The Journal of Systems & Software 179 (2021) 110990

Contents lists available at ScienceDirect

The Journal of Systems & Software

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

Evaluating T-wise testing strategies in a community-wide dataset of configurable software systems ☆

Fischer Ferreira [a,*], Gustavo Vale [b], João P. Diniz [a], Eduardo Figueiredo [a]

[a] Federal University of Minas Gerais, Belo Horizonte, Brazil
[b] Saarland University, Saarbrücken, Germany

ARTICLE INFO

Article history:
Received 21 August 2020
Received in revised form 18 March 2021
Accepted 28 April 2021
Available online 4 May 2021

Keywords:
Testing configurable systems
Software faults
Feature interactions
T-wise sampling strategies

ABSTRACT

Configurable software systems allow developers to maintain a unique platform and address a diversity of deployment contexts and usages. Testing configurable systems is essential because configurations that fail may potentially hurt users and degrade the project reputation. As extensively testing all valid configurations is infeasible in practice, several testing strategies have been proposed to recommend an optimal sample of configurations able to find most existing faults. However, up to now, we could not find studies comparing testing strategies with a community-wide dataset. Aiming at (i) comparing sampling testing strategies and (ii) understanding the location of faults, we use a community-wide dataset from the literature and compare suggested configurations from variations of five t-wise testing strategies (e.g., *ICPL-T2*, *Chvatal-T4*, and *IncLing-T2*). This comparison aims to find which strategies are faster, more comprehensive, effective on identifying faults, time-efficient, and coverage-efficient in this community-wide dataset and the reasons why a strategy fared better in one investigated property. Complementary, we investigate the dispersion of faults over classes and features from the dataset. As a result, we found that the dispersion of faults are usually concentrated in a few classes and features. Furthermore, fault-prone classes and features are distinguishable from classes and features safe of faults. Overall, we believe that with our results practitioners acquire the necessary knowledge to choose a testing strategy that best fits their needs. Moreover, researchers and tool builders are served with a bunch of opportunities to improve existing testing strategies and tools. For instance, they may incorporate information from fault-prone classes and features when selecting configurations to be tested in their testing strategies.

Published by Elsevier Inc.

1. Introduction

Configurable software systems (for short, *configurable systems*) are software systems that can be adapted based on a set of features that fits specific customer needs (Apel et al., 2013a; Pohl and Metzger, 2006; Svahnberg et al., 2005). To address a diversity of deployment contexts and usages, developers only need to activate or deactivate features. Although configurable systems increase code reuse and productivity, developers have to deal with several configuration options. Hence, to ensure that all configurations correctly compile, build, and run, developers spend considerable effort testing their systems. This effort is necessary mainly because configurations that fail may hurt potential users and degrade the reputation of a project (Halin et al., 2019).

Testing configurable systems is challenging due to the high number of possible configurations, which makes exhaustive testing prohibitively expensive and practically infeasible. Alternatively, developers may test a sample of valid configurations (Greiler et al., 2012; Machado et al., 2014). Several strategies for choosing a sample of configurations to test have been proposed (Al-Hajjaji et al., 2014; Kruse et al., 2014; Liebig et al., 2013; Medeiros et al., 2016; Souto et al., 2017). Some of them use information only from the *feature model* (Al-Hajjaji et al., 2016; Cohen et al., 2003; Johansen et al., 2011, 2012b; Kuhn et al., 2010, 2004; Medeiros et al., 2016; Nie and Leung, 2011), while others also use information from the source code (Kim et al., 2012, 2013; Liebig et al., 2013; Nguyen et al., 2014; Souto et al., 2017). Even with a large number of testing strategies in the literature and used in practice, previous work (Medeiros et al., 2016; Souto et al., 2017) reports a lack of an empirical evaluation based on a community-wide dataset to guide the comparison of different testing strategies.

T-wise interaction sampling is a cost-effective sampling technique for discovering interaction faults in configurable systems

☆ Editor: Aldeida Aleti.

* Corresponding author.
E-mail address: fischerjf@dcc.ufmg.br (F. Ferreira).

https://doi.org/10.1016/j.jss.2021.110990
0164-1212/Published by Elsevier Inc.

(Henard et al., 2014). This approach has attracted the interest of several researchers because it achieves effective results with lower cost by minimizing the number of configurations to be tested even when using small values for t (e.g., 1 or 2) (Al-Hajjaji et al., 2016; Garvin et al., 2011; Henard et al., 2014; Johansen et al., 2011; Kaltenecker et al., 2019; Krieter et al., 2020; Kuhn and Reilly, 2002; Nie and Leung, 2011; Xiang et al., 2021). In this work, we chose to evaluate the t-wise strategies implemented in FeatureIDE (Thüm et al., 2014).

Our goal in this paper is twofold. First, we provide a comparison of sampling testing strategies. Second, we provide a deep understanding of faults found in a community-wide dataset. While the comparison of testing strategies may benefit practitioners supporting their choice of a testing strategy that best fits their needs, a deep understanding of faults may help practitioners to learn characteristics of classes and features prone to fail, to avoid the introduction of similar faults, and to guide them to increase the test coverage in these fault-prone classes and features. On the other hand, this study may also benefit researchers and tool builders by showing them opportunities for improving existing testing strategies and tools.

To achieve our first goal, we choose (i) a dataset, (ii) testing strategies to be compared, (iii) the comparison criteria, and (iv) a reference list of faults. We use a community-wide dataset with 30 configurable systems and test suite available for each of them (Ferreira et al., 2020c). As testing strategies to be compared, we selected variations of five t-wise sampling testing strategies *CASA* (Garvin et al., 2011), *Chvatal* (Johansen et al., 2012b), *ICPL* (Johansen et al., 2011), *IncLing* (Al-Hajjaji et al., 2016), and YASA (Krieter et al., 2020) because t-wise strategies assure a degree of testing coverage on the recommended configurations (see details of our choice in Sections 2.2 and 2.3). At the end, we compare sixteen t-wise strategies (*CASA-T1*, *CASA-T2*, *CASA-T3*, *CASA-T4*, *Chvatal-T1*, *Chvatal-T2*, *Chvatal-T3*, *Chvatal-T4*, *ICPL-T1*, *ICPL-T2*, *ICPL-T3*, *IncLing-T2*, *YASA-T1*, *YASA-T2*, *YASA-T3*, and *YASA-T4*) and two baselines (brute force and random selection). As comparison criteria, we evaluate which testing strategies are faster, more comprehensive (i.e., with greater coverage), more effective in identifying faults, time-efficient, and coverage-efficient. To build a reference list of faults, we use the union of all faults found by the subject strategies. To make our study feasible, we limit the number of recommended configurations for the baselines for up to 250 configurations. This limitation was needed due to the time to test all configurations of some subject configurable systems.

To achieve our second goal, we look at the dispersion of faults found over classes and features in the subject dataset. Then, we measure each class and feature with metrics commonly used in practice (Chidamber et al., 1998). Number of lines of code (*LoC*) (CK, 2020), weighted methods per class (*WMC*) (CK, 2020), and response for a class (*RFC*) (CK, 2020) are examples of metrics at class-level. Feature scattering and feature tangling are examples of metrics at feature-level. Finally, we compute Spearman's rank correlation between the number of faults in a component (i.e., class or feature) and a given metric.

As a result of our first goal, we analyze our results by grouping the test strategies by t-wise groups. Thus, for groups 1-, 2-, 3-, and 4-wise, we find the following highlights. *ICPL-T1*, *ICPL-T2*, *ICPL-T3* and *YASA-T4* are faster testing strategies. *Chvatal-T1*, *IncLing-T2*, *Chvatal-T3*, and *Chvatal-T4*, are most comprehensive testing strategies. In this work, we define the testing strategy as most comprehensive when it presents high coverage of valid configurations. *Chvatal-T1*, *ICPL-T2*, *Chvatal-T3*, and *Chvatal-T4* recommends configurations able to find the greatest number of faults. *Chvatal-T1*, *ICPL-T2*, *Chvatal-T3*, and *YASA-T4* are the testing strategies that recommends configurations with the best balance among faults found and time. However, *Chvatal-T1*, *CASA-T2*, *Chvatal-T3*, and *Chvatal-T4*, recommends configurations with the best balance among faults found and the number of recommended configurations.

As a result of our second goal, we found at least one fault in 16 out of 30 systems in the subject dataset. Regarding the class-level analysis, we found that faults are concentrated in only 0.8% of classes of the subject dataset and high values of some source code metrics, such as *LoC*, *WMC*, and *RFC*, appear to be related to fault-prone classes. Regarding the feature-level analysis, we found that faults are concentrated in a few features with high values of some source code metrics, such as feature scattering and tangling.

Our discussions show that testing strategies with similar t usually present similar results (e.g., *ICPL-T2*, *Chvatal-T2*, *IncLing-T2* are similarly coverage-efficient). Despite t-wise strategies being able to find faults, the most effective strategy found only 30.62% of the known faults. Furthermore, we also perceived that strategies crash when their t is larger than the total number of valid configurations of a subject system (e.g., CASA-T4 against the IntegerSetSPL system, which has only 3 distinct configurations). Moreover, some 4-wise strategies take more than eight testing hours overall (e.g., Chavtal-T4 against the Checkstyle system). With all, we believe that our results in fact support practitioners on choosing the sampling testing strategy and guide them on creating test cases in classes and features prone to conflict. On the other hand, we have drawn concrete directions for researchers and tool builders to improve current testing strategies and tools, specially because we show that the fault-prone classes and features have distinct characteristics from classes and features free of faults.

Overall, we make the following contributions:

- We provide evidence of which testing strategies are faster, more comprehensive, more effective on finding faults, more time-efficient, and coverage-efficient based on data from a community-wide dataset;
- We show that faults are usually concentrated in few classes and features. Moreover, these fault-prone classes and features have distinguish characteristics from fault-free classes and features and these differences can be found based on commonly used source code metrics;
- We make our infrastructure and data publicly available for follow-up studies on a supplementary Web site (Ferreira et al., 2020c).

The remainder of this paper is organized as follows. Section 2 presents background information on configurable software systems and testing strategies. Section 3 presents a dataset of configurable software systems used in this paper. Section 4 describes our study settings. Section 5 presents the results of our empirical study. Section 6 shows discussions about our results and implications for researchers and practitioners. Section 7 describes the main limitations and threats to validity of this work. Section 8 discusses some related work. Finally, Section 9 concludes this study and points directions for future work.

2. Background

In this section, we present an overview of approaches, techniques, and challenges to develop configurable systems (Section 2.1), testing configurable systems (Section 2.2), and t-wise sampling strategies (Section 2.3).

2.1. Developing configurable systems

Configurable systems can be efficiently customized according to a set of features (configuration options) that satisfies different customer needs (Svahnberg et al., 2005). The source code that belongs to deactivated features is ignored in a determined configuration. To develop configurable systems, developers may choose between the two main strategies: *compile-* and *execution-time* (Apel et al., 2013a).

In compile-time strategies, developers activate a set of features that is (pre)processed in order to generate the final product (Apel et al., 2013a; Czarnecki and Eisenecker, 2000; Kastner, 2010). This strategy can be divided into annotative and compositional approaches. The main difference among these approaches is that in the annotative strategy, developers annotate their code (e.g., with `#ifdef`-like directives) to represent variation points. In compositional approaches, developers implement each variation point in modularized components (e.g., features Batory, 2005 to FOP, aspects Schaefer et al., 2011 to AOP or deltas Schaefer et al., 2010 to DOP). Conditional compilation (CC) is an example of a technique that uses the annotative approach (Apel et al., 2013b; Post and Sinz, 2008). Feature-oriented programming (FOP) (Batory, 2005), aspectual feature modules (AFM) (Apel et al., 2008), and delta-oriented programming (DOP) (Schaefer et al., 2010) are examples of techniques that use a compositional approach. The Linux Kernel (Linux, 2020) and the Firefox Web Browser (Mozilla, 2020; Garvin and Cohen, 2011) are examples of configurable systems developed with compile-time strategies.

In execution-time strategies, developers activate features that contain code blocks at execution-time (Post and Sinz, 2008). Variability encoding is an example of execution-time strategy (Apel et al., 2013a,b). Similar to conditional compilation, developers should create conditional structures (e.g., `if/else` statements or ternary operator `?:`), and generate the so-called *meta-products* (Thüm et al., 2012). Then, developers should create configuration files determining features to be active in a target configuration. This way, all features can be activated or deactivated at execution time (Kim et al., 2013; Meinicke et al., 2016; Wong et al., 2018). Android family is a (set of) configurable system(s) developed using variability encoding (Galindo et al., 2016; Li et al., 2016). In this study, configurable systems are systems that, through manipulating functional features, are added in run-time. We do not consider systems in which program parameters (Apel et al., 2013a) are used to determine configurations.

Feature interactions occur when features influence the behavior of other features. It becomes an issue when developers look at features together and find an unexpected behavior that does not occur when they look at features in isolation (Soares et al., 2018b). Unexpected behavior can introduce faults that manifest themselves in specific configurations. Feature interactions are among the greatest challenges in developing configurable systems (Abal et al., 2014; Apel et al., 2011; Garvin and Cohen, 2011; Kim et al., 2010; Machado et al., 2014; Nguyen et al., 2019; Schuster et al., 2014; Siegmund et al., 2012; Soares et al., 2018b), since they enforce the need of creating testing suites which cover all potential interactions (Cohen et al., 2008; Oster et al., 2011).

2.2. Testing configurable systems

A major challenge for developers of configurable systems is to ensure that all configurations correctly compile, build, and run. The fact that the number of product variants grows exponentially with the number of variation points makes it infeasible to test all possible feature combinations after a certain number of features. This way, practitioners have to choose somehow to test only a sample of configurations.

Over the years, various strategies have been developed to test configurable systems (Engström and Runeson, 2011; Ferreira et al., 2019; Machado et al., 2014; da Mota et al., 2011; Puoskari et al., 2013). These strategies can be classified into: variability-aware testing (Kim et al., 2013; Meinicke et al., 2016; Wong et al., 2018) and configuration sampling testing (Al-Hajjaji et al., 2016; Johansen et al., 2012a; Souto et al., 2017). *Variability-aware testing strategies* explore dynamically all reachable configurations from a given test, by monitoring feature variable accesses during test execution. *Configuration sampling testing strategies* sample a subset of valid configurations and test them individually. We focus on configuration sampling testing strategies because they are more often used in practice (Varshosaz et al., 2018).

Configurable sampling testing strategies (for short, *sampling strategies*) can be classified into four groups (Varshosaz et al., 2018): *manual selection*, *semi-automatic selection*, *automatic selection*, and *coverage*. In the first, practitioners should manually select the configurations to be tested. In the second, the selection of configurations requires an input representing the stop criteria (e.g., the number of products to be generated, the time for sampling, or a degree of coverage). In the third, the selection of configurations has support of greedy (i.e., optimal interactive choice) or meta-heuristic algorithms (e.g., local search or population-based search). In the fourth, the selection of configurations uses the coverage criteria to assure the quality of product sampling (e.g., source code coverage or *feature interaction coverage*). From now on, we call this last group *t-wise* strategies.

Note that sampling strategies from one group may have the same goal of other groups and, for that reason, provide similar outcomes. For instance, a semi-automatic algorithm that covers all source code will provide a similar set of configurations that a coverage algorithm provides. Alternatively, a greedy automated algorithm that aims at selecting an optimal number of configurations that test all pairs of features individually, provides a similar set of configurations that a 2-wise algorithm provides (Lübke et al., 2019). In the next section, we focus on t-wise strategies since we are interested in investigating sampling strategies that assure product sampling quality.

2.3. T-wise techniques

T-wise techniques select a subset of configurations that covers a valid group of t features being activated and deactivated simultaneously (Henard et al., 2014; Xiang et al., 2021). This subset of configurations should respect constraints in the feature model to be called valid configurations. For instance, an 1-wise algorithm should select a set of configurations that all optional features are active and deactivate at least once. On the other hand, a 2-wise algorithm (also known as pair-wise) should select a set of valid configurations that all pairs of optional features (*F1* and *F2*) are simultaneously activate (*F1* & *F2*), alternatively activate (*F1* & $\sim$*F2* and $\sim$*F1* & *F2*), and mutually deactivated $\sim$*F1* & $\sim$*F2*). Note that the set of configurations may vary depending on the algorithm/strategy used.

To illustrate, consider the feature model of the Elevator configurable system shown in the left-side of Fig. 1. This system has six features: one mandatory (*Base*) and five optional (*Empty*, *ExecutiveFloor*, *Overloaded*, *TwoThirdsFull*, and *Weight*). The number of valid configurations is 20. In the right-side of Fig. 1, we see the selected configurations to fulfill the requirements of 1-, 2-, 3-, and 4-wise using the *Chvatal* (Johansen et al., 2012b) strategy (see details of the testing strategy in Section 4.2). For instance, to fulfill 1- and 2-wise requirements, this strategy selects three and six different configurations, respectively.

