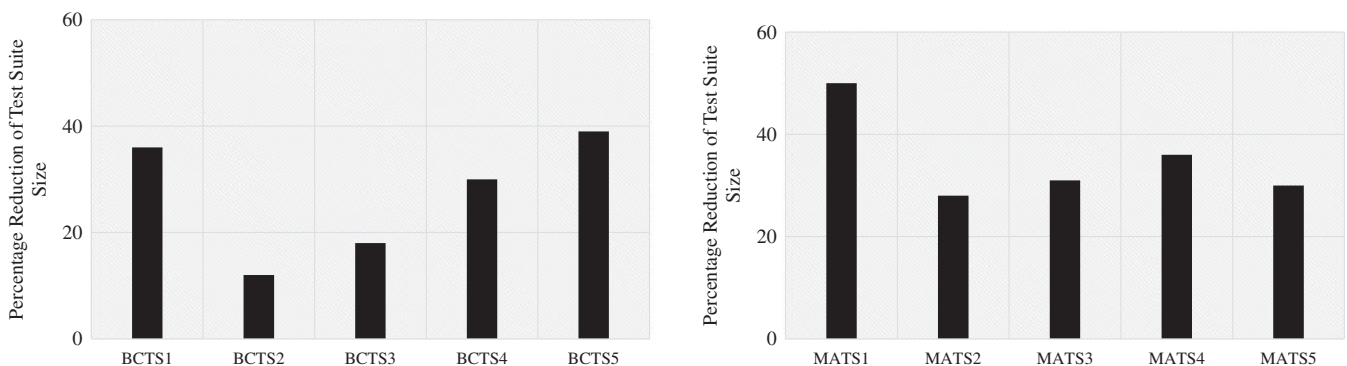


FIGURE 9 (Left) Percentage reduction of test suite size for RHS compared to current industry practice (CIP) for video conferencing software. (Right) Percentage reduction of test suite size for RHS compared to CIP for mobile application software

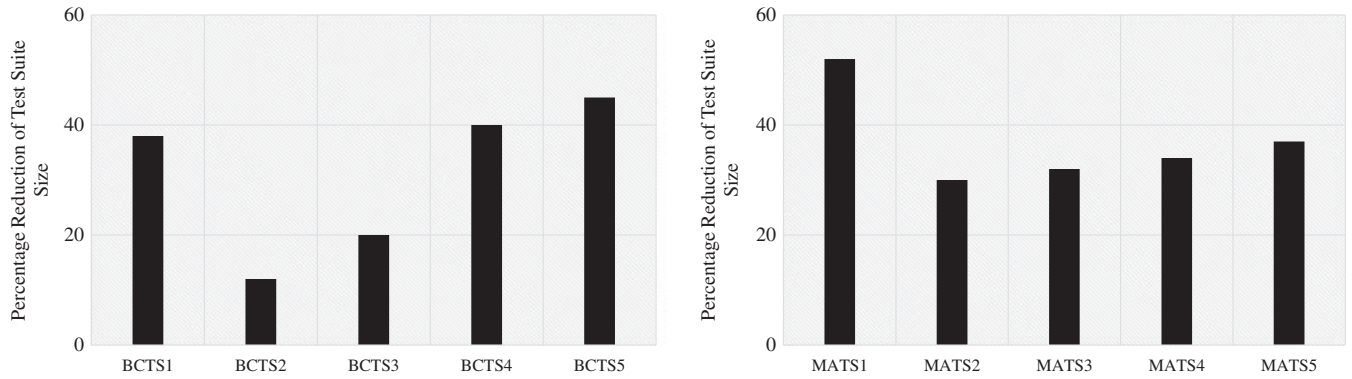


FIGURE 10 (Left) Percentage reduction of test suite size for RHS compared to the modified retest-all (MRA) approach for video conferencing software. (Right) Percentage reduction of test suite size for RHS compared to the MRA approach for mobile application software

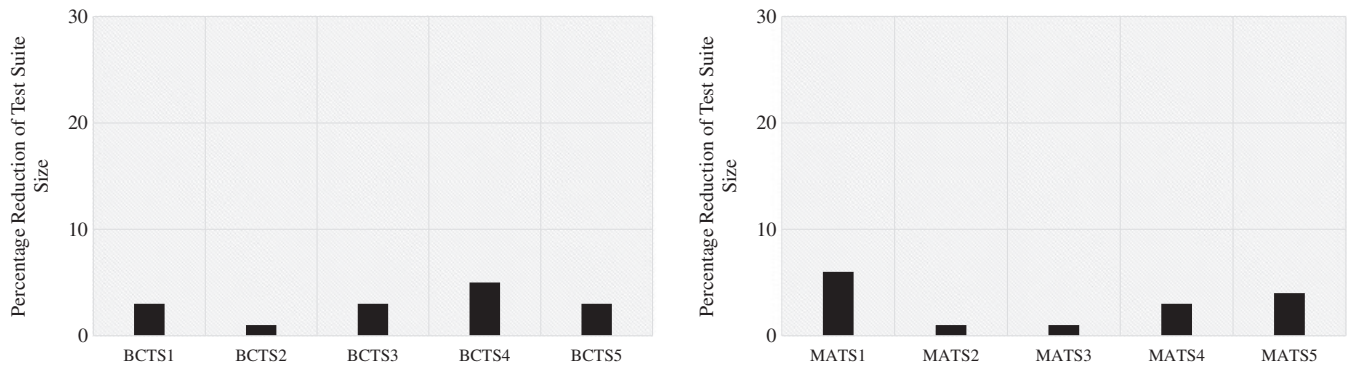


FIGURE 11 (Left) Percentage reduction of test suite size for RHS compared to FLOWER for video conferencing software. (Right) Percentage reduction of test suite size for RHS compared to FLOWER for mobile application software