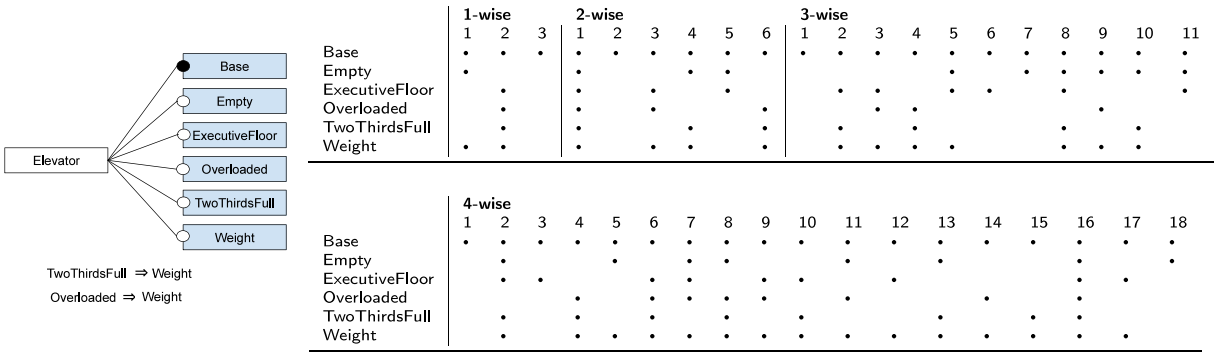


Fig. 1. Showing the feature model and suggested configurations using a 1-, 2-, 3-, and 4-wise strategies, respecting feature model constraints for Elevator configurable system.

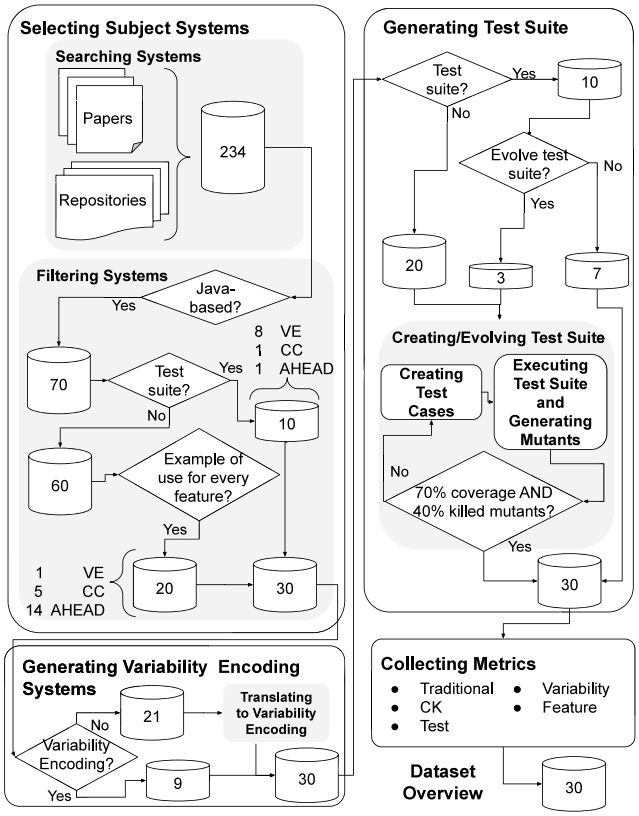


Fig. 2. Dataset creation process.

3. Dataset of configurable systems

In a previous work (Ferreira et al., 2020a), we proposed a test-enriched dataset of configurable systems. This dataset has been used by the community to investigate the testing challenges in configurable systems (Ferreira et al., 2020b). This section briefly describes the creation of such a dataset, since it is used in the empirical study presented in this paper. Next, we describe the four tasks on the dataset creation as well as the dataset itself. Fig. 2 presents the general overview of this process.

Selecting Subject Systems. We searched for configurable systems on six survey papers about testing configurable systems (Engström and Runeson, 2011; Lamancha et al., 2013; Lee et al., 2012; Lopez-Herrejon et al., 2015; Machado et al., 2014; da Mota et al., 2011) and on all primary studies found on them. In addition, we included configurable systems of three well-known repositories of configurable systems: SPL2go (SPL2go, 2020), SPL Repository (Vale et al., 2015), and ESPLA Catalog (Martinez et al., 2017). At the end of this initial search, we found 234 configurable systems. Then, given tool constraints, we limited our dataset to configurable systems developed in a Java-based programming language (i.e., in Java with variability encoding, conditional compilation, or AHEAD). As a result, only 70 configurable systems remain in our dataset. By analyzing these 70 systems, we noted that only 10 of them have a test suite available (8 developed with variability encoding, 1 with conditional compilation, and 1 with AHEAD). Consequently, only these 10 systems would compose our dataset.

Aiming at increasing the number of systems in our dataset, we looked at the remaining 60 configurable systems developed in a Java-based programming language without a test suite. Considering that we need a deep understanding of the systems as well as their features to develop a test suite, we looked at each system's documentation searching for that information. Once we found an example of use for each available feature of a target configurable system, we included it into our dataset. As an outcome of this analysis, we selected 20 additional configurable systems (i.e., 1 with variability encoding, 5 with conditional compilation, and 14 in AHEAD). At the end, we selected 30 configurable systems to our dataset.

Generating Variability Encoding Systems. Aiming at facilitating our testing suite execution, we translated the 6 conditional compilation and 15 AHEAD systems into variability encoding systems. For conditional compilation systems, we manually converted the compilation directives into variability encoding. For instance, ArgoUML-SPL (Couto et al., 2011) relies on conditional compilation and use JavaPP (JavaPP, 2021), since Java does not provide such support. As the translation is one-to-one (e.g., `#ifdef(FEATURE)` into `if(Configuration.FEATURE)`), chances of manual errors are minimized. For AHEAD systems, we relied on FeatureIDE to automatically translate AHEAD code into a variability encoding code.

Generating Test Suites. For the 20 systems without a test suite, we created the test suite from scratch. Aiming at enriching discussions and test suite coverage, we extended the test suite of other three systems. The creation/extension of the test suites consisted of creating testing cases, executing the test suite with mutants, and checking if the stop criteria was satisfied, as we explain next.

Creating test cases. We used JUnit framework (JUnit, 2020), *FEST* (FEST, 2020), and Mockito (Mockito, 2020) for creating testing cases, writing tests for systems with graphical user interfaces, and creating mock objects which simplifies the development of tests for classes with external dependencies, respectively. To design a test case, we checked which features should be activated

and placed this condition as a sentinel in the test case. Therefore, only tests related to active features would run.

Executing mutation testing. For each subject system, we generated a single configuration containing as many active features as possible. Then, we used PIT (Coles et al., 2016) to generate mutants on each configuration and execute the respective test suite against them. Based on the PIT's execution report, we decided whether it was necessary to create more test cases.

Checking stop criteria fulfillment. We used JaCoCo (JaCoCo, 2020) to retrieve code coverage, we created testing cases until we achieved for each subject system: (i) 70% of testing coverage (Inozemtseva and Holmes, 2014; Ivanković et al., 2019) and (ii) 40% of the mutants killed (Just et al., 2014).

Collecting Metrics. Aiming for a better overview of the subject configurable systems in our dataset, we collected a set of 14 software metrics. These metrics include traditional, CK, test, variability, and feature measurements. With Metrics (Metrics, 2020) and CK tool (CK, 2020), we computed traditional and CK metrics (e.g., number of lines of code and number of packages). With FeatureIDE, we extracted metrics related to variability (e.g., the number of features and valid configurations). As mentioned, with JaCoCo (JaCoCo, 2020) and PIT (Coles et al., 2016), we retrieved metrics related to the test suite. To collect metrics related to each feature, we wrote our own script. For each system, our script searches for code snippets containing specific variability statements. Then, for each feature, it computed the number of classes, methods, constructors, and lines of code containing variability statements. We increased internal validity of our study through manually checking our script outcomes.

Dataset Overview. The 30 configurable systems of our dataset belong to several domains such as games, text editor, media management, and file compression. In Table 1, we present an overview of these systems divided into size, variability, and test suite measures. Additional information of the dataset is available in our supplementary Website (Ferreira et al., 2020c).

Size measures. We selected systems from a large variation of sizes regarding their number of lines of code (*#LoC*), packages (*#Pack.*), classes (*#Clas.*), and methods (*#Meth.*). For instance, subject configurable systems vary from 189 lines of code (BankAccount) to more than 150 000 lines of code (ArgoUML-SPL). Similarly, while IntegerSetSPL has only 3 classes, ArgoUML-SPL has almost 2000 classes.

Variability measures. We selected systems with different variability. For instance, while CheckStyle has 141 features, Chess, TaskObserver, and Telecom have only three features (*#Features*). We can also see a similar variation in the number of valid configurations (*#VConf.*). For instance, while Elevator has 20 valid configurations, FeatureAMP3 has 20 500 valid configurations.

Test suite measures. In the *#FTC* column, we see the number of test cases for each system of our dataset and the percentage increased compared to the initial number of test cases (*#ITC* column). Note that the number of lines of testing code varies from 207 for FeatureAMP6 to 17 014 for ArgoUML-SPL. As expected, smaller systems often have fewer lines of testing code and test cases. At the end, our dataset has a total of 3182 test cases of which we created 727 test cases for 20 configurable systems, 90 tests for the three systems we extended test suites and, the remaining 2 365 test cases come from systems that already had test suite.

4. Study settings

This section describes the experimental procedures of our study. Section 4.1 presents our goal and research questions. Section 4.2 describes subject testing strategies. Sections 4.3 and 4.4 describe how we acquire data and operationalize the answer of research questions, respectively.

4.1. Goal and research questions

Based on the goal question metric (GQM) template (Basili and Rombach, 1988), we systematically defined our goal. We ***analyze*** sixteen testing strategies and the dispersion of faults over systems, classes, and features found on configurations suggested by these strategies; ***for the purpose*** of (i) identifying the fastest, most comprehensive, most effective, and most efficient and (ii) deeply understanding feature interaction faults; ***with respect*** to support practitioners and researchers to choose the most appropriate strategy to fulfill their needs and improve existing testing strategies; ***from the viewpoint*** of researchers and software developers with expertise in software testing ***in the context*** of a previously proposed dataset of configurable software systems with test suite available.

Previous work (Engström and Runeson, 2011; Lopez-Herrejon et al., 2015; Machado et al., 2014) reported a lack of empirical evaluation based on a community-wide dataset to guide the comparison of different testing strategies. Motivated by that, we formulate the following research questions.

RQ$_1$: *What are the fastest strategies for testing configurable systems?*

RQ1.1: *Which are the fastest testing strategies on generating a list of configurations?*

RQ1.2: *Which configurations suggested by testing strategies do execute faster?*

RQ$_2$: *Which testing strategies do suggest a list of configurations that covers most configurations in configurable systems?*

RQ$_3$: *Which testing strategies are more effective on finding faults in configurable systems?*

RQ$_4$: *Which testing strategies are more time-efficient on finding faults in configurable systems?*

RQ$_5$: *Which testing strategies are more coverage-efficient on finding faults in configurable systems?*

These research questions show which testing strategies are faster, more comprehensive, more effective, and more efficient. This comparison with a community-wide dataset may benefit software testers because, from now on, they have results for several configurable systems to support their choice of a testing strategy that best fits their needs. Note that in the first two research questions, we provide a broader view of the configurable systems in the subject dataset. In the last three research questions, we look only at faulty systems.

By comparing testing strategies, we may find several faults. A great understanding of these faults may benefit both practitioners and researchers because they can (i) learn patterns from previous faults, (ii) use these patterns to avoid the emergence of similar faults, and (iii) improve existing testing strategies. Aiming at investigating the dispersion of faults over classes and features, we formulate the following two research questions.

RQ$_6$: *What are the characteristics of the fault-prone classes of the configurable systems?*

RQ$_7$: *What are the characteristics of the fault-prone features of the configurable systems?*

4.2. Selected testing strategies

We focus on t-wise testing strategies because (i) they ensure certain quality of the set of suggested configurations (see Sections 2.2 and 2.3), and (ii) the literature lacks a comparison among them on a wide-community dataset (Engström and Runeson, 2011; Lopez-Herrejon et al., 2015; Machado et al., 2014). We are confident that we selected well-known testing strategies, once all selected strategies are developed in the FeatureIDE - an integrated development environment (IDE) widely used by developers of configurable systems (Thüm et al., 2014). Some

Table 1
Dataset measurements overview.

System name	Size measures				Variability measures				Test suite measures					
	#LoC	#Pack.	#Clas.	#Meth.	#Feat.	#VConf.	#Var.	Tech.	#ITC	#FTC	#LoTC	%Cov.	%KM	TS
ATM Santos et al. (2016)	1 160	2	27	100	7	80	44	CC	0	76(100%)	1 371	91%	79%	C
ArgoUML-SPL Couto et al. (2011)	153 977	92	1812	13 034	8	256	1388	CC	1326	1 326(0%)	17 014	17%	9%	O
BankAccount SPL2go (2020)	189	3	9	22	10	144	13	A	0	42(100%)	539	92%	62%	C
CheckStyle Wong et al. (2018)	61 435	14	78	719	141	$>2^{135}$	180	VE	719	719(0%)	13 606	38%	5%	O
Chess Santos et al. (2016)	2 149	7	22	162	3	8	20	CC	0	77(100%)	1 296	72%	72%	C
Companies Souto et al. (2017)	2 477	16	50	244	10	192	255	VE	42	42(0%)	1 850	70%	46%	O
Elevator Meinicke et al. (2014)	426	2	7	59	5	20	9	VE	0	59(100%)	683	92%	73%	C
Email Souto et al. (2017)	429	3	7	49	8	40	30	VE	30	85(65%)	1 429	97%	61%	E
FeatureAMP1 SPL2go (2020)	1 350	4	15	93	28	6732	40	A	0	18(100%)	977	85%	46%	C
FeatureAMP2 SPL2go (2020)	2 033	3	14	167	34	7020	55	A	0	18(100%)	698	72%	43%	C
FeatureAMP3 SPL2go (2020)	2 575	8	16	223	27	20 500	93	A	0	15(100%)	725	77%	42%	C
FeatureAMP4 SPL2go (2020)	2 147	2	57	203	27	6732	57	A	0	12(100%)	622	82%	40%	C
FeatureAMP5 SPL2go (2020)	1 344	3	9	895	29	3810	36	A	0	17(100%)	730	91%	49%	C
FeatureAMP6 SPL2go (2020)	2 418	8	30	202	38	21 522	76	A	0	9(100%)	207	31%	43%	C
FeatureAMP7 SPL2go (2020)	5 644	3	46	220	29	15 795	57	A	0	8(100%)	180	28%	40%	C
FeatureAMP8 SPL2go (2020)	2 376	2	6	106	27	15 708	48	A	0	78(100%)	1 637	82%	42%	C
FeatureAMP9 SPL2go (2020)	1 859	3	8	134	24	6732	53	A	0	105(100%)	1 975	83%	63%	C
GPL Souto et al. (2017)	1 235	3	17	78	13	73	59	VE	45	51(12%)	1 162	83%	60%	E
IntegerSetSPL SPL2go (2020)	200	2	3	20	3	2	7	A	0	19(100%)	286	100%	80%	C
Jtopas Souto et al. (2017)	4 397	7	43	472	5	32	10	VE	87	87(0%)	6 703	67%	50%	O
MinePump SPL2go (2020)	244	2	7	26	7	64	4	A	0	34(100%)	459	91%	65%	C
Notepad Souto et al. (2017)	1 564	4	17	90	17	256	24	VE	25	25(0%)	1 790	59%	15%	O
Paycard SPL2go (2020)	374	2	8	27	4	6	10	CC	0	13(100%)	453	88%	61%	C
Prop4J SPL2go (2020)	1 138	2	15	90	17	5029	17	A	63	63(0%)	504	71%	67%	O
Sudoku Souto et al. (2017)	949	2	13	51	6	20	53	VE	6	35(82%)	650	80%	67%	E
TaskObserver Santos et al. (2016)	486	2	10	33	4	8	9	CC	0	24(100%)	280	91%	71%	C
Telecom Santos et al. (2016)	273	2	40	11	3	4	6	CC	0	26(100%)	391	99%	65%	C
UnionFindSPL SPL2go (2020)	335	2	36	5	13	10	12	A	0	40(100%)	616	84%	66%	C
VendingMachine Martinez et al. (2017)	472	2	7	21	8	256	7	CC	0	37(100%)	297	97%	83%	C
ZipMe Souto et al. (2017)	4 647	3	311	33	13	24	343	VE	22	22(0%)	703	41%	19%	O

#LoC: Number of lines of code, ***#Pack.***: Number of packages, ***#Clas.***: Number of classes, ***#Meth.***: Number of methods, ***#Feat.***: Number of features, ***#VConf.***: Number of valid configurations, ***#Var.***: Number of occurrences of variability in the source code, ***Tech.***: Variability technique being possible variability encoding (VE), AHEAD (A), and conditional compilation (CC), ***#ITC***: Initial number of test cases, ***#FTC***: Final number of test cases, ***#LoTC***: Lines of testing code, ***%Cov.***: Percentage of test suite coverage, ***%KM***: Percentage of killed mutants, ***TS***: Test suite being created by us (C), extended by us (E), or original (O).

strategies like **Chvatal** presents versions for 1-, 2-, 3-, and 4-wise. However, other strategies, such as **IncLing**, are only available for 2-wise tests. We investigate all testing strategies versions available in FeatureIDE. The constraint to 4-wise is due to the number of suggested configurations. That is, the running time would increase significantly and make our study unfeasible.

At the end, we compared 16 t-wise strategies and two baselines. The baselines are mainly important to identify faults in the subject systems. For short, we selected *brute force* (baseline 1), *random* (baseline 2), four variations of *CASA*, four variations of *Chvatal*, three variations of *ICPL*, one variation of *IncLing*, and four variations of *YASA*. Next, we briefly present all testing strategies investigated.

Brute Force (baseline 1) (Al-Hajjaji et al., 2017; Thüm et al., 2014) is a strategy that generates all distinct valid configurations. With this strategy, it is possible to generate all or a fixed number of valid configurations for a configurable system.

Random (baseline 2) (Al-Hajjaji et al., 2017; Thüm et al., 2014) is a SAT solver-based strategy that randomly generates a pre-defined number of valid configurations.

CASA (Garvin et al., 2011) is a greedy algorithm for sampling test configurations (more specifically, a simulated annealing algorithm). It works on two iterated steps. First, it minimizes the number of created configurations. Second, it ensures that a certain degree of coverage is achieved. We use *CASA* versions 1-, 2-, 3-, and 4-wise.

Chvatal (Johansen et al., 2011) is an adaptation of a greedy algorithm proposed by *Chvatal* (Chvatal, 1979) to solve the covering array problem. At the end, it is a heuristic that selects a subset of possible configurations with a t-wise covering array. We use *Chvatal* versions 1-, 2-, 3-, and 4-wise.

ICPL (Johansen et al., 2012b) is an algorithm for t-wise covering arrays. This strategy is also based on the *Chvatal* algorithm (Chvatal, 1979). However, it contains optimizations for increasing its performance. We use *ICPL* versions 1-, 2-, and 3-wise.

IncLing (Al-Hajjaji et al., 2016) is an incremental sampling for 2-wise (i.e., pair-wise) interaction testing. The main difference between *IncLing* and other testing strategies is that *IncLing* generates configurations one at a time to enhance sampling efficiency in terms of interaction coverage rate.

YASA (Krieter et al., 2020) is based on the traditional *IPOG* algorithm (Lei et al., 2008; Yu et al., 2013), which starts with a given empty sample and then iterates over all t-wise once at the time. Through the application of different heuristics and caching methods, this testing strategy, in theory, improves its sampling time compared to other t-wise sampling strategies. We use *YASA* versions 1-, 2-, 3-, and 4-wise.

4.3. Data acquisition

Our data acquisition consists basically of three tasks: (i) run the testing strategies presented in Section 4.2 for each configurable system in the subject dataset (Section 3), (ii) extract information from logs creating a set of true faults (reference list), and (iii) collect metrics. In what follows, we give details on how we automated these tasks.

Running Testing Strategies. Once we run a testing strategy, it returns a list of configurations. Hence, for each configuration, we selected the set of features of the target configuration, ran the testing suite related to it, and analyzed the outcome logging. As it is a time-consuming and error-prone task, we created a script to automate this task. Considering time constraints, we defined

an upper threshold of 250 configurations per subject system and testing strategy.

Creating a Reference List. We analyzed the execution log of the 18 (16 t-wise strategies and two baselines) testing strategies against the 30 subject systems. After parsing the log, we found test cases of which a fault emerged as well as the reason why it happened. Hence, we investigated each reported fault to confirm that it is truly a fault and if it arose due to a feature interaction. The reference list is the union of all faults found on all configurations suggested by all testing strategies investigated in this study.

Metrics Collection. We use different tools to extract metrics used in our study. Once we did not find tools to compute metrics used to answer our research questions, we computed them with our scripts (see Section 3).

Our analysis scripts (written in Java) are open-source. All data necessary for replicating this study are stored in csv files. All tools, links to subject projects, reports of faults for each testing strategy, and data used in this study are available at our supplementary Website (Ferreira et al., 2020c).

4.4. Operationalization

For the first five research questions, we defined the following null and alternative hypotheses.

(H0) *All testing strategies present similar results.*

(H1) *At least one testing strategy differ from the others.*

To perform a normality test, we used the Shapiro–Wilk method. As a result, this test failed in all cases indicating that our data do not follow a normal distribution. Taking this information into account, we used Friedman's Test to verify the hypotheses formulated in our study since it is used for one-way repeated measure analysis of variance by ranks. For short, this test detects differences in treatments across multiple test attempts (Sheskin, 2020). In our case, it is similar to the Kruskal–Wallis one-way analysis of variance by ranks. When it was possible to reject the null hypothesis, we used Post Hoc Analysis to identify which subject testing strategies tend to statistically differ from the others.

Next, we detail how we answered each research question. For the first five research questions, the independent variables are the 16 t-wise testing strategies.

Answering RQ_1. To answer RQ_1, we compute the time (in seconds) to generate configurations (RQ1.1) and the time (in seconds) to run the test suite for the configurations suggested by each subject testing strategy (RQ1.2). In summary, number of seconds (*#Seconds*) consists of the sum of the time to generate configurations and to execute the suggested configurations. To mitigate random effects on time measurement, we use the average of seconds, performing these analyses ten times. We use a computer with 16 GB of RAM, i7 processor 3.60 GHz, Windows 10, and JVM with 2 GB of memory. The lower the *#Seconds*, the faster the testing strategy is. We report results for each configurable system and show which strategies performed better for a greater number of configurable systems. In RQ_1, *#Seconds* is our dependent variable.

Answering RQ_2. To answer RQ_2, we compute the percentage of the number of configurations reported by a testing strategy (*#Configurations*) over the number of valid configurations for each configurable system. The higher the percentage, the higher the testing strategy coverage. We report results for each configurable system and also show which testing strategies are more comprehensive for a greater number of configurable systems. In RQ_2, *#Configurations* is our dependent variable.

Answering RQ_3. To answer RQ_3, we use *recall* (Eq. (1)). In our context, recall is the number of correct faults found by the configurations suggested by a target testing strategy (i.e., true positive faults) divided by the number of existing faults in our reference list (i.e., the sum of true positive and false negative faults). The higher the recall, the more effective the testing strategy is. We report the recall calculated for each testing strategy over each configurable system from which at least one feature interaction fault exists. In RQ_3, *recall* is our dependent variable.

$$Recall = \frac{TP\ faults}{(TP\ faults + FN\ faults)} \tag{1}$$

Answering RQ_4. To answer RQ_4, we compute *time-efficiency* (Eq. (2)). Time-efficiency is the number of correct faults found by the configurations suggested by a target strategy divided by *#Seconds* used to answer RQ_1. The higher the *time-efficiency*, the more efficient the strategies. We report results for each configurable system and show which strategies performed better for the higher number of systems. In RQ_4, *time-efficiency* is our dependent variable.

$$TimeEfficiency = \frac{TP\ faults}{\#Seconds} \tag{2}$$

Answering RQ_5. To answer RQ_5, we compute *coverage-efficiency* (Eq. (3)). Coverage-efficiency is the number of faults found by the configurations suggested by a target strategy divided by *#Configurations* used to answer RQ_2. The higher the *coverage-efficiency*, the more efficient the testing strategy. We report results for each configurable system and show which testing strategies performed better for a higher number of configurable systems. In RQ_5, *coverage-efficiency* is our dependent variable.

$$CoverageEfficiency = \frac{TP\ faults}{\#Configurations} \tag{3}$$

Answering RQ_6. To answer RQ_6, we first retrieve a list of all classes that failed and discuss their dispersion over each subject configurable system. Then, aiming at discovering whether these faulty classes have distinct characteristics from other classes, we compute traditional and CK metrics for each class of each subject system (see Section 3). After, we compute the Spearman's rank correlation to see whether faulty classes are often larger and more complex, for instance. Spearman's rank correlation is adequate to this analysis since the measures of our classes represent continuous values and do not follow a normal distribution.

Answering RQ_7. To answer RQ_7, we did a similar analysis to answer RQ_6. The main difference is that instead of investigating characteristics of faulty classes, we investigate characteristics of faulty features. Naturally, we use metrics related to features, such as the number of classes and methods a feature is located (scattering). Spearman's rank correlation is adequate to this analysis since the measures of our features represent continuous values and do not follow a normal distribution. We detail the used metrics when answering the research question.

5. Results

This section presents the results of our study structured according to our research questions. Note that the first two subsections are simply measuring time and coverage of subject testing strategies. Later subsections present more sophisticated analyses. The idea behind these analyses is offering practitioners a broader practical view of how each testing strategy behaves depending on the system characteristics (e.g., number of lines of code, configurations, and test cases).

5.1. The fastest testing strategies (RQ1)

In Table 2, we present the sum of the time to generate configurations (RQ1.1) and execute the test suite (RQ1.2) for all

Table 2
Total time in seconds spent by the target testing strategies.

(a) Time spent in seconds by the 1- and 2-wise testing strategies

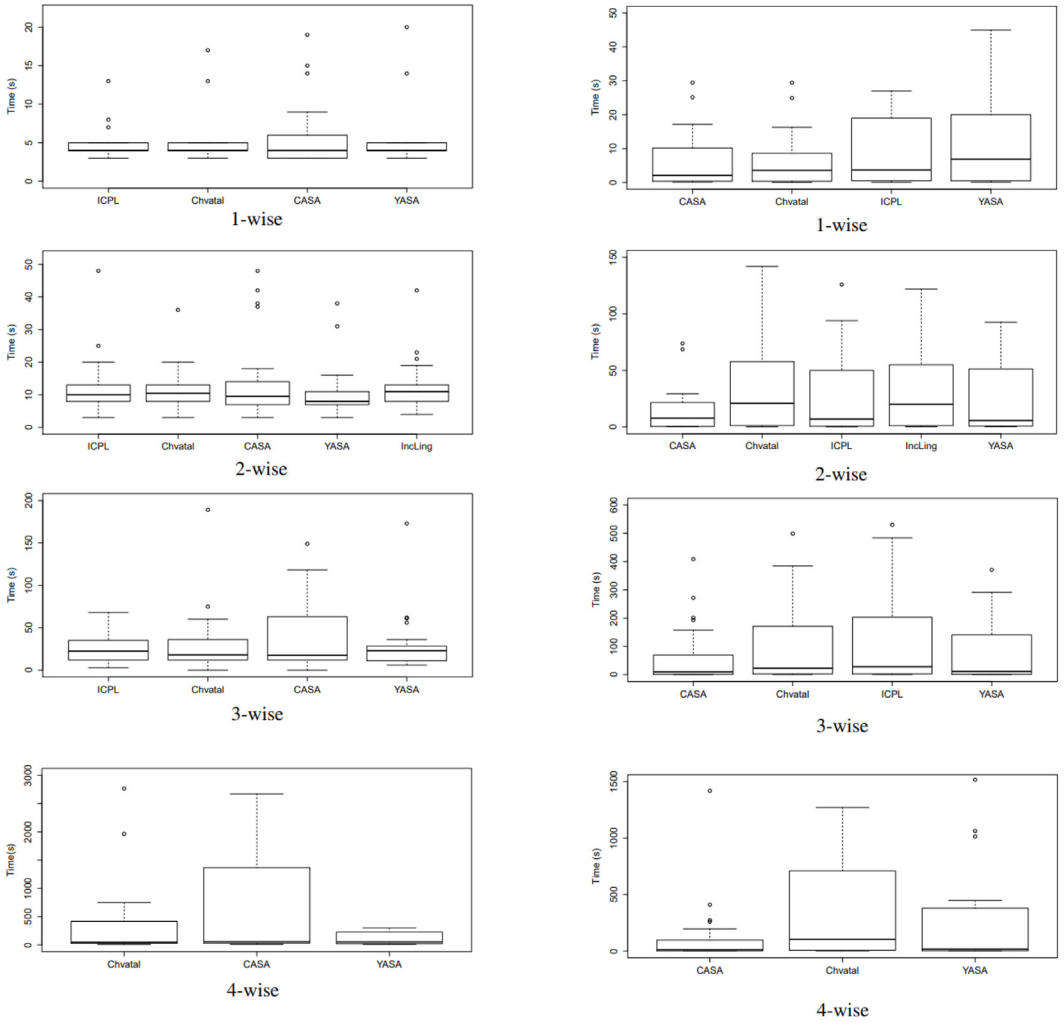
Name	Baseline1	Baseline2	CASA-T1	Chvatal-T1	ICPL-T1	YASA-T1	CASA-T2	Chvatal-T2	ICPL-T2	IncLing-T2	YASA-T2
ArgoUML-SPL	2 376.00	3 336.00	4.50	15.00	31.00	8.48	8.00	41.00	38.00	27.00	17.88
ATM	532.87	700.33	8.27	28.59	8.48	11.73	27.55	33.85	14.48	30.01	32.92
BankAccount	142.75	207.06	3.33	4.12	3.06	3.33	7.50	8.24	9.06	10.25	8.38
CheckStyle	1 007.00	1 048.00	31.16	36.00	12.00	15.26	116.65	118.00	65.00	78.00	66.93
Chess	21.23	259.23	7.92	11.17	5.61	8.88	14.65	12.13	7.61	35.00	9.87
Companies	94.41	264.41	5.44	4.51	5.52	4.03	15.14	9.26	9.52	12.30	8.86
Elevator	9.40	243.40	3.39	4.10	5.10	4.38	6.40	7.20	7.10	8.20	6.36
Email	172.00	470.00	9.69	14.00	12.00	11.68	36.25	23.00	16.00	319.00	17.00
FeatureAMP1	2 410.00	2 629.00	4.14	5.90	222.68	4.13	17.20	31.00	228.68	231.30	16.68
FeatureAMP2	5 234.00	5 426.00	14.23	28.00	25.65	11.68	21.14	92.00	32.65	213.95	76.38
FeatureAMP3	1 155.00	789.00	15.65	25.00	17.92	6.97	25.65	68.00	32.92	40.00	65.67
FeatureAMP4	2 718.00	2 427.00	33.45	15.00	25.24	33.45	39.45	109.00	32.24	57.00	226.36
FeatureAMP5	7 470.00	6 809.00	15.39	66.00	59.00	20.31	23.65	155.00	68.00	133.00	104.56
FeatureAMP6	781.00	920.00	18.87	12.00	12.00	9.56	63.67	39.00	23.00	57.00	20.91
FeatureAMP7	660.00	234.30	9.32	4.50	4.70	4.29	12.34	13.70	11.70	14.00	10.37
FeatureAMP8	1 998.00	2 161.00	20.12	48.00	17.00	17.68	39.26	45.00	24.00	55.00	71.40
FeatureAMP9	1 975.00	2 534.00	6.44	23.00	24.00	27.90	82.91	87.00	33.00	103.00	66.72
GPL	41.00	102.00	7.38	5.50	5.70	5.43	14.55	16.20	13.70	18.20	12.71
IntegerSetSPL	4.00	84.00	3.36	3.10	3.10	3.34	3.41	3.10	3.10	4.20	3.40
Jtopas	22.00	91.00	3.81	3.70	3.50	4.34	6.75	7.70	7.50	6.80	7.30
MinePump	666.00	703.00	3.44	3.40	3.50	3.37	6.44	10.00	8.50	9.20	6.49
Notepad	22 460.00	22 557.00	201.30	299.00	235.00	197.34	450.00	1646.00	241.00	1077.00	721.92
Paycard	442.00	539.00	168.48	68.45	72.00	68.58	171.48	207.31	74.00	172.29	171.36
Prop4J	11.00	108.00	3.38	4.30	3.20	4.40	10.44	15.40	11.20	12.80	8.49
Sudoku	14.80	103.40	3.50	5.00	3.80	4.53	7.80	8.70	7.80	9.80	7.92
TaskObserver	77.00	778.00	23.80	25.00	25.00	24.80	45.50	82.70	60.00	88.00	39.72
Telecom	22.00	778.00	16.77	20.00	9.90	16.73	20.73	27.00	27.90	17.00	40.70
UnionFindSPL	10.90	103.80	9.80	11.80	10.70	10.20	19.54	12.80	8.70	17.90	11.79
VendingMachine	110.40	103.80	6.00	4.40	4.40	6.32	8.10	15.80	10.40	10.70	12.25
ZipMe	15.00	99.00	4.20	4.90	4.80	5.32	7.30	11.00	10.80	7.70	7.37