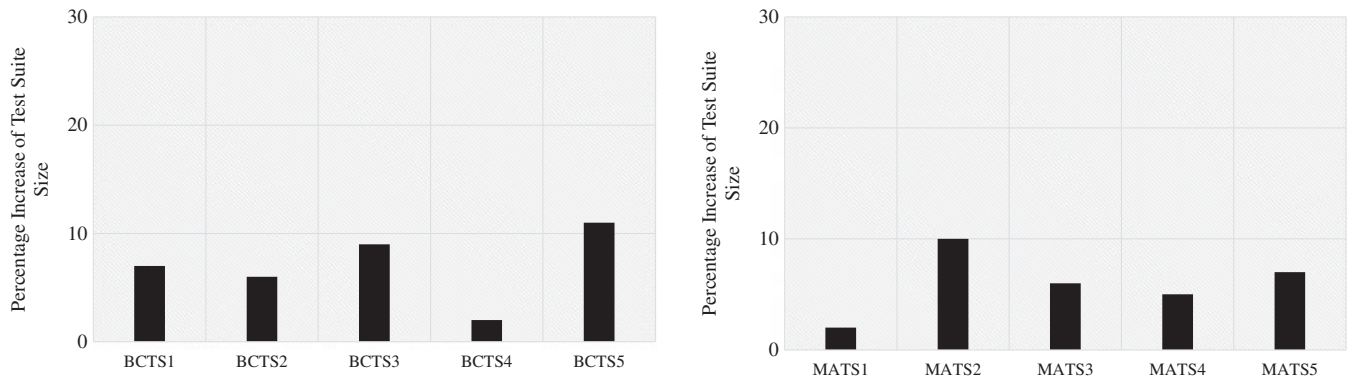


FIGURE 12 (Left) Percentage increase of test suite size for RHS compared to ROCKET for video conferencing software. (Right) Percentage increase of test suite size for RHS compared to ROCKET for mobile application software

7.3 | Fault-detection effectiveness

In experiment E3, we evaluate the effect of test suite reduction on the fault-detection effectiveness of the reduced test suite, in comparison with CIP, RA, RS, FLOWER, and ROCKET.

For both *BCTS* and *MATS*, the results indicate that RHS exhibited similar performance compared to CIP. The percentage difference in fault-detection effectiveness is under 1%. The results are shown in Figure 13. The x-axis represents test suites, and the y-axis is the percentage of fault-detection effectiveness loss for RHS, compared to CIP. Positive values signify the reduction in fault-detection effectiveness (ie, CIP detected more faults), and negative values signify the gain in fault-detection effectiveness (ie, an RHS-reduced test suite detected more faults). The latter is the case for *BCTS3*, where

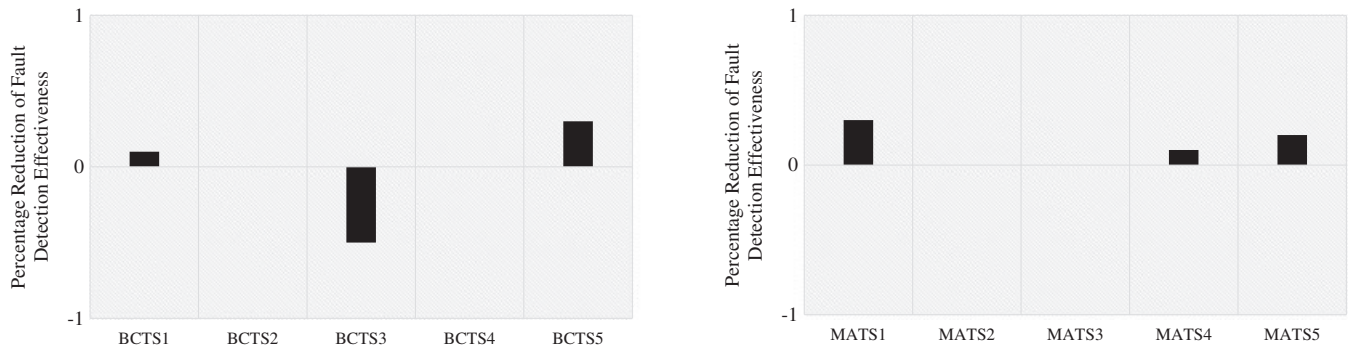


FIGURE 13 (Left) Percentage reduction of fault-detection effectiveness for RHS compared to current industry practice (CIP) for video conferencing software. (Right) Percentage reduction of fault-detection effectiveness for RHS compared to CIP for mobile application software

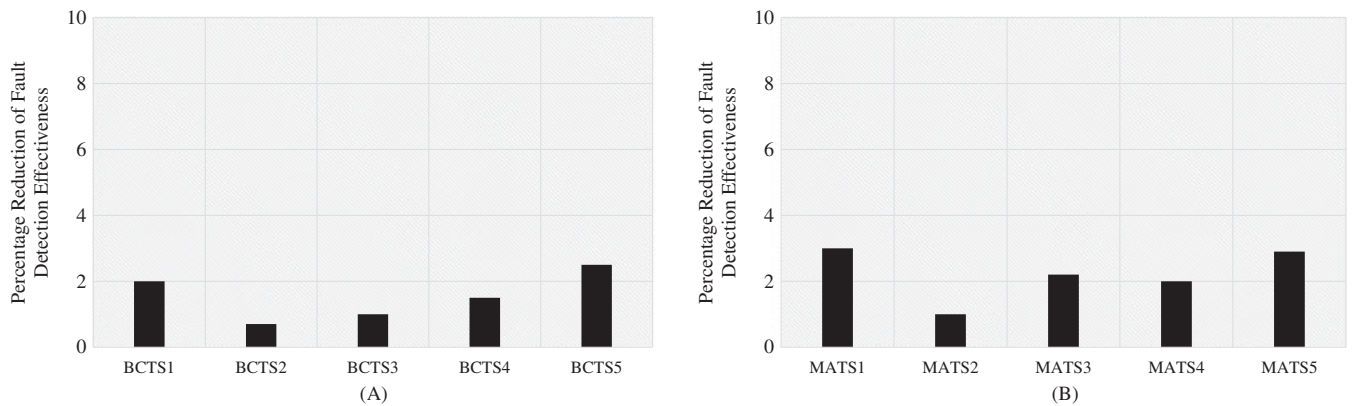


FIGURE 14 (Left) Percentage reduction of fault-detection effectiveness for RHS compared to the modified retest-all (MRA) approach for video conferencing software. (Right) Percentage reduction of fault-detection effectiveness for RHS compared to the MRA approach for mobile application software

RHS exhibited a 0.5% gain in fault-detection effectiveness compared to CIP. This is because in the CIP of video conferencing software, after automatically obtaining a test suite affected by changes, test engineers perform further manual reduction, given their experience and domain expertise. On the basis of the expertise of the engineer analyzing the test suite, some effective test cases can be removed, which was the case of *BCTS3*. Compared to MRA, the results show that RHS made a 1.5% loss in fault-detection effectiveness on average for *BCTS* and a 2% loss in fault-detection effectiveness on average for *MATS*. The results are shown in Figure 14. Compared with the results of fault-detection effectiveness obtained in comparison with CIP, we see that the proposed algorithm achieves slightly lower performance, because MRA produces bigger test suites than CIP. In comparison with RS, RHS consistently exhibits better performance, producing test suites capable of detecting 81% more faults on average compared to RS, for *BCTS*, with a deviation of about 7 on average. For *MATS*, RHS achieves better performance across all subjects, producing test suites capable of detecting 73% more faults on average compared to RS, with a deviation of about 9 on average. The results are shown in Figure 15. The boxplot shows the distribution of the percentage of the fault-detection effectiveness loss of RS compared to RHS for each test suite, averaged over 100 runs (see Section 6.2). The statistical Mann-Whitney U tests showed that the improvements in fault-detection effectiveness are statistically significant for both *BCTS* and *MATS*. In comparison with FLOWER, RHS achieved higher fault-detection effectiveness by 10% on average for *BCTS* and 8% on average for *MATS*. The results are shown in Figure 16. This is because FLOWER preserves pairwise feature coverage in test reduction, while there are distinct combinations of feature interactions higher than pairwise, which may lead to unique faults. In comparison with ROCKET, RHS increased fault-detection effectiveness by 15% on average for *BCTS* for a given time budget and 17% on average for *MATS* for a given time budget. This is because ROCKET is designed to order higher test cases with the shortest execution time, which may lead to ordering lower some of the tests with high fault-detection effectiveness. The results are shown in Figure 17.

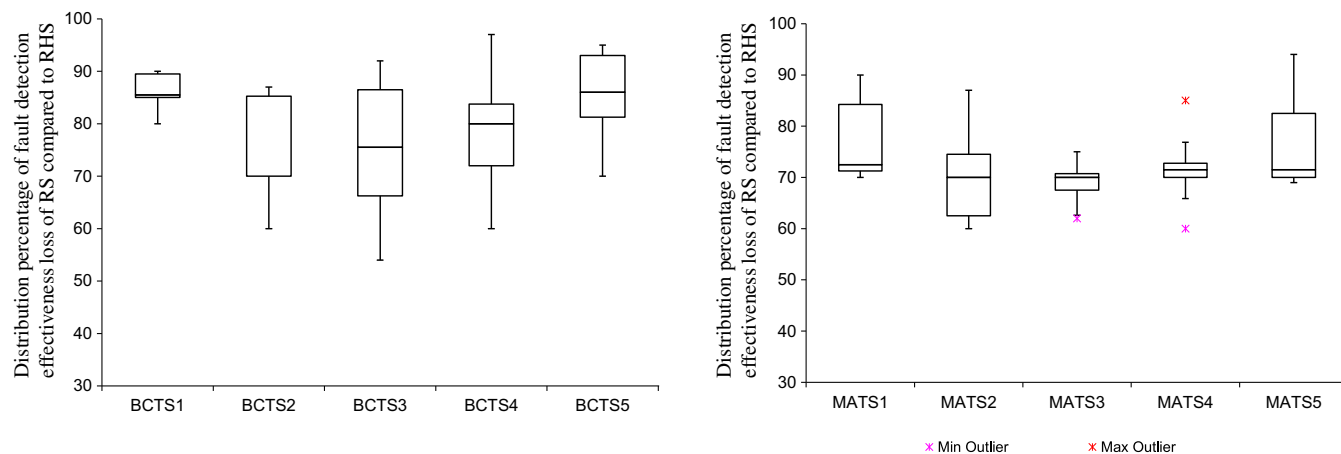


FIGURE 15 (Left) Distribution of the percentage reduction of fault-detection effectiveness for random test selection (RS) compared to RHS for video conferencing software. (Right) Distribution of the percentage reduction of fault-detection effectiveness for RS compared to RHS for mobile application software [Colour figure can be viewed at wileyonlinelibrary.com]