(b) Time spent in seconds by the 3- and 4-wise testing strategies

Name	Baseline1	Baseline2	CASA-T3	Chvatal-T3	ICPL-T3	YASA-T3	CASA-T4	Chvatal-T4	YASA-T4
ArgoUML-SPL	2 376.00	3 336.00	27.84	66.00	39.00	18.77	63.50	153.00	89.19
ATM	532.87	700.33	39.08	122.27	45.08	65.62	125.53	171.55	146.55
BankAccount	142.75	207.06	17.52	19.42	18.18	16.52	49.40	38.25	35.00
CheckStyle	1 007.00	1 048.00	3 487.00	364.08	2 869.00	368.76	[b]	[b]	18 691.00
Chess	21.23	259.23	23.87	24.32	19.43	14.83	17.00	24.01	16.00
Companies	94.41	264.41	19.29	23.13	20.16	20.18	43.38	53.84	40.20
Elevator	9.40	243.40	12.41	12.30	12.30	10.48	19.54	17.50	22.49
Email	172.00	470.00	47.92	70.00	272.00	26.93	58.88	1 012.00	42.28
FeatureAMP1	2 410.00	2 629.00	194.00	55.40	265.00	158.67	1 304.00	1 791.00	1 226.22
FeatureAMP2	5 234.00	5 426.00	461.42	420.00	111.32	398.84	1 235.01	1 249.00	1 267.60
FeatureAMP3	1 155.00	789.00	214.42	283.00	97.00	175.45	1 379.63	2 614.00	705.00
FeatureAMP4	2 718.00	2 427.00	44.45	389.00	127.00	245.27	2 681.39	3 758.00	614.35
FeatureAMP5	7 470.00	6 809.00	312.91	540.00	163.00	327.04	1 223.77	2 222.00	1 752.37
FeatureAMP6	781.00	920.00	2759.71	279.00	119.00	135.48	28 863.25	6 881.00	526.40
FeatureAMP7	660.00	234.30	95.41	45.20	37.70	35.42	28 800.42	751.50	272.66
FeatureAMP8	1 998.00	2 161.00	62.87	145.00	64.00	182.69	28 897.42	1 105.00	584.86
FeatureAMP9	1 975.00	2 534.00	306.36	321.00	33.00	177.15	28 963.06	1 088.00	706.68
GPL	41.00	102.00	29.18	34.70	40.50	29.32	98.85	71.40	60.64
IntegerSetSPL	4.00	84.00	3.41	3.10	3.10	[c]	[a]	[a]	[a]
Jtopas	22.00	91.00	9.78	11.40	11.70	10.89	13.70	22.30	16.01
MinePump	666.00	703.00	10.55	39.00	12.60	11.53	29.38	149.00	21.81
Notepad	22 460.00	22 557.00	496.00	2149.00	836.00	3 008.98	9 782.11	5 234.00	3 613.55
Paycard	442.00	539.00	207.40	207.34	171.01	206.51	203.28	205.90	400.11
Prop4J	11.00	108.00	38.50	30.80	30.50	24.66	249.76	37.20	63.01
Sudoku	14.80	103.40	13.80	16.00	13.80	14.25	19.10	21.00	20.48
TaskObserver	77.00	778.00	84.21	107.00	114.00	72.13	85.65	101.00	76.16
Telecom	22.00	778.00	22.68	33.20	45.00	[c]	[a]	[a]	[a]
UnionFindSPL	10.90	103.80	12.61	12.80	12.90	14.45	17.69	14.20	14.55
VendingMachine	110.40	103.00	25.30	16.20	16.70	25.30	45.22	41.90	47.70
ZipMe	15.00	99.00	13.40	14.70	13.70	11.45	33.40	28.00	21.63

[a]Number of features of target configurable systems is less than t.
[b]Could not generate configurations, time greater than 8 h.
[c]The strategy did not generate any configuration.

configurations suggested by each subject testing strategy. We highlight the time of the fastest testing strategy for each configurable system according to the t-wise group. Once multiple strategies have the same shortest time, we consider those as the fastest ones. Note that the time to generate configurations and execute these configurations varied from 3 to 28 963 s (≈ 8 h) and in general, 1- and 4-wise strategies were respectively faster and slower than the other testing strategies. In addition, in most cases,



(a) Time spent to generate the list of configurations

(b) Time spent running the test configurations

Fig. 3. Time spent to generate the list of configurations.

the time to generate the configurations were far greater than the time to execute the configurations. As an extreme example, for the FeatureAMP9 using *CASA-T4*, it was necessary 28 800 s to generate the configurations and 163 s to execute the test suite for the suggested configurations.

As a result, we can see that for the 1-wise group, *ICPL-T1* is faster than the other testing strategies for ten configurable systems. *CASA-T1* and *YASA-T1* are the fastest for nine configurable systems. For the 2-wise group, *ICPL-T2* is faster than the other testing strategies for 12 configurable systems. *CASA-T2* is the fastest for ten configurable systems. *YASA-T2* is the fastest for seven configurable systems. For the 3-wise group, *ICPL-T3* is faster than the other testing strategies for 11 configurable systems. *CASA-T3* is the fastest for nine configurable systems. *YASA-T3* is the fastest for eight configurable systems. For the 4-wise group, *YASA-T4* is faster than the other testing strategies for fourteen configurable systems. *CASA-T4* is the fastest for eight configurable systems. *Chavatal-T4* is the fastest for five configurable systems. *YASA-T4* was the only strategy that was able to generate a list of configurations for CheckStyle. This configurable system has 141 features, which lead to 2^{135} valid configurations.

RQ1.1. Comparing the time to generate the list of configurations. In Fig. 3a, we present boxplots with the time taken to generate the list of configurations for each testing strategy organized according to the t-wise group. To improve visualization, we removed outliers. The horizontal axis presents the testing strategy, while the vertical axis presents the time spent in seconds to generate the list of configurations. For instance, in Fig. 3a '*1-wise*', we show the results for 1-wise group. As expected, t-wise testing strategies with smaller '*t*' took less time than strategies with larger '*t*'. For instance, while 1-wise testing strategies took around 4 s, 4-wise testing strategies took around 136 s.

Table 3
P-value for RQ1 (time to generate and execute configurations)

RQ	1-wise	2-wise	3-wise	4-wise
RQ1.1	0.1754	0.0013	0.3441	0.5273
RQ1.2	0.0795	0.0013	0.0013	0.9636
RQ1	0.0907	0.0003	0.9658	0.1690

In the first row of Table 3, we present the results of the *p-value* for the Friedman's Test grouped according to the t-wise group. As we can see, only the time to generate the list of configurations of 2-wise testing strategies is statistically significant different from the other testing strategies (i.e., $p\text{-value} < 0.05$). It means that for 1-, 3-, and 4-wise strategies, we could not find a statistical difference on the time to generate the configurations. Therefore,

we reject the null hypothesis (H0) and accept the alternative hypothesis (H1) only for the 2-wise testing strategies.

Aiming at finding out which 2-wise testing strategies differ from each other, we used the Post Hoc analysis (see our operationalization in Section 4.4). As a result, *CASA* and *YASA* are the ones that statistically differ for the other 2-wise testing strategies. We noted that *YASA* took less time to generate the configurations for systems with more than 20 features. *CASA*, on the other hand, spent more time for systems with more than 20 features.

RQ1.2. Comparing the time to execute the test suite for the suggested configurations. Similar to Fig. 3a, in Fig. 3b, we show the boxplots with the time taken to execute the test suite for the suggested configuration by each testing strategy organized by t-wise group. In general, *CASA* took less time than the other testing strategies and *Chvatal* took more time than the other testing strategies. The greater exception was on for the 1-wise group, of which *Chvatal* was the fastest testing strategy on running the test suite.

In the second row of Table 3, we present the results of the *p-value* performing the Friedman's Test. We found statistical difference for the 2- and 3-wise groups. Therefore, for these groups, we can reject the null hypothesis (H0) and accept the alternative hypothesis (H1). Regarding the Post Hoc analysis for these two groups, we found that: (i) in the case of 2-wise testing strategies, *CASA*, *ICPL*, and *YASA* are statistically different from the other testing strategies. For the 3-wise group, *CASA* and *YASA* testing strategies are statistically different from the others. In general, *CASA* configurations were faster on running the test suite of the suggested configurations than *ICPL* and *YASA* testing strategies.

RQ1. Considering both the time to generate and execute the suggested configurations. Finally, we performed the analysis for the sum of the time to generate and execute the configurations. In the third row of Table 3, we present the results of the *p-value* for the Friedman's Test grouped according to the t-wise group. As we can see, only the time of 2-wise testing strategies is statistically significant different from the other testing strategies (i.e., $p\text{-value} < 0.05$). As a result of the Post Hoc Analysis for 2-wise group, the time of *IncLing*, *Chvatal*, and *CASA* testing strategies are statistically different from the other testing strategies. .

Implications. Practitioners should look at our results to estimate how long their test suite might take for each testing strategy. To do so, they can compare the time that systems with similar characteristics to theirs last for each testing strategy. For instance, ArgoUML-SPL, with 153 977 lines of code, 256 valid configurations, and with 1326 test cases took around 15 s for 1-wise strategies, 25 s for 2-wise strategies, 35 s for 3-wise strategies, and 90 s for 4-wise strategies

RQ_1 Summary. Although we found statistically significant results only in some cases, data of Table 2 show that, as expected, strategies with smaller '*t*', took less time to generate and execute configurations than strategies with greater '*t*'. Looking the result for the same t-wise group, *ICPL-T1*, *ICPL-T2*, *ICPL-T3*, and *YASA-T4* were faster than the other testing strategies.

5.2. The most comprehensive strategies (RQ_2)

In Table 4, we present the number of valid configurations (*#Conf.*) and the percentage of configurations recommended by each testing strategy and configurable system. We highlight the greatest percentage for each configurable system. Once multiple strategies have the same greatest percentage and it is greater than 0, we consider those as the most comprehensive ones. We included baseline results in Table 4 aiming at fostering discussions (Section 6).

As a result, for 1-wise group, *Chvatal-T1* is the most comprehensive testing strategy for 15 configurable systems. *ICPL-T1*, *YASA-T1*, and *CASA-T1* are the most comprehensive testing strategies for 9, 7 and 4 configurable systems, respectively. In the 2-wise group, *IncLing-T2* is the most comprehensive testing strategy for 15 configurable systems. *ICPL-T2*, *Chvatal-T2*, *CASA-T2*, and *YASA-T2* are the most comprehensive testing strategies for 11, 9, 1 and 1 configurable systems, respectively. For 3-wise group, *Chvatal-T3* is the most comprehensive testing strategy for 20 configurable systems. Next, *ICPL-T3*, and *YASA-T3* are more comprehensive for 9 and 2 configurable systems, respectively. *CASA-T3* did not have better results for any configurable system. For the group 4-wise, *Chvatal-T4* is the most comprehensive testing strategy for 17 configurable systems. After, *CASA-T4*, and *YASA-T4* are the most comprehensive testing strategies for 8 and 3 configurable systems, respectively.

As expected, increasing the '*t*', the number of suggested configurations also increases. For systems with more than 20 features for groups 1- and 2-wise, the percentage of configurations is less than 1% of configurations. For groups 3- and 4-wise, for configurable systems with more than 20 features, the percentage of configurations is less than 3%. For systems with 10 to 20 features for groups 1- and 2-wise, the highest percentage of configurations was for UnionFindSPL through *Chvatal-T2* with 60% of configurations. For 3- and 4-wise groups with systems from 10 to 20 features, the greatest percentage was 100% for UnionFindSPL through *Chvatal-T3* and *Chvatal-T4*.

In three specific scenarios, we were not able to generate the configurations, for IntegerSetSPL, 4-wise testing strategies did not work because that the number of valid configurations was smaller than 4 (see Section 2.3). For Telecom and IntegerSetSPL, *YASA-T3* could not generate any configuration for testing. CheckStyle, *CASA-T4* and *Chvatal-T4* did not generate any configuration after eight hours and we interrupted their execution. It was not possible to reject the null hypothesis for RQ_2. In Table 5, we show the p-values obtained running the Friedman's Test.

Practitioners should look at our results to estimate how comprehensive the test suite of their system is for each testing strategy as well as know the constraints and limitations of testing strategies themselves. For instance, for systems with up to 8 valid configurations, most testing strategies recommend at least half of the configurations. For systems with more than 6000 configurations, testing strategies recommend no more than 2% of the valid configurations. For systems with a very small or very large number of configurations, 3- and 4-wise testing strategies may not work correctly because it did not fulfill the requirements of the t-wise strategies or did not have enough memory to run all configurations, respectively.

RQ_2 Summary. Even though the Friedman's Test does not show that the testing strategies differ in the number of suggested configuration, our data in Table 4 show that *Chvatal* seems the most comprehensive testing strategy for 1-, 3-, and 4-wise testing strategies. Among 2-wise strategies, *IncLing-T2* is likely the most comprehensive testing strategy.

5.3. The most effective strategies (RQ_3)

As we found faults in only 16 configurable systems, it is meaningfulness report recall for all configurable systems of the subject dataset. Hence, in this section, we focus on the 16 faulty systems. In Table 6, we present the number of faults (*#faults*) found and the recall of each testing strategy for each configurable system with faults. To increase readability, we replace 0% to "-". We highlight the most effective testing strategy for each configurable

Table 4
Percentage of configurations analyzed by strategies.

(a) Percentage of configurations analyzed by 1- and 2-wise testing strategies

Name	#Conf.	Baseline1	Baseline2	CASA-T1	Chvatal-T1	ICPL-T1	YASA-T1	CASA-T2	Chvatal-T2	ICPL-T2	IncLing-T2	YASA-T2
ArgoUML-SPL	256	97.66%	97.66%	0.78%	0.78%	0.78%	0.78%	0.78%	2.73%	2.73%	2.73%	0.78%
ATM	80	100.00%	100.00%	2.50%	2.50%	1.25%	2.50%	7.50%	8.75%	8.75%	8.75%	7.50%
BankAccount	144	100.00%	100.00%	1.39%	2.08%	1.39%	1.39%	4.17%	4.86%	4.86%	5.56%	4.17%
CheckStyle	$>2^{135}$	∼0.00%	∼0.00%	∼0.00%	∼0.00%	∼0.00%	∼0.00%	∼0.00%	∼0.00%	∼0.00%	∼0.00%	∼0.00%
Chess	8	87.50%	87.50%	25.00%	25.00%	25.00%	25.00%	50.00%	37.50%	62.50%	50.00%	50.00%
Companies	192	100.00%	100.00%	2.08%	2.08%	2.08%	1.04%	6.25%	6.77%	6.77%	6.77%	6.25%
Elevator	20	100.00%	100.00%	10.00%	15.00%	10.00%	15.00%	25.00%	30.00%	35.00%	30.00%	25.00%
Email	40	100.00%	100.00%	5.00%	7.50%	5.00%	5.00%	15.00%	15.00%	15.00%	17.50%	15.00%
FeatureAMP1	6732	3.71%	3.71%	0.03%	0.04%	0.04%	0.03%	0.12%	0.16%	0.15%	0.21%	0.12%
FeatureAMP2	7020	3.56%	3.56%	0.03%	0.04%	0.04%	0.04%	0.03%	0.17%	0.16%	0.19%	0.04%
FeatureAMP3	20 500	1.22%	1.22%	0.01%	0.01%	0.01%	0.01%	0.01%	0.07%	0.08%	0.10%	0.01%
FeatureAMP4	6732	3.71%	3.71%	0.03%	0.04%	0.04%	0.03%	0.12%	0.15%	0.16%	0.15%	0.12%
FeatureAMP5	3810	6.56%	6.56%	0.05%	0.08%	0.08%	0.05%	0.26%	0.31%	0.31%	0.29%	0.26%
FeatureAMP6	21 522	1.16%	1.16%	0.02%	0.01%	0.01%	0.01%	0.07%	0.06%	0.05%	0.10%	0.07%
FeatureAMP7	15 795	1.58%	1.58%	0.02%	0.02%	0.02%	0.02%	0.08%	0.08%	0.09%	0.09%	0.08%
FeatureAMP8	15 708	1.59%	1.59%	0.01%	0.02%	0.02%	0.01%	0.07%	0.07%	0.07%	0.07%	0.07%
FeatureAMP9	6732	3.71%	3.71%	0.03%	0.04%	0.04%	0.03%	0.13%	0.18%	0.16%	0.18%	0.13%
GPL	73	100.00%	100.00%	5.48%	5.48%	6.85%	8.22%	17.81%	17.81%	23.29%	19.18%	17.81%
IntegerSetSPL	2	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
Jtopas	32	100.00%	100.00%	6.25%	9.38%	6.25%	6.25%	18.75%	18.75%	18.75%	18.75%	18.75%
MinePump	64	100.00%	100.00%	3.13%	4.69%	3.13%	3.13%	10.94%	10.94%	12.50%	10.94%	10.94%
Notepad	256	97.66%	97.66%	1.17%	1.56%	1.17%	1.17%	3.13%	5.47%	4.30%	5.47%	3.13%
Paycard	6	100.00%	100.00%	33.33%	33.33%	33.33%	33.33%	83.33%	100.00%	83.33%	83.33%	83.33%
Prop4J	5029	4.97%	4.97%	0.04%	0.06%	0.04%	0.06%	0.16%	0.22%	0.24%	0.28%	0.16%
Sudoku	20	100.00%	100.00%	10.00%	10.00%	10.00%	10.00%	35.00%	35.00%	40.00%	40.00%	35.00%
TaskObserver	8	100.00%	87.50%	25.00%	25.00%	25.00%	25.00%	37.50%	87.50%	50.00%	50.00%	37.50%
Telecom	4	100.00%	75.00%	50.00%	50.00%	50.00%	75.00%	100.00%	75.00%	75.00%	100.00%	100.00%
UnionFindSPL	10	20.00%	20.00%	40.00%	40.00%	40.00	40.00%	40.00%	60.00%	50.00%	20.00%	50.00%
Vending machine	256	97.66%	97.66%	0.78%	1.56%	1.95%	0.78%	2.34%	13.67%	2.34%	2.34%	2.34%
ZipMe	24	100.00%	100.00%	8.33%	12.50%	20.83%	12.50%	8.33%	29.17%	25.00%	25.00%	12.50%