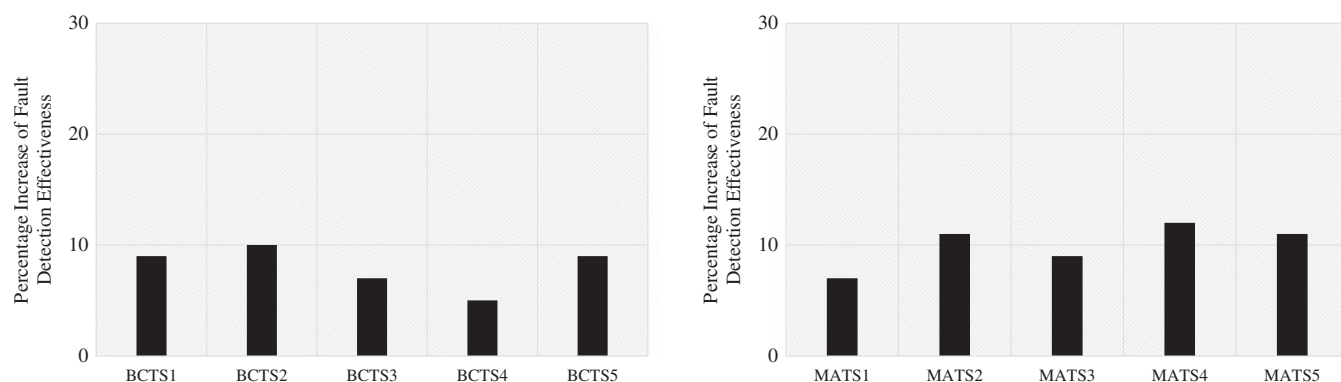


FIGURE 16 (Left) Percentage increase of fault-detection effectiveness for RHS compared to FLOWER for video conferencing software. (Right) Percentage increase of fault-detection effectiveness for RHS compared to FLOWER for mobile application software

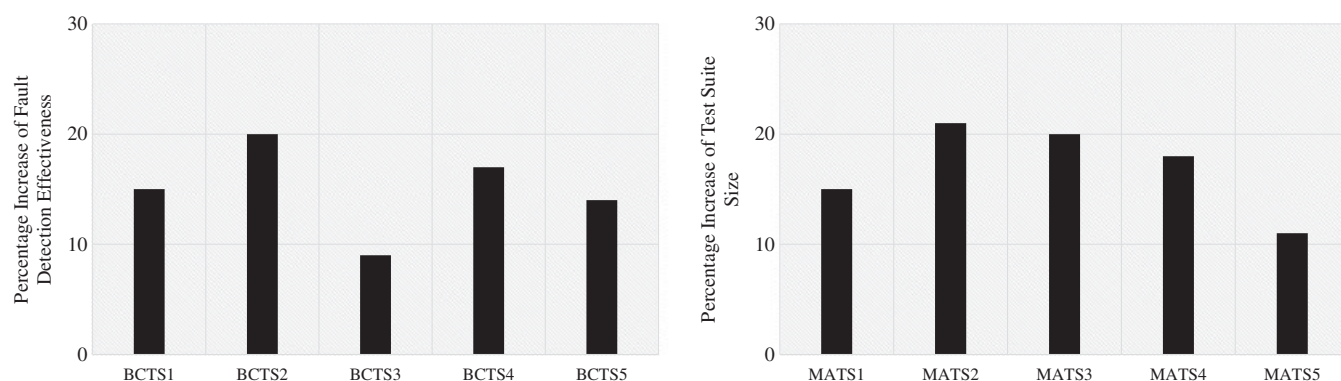


FIGURE 17 (Left) Percentage increase of fault-detection effectiveness for RHS compared to ROCKET for video conferencing software. (Right) Percentage increase of fault-detection effectiveness for RHS compared to ROCKET for mobile application software

7.3.1 | Fault-detection effectiveness with different time budgets

One important advantage of the proposed algorithm for test reduction is the ability to generate reduced test suites of different sizes for different time budgets. In Figure 18, we evaluate the fault-detection effectiveness of the test suites *BCTS*

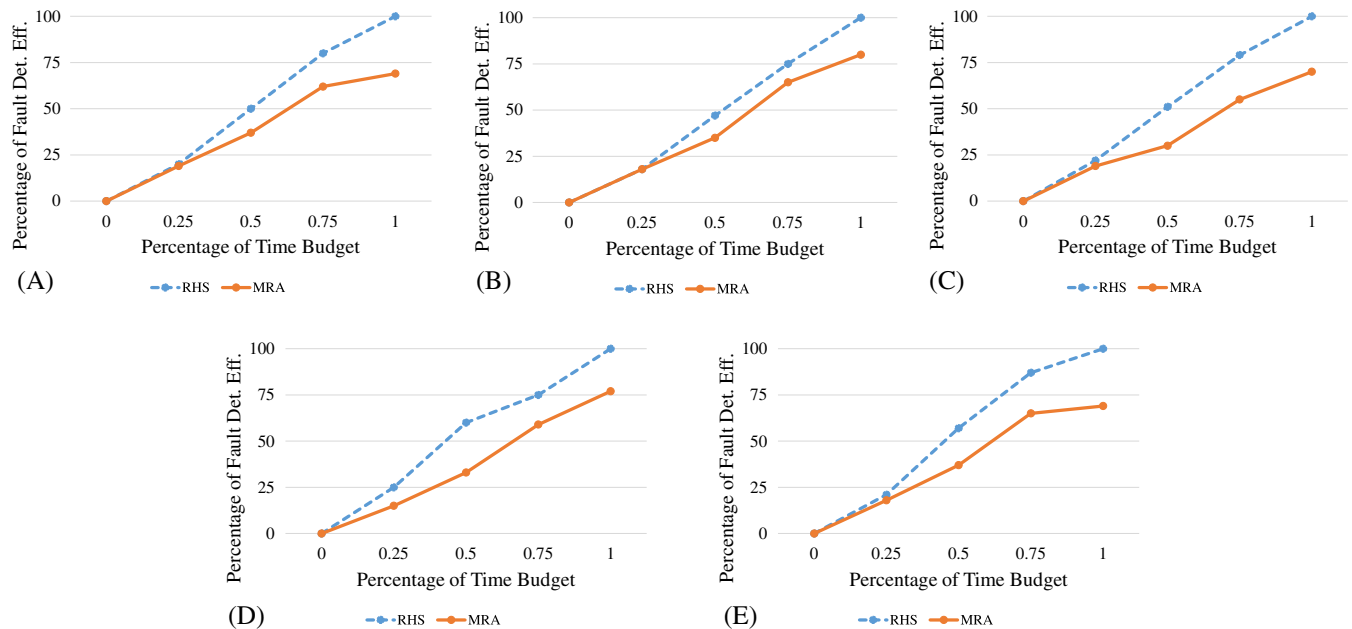


FIGURE 18 Fault-detection effectiveness of test suites for different time budgets, generated by RHS compared with the modified retest-all approach. A, BCTS1; B, BCTS2; C, BCTS3; D, BCTS4; E, BCTS5 [Colour figure can be viewed at [wileyonlinelibrary.com](https://onlinelibrary.wiley.com)]

reduced by RHS, when only parts of the test suites are executed, ie, when only a fraction of the total time required for running the test suites is available. We compare the proposed RHS algorithm with MRA. The x-axis represents different time budgets, starting at 25% of the overall test execution time needed for test suites generated by RHS, with increments of 25%. The y-axis represents fault-detection effectiveness as the percentage of unique faults detected by the test suite in the given time budget.

The results show that for smaller time budgets, the proposed algorithm achieves better performance compared to MRA. This is because the test suites generated by MRA contain redundant test cases that typically do not detect unique faults. Furthermore, test suites generated by MRA do not order sooner in execution tests that have shown higher fault-detection effectiveness. This means that test suites require more time to achieve good fault-detection effectiveness. The results also show that for RHS, the performance typically increases after some time, which is required to include effective redundant test cases that are predicted to have high fault-detection effectiveness. Finally, as the maximum time budget for comparing the fault-detection effectiveness of RHS and MRA is the time needed for executing test suites obtained by RHS (less time than is needed for executing test suites obtained by MRA), MRA never reaches full fault-detection effectiveness.

In summary, the results show that the proposed algorithm achieves similar fault-detection effectiveness as in the industry testing practice. It does not significantly compromise fault-detection performance compared to the RA approach. It improves the performance compared to ROCKET and FLOWER algorithms.

7.4 | Prediction performance

In experiment E4, we evaluate the performance of the regression model in predicting the fault-detection effectiveness of partially redundant test cases. The results for the four measures, namely, precision, recall, accuracy, and the F-score, are shown in Table 2. The mean and standard deviation are shown for *BCTS* and *MATS*, each for the five versions, as we have test execution history for 10 consecutive runs for these test suites, and as explained in E1-E3, we run the experiments five times using the history window of size 6. A *Mean* value closer to 1 indicates better performance of prediction of our regression model. The results in Table 2 show consistent performance of the prediction for all 10 test suites. The overall precision is in the range of 0.88-0.94, the overall recall is in the range of 0.89-0.97, the overall accuracy is in the range of 0.85-0.95, and the overall F-score is in the range of 0.88-0.93. These results demonstrate good performance of the regression model in detecting inefficient partially redundant test cases, which suggests that the proposed algorithm for test redundancy reduction can be a useful and reliable tool for reducing the effort and improving the quality of CI testing.

TABLE 2 Prediction performance measures for the industrial test suites

	Precision		Recall		Accuracy		F-Score	
	Mean	SDev	Mean	SDev	Mean	SDev	Mean	SDev
BCTS_1	0.89	0.09	0.91	0.17	0.90	0.07	0.90	0.18
BCTS_2	0.91	0.04	0.89	0.08	0.91	0.09	0.88	0.09
BCTS_3	0.89	0.11	0.91	0.09	0.89	0.18	0.88	0.05
BCTS_4	0.90	0.09	0.92	0.04	0.88	0.12	0.90	0.09
BCTS_5	0.88	0.07	0.95	0.09	0.92	0.09	0.93	0.21
MATS_1	0.91	0.06	0.89	0.08	0.95	0.20	0.92	0.30
MATS_2	0.94	0.11	0.89	0.21	0.89	0.09	0.90	0.11
MATS_3	0.93	0.06	0.97	0.05	0.90	0.05	0.90	0.22
MATS_4	0.90	0.25	0.92	0.12	0.87	0.45	0.89	0.07
MATS_5	0.88	0.21	0.91	0.15	0.85	0.32	0.88	0.09

8 | THREATS TO VALIDITY

External validity: A threat to the external validity of the experimental results relates the fact that the experiment subjects are not sufficiently representative. In this paper, we reduce this threat by evaluating the algorithm on two distinct industrial data sets from two different HCS domains.