(b) Percentage of configurations analyzed by 3- and 4-wise testing strategies

Name	#Conf.	Baseline1	Baseline2	CASA-T3	Chvatal-T3	ICPL-T3	YASA-T3	CASA-T4	Chvatal-T4	YASA-T4
ArgoUML-SPL	256	97.66%	97.66%	4.69%	7.03%	6.25%	5.47%	9.38%	14.06%	12.89%
ATM	80	100.00%	100.00%	7.50%	20.00%	20.00%	17.50%	30.00%	41.25%	37.50%
BankAccount	144	100.00%	100.00%	11.11%	14.58%	13.89%	11.81%	25.00%	29.17%	27.78%
CheckStyle	$>2^{135}$	∼0.00%	∼0.00%	∼0.00%	∼0.00%	∼0.00%	∼0.00%	[b]	[b]	∼0.00%
Chess	8	87.50%	87.50%	87.50%	87.50%	87.50%	87.50%	100.00%	87.50%	87.50%
Companies	192	100.00%	100.00%	13.02%	16.67%	16.67%	11.46%	31.25%	40.10%	23.44%
Elevator	20	100.00%	100.00%	55.00%	55.00%	60.00%	55.00%	90.00%	90.00%	90.00%
Email	40	100.00%	100.00%	30.00%	40.00%	32.50%	32.50%	55.00%	70.00%	55.00%
FeatureAMP1	6732	3.71%	3.71%	0.36%	0.48%	0.48%	0.45%	0.98%	1.25%	1.19%
FeatureAMP2	7020	3.56%	3.56%	0.37%	0.54%	0.50%	0.47%	0.37%	1.34%	1.23%
FeatureAMP3	20 500	1.22%	1.22%	0.20%	0.25%	0.26%	0.23%	0.20%	0.56%	0.66%
FeatureAMP4	6732	3.71%	3.71%	0.37%	0.49%	0.49%	0.46%	0.59%	1.35%	0.58%
FeatureAMP5	3810	6.56%	6.56%	0.73%	0.92%	0.97%	0.89%	0.73%	2.57%	2.44%
FeatureAMP6	21 522	1.16%	1.16%	0.26%	0.21%	0.21%	0.31%	0.26%	1.01%	0.54%
FeatureAMP7	15 795	1.58%	1.58%	0.23%	0.28%	0.28%	0.26%	0.23%	0.80%	0.75%
FeatureAMP8	15 708	1.59%	1.59%	0.16%	0.21%	0.20%	0.18%	0.16%	0.55%	0.55%
FeatureAMP9	6732	3.71%	3.71%	0.43%	0.51%	0.49%	0.46%	0.43%	1.28%	1.17%
GPL	73	100.00%	100.00%	42.47%	47.95%	47.95%	58.90%	42.47%	86.30%	83.56%
IntegerSetSPL	2	100.00%	100.00%	100.00%	100.00%	100.00%	[c]	[a]	[a]	[a]
Jtopas	32	100.00%	100.00%	31.25%	65.63%	37.50%	50.00%	31.25%	65.63%	50.00%
MinePump	64	100.00%	100.00%	23.44%	32.81%	29.69%	20.31%	23.44%	56.25%	42.19%
Notepad	256	97.66%	97.66%	8.59%	11.72%	10.94%	9.77%	8.59%	23.44%	20.31%
Paycard	6	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
Prop4J	5029	4.97%	4.97%	0.52%	0.66%	0.66%	0.58%	0.52%	1.59%	1.59%
Sudoku	20	100.00%	100.00%	70.00%	75.00%	70.00%	70.00%	70.00%	100.00%	100.00%
TaskObserver	8	100.00%	87.50%	87.50%	100.00%	100.00%	87.50%	100.00%	100.00%	87.50%
Telecom	4	100.00%	75.00%	100.00%	100.00%	60.00%	[c]	[a]	[a]	[a]
UnionFindSPL	10	20.00%	20.00%	60.00%	100.00%	60.00%	50.00%	30.00%	100.00%	60.00%
Vending machine	256	97.66%	97.66%	4.69%	31.25%	33.20%	5.47%	10.16%	68.36%	12.11%
ZipMe	24	100.00%	100.00%	41.67%	66.67%	58.33%	45.83%	41.67%	91.67%	95.83%

#Conf.: number of valid configurations.
[a]Number of features of target configurable systems is less than t.
[b]Could not generate. configurations, time greater than 8 h.
[c]The strategy did not generate any configuration.

system (i.e., the greatest recall). Once multiple strategies have the same greatest recall and it is greater than 0%, we consider those as the most effective ones. Similar to previous sections, we present the results for baseline strategies in Table 6 aiming at fostering discussions in Section 6.

Regarding the 1-wise testing strategies, *Chvatal-T1* faced better finding more faults in 3 configurable systems and 4.25% of

Table 5
P-value results for percentage of configurations analyzed.

	1-wise	2-wise	3-wise	4-wise
p-value	0.9864	0.9493	0.1000	0.1292

the total of faults in the reference list. *CASA-T1*, *YASA-T1*, and *ICPL-T1* come next finding 3.89%, 3.54%, and 3.01% of the faults. Note that for three configurable systems (FeatureAMPP1, FeatureAMPP2, and MinePump) only one testing strategy suggested configurations that the test suite identified faults (*ICPL-T1* in the first configurable system and *Chvatal-T1* in the others). Regarding 2-wise testing strategies, *ICPL-T2* is the most effective testing strategy finding 15.61% of the faults. Next, *CASA-T2*, *Chvatal-T2*, *IncLing-T2*, and *YASA-T2* found 13.31%, 13.18%, 9.46%, and 5.27% of the faults from the reference list, respectively. Regarding 3- and 4-wise groups, *Chvatal-T3* and *Chvatal-T4* are the most effective testing strategy with 23.12% and 30.17% of the faults. Surprisingly, *CASA-T3* and *CASA-T4* downgraded the results from *CASA-T2* reducing the percentage of faults found (from 13.31% to 13.02% and 9.74%, respectively).

Looking at the effectiveness per configurable system, *Chvatal-T4* and *YASA-T4* achieved much greater recall than *CASA-T4*. Only for FeatureAMP1, *CASA-T4* achieved a greater recall than these two other testing strategies. Note that *YASA-T4* retrieves greater recall for four subject systems (FeatureAMP2, FeatureAMP5, FeatureAMP8, and FeatureAMP9) compared to the other strategies. For the configurable systems that *YASA-T4* achieved greater recall have more than 20 features. On the other hand, *Chvatal-T4* achieves greater recall for the five subject systems Companies, FeatureAMP1, FeatureAMP3, GPL, and Sudoku that have from 10 to 28 features. Note that for only three configurable systems (Chess, Paycard, and Sudoku) the suggested configurations of a t-wise strategy was able to find all faults in the reference list.

Practitioners should be aware that, as expected, the more configurations they test, the greater are the chances of finding faults. In addition, despite t-wise strategies suggested configurations which cover all features of a configurable system, these strategies are still far to recommend configurations that capture most of the faults. For instance, even though *Chvatal-T4* recommends 41.25% of the valid configurations of ATM, these configurations find no fault. Researchers may see it as an opportunity to propose sampling strategies that somehow recommend fault-prone configurations. Our assumption is that some classes and features are fault-prone and, for that reason, sampling strategies should prioritize testing configurations with these fault-prone features and classes. We investigate this assumption when answering RQ_6 and RQ_7.

In Table 7, we present the *p-values* for the Friedman's Test according to the t-wise group. As we can see, only 1-wise testing strategies is statistically significant different from the other testing strategies (i.e., *p-value* < 0.05) on being effective on finding faults. It means that for 2-, 3-, and 4-wise strategies, there is no statistical difference on the time to generate the configurations. Therefore, we reject the null hypothesis (H0) and accept the alternative hypothesis (H1) only for the 1-wise testing strategies. With the Post Hoc analysis for the 1-wise group, we did not obtain a significant difference among the strategies. It might be because the *p*-value is close to our confidence level.

Table 6
Recall of testing strategies.

(a) Recall of 1- and 2-wise testing strategies

Name	#faults	Baseline1	Baseline2	CASA-T1	Chvatal-T1	ICPL-T1	YASA-T1	CASA-T2	Chvatal-T2	ICPL-T2	IncLing-T2	YASA-T2
ATM	4	100.00%	100.00%	–	–	–	–	–	–	–	–	–
BankAccount	4	100.00%	100.00%	–	–	–	–	25.00%	–	–	–	–
Chess	30	100.00%	100.00%	16.67%	16.67%	3.33%	16.67%	60.00%	30.00%	70.00%	56.67%	16.67%
Companies	17	100.00%	100.00%	11.67%	–	5.88%	–	–	29.41%	11.76%	5.88%	–
FeatureAMP1	439	18.91%	47.84%	–	–	0.23%	0.46%	–	0.46%	0.23%	0.23%	0.23%
FeatureAMP2	148	9.46%	28.38%	–	0.68%	–	–	–	1.35%	0.68%	0.68%	0.68%
FeatureAMP3	180	5.56%	29.44%	–	–	0.56%	–	–	1.11%	5.56%	0.56%	2.78%
FeatureAMP4	147	13.61%	78.91%	–	–	–	–	–	–	–	–	0.68%
FeatureAMP5	5	80.00%	–	–	–	–	–	–	–	–	–	–
FeatureAMP6	24	–	79.17%	–	–	–	–	4.17%	–	–	–	–
FeatureAMP8	4	50.00%	–	–	–	–	–	–	–	–	–	–
FeatureAMP9	236	11.44%	22.88%	0.42%	0.42%	0.42%	0.42%	2.54%	2.97%	2.97%	3.81%	2.12%
GPL	23	100.00%	100.00%	–	4.35%	4.35%	4.35%	8.70%	13.04%	26.09%	4.35%	8.70%
MinePump	24	100.00%	100.00%	–	12.05%	–	–	12.50%	12.50%	12.50%	12.50%	12.50%
Paycard	3	100.00%	100.00%	33.33%	33.33%	33.00%	33.00%	100.00%	100.00%	100.00%	67.00%	–
Sudoku	5	100.00%	100.00%	–	–	–	–	–	20.00%	20.00%	–	40.00%
All systems	1293	61.81%	67.91%	3.89%	4.25%	3.01%	3.45%	13.31%	13.18%	15.61%	9.46%	5.27%

(b) Recall of 3- and 4-wise testing strategies

Name	#faults	Baseline1	Baseline2	CASA-T3	Chvatal-T3	ICPL-T3	YASA-T3	CASA-T4	Chvatal-T4	YASA-T4
ATM	4	100.00%	100.00%	–	–	–	–	–	–	–
BankAccount	4	100.00%	100.00%	–	25.00%	–	–	–	25.00%	25.00%
Chess	30	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	16.67%	100.00%	100.00%
Companies	17	100.00%	100.00%	17.65%	23.53%	41.18%	17.65%	29.41%	47.06%	35.29%
FeatureAMP1	439	18.91%	47.89%	3.64%	0.68%	0.23%	5.69%	17.54%	0.68%	2.96%
FeatureAMP2	48	9.46%	28.38%	3.38%	4.05%	3.38%	4.73%	12.16%	6.76%	23.65%
FeatureAMP3	180	5.56%	29.44%	5.56%	14.44%	6.11%	1.67%	5.00%	14.44%	7.22%
FeatureAMP4	147	13.61%	78.91%	–	–	–	0.68%	–	5.44%	0.68%
FeatureAMP5	5	80.00%	–	–	–	–	–	–	–	20.00%
FeatureAMP6	24	–	79.17%	4.17%	–	–	12.50%	–	–	–
FeatureAMP8	4	50.00%	–	–	–	–	25.00%	–	–	25.00%
FeatureAMP9	236	11.44%	22.88%	5.51%	4.24%	5.93%	8.90%	6.36%	7.20%	11.44%
GPL	23	100.00%	100.00%	43.48%	13.04%	13.04%	17.39%	8.70%	26.09%	17.39%
MinePump	24	100.00%	100.00%	25.00%	25.00%	25.00%	12.50%	–	50.00%	50.00%
Paycard	3	100.00%	100.00%	–	100.00%	100.00%	–	–	100.00%	–
Sudoku	5	100.00%	100.00%	–	60.00%	40.00%	40.00%	60.00%	100.00%	60.00%
All systems	1293	61.81%	67.91%	13.02%	23.12%	20.93%	15.42%	9.74%	30.17%	23.66%

Table 7
P-value results for recall of testing strategies.

	1-wise	2-wise	3-wise	4-wise
p-value	0.0380	0.5846	0.0732	0.6677

Table 9
P-value results for time efficiency by the strategies.

	1-wise	2-wise	3-wise	4-wise
p-value	0.2939	0.9716	0.7581	0.6065

RQ_3 Summary. Although we found statistically significant results only for 1-wise strategies, data of Table 6 show that *Chvatal-T4*, *YASA-T4*, and *Chvatal-T3* seems to be the testing strategies that suggested configurations able to find fault mostly effective. Nevertheless, there are still space for improvements since the strategy that faced better found 30.17% of the faults in the reference list.

5.4. The most time-efficient strategy (RQ_4)

In Table 8, we present the *time-efficiency* of each testing strategy for each configurable system with faults. We use "-" to represent testing strategies of which no fault was found. We highlight the most time-efficient testing strategy for each configurable system. Once multiple strategies have the same greatest *time-efficiency*, we consider those as the most *time-efficiency* ones. Similar to previous sections, we present baseline results for fostering discussions (Section 6).

As a result, *Chvatal-T1* is the most *time-efficiency* testing strategy in the 1-wise group for 4 configurable systems. *YASA-T1*, *ICPL-T1*, and *CASA-T1* are the most time-efficient testing strategy for 3, 2 and 1 configurable systems, respectively. For 2-wise group, *ICPL-T2* is the most time-efficient testing strategy for 6 configurable systems. *CASA-T2*, *Chvatal-T2*, and *YASA-T2* are the most time-efficient testing strategy for 3, 2 and 2 configurable systems, respectively. *IncLing-T2* was the most time efficient testing strategy for no system. *ICPL-T3* is the most time-efficient testing strategy to the 3-wise group for 5 configurable systems. *YASA-T3*, *Chvatal-T3*, and *CASA-T3* are the most time-efficient testing strategy for 4, 2 and 2 configurable systems, respectively. Finally, in the 4-wise group, *YASA-T4* was far the most time-efficient testing strategy compared to the other testing strategies. While it was the most time-efficient for 10 configurable systems, *Chvatal-T4* and *CASA-T4* were the most time-efficient testing strategies for only 4 and 1 configurable systems, respectively. Regarding the Friedman's Test, it was not possible to reject the null hypothesis. In Table 9, we show the *p-values* obtained with the Friedman's Test.

RQ_4 Summary. Although the Friedman's Test does not show statistically difference on the time-efficiency among testing strategies from the same t-wise group, our data in Table 4 show that *Chvatal-T1*, *ICPL-T2*, *ICPL-T3*, and *YASA-T4* are probably the testing strategies that suggested configurations able to find the best balance among the number of faults and time for the 1-, 2-, 3-, and 4-wise groups, respectively.

Table 8
Time efficiency of testing strategies.