Internal validity: A threat to internal validity relates potential faults in our implementation of RHS, MRA, and RS. To mitigate this threat, we have carefully analyzed and thoroughly tested all implementations used in the experimentation. Another threat to internal validity may be the effect of randomness in the evaluation. To mitigate this threat, we repeated the experiments 100 times and used nonparametric Mann-Whitney U tests, with the significance level of 0.01, to confirm the statistical significance of the results.

Construct validity: A threat to construct validity is in the choice of experimental measures. To mitigate this threat, we used the measures commonly used in the evaluation of regression test selection, such as test execution time reduction, test suite size reduction, and the fault-detection effectiveness of a test suite, as well as measures commonly used to evaluate prediction performance, such as precision, recall, accuracy, and F-score.

9 | RELATED WORK

Our work touches upon and extends the following two principal research areas.

CI regression test optimization. Several approaches have been proposed for improving the effectiveness of regression test selection. Ren et al used change impact analysis to identify regression tests affected by a change, by constructing a call graph for each test that consists of interdependent changes at the method level.²¹ Orso et al used change impact analysis and field data, although the approach is considered unsafe.²² Other techniques include regression test selection based on specification changes,^{23–25} as well as code changes,^{23,26–28} or adaptive test prioritization.²⁹ Several researchers have proposed the use of model-based techniques for regression test selection. Another approach has been proposed to dynamically skip tests when the expected cost of running the test exceeds the expected cost of removing the test.³⁰ Korel et al used an extended finite-state machine model dependence analysis for identifying modifications.³¹ Andrews et al applied model-based selective regression testing to a railroad crossing control system based on the behavioral model of a system and a behavioral test suite.³² However, a common limitation of model-based regression testing techniques is the ambiguous quality of regression test suites, because these techniques use only a modified system model instead of an original.³¹ Another limitation of these approaches is that they give limited attention to the problem of regression testing of HCS developed in CI, especially in the case of highly interleaved tests. Several other approaches were proposed in the direction of CI testing. An approach has been proposed for regression test selection with dynamic test dependencies,³³ where, for each test method or a test class file, dependencies are computed. Another approach by this author has been to dynamically detect, at the operating system level, all file artifacts a test depends on.³⁴ Legunsen et al reported a study³⁵ evaluating the performance of static regression test selection techniques, and the authors reported that class-level approaches perform better than method-level approaches. Zhang proposed a hybrid regression test selection approach combining method and file level selection.³⁶ Qu et al proposed an approach for running impact analysis of configuration changes for test case selection.³⁷ However, all these approaches work at the source code level, which is not available in our case.

The requirement of our industry partner was to develop an approach that will not require access to source code, but only feature tags indicating code changes. A blackbox approach was proposed for test prioritization in CI by Marijan et al.² The proposed algorithm orders test cases given their historical fault-detection effectiveness and time efficiency. However, the algorithm does not consider test redundancy in test prioritization, which may lead to lower fault-detection effectiveness compared to our approach.

Test redundancy. The existing approaches for testing redundancy detection are based on different test coverage metrics, such as statement coverage,³⁸ branch coverage,³⁹ decision coverage, condition coverage, or path coverage. However, Koochakzadeh et al reported that coverage-based metrics are imprecise in test redundancy detection and should be combined with additional information.^{16,40} Fraser and Wotawa presented an approach where redundancy in tests generated with a model checker is identified based on the analysis of paths of the Kripke structure.⁴¹ In a later stage, the algorithm modifies redundant parts of tests to eliminate redundancy. However, the authors reported the high runtime complexity of the algorithm as a drawback. Jeffrey and Gupta proposed to use a heuristic⁴² for selectively retaining redundant tests in a suite based on two sets of test requirements, namely, branch coverage and all-uses coverage, while preserving fault-detection effectiveness.^{43,44} However, the approach has limitations due to a large size of a reduced test suite. Test redundancy has been studied in the context of test suite minimization techniques,^{3,12,38,39,45-48} which aim to reduce the size of the original test suite while preserving its fault-detection effectiveness. However, it was shown that test minimization techniques often compromise the fault-detection effectiveness of a test suite. Rothermel et al analyzed the effect of test suite minimization on its fault-detection capability and showed that minimizing a test suite can significantly lower its effectiveness in detecting faults.^{39,49} Heimdahl and George found that test suite reduction based on structural coverage exhibits high loss in fault-detection capability.⁵⁰ These findings imply that the performance of test suite reduction could be improved by improving the effectiveness of precisely identifying redundant tests. This is particularly important for HCS, where redundancy incurs high testing and maintenance costs. Unlike other approaches, our work uses classification trees in combination with test coverage metrics to improve the precision of identifying and reducing redundant test cases.

10 | CONCLUSIONS

In this paper, we have addressed the challenge of improving the efficiency of CI development and testing practice for HCS. Regression testing, which runs in CI frequently, needs to be time efficient. However, HCS environments typically imply complex and sizable test setups, due to the large number of software features, which need to be tested in different combinations. To enable efficient CI development and testing of software systems bound by these conflicting objectives, in this paper, we have presented a practical algorithm for improving the efficiency of CI regression test selection based on identifying and reducing test redundancy. The algorithm uses coverage metrics to classify tests as unique, totally redundant, or partially redundant. Then, for partial redundancy, regression models are built to classify partially redundant test cases as being likely or not likely to detect failures, based on their historical fault-detection effectiveness. Finally, on the basis of this information, partially redundant tests are classified into effective and ineffective tests, and a regression test suite is created consisting of only unique and effective partially redundant tests. The algorithm has been evaluated using a large set of subjects, including one industrial HCS video conferencing system and one HCS mobile application software. The results show that the proposed algorithm improves CIP, enabling faster regression test feedback (better time efficiency in CI), without compromising fault-detection effectiveness. The results further show that our proposed algorithm can reduce test feedback compared to an advanced RA approach without significantly compromising the fault-detection effectiveness of a regression suite. The results also show that the proposed algorithm can significantly improve fault-detection effectiveness compared to random testing, which is often used to reduce the effort of regression testing in time- and resource-constrained environments. Finally, the results show that the proposed algorithm for predicting the fault-detection effectiveness of tests achieves good prediction performance, in terms of accuracy and precision. As part of future work, we plan to conduct research in the following directions. First, we will conduct a performance analysis of the algorithm. Second, we plan to perform further comparison for the variable time budgets with the test prioritization and reduction approaches. Third, we will perform an analysis of the major causes of false positives and false negatives encountered while evaluating the prediction performance of our algorithm.

ACKNOWLEDGEMENT

This work was supported by the Norwegian Research Council through the Certus Centre (SFI) project.

ORCID

Dusica Marijan  <http://orcid.org/0000-0001-9345-5431>

REFERENCES

1. Spieker H, Gotlieb A, Marijan D, Mossige M. Reinforcement learning for automatic test case prioritization and selection in continuous integration. In: Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA); 2017; Santa Barbara, CA.
2. Marijan D, Gotlieb A, Sen S. Test case prioritization for continuous regression testing: an industrial case study. Paper presented at: 2013 IEEE International Conference on Software Maintenance (ICSM); 2013; Eindhoven, The Netherlands.
3. Gotlieb A, Marijan D. FLOWER: optimal test suite reduction as a network maximum flow. In: Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA); 2014; San Jose, CA.
4. Elbaum S, Rothermel G, Penix J. Techniques for improving regression testing in continuous integration development environments. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering; 2014; Hong Kong, China.
5. Marijan D, Liaaen M. Practical selective regression testing with effective redundancy in interleaved tests. In: Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP); 2018; Gothenburg, Sweden.
6. Cohen DM, Dalal SR, Fredman ML, Patton GC. An AETG system: an approach to testing based on combinatorial design. *IEEE Trans Softw Eng.* 1997;23(7):437-444.
7. Cohen MB, Gibbons PB, Mugridge WB, Colbourn CJ. Constructing test suites for interaction testing. In: Proceedings of the 25th International Conference on Software Engineering (ICSE); 2003; Portland, OR.
8. Bell KZ, Vouk MA. On effectiveness of pairwise methodology for testing network-centric software. Paper presented at: 2005 International Conference on Information and Communication Technology; 2005; Cairo, Egypt.
9. Engström E, Runeson P. Test overlay in an emerging software product line - an industrial case study. *Inf Softw Technol.* 2013;55(3):581-594.
10. Selby RW, Porter AA. Learning from examples, generation and evaluation of decision trees for software resource analysis. *IEEE Trans Softw Eng.* 1988;14(12):1743-1756.
11. Kang KC, Cohen SG, Hess JA, Nowak WE, Peterson AS. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Technical Report. 1990.
12. Hervieu A, Marijan D, Gotlieb A, Baudry B. Practical minimization of pairwise-covering test configurations using constraint programming. *Inf Softw Technol.* 2016;71(C):129-146.
13. Wu X, Kumar V, Quinlan JR, et al. Top 10 algorithms in data mining. *Knowl Inf Syst.* 2008;14(1):1-37.
14. Li H, Sun J, Wu J. Predicting business failure using classification and regression tree: an empirical comparison with popular classical statistical methods and top classification mining methods. *Expert Syst Appl.* 2010;37(8):5895-5904.
15. Gokhale SS, Lyu MR. Regression tree modeling for the prediction of software quality. In: Proceedings of the Third ISSAT International Conference on Reliability and Quality in Design; 1997; Anaheim, CA.
16. Koochakzadeh N, Garousi V, Maurer F. Test redundancy measurement based on coverage information: evaluations and lessons learned. Paper presented at: 2009 International Conference on Software Testing Verification and Validation; 2009; Denver, CO.
17. Kim J-M, Porter A. A history-based test prioritization technique for regression testing in resource constrained environments. In: Proceedings of the 24th International Conference on Software Engineering (ICSE); 2002; Orlando, FL.
18. Marijan D, Liaaen M. Effect of time window on the performance of continuous regression testing. Paper presented at: 2016 IEEE International Conference on Software Maintenance and Evolution (ICSME); 2016; Raleigh, NC.
19. Quinlan JR. *C4.5: Programs for Machine Learning*. Burlington, MA: Morgan Kaufmann Publishers; 1993.
20. Witten IH, Frank E, Hall MA. *Data Mining: Practical Machine Learning Tools and Techniques*. 3rd ed. Elsevier: Morgan Kaufmann; 2011:1-629. ISBN 9780123748560.
21. Ren X, Shah F, Tip F, Ryder BG, Chesley O. Chianti: a tool for change impact analysis of Java programs. In: Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA); 2004; Vancouver, Canada.
22. Orso A, Apiwattanapong T, Harrold MJ. Leveraging field data for impact analysis and regression testing. In: Proceedings of the 9th European Software Engineering Conference (ESEC/FSE); 2003; Helsinki, Finland.
23. Chen Y, Probert RL, Sims DP. Specification-based regression test selection with risk analysis. In: Proceedings of the 2002 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON); 2002; Toronto, Canada.
24. Mao C, Lu Y. Regression testing for component-based software systems by enhancing change information. In: Proceedings of the 12th Asia-Pacific Software Engineering Conference (APSEC); 2005; Taipei, Taiwan.
25. Sajeev ASM, Wibowo B. Regression test selection based on version changes of components. In: Proceedings of the Tenth Asia-Pacific Software Engineering Conference Software Engineering Conference (APSEC); 2003; Chiang Mai, Thailand.
26. Gligoric M, Eloussi L, Marinov D. Ekstazi: lightweight test selection. In: Proceedings of the 37th International Conference on Software Engineering - Volume 2 (ICSE); 2015; Florence, Italy.
27. Gupta R, Harrold MJ, Soffa ML. Program slicing-based regression testing techniques. *J Softw Test Verification Reliab.* 1996;6(2):83-111.
28. Rothermel G, Harrold MJ. A safe, efficient regression test selection technique. *ACM Trans Softw Eng Methodol.* 1997;6(2):173-210.
29. Schwartz A, Do H. Cost-effective regression testing through adaptive test prioritization strategies. *J Syst Softw.* 2016;115:61-81.