(a) Time efficiency of 1- and 2-wise testing strategies

Name	Baseline1	Baseline2	CASA-T1	Chvatal-T1	ICPL-T1	YASA-T1	CASA-T2	Chvatal-T2	ICPL-T2	IncLing-T2	YASA-T2
ATM	0.008	0.006	–	–	–	–	–	–	–	–	–
BankAccount	0.028	0.019	–	–	–	–	0.133	–	–	–	–
Chess	1.413	0.116	0.631	0.448	0.178	0.563	1.229	0.742	2.760	0.631	0.507
Companies	0.180	0.064	0.368	–	0.181	–	–	0.540	0.210	0.368	–
FeatureAMP1	0.034	0.080	–	–	0.004	0.484	–	0.065	0.004	0.004	0.060
FeatureAMP2	0.005	0.008	–	0.036	–	–	–	0.022	0.031	0.005	0.013
FeatureAMP3	0.009	0.067	–	–	0.056	–	–	0.029	0.304	0.025	0.076
FeatureAMP4	0.024	0.048	–	–	–	–	–	–	–	–	0.004
FeatureAMP5	0.001	–	–	–	–	–	–	–	–	–	–
FeatureAMP6	0.026	0.021	–	–	–	–	0.016	–	–	–	–
FeatureAMP8	0.001	–	–	–	–	–	–	–	–	–	–
FeatureAMP9	0.024	0.021	0.155	0.043	0.042	0.036	0.072	0.080	0.212	0.155	0.075
GPL	0.561	0.575	–	0.182	1.428	0.184	0.137	0.185	0.438	0.055	0.157
MinePump	0.036	0.039	–	0.882	–	–	0.466	0.300	0.353	0.326	0.462
Paycard	0.007	0.006	0.006	0.015	0.014	0.015	0.017	0.014	0.041	0.006	–
Sudoku	0.338	0.048	–	–	–	–	–	0.115	0.128	–	0.253

(b) Time efficiency of 3- and 4-wise testing strategies

Name	Baseline1	Baseline2	CASA-T3	Chvatal-T3	ICPL-T3	YASA-T3	CASA-T4	Chvatal-T4	YASA-T4
ATM	0.008	0.006	–	–	–	–	–	–	–
BankAccount	0.028	0.019	–	0.051	–	–	–	0.026	0.029
Chess	1.413	0.116	1.257	1.234	1.544	2.023	0.294	1.249	1.875
Companies	0.180	0.064	0.156	0.173	0.347	0.149	0.116	0.149	0.149
FeatureAMP1	0.034	0.080	0.082	0.054	0.024	0.158	0.059	0.002	0.011
FeatureAMP2	0.005	0.008	0.011	0.014	0.045	0.018	0.015	0.008	0.028
FeatureAMP3	0.009	0.067	0.047	0.092	0.113	0.017	0.007	0.010	0.018
FeatureAMP4	0.024	0.048	–	–	–	0.004	–	0.002	0.002
FeatureAMP5	0.001	–	–	–	–	–	–	–	0.001
FeatureAMP6	0.026	0.021	0.001	–	–	0.022	–	–	–
FeatureAMP8	0.001	–	–	–	–	0.006	–	–	0.002
FeatureAMP9	0.024	0.021	0.042	0.031	0.424	0.006	0.001	0.016	0.038
GPL	0.561	0.575	0.343	0.086	0.074	0.136	0.020	0.084	0.066
MinePump	0.036	0.039	0.569	0.154	0.476	0.260	–	0.081	0.550
Paycard	0.007	0.006	–	0.014	0.018	–	–	0.015	–
Sudoku	0.338	0.048	–	0.188	0.154	0.140	0.157	0.238	0.146

Table 10
Coverage efficiency of testing strategies.

(a) Coverage efficiency of 1- and 2-wise testing strategies

Name	Baseline1	Baseline2	CASA-T1	Chvatal-T1	ICPL-T1	YASA-T1	CASA-T2	Chvatal-T2	ICPL-T2	IncLing-T2	YASA-T2
ATM	0.050	0.050	–	–	–	–	–	–	–	–	–
BankAccount	0.028	0.028	–	–	–	–	0.143	–	–	–	–
Chess	4.286	4.286	2.500	2.500	0.500	2.500	4.500	3.000	4.200	4.250	2.500
Companies	0.089	0.089	0.500	–	0.250	–	–	0.385	0.154	0.077	–
FeatureAMP1	0.332	1.240	–	–	0.333	1.000	–	0.182	0.100	0.071	0.100
FeatureAMP2	0.112	0.276	–	0.333	–	–	–	0.167	0.091	0.077	0.091
FeatureAMP3	0.040	0.316	–	–	0.333	–	–	0.133	0.625	0.048	0.556
FeatureAMP4	0.080	0.516	–	–	–	–	–	–	–	–	0.100
FeatureAMP5	0.016	–	–	–	–	–	–	–	–	–	–
FeatureAMP6	–	0.080	–	–	–	–	0.071	–	–	–	–
FeatureAMP8	0.008	–	–	–	–	–	–	–	–	–	–
FeatureAMP9	0.188	0.296	0.500	0.333	0.333	0.500	0.667	0.583	0.636	0.750	0.500
GPL	0.258	0.258	–	0.250	0.200	0.167	0.143	0.231	0.120	0.071	0.118
MinePump	0.375	0.375	–	1.000	–	–	0.429	0.429	0.375	0.429	0.500
Paycard	0.500	0.500	0.500	0.500	0.500	0.500	0.600	0.500	0.600	0.400	–
Sudoku	0.250	0.250	–	–	–	–	–	0.143	0.125	–	0.286

(b) Coverage Efficiency of 3- and 4-wise testing strategies

Name	Baseline1	Baseline2	CASA-T3	Chvatal-T3	ICPL-T3	YASA-T3	CASA-T4	Chvatal-T4	YASA-T4
ATM	0.050	0.050	–	–	–	–	–	–	–
BankAccount	0.028	0.028	–	0.048	–	–	–	0.024	0.024
Chess	4.286	4.286	3.750	4.286	4.286	3.750	0.625	4.286	3.750
Companies	0.089	0.089	0.125	0.125	0.219	0.136	0.083	0.104	0.133
FeatureAMP1	0.332	1.240	0.667	0.094	0.031	0.833	1.167	0.036	0.160
FeatureAMP2	0.112	0.276	0.192	0.158	0.143	0.212	0.692	0.106	0.407
FeatureAMP3	0.040	0.316	0.250	0.500	0.204	0.064	0.136	0.228	0.096
FeatureAMP4	0.080	0.516	–	–	–	0.032	–	0.088	0.012
FeatureAMP5	0.016	–	–	–	–	–	–	–	0.011
FeatureAMP6	–	0.080	0.018	–	–	0.045	–	–	–
FeatureAMP8	0.008	–	–	–	–	0.033	–	–	0.011
FeatureAMP9	0.188	0.296	0.448	0.294	0.424	0.677	0.517	0.198	0.342
GPL	0.258	0.258	0.250	0.086	0.086	0.091	0.063	0.095	0.065
MinePump	0.375	0.375	0.400	0.286	0.316	0.231	–	0.333	0.444
Paycard	0.500	0.500	–	0.500	0.500	–	–	0.500	–
Sudoku	0.250	0.250	–	0.200	0.143	0.143	0.214	0.250	0.150

Table 11
P-value results for coverage efficiency of testing strategies.

	1-wise	2-wise	3-wise	4-wise
p-value	0.1893	0.8847	0.7961	0.5258

5.5. The most coverage-efficient strategy (RQ_5)

In Table 10, we present the *coverage-efficiency* of each testing strategy for each configurable system with faults. We use "-" for representing testing strategies that no fault was found. We highlight the most coverage-efficient testing strategy for each configurable system. Once multiple strategies have the same greatest coverage-efficiency, we consider those as the most coverage-efficient ones. Similar to previous sections, we present results for baseline testing strategies aiming at fostering discussions (Section 6).

As a result for the 1-wise group, *Chvatal-T1* is the most coverage efficient testing strategy for 3 configurable systems. *CASA-T1*, *YASA-T1*, and *ICPL-T1* are the most coverage-efficient testing strategies for 2, 2, and 1 configurable systems, respectively. For the 2-wise group, *CASA-T2* and *Chvatal-T2* are the most coverage-efficient testing strategy for 5 configurable systems. *ICPL-T2*, *IncLing-T2*, and *YASA-T2* are the most coverage-efficient testing strategies for 2, 2, and 1 configurable systems, respectively. For the 3-wise group, *Chvatal-T3* is the most coverage-efficient testing strategy for 5 configurable systems. *ICPL-T3*, *YASA-T3* , and *CASA-T3* are the most coverage-efficient testing strategies for 3, 3, and 2 configurable systems, respectively. For the 4-wise group, *Chvatal-T4* is the most coverage-efficient testing strategy for 6 configurable systems. *CASA-T4* and *YASA-T4* are the most coverage-efficient testing strategies for 2 configurable systems, respectively.

Note that since each t-wise testing strategy faced better in at least one configuration system, the *coverage-efficiency* analysis presented more dispersed results than the previous ones. In Table 11, we show the *p-values* obtained with the Friedman's Test. As seen, it was not possible to reject the null hypothesis in any case.

RQ_5 Summary. Although the Friedman's Test does not show statistically difference on the coverage-efficiency among testing strategies from the same t-wise group, our data of Table 10 show that *Chvatal* is the testing strategy that recommends configurations able to find the best balance among faults and the number of recommended configurations in all groups (1-, 2-, 3-, and 4-wise).

5.6. Dispersion of faults over classes (RQ_6)

In Table 12, we report the 22 classes with faults distributed over 16 configurable systems. For each faulty class, we present traditional source code metrics, such as Coupling between Object Classes (CBO), Weighted Methods per Class (WMC), Depth Inheritance Tree (DIT), Number of Methods (NOM), Lines of Code (LoC), and faults found (#F). We analyze the log generated by JUnit to understand each fault found. Through the JUnit log, we evaluated each faulty class. In Table 13, we present the 22 classes with faults concerning feature metrics, such as Total number Features that handle the related class (TNF), Scattering over classes (ScC), and Tangling in class (TaC).

Table 12
Faults found by class with traditional metrics.

System name	Class name	CBO	WMC	DIT	RFC	LCOM	NOM	NOPM	NOSM	NOF	NOPF	NOSF	NOSI	LoC	#F
ATM	ATMUserInterface	12	62	6	53	0	34	23	0	43	17	19	10	447	3
	Withdrawal	6	31	2	11	0	5	4	0	4	0	1	0	159	1
Bankaccount	Transaction	1	19	1	6	3	3	1	1	1	0	0	1	38	4
Chess	Model	6	75	1	48	76	20	16	1	18	1	1	0	289	23
	Pawn	1	24	2	9	1	2	2	0	0	0	0	0	69	1
	Board	13	196	1	47	124	40	27	0	14	0	0	0	596	6
Companies	Controller	12	61	1	18	0	7	7	0	2	0	0	0	181	17
FeatureAMP1	PlaylistHandler	4	56	1	29	13	15	10	0	9	0	0	1	158	33
	App	7	159	1	124	848	48	20	0	43	0	0	25	677	406
FeatureAMP2	Gui	11	310	7	178	3137	66	26	0	44	6	1	10	1161	148
FeatureAMP3	FeatureAmp	20	470	6	251	6511	117	17	1	57	0	5	12	1537	180
FeatureAMP4	FeatureAmp	13	78	1	56	404	33	10	3	12	1	1	8	285	93
	PlayerBar	11	16	1	17	1	10	9	0	5	0	0	0	108	55
FeatureAMP5	Main	9	211	6	159	2889	60	3	2	34	0	4	22	860	5
FeatureAMP6	Playlist	1	62	1	23	116	24	18	0	4	2	0	0	185	10
	Kernel	7	114	1	71	940	52	29	0	14	0	0	9	411	14
FeatureAMP8	Application	8	355	2	173	3987	75	39	1	37	1	0	11	1565	4
FeatureAMP9	Gui	7	311	6	176	2467	61	22	0	40	3	1	13	1172	236
GPL	Graph	10	175	1	48	241	26	25	4	3	3	1	27	797	23
MinePump	PL_Interface	3	31	1	16	34	9	8	4	5	2	5	7	186	24
PayCard	PayCard	2	31	1	6	25	10	7	1	5	0	0	0	127	3
Sudoku	BoardManager	7	99	1	33	54	21	15	0	4	0	0	12	295	5

CBO: Coupling between objects; **WMC**: Weight Method Class; **DIT**: Depth Inheritance Tree; **RFC**: Response for a Class; **LCOM**: Lack of Cohesion of Methods; **NOM**: Number of Methods; **NOPM**: Number of Public Methods; **NOSM**: Number of Static Methods; **NOF**: Number of Fields; **NOPF**: Number of Public Fields; **NOSF**: Number of Static Fields; **NOSI**: Number of static invocations; **LoC**: Lines of code; **#F**: Number of faults found.

Table 13
Faults found by class with variability metrics.

System	Class name	TNF	ScC	VarC	TaC	MFA	LoCIF	#F
ATM	ATMUserInterface	6	10	32	15	2	40	3
	Withdrawal	2	4	3	7	3	59	1
Bankaccount	Transaction	1	1	1	1	1	1	4
Chess	Model	0	0	0	0	0	0	1
	Board	3	10	20	6	8	22	23
	Pawn	0	0	0	0	0	0	6
Companies	Controller	5	119	155	85	3	16	17
FeatureAMP1	PlaylistHandler	3	8	11	7	5	10	33
	APP	17	23	30	22	24	51	406
FeatureAMP2	GUI	19	35	55	29	37	85	148
FeatureAMP3	FeatureAmp	26	47	90	39	58	163	180
FeatureAMP4	FeatureAmp	12	50	35	24	2	61	93
	PlayerBar	1	9	3	2	1	28	55
FeatureAMP5	Main	16	30	32	22	26	52	5
FeatureAMP6	Playlist	2	12	17	8	7	10	10
	Kernel	8	47	50	29	18	35	14
FeatureAMP8	Application	23	27	41	26	38	72	4
FeatureAMP9	Gui	19	32	53	26	40	92	236
GPL	Graph	11	28	58	5	15	474	23
MinePump	PL_Interface	0	0	0	0	0	0	24
PayCard	PayCard	3	10	5	4	5	7	3
Sudoku	BoardManager	5	16	43	16	14	126	5

TNF: Total number Features that the related class handle; **ScC** Scattering against class features: Counts the number of classes that implement the features that the analyzed class handles; **VarC**: Number of occurrences of variability in the class; **TaC** Tangling in class: Counts the number of context switching features manipulated in the classes; **MFA**: Counts the number of methods of the class that handle the optional features.; **LoCIF**: Counts the number of lines of code for optional features in the analyzed class; **#F**: Number of faults found.

Considering that the subject dataset has 2740 classes and we found faults in 22, only 0.8% of the classes failed. It shows that faults are highly concentrated in a few classes. By looking only at the 16 faulty systems, we see that in 11 systems the faults are concentrated in only one class and in the other 5 systems (ATM, Chess, FeatureAMP1, FeatureAMP4, and FeatureAMP6), the faults are located in up to three classes.

Looking at the number of faults per faulty class, we see that most of the faults found are in few classes. For instance, 31.40% of the faults are in the class *App* of FeatureAMP1, 18.35% of the faults are in the class *Gui* of FeautureAMP9, and 13.92% of the faults are in the class *FeatureAMP* of FeautureAMP3. Anyway, we found more than 10 faults in 12 out of 22 faulty classes.

High values for coupling and complexity metrics such as Response for a Class (RFC) and Weighted Methods per Class (WMC) might be related to fault-prone classes in configurable systems. For all systems with more than one class with faults, WMC and RFC are greater for classes with more faults than classes with fewer faults. For example, in the FeatureAMP1 system, the *App* class, with 406 faults, is greater for WMC and RFC than the *PlaylistHandler* class, which has 33 faults. We can also verify how the features are implemented by several classes of the system and mix up with other features through the metrics of features used in this study. We observed that Scattering against class Features (ScC), Number of occurrences of variability in the class (VarC), and Tangling in class (TaC) might be related to fault-prone classes in configurable systems. For example, in FeatureAMP6, *Kernel* class, of which were identified 14 faults, has greater SsC, VarC, and TaC values than the *Playlist* class, which were identified 9 faults.

Aiming at investigating whether faulty classes have some characteristics that differ them from non-faulty classes, we compute the Spearman's rank correlation among all classes in the dataset for all metrics shown in Table 12. Due to the small number of failing classes in our dataset, we only found a weak correlation (less than 0.15) between the traditional source code metrics and the faults found. The higher correlations (between 0.10 and 0.15) were found to WMC (0.13), NOF (0.12), NOM (0.11), NOPF (0.11), LoC (0.11), RFC (0.10), and NOPM (0.10). Concerning feature metrics, we found a slightly highest correlation compared to traditional metrics. However, we were able to find only weak correlations (less than 0.25) between the feature metrics and the faults found. The higher correlation (0.23) was found for TaC. We found a slightly lower correlation to NTF (0.19), LoCIF (0.18), and MFA (0.18) metrics. For the other feature metrics, we found 0.16 for ScC and VarC. These correlations are statistically significant at 99% confidence level (i.e., p-value <0.01). For DIT, NOC, NOSF, and NOSI, no statistically significant correlation was found.

We can see that the feature metrics had a slightly higher correlation with faults than traditional metrics from source code. This result may indicate that classes that are handled by features may be more likely to have feature interaction faults than other classes that are not handled by optional features. Thus, a reduction in the cost of testing would be to test more rigorously only the classes that implement the features with many points of variability. In another direction, it is known that classes with high coupling make it challenging to understand the class, which makes the test suite creation more challenging. In this way, the metrics we have pointed out can be good to indicate faults. This result may indicate fault-prone classes of which testers may prioritize them. In this way, developers can avoid creating classes high values for these metrics to decrease the chances of feature interaction faults.

Although our fault group is very small, we did not find high correlations for the metrics as indicators of faults. The small percentage of feature interaction faults can be a characteristic of configurable systems. As far as we know, we lack an empirical study to report the average percentage of feature interaction faults in configurable systems at the level of the source code. The lack of a large dataset with automated tests may be the reason for the lack of studies evaluating it.

Regarding faults at the configuration level, there is a consensus in the literature that faults occur only in a subset of all configurations (Soares et al., 2018a; Medeiros et al., 2016). Our results indicate that faults also occur in a few classes of configurable systems. Even though in a small amount, these faults are difficult to find and decrease the quality of the configurable system. Our findings might inspire researchers to deeply investigate characteristics of feature interaction faults considering not only the feature model but also the source code of configurable systems.

RQ_6 Summary. We found that faults are concentrated in only 0.8% of classes of the subject dataset. Two classes are responsible for 49.75% of the faults. Our results also show that source code metrics, such as LoC, WMC, and RFC, and the feature metrics ScC, VarC, and TaC appear to be related to fault-prone classes. Therefore, practitioners should be aware that large, complex, and highly coupled classes and classes that implement features that have high scattering and interlacing are often faulty.

5.7. Dispersion of faults over features (RQ_7)

In this section, we report the dispersion of faults found regarding features. Several studies on feature interaction faults analyze the features only concerning the feature model and do not analyze the implementation of features (Al-Hajjaji et al., 2016; Hervieu et al., 2011; Johansen et al., 2011; Oster et al., 2011). Hence, there is still a lack of evidence on the characteristics of features that are fault-prone. To achieve this goal, we compared characteristics of faulty features with characteristics of no faulty features. At the end, we investigated each failed configuration to discover the active features. Looking at the configurable system's source code, we analyze the active features related to the fault location. Hence, we compute the active features in the failed configurations, and we use feature metrics to measure these features. In Table 14, we show the characteristics of the features frequently active in failed configurations. The main difference of the metrics in this section and metrics from the previous section is that while in this section we measure features concerning the entire configurable system, in the previous section we measure the features concerning each class. For example, in the previous section, we measured Scattering and Tangling for features concerning the class. In this section, we use the Scattering and Tangling to measure each feature concerning the entire project.

For short, we measure the number of variability points containing other features using the Scattering (Sc.). Some features change the behavior of a more significant amount of code, and others a specific part of the code. In this way, we compute the number of times that features appear in the constructors (Co.) and methods (Me.). The influence of the features in the configurable system through the number of code lines (LoC) that the feature handles. Some features cause many points of variability in the source code. We calculate these variability points by feature using the variability points (VP). Some features handle various classes of the configurable system and others only in specific classes. To verify this characteristic, we compute the number of classes that the feature manipulates using the Tangling (Ta.). Finally, we use a metric to show the depth of a feature relative to the feature model (DT). This metric calculates the distance of a feature from the root feature of its feature model.

We analyze the characteristics of the faulty features in the same way used in the faulty classes. We compute the Spearman's rank correlation among all features in the dataset for all metrics shown in Table 14. We investigated a total of 350 features and found values for correlation between 0.41 and 0.66 across all seven metrics in Table 14. We have been found for Ta. (0.66), Co. (0.41), Me. (0.64), LoC (0.64), VP (0.64), Ta. (0.65), and DT

Table 14
Faults found by feature.

Name	Feature	Sc.	Co.	Me.	LoC	VP	Ta.	DT	#F	%F
ATM	DEPOSITING	3	1	17	264	7	3	2	4	5.00
	USER_INTERFACE	3	3	24	237	12	4	3	4	5.00
	WITHDRAWING	1	0	4	89	7	3	3	3	3.75
Bankaccount	OVERDRAFT	1	0	0	6	0	0	2	6	4.17
	BANKACCOUNT	4	2	21	210	2	2	2	6	4.17
	CREDITWORTHINESS	1	0	2	16	2	2	2	3	2.08
	DAILYLIMIT	2	0	4	57	3	2	2	6	4.17
Chess	AI_PLAYER	4	4	38	690	10	2	2	8	100.00
	OFFLINE_PLAYER	3	3	38	663	5	2	2	8	100.00
	ONLINE_PLAYER	3	3	38	665	5	2	2	8	100.00
Companies	LOGGING	4	1	8	187	9	4	2	8	4.17
	PRECEDENCE	4	1	9	186	13	4	2	16	8.33
	TOTAL_WALKER	16	5	32	1119	31	17	3	16	8.33
	TOTAL_REDUCER	16	5	23	1000	31	17	3	1	0.52
	CUT_WHATEVER	29	9	49	1399	31	17	3	17	8.85
	CUT_NO_MANAGER	29	8	46	1345	31	17	3	1	0.52
	GUI	7	5	33	492	48	9	2	9	4.69
FeatureAMP1	PLAYLIST	3	1	13	163	4	2	4	416	45.66
	PROGRESSBAR	1	0	2	24	2	1	5	416	45.66
	SHOWTIME	1	0	2	22	2	1	5	415	45.55
FeatureAMP2	PLAYLIST	3	1	24	321	5	2	3	126	14.03
	PROGRESSBAR	1	0	4	37	4	1	4	129	14.36
	SHOWCOVER	1	0	3	30	3	1	3	90	10.02
FeatureAMP3	REMOVETRACK	1	1	4	54	4	1	5	107	10.09
	VOLUMECONTROL	1	1	4	55	4	1	3	121	11.42
	PLAYLIST	2	2	28	282	8	2	3	182	17.17
	PROGRESSBAR	1	0	6	74	6	1	4	183	17.26
	SHUFFLEREPEAT	2	1	6	141	5	1	5	74	6.98
FeatureAMP4	PLAYER_BAR	9	1	11	154	3	2	2	146	17.16
	PLAYER_CONTROL	2	1	5	47	3	2	3	117	13.75
	PLAYLIST	9	2	29	315	6	4	2	117	13.75
	PROGRESS_BAR	4	1	5	59	3	2	3	125	14.69
FeatureAMP5	PLAYLIST	4	1	25	259	6	1	4	5	0.54
	PROGRESSBAR	1	0	2	23	2	1	5	5	0.54
	SHUFFLEREPEAT	3	0	8	128	3	2	5	3	0.33
FeatureAMP6	REORDER	3	1	7	124	2	2	4	19	1.59
	PROGRESSBAR	2	1	4	86	3	1	4	22	1.84
	PLAYLIST	7	3	31	451	8	5	2	21	1.76
FeatureAMP8	PLAYLIST	1	0	14	274	4	1	3	2	0.23
	QUEUETRACK	1	0	8	292	6	1	4	4	0.45
FeatureAMP9	LOADFOLDER	1	0	5	91	3	1	4	230	26.05
	PLAYLIST	2	1	20	244	6	2	3	230	26.05
	PROGRESSBAR	1	0	3	48	3	1	4	231	26.16
GPL	SEARCH	2	0	6	77	6	2	5	4	1.60
	NUMBER	3	1	4	19	5	3	4	2	0.80
	STRONGLYCONNECTED	4	2	7	66	9	4	4	4	1.60
MinePump	STOPCOMMAND	2	0	2	21	1	1	4	12	18.75
	HIGHWATERSENSOR	2	0	3	27	1	1	4	24	37.50
Paycard	PAYCARD	6	5	16	339	5	4	1	3	50.00
	LOCKOUT	1	0	2	33	2	1	2	3	50.00
	LOGGING	3	2	5	94	3	3	2	2	33.33
Sudoku	COLOR	1	3	0	48	3	1	2	3	15.00
	SOLVER	7	4	19	467	23	7	2	4	20.00
	GENERATOR	5	1	7	195	8	5	2	3	15.00
	STATES	4	1	13	236	13	4	2	4	20.00
	EXTENDEDSUDOKU	2	0	2	45	2	2	2	2	10.00
	UNDO	2	0	2	44	2	2	2	4	20.00

Sc. (Scattering) column shows the number of classes that the feature manipulates. **Co.**, and **Me.** columns show the number constructors, and methods the feature is inserted into, respectively. **LoC**: Lines of code; shows the number of lines of code handled by the feature. VP: variability points, occurrences of variability in source code related to feature. **Ta.** (Tangling) show the number of variability points containing other features. **DT**: Depth of tree; shows the depth of the feature relative to the feature model. **#F**: The number of faults found where features are present. **%F**: Percentage of the active feature concerning the total configurations analyzed.

(0.66). All these correlations are statistically significant at 99% confidence level (i.e., $p\text{-}value < 0.01$).

The features that stand out with the greatest percentage of faults within the configurations have high values for the Scattering and Tangling metrics. In this way, the features implemented by several modules of the system and mixed up with other features are good indicators of fault-prone features in configurable systems. We observed that the features with the greatest values for the metrics Sc., Me., LoC, VP, and Ta are more related to faults. These metrics are related to size and measure how much the features manipulate parts of the configurable system source code.

Our results show that the more the feature handles parts of the source code, the more fault-prone this feature will be. This result can improve the set of configurations for testing, as the features that have the most influence on the source code can be prioritized for testing. Thus, researchers might prioritize these features in their testing approaches. On the other hand, practitioners can be more careful on touching features that have greater influence on the configurable system code and they can refactor these features dividing them into multiple features. In this way, configurable system operations will not be concentrated in a small number of features.

RQ_7 Summary. We find that faults are concentrated in small groups of features, and the features that have the most influence on the configurable system are fault-prone. Our results also reveal that feature metrics, such as Scattering and Tangling, seem to be good indicators to identify fault-prone features.

6. Discussion

In this section, we discuss our results and present implications of our study. In Section 6.1, we show a brief discussion of the occurrence of faults in different testing suite types. In Section 6.2, we discuss the relation between the characteristics of features and classes with faults and how it can influence the proposal of a new strategy. In Sections 6.3 and 6.4 , we present the implications for practitioners, researchers, and tool builders.

6.1. Investigating faults in testing suite types

As seen in Section 3, we have three types of testing suites: (i) the ones we create from scratch, (ii) the ones we evolve, and (iii) the ones present in the literature. From the 20 configurable systems that we created the test suite from scratch, we find faults in 13 of them (65%). From the three systems that we evolve the testing suite, we found faults in two of them (66.67%). From the seven that we used in the original testing suite, we found faults in only one of them (14.29%). Hence, the following question emerges:

Why do we find less faults in the configurable systems that already had a testing suite than in the configurable systems that we create or evolve the testing suite?

The answer is twofold. First, for systems that we created or evolved the testing suite, our test cases represent feature interactions that developers did not think upfront. Second, for systems that already had a testing suite, developers already tested at least a sample of configurations and fixed identified faults. Note that both cases highlight the importance of testing several (or all) configurations to avoid hurting potential users and degrading the reputation of a project.

6.2. On the relation between faulty classes and faulty features

In Fig. 4, we show the distribution of faults found in the subject dataset. In Fig. 4a, we show the occurrence of faults in our dataset and which strategies found faults. Rows indicate systems and columns indicate the testing strategies. We highlight in black the occurrence of faults for the configurable system. For example, for FeatureAMP5, only the *YASA-T4* strategy found faults. For FeatureAMP9, all 16 testing strategies found faults. The only exception, is ATM, of which, only the baselines found faults. In other words, no t-wise strategy found faults on ATM. In Fig. 4b, we show faulty classes and faulty features for all optional code (i.e., code of non-mandatory features). The internal layer shows the configurable systems, the mid layer shows faulty classes, and the external layer shows faulty features. We represent only optional code because they may change across configurations impacting on the testing process. The larger the representation of an instance (i.e., project, class, or feature), the more variation points are related to it. The darker the representation, the larger the number of faults found. For example, in FeatureAMP4, there are four classes with variability points (*PlayerBar*, *Playlist*, *AudioFactory*, and *FeatureAMP*) and only two of them are faulty. We found 55 and 93 faults in the *PlayerBar* and *FeatureAmp* classes, respectively. The failing code of these classes belongs to four features: PLAYER_BAR, PROGRESS_BAR, PLAYER_CONTROL, and PLAYLIST. Faults found in the other classes and features were pointed only by our baselines. Note that a great number of variation points are often related to a great number of faults. For instance, looking at FeatureAMP3, FeatureAMP9, FeatureAMP1, and FeatureAMP4, we see that they concentrate most variation points and also most faults.

As mentioned in Section 2.3, t-wise strategies need to satisfy a certain degree of feature coverage. For example, a 2-wise strategy has to assure that the suggested configurations cover all interactions between two features. Therefore, if there is a fault between two features, a 2-wise strategy will find this fault. To illustrate it with a real example, we found a feature interaction fault in Companies when features PRECEDENCE and CUT_WHATEVER are active. This fault is related to the functionality of assigning a new salary to an employee of a company when a company expense cut is made. All 2-wise strategies (or with greater t) identified this fault because, in a target configuration, the features were active. However, the *Chvatal-T1* strategy did not find this fault because none of the recommended configurations have these two features simultaneously active. There is no guarantee that a 1-wise strategy will retrieve a fault that occurs with the interaction among two features or that a 3-wise strategy will retrieve a fault that occurs given the interaction of four features.

As seen in Section 5.2, the greater the number of valid configurations of a project, the smaller is the percentage of configurations tested. However, once a fault that occurs among the interaction of 5 features is found (e.g., when these five features are active), fixing it will remove faults that may appear in several other configurations. Hence, at the end, it is not about the number of configurations tested, but the number of *right* configurations tested (i.e., configurations that may fail and hurt potential users). The problem of t-wise strategies is that they prioritize a degree of coverage and ignore characteristics of fault-prone-classes and -features (see the characteristics in Sections 5.6 and 5.7).

A possible solution is to create a hybrid strategy that, besides, to consider feature coverage, also takes characteristics of features and classes into account on the selection of configurations. This way, it may also consider a different t depending on the number of valid configurations. Even though a large t is mostly desired, a large t for a system with a very small number of valid configurations may not fulfill the t-wise requirements and large t for systems with a very large number of valid configurations may run out of memory.

6.3. Implications for practitioners

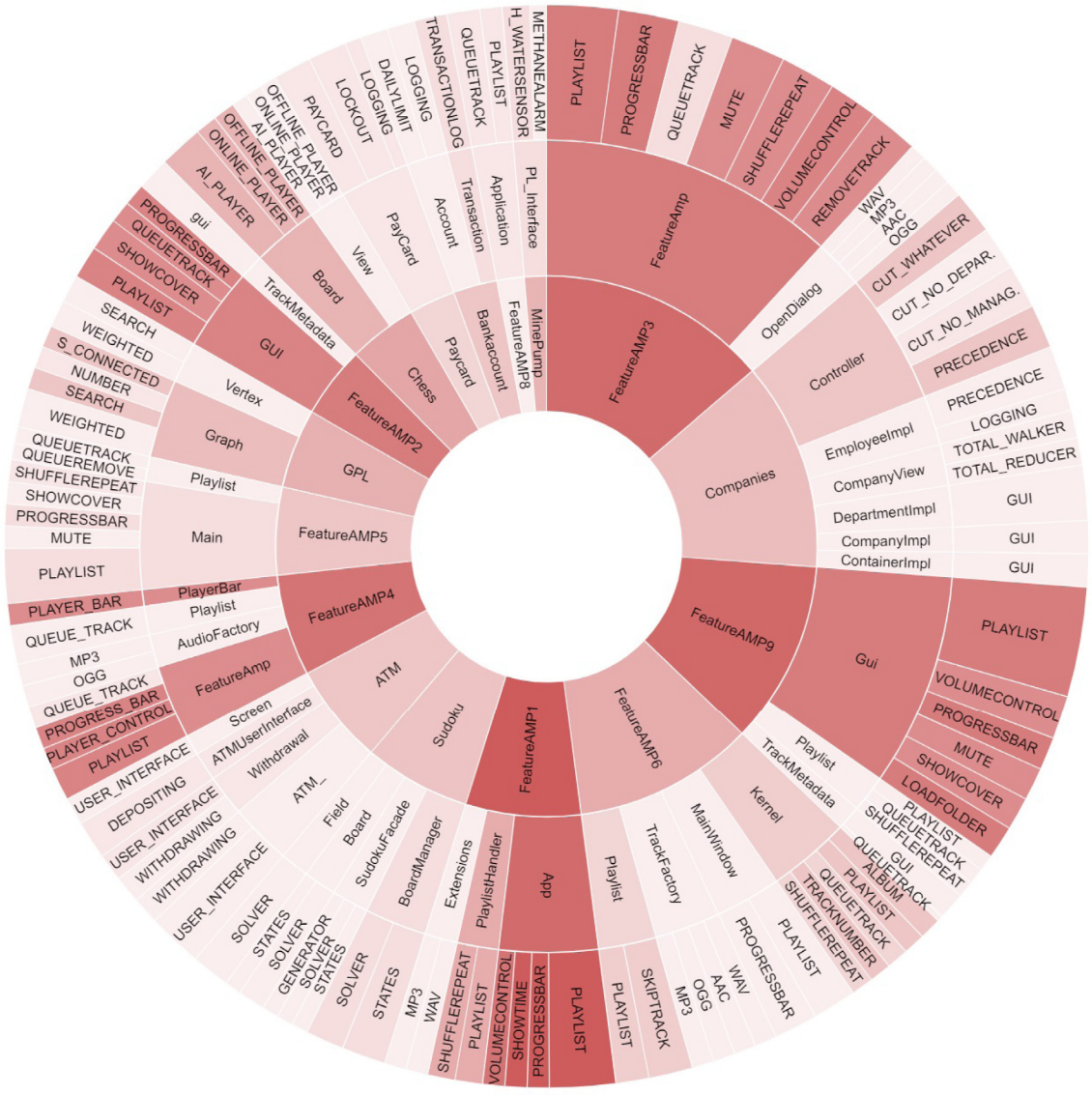
After reading this paper a question may arise:

Which testing strategy should I use?

The answer varies depending on your *system*, *project life-cycle*, and *organizational constraints*. In the ideal case, you should test all configurations. In practice, the maximal number of feasible configurations, as explained next. Looking at the results of Sections 5.1 and 5.2, practitioners have a practical estimation of time and coverage of each testing strategy. For instance, for small

Name	CASA-T1	Chvatal-T1	ICPL-T1	YASA-T1	CASA-T2	Chvatal-T2	ICPL-T2	IncLing-T2	YASA-T2	CASA-T3	Chvatal-T3	ICPL-T3	YASA-T3	CASA-T4	Chvatal-T4	YASA-T4
ATM																
Bankaccount					■						■				■	■
Chess	■	■	■	■	■											
Companies	■		■			■	■	■		■	■	■	■	■	■	■
FeatureAMP1			■	■		■	■	■	■	■	■	■	■	■	■	■
FeatureAMP2		■				■	■	■	■	■	■	■	■	■	■	■
FeatureAMP3			■			■	■	■	■	■	■	■	■	■	■	■
FeatureAMP4									■				■		■	■
FeatureAMP5																■
FeatureAMP6					■					■			■			
FeatureAMP8																■
FeatureAMP9	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■
GPL		■	■	■	■	■	■	■	■	■	■	■	■	■	■	■
MinePump		■			■	■	■	■		■	■	■	■		■	
Paycard	■	■	■	■	■	■	■	■			■	■			■	
Sudoku						■	■		■		■	■	■	■	■	■

(a) Faults distribution found by configurable systems



(b) Faults distribution found by classes and features

Fig. 4. Faults distribution found by configurable systems.

projects such as GPL (1235 lines of code, 13 features, and 51 test cases), it is reasonable waiting around 10 ms to run the testing suite of all 73 valid configurations. On the other hand, for large projects such as CheckStyle (61 435 lines of code, 141 features, and 719 test cases), it is not feasible or meaningful wait until finishing running the test suite of all $>2^{135}$ configurations. Our suggestion is to first run configurations that test the source code of changed features. Then, choose a t-wise strategy that fits your time- and coverage-constraints to have a general view of the project.

Aiming at supporting the choice of a testing strategy, we condensed our results in Fig. 5 representing each t-wise group separately, only t-wise strategies, and all testing strategies used in this study (i.e., including the baselines). For example, in Fig. 5a, we show the 1-wise testing strategies, in Fig. 5b, we show the 2-wise group, in Fig. 5e, we show the comparison between t-wise groups, and, in Fig. 5f, we show all testing strategies used in this study. The broader the line, the better the strategy is for a target characteristic. Next, we discuss each sub-figure individually.

1-Wise testing strategies (Fig. 5a). Comparing the 1-wise strategies, *ICPL-T1* stood out as the fastest strategy. *Chvatal-T1* was the strategy with the greatest coverage and recall, and was the most time- and coverage-efficient.

2-Wise testing strategies (Fig. 5b). Comparing the 2-wise strategies, *ICPL-T2* was fastest and time-efficient and got the greatest coverage and recall. *CASA-T2* and *Chvatal-T2* were the most coverage-efficient strategies.

3-Wise testing strategies (Fig. 5c). Comparing the 3-wise strategies, *ICPL-T3* was fastest and time-efficient testing strategy. *Chvatal-T3* achieved better coverage, recall, and coverage-efficient.

4-Wise testing strategies (Fig. 5d). Comparing the 4-wise strategies, *YASA-T4* was the fastest and time-efficient testing strategy. *Chvatal-T4* covered more configurations, achieved the greatest recall and was the most coverage-efficient testing strategy.

T-wise testing strategies (Fig. 5e). If there is a great time-constraint, 1-wise strategies might be the right option. 2-wise strategies found faults that no 1-wise strategy found in BankAccount, FeatureAMP4, FeatureAMP6, and Sudoku. For practitioners who want to test more configurations than the suggested by 1-wise strategies and still has a great time-constraint, 2-wise strategies is a good option. 3-wise strategies obtained greater time-efficiency and coverage-efficiency results than the other groups. Note that 3-wise strategies recall was close to the recall of 4-wise strategies. Practitioners might choose 3-wise group when prioritizing time- and coverage-efficiency. When generating configurations using 4-wise strategies we noticed that they took a while for systems with more than 20 features. However, considering that practitioners do not need to generate the configurations every time (e.g., only when the feature module changes), they can reuse the suggested list of configurations multiple times. This point make 4-wise strategies more usable in practice.

Comparing all testing strategies in the study (Fig. 5f). As expected, the baselines used in the study got better results for recall, percentage of configurations, and coverage efficiency than the t-wise strategies. It happened because our baselines test all possible configurations for systems with less than 250 valid configurations. When the total number of valid configurations is small and there is not a strong time-constraint, testing all configurations is the best option. However, for systems with a much greater number of valid configurations, it is infeasible in practice.

We started using the baselines to increase the number of configurations tested. However, our baselines remembered us that, looking at a minimized number of configurations leaves a bunch of configurations behind where feature interactions faults may occur. For instance, while baseline 1 and 2 found 856 and 795 faults in the subject dataset, *Chvatal-T4* found only 353 faults. It is less than half of the faults found by the baselines. Hence, depending on the number of valid configurations, it is meaningful to test all of them.

6.4. Implications for researchers and tool builders

Our study does not aim to point out which testing strategy has a better performance on finding configurations that will be recommended. We only use the testing strategies' outcome (i.e., their suggested configurations) to run the respective testing suite for the recommended configurations and report results from the subject dataset. Therefore, we are not comparing the testing strategies themselves but their recommended configurations.

Our effectiveness results (Section 5.3) show that even when using sophisticated t-wise testing strategies, it is difficult to find faults with the generated configurations. In the best case using a t-wise strategy, we retrieve 30.62% of faults. Then, when we investigate where the faults arise, we found out that faults are concentrated in 1% of the classes and in only few features (see Sections 5.6 and 5.7). Hence, we conclude that these components need to be exhaustively tested. In other words, these components need more attention than others once developers aim at minimizing the number of faults in the production-phase.

We see two prominent directions to improve testing strategies and to increase the chance of finding more feature interaction faults. The first uses artificial intelligence techniques to retrieve information of faulty-features and faulty-classes (e.g., using metrics such we used in Sections 5.6 and 5.7 or bug reports history). Hence, this information can be used as an additional source for t-wise strategies. The second regards the creation of a strategy that interactively asks developers for components that they want to prioritize. Hence, the strategy should generate a set of configurations that exhaustively investigate the target components.

Both researchers and tool builders benefit from these directions. While researchers have the opportunity to propose testing strategies using information not yet used for this purpose, tool builders can automate new strategies leaving them ready for practitioners to use.

7. Threats to validity

Even with a careful planning, this research can be affected by different factors which might threat our findings. We discuss these factors and decisions to mitigate their impact on our study divided into external and internal threats to validity below.

External Validity. External validity is threatened mainly by two factors. First, our restriction to variability encoding as variability approach and Java as programming language. The generalization to other variability approaches, a programming languages, and configurable systems is limited. This limitation of the sample was necessary to reduce the influence of confounds, increasing internal validity, though (Siegmund and Schumann, 2015). While more research is needed to generalize to other variability approaches, programming languages, and configurable systems, we are confident that we selected and analyzed a practically relevant variability approach and a substantial number of configurable systems from various domains, longevity, size, and valid configurations. Limiting the programming language is a common limitation of several research studies on configurable software systems (Kim et al., 2012, 2013; Meinicke et al., 2014; Souto et al., 2017; Wong et al., 2018).

Second, the testing suite and faults found in the dataset. Our results are restricted to the test suite and faults found in the subject dataset. Using other systems, different versions of the subject systems, or different testing suites may come up with other results. Aiming at minimizing this threat, we chose a dataset previously proposed (Ferreira et al., 2020a) that follows two reasonable thresholds to increase the quality of the testing suite: 70% of code coverage and 40% of killed mutants. We provided an overview of it in Section 3.

Internal Validity. There are three major threats to the internal validity of our study. First, we could have wrongly implemented the subject testing strategies. However, as we use the implementation provided on FeatureIDE, we are confident that they were correctly implemented.

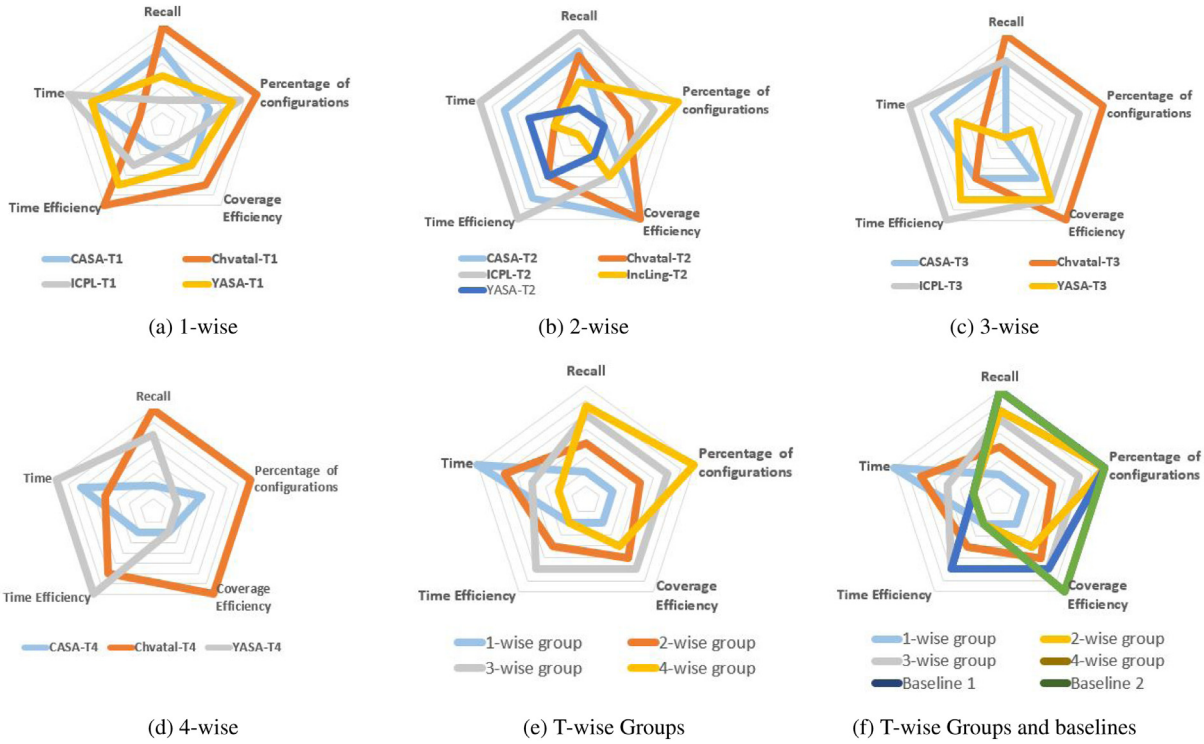


Fig. 5. Summary of t-wise strategies comparison.

Second, we cannot ensure that we identified all faults present in the subject systems. To increase testing coverage, we used two baselines. While the first baseline runs the test suite in all valid configurations for up to 250 valid configurations sequentially chosen, the second baseline runs the testing suite for up to 250 valid configurations randomly chosen. This way we run all valid configurations for several subject systems. In addition, our reference list is the union set of all faults found by all configurations run by the ten subject testing strategies. Hence, we are not prioritizing faults found by one testing strategy, and all testing strategies have the same chance of finding faults.

Third, we may have not chosen the best metrics to represent systems, features, and classes. We selected several well-known metrics to quantify the size, features, classes, and test suite that compose the configurable systems of the subject dataset. To make this measurement process simpler and automated, we used well-known tools, such as CK Tool (CK, 2020) and JaCoCo (JaCoCo, 2020) and, for the metrics that well-known tools are not able to compute, we create a script. We manually checked the measurement of 8% of the components (e.g., classes and features) to confirm their results. Hence, we believe that similar conclusions would also be achieved using different metrics and tools that quantify similar attributes for the same set of configurable systems and components.

8. Related work

Even though there are dozens of papers related to testing configurable systems (Engström and Runeson, 2011; Lamancha et al., 2013; Lee et al., 2012; Lopez-Herrejon et al., 2015; Machado et al., 2014; da Mota et al., 2011; Pohl and Metzger, 2006), there is a lack of studies comparing testing strategies for a wide-community dataset. In this section, we present studies that investigate faults, feature interactions, and compare testing strategies for configurable systems.

Faults on Configurable Systems. Lopez-Herrejon et al. (2014) proposed a dataset with 19 configurable systems obtained through studies published in the five events (SPLC, VaMoS, ICSE, ASE, and FSE), and four repositories publicly available (SPL Conqueror, FeatureHouse, SPL2go, and SPLOT). Our study has three systems in common (ArgoUML-SPL, GPL, and ZipMe). Their work analyzed configurations recommended by *CASA*, *PGS*, and *ICPL* testing strategies in terms of size, performance, similarity, and frequency, considering information from the feature model. The objective of their work is to evaluate the effectiveness of the testing approaches which is similar to one of our goals. In addition of having additional goals, we evaluate a larger set of systems, other testing strategies (*Chvatal* and *IncLing*), and other properties (e.g., time, comprehensiveness, time-efficient, and coverage-efficient). In line with our results, *ICPL* was one of the fastest strategies.

Sánchez et al. (2017) mined a list of faults from Drupal. Drupal is a modular web content management framework written in PHP with 48 features and 21 cross-tree constraints. They identified 3392 faults, and report feature interactions associated with these faults in two analyzed Drupal versions (v7.22 and v7.23). They characterized Drupal through the number of changes (during two years), cyclomatic complexity, number of test cases, number of test assertions, number of developers, and number of reported installations. Different from them, we use a much larger set of metrics and multiple systems. Our results are inline with their study concerning the number of code lines handled by the feature and weight method class can be used as good estimators for identifying fault-prone features and classes.

Fischer et al. (2018) proposed a dataset with six configurable systems for evaluating the fault detection capabilities of configurable systems testing strategies. Our study has two configurable systems in common (GPL and Notepad). They reported that one of the main limitations of their work was the automatic generation of test cases using the EvoSuite tool. Our study addresses this limitation since we manually developed a test suite for systems without one and extending the test suite of other systems that already had one. We believe that our test suites are more comprehensive than theirs, which reflects also on the number of

faults we found. Similar to their work, we introduced mutations to emulate faults and verify the effectiveness of the test suite created.

Feature Interaction Investigations. Considering that identify feature interaction faults is expensive and challenging, several studies in the literature have proposed different approaches to deal with the feature interaction (Garvin and Cohen, 2011; Machado et al., 2014; Schuster et al., 2014; Siegmund et al., 2012; Soares et al., 2018a,b). As an example, Soares et al. (2018a) proposed a strategy called VarXplorer, which focuses on pairwise feature interactions. VarXplorer is an incremental and interactive lightweight process to detect problematic interactions dynamically. Through VarXplorer, the user identifies features that should not interact, and this way, a specification of the features is created. Our work also investigates feature interaction faults. However, we use the suite of automated and configurations recommended by eight t-wise strategies to inspect features. Our results demonstrate that it is possible to identify feature interaction problems through automated tests. The creation of a formal specification is costly, may be incomplete, ambiguous, or have errors. As an alternative to find feature interaction faults, automated testing can be a viable alternative