30. Herzig K, Greiler M, Czerwonka J, Murphy B. The art of testing less without sacrificing quality. In: Proceedings of the 37th International Conference on Software Engineering - Volume 1 (ICSE); 2015; Florence, Italy.
31. Korel B, Tahat LH, Vaysburg B. Model based regression test reduction using dependence analysis. In: Proceedings of the International Conference on Software Maintenance; 2002; Montreal, Canada.
32. Andrews A, Elakeili S, Alhaddad A. Selective regression testing of safety-critical systems: a black box approach. In: Proceedings of the 2015 IEEE International Conference on Software Quality, Reliability and Security - Companion (QRS-C); 2015; Vancouver, Canada.
33. Gligoric M, Eloussi L, Marinov D. Practical regression test selection with dynamic file dependencies. In: Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA); 2015; Baltimore, MD.
34. Celik A, Vasic M, Milicevic A, Gligoric M. Regression test selection across JVM boundaries. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE); 2017; Paderborn, Germany.
35. Legunsen O, Hariri F, Shi A, Lu Y, Zhang L, Marinov D. An extensive study of static regression test selection in modern software evolution. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE); 2016; Seattle, WA:583-594.
36. Zhang L. Hybrid regression test selection. Paper presented at: 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE); 2018; Gothenburg, Sweden.
37. Qu X, Acharya M, Robinson B. Impact analysis of configuration changes for test case selection. In: Proceedings of the 2011 IEEE 22nd International Symposium on Software Reliability Engineering (ISSRE); 2011; Hiroshima, Japan.
38. Offutt J, Pan J, Voas JM. Procedures for reducing the size of coverage-based test sets. In: Proceedings of the 12th International Conference Testing Computer Software; 1995; Washington, DC.
39. Rothermel G, Harrold MJ, Ostrin J, Hong C. An empirical study of the effects of minimization on the fault detection capabilities of test suites. In: Proceedings of the International Conference on Software Maintenance (ICSM); 1998; Bethesda, MD.
40. Koochakzadeh N, Garousi V. A tester-assisted methodology for test redundancy detection. *Adv Softw Eng*. 2010;6:1-6:13.
41. Fraser G, Wotawa F. Redundancy based test-suite reduction. In: *Fundamental Approaches to Software Engineering: 10th International Conference, FASE 2007, Held as Part of the Joint European Conferences, on Theory and Practice of Software, ETAPS 2007, Braga, Portugal, March 24 - April 1, 2007. Proceedings*. Berlin, Germany: Springer-Verlag Berlin Heidelberg; 2007:291-305.
42. Harrold MJ, Gupta R, Soffa ML. A methodology for controlling the size of a test suite. *ACM Trans Softw Eng Methodol*. 1993;2(3):270-285.
43. Jeffrey D, Gupta N. Test suite reduction with selective redundancy. In: Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM); 2005; Budapest, Hungary.
44. Jeffrey D, Gupta N. Improving fault detection capability by selectively retaining test cases during test suite reduction. *IEEE Trans Softw Eng*. 2007;33(2):108-123.
45. Masri W, Podgurski A, Leon D. An empirical study of test case filtering techniques based on exercising information flows. *IEEE Trans Softw Eng*. 2007;33(7):454-477.
46. Hsu H-Y, Orso A. MINTS: a general framework and tool for supporting test-suite minimization. In: Proceedings of the 31st International Conference on Software Engineering (ICSE); 2009; Vancouver, Canada.
47. Zhang L, Marinov D, Zhang L, Khurshid S. An empirical study of junit test-suite reduction. In: Proceedings of the 2011 IEEE 22nd International Symposium on Software Reliability Engineering (ISSRE); 2011; Hiroshima, Japan.
48. Baller H, Lity S, Lochau M, Schaefer I. Multi-objective test suite optimization for incremental product family testing. Paper presented at: 2014 IEEE Seventh International Conference on Software Testing, Verification and Validation (ICST); 2014; Cleveland, OH.
49. Rothermel G, Harrold MJ, von Ronne J, Hong C. Empirical studies of test-suite reduction. *J Softw Test Verification Reliab*. 2002;12(4):219-249.
50. Heimdahl MPE, George D. Test-suite reduction for model based tests: effects on test quality and implications for testing. In: Proceedings of the 19th IEEE International Conference on Automated Software Engineering (ASE); 2004; Linz, Austria.

How to cite this article: Marijan D, Gotlieb A, Liaaen M. A learning algorithm for optimizing continuous integration development and testing practice. *Softw: Pract Exper*. 2019;49:192-213. <https://doi.org/10.1002/spe.2661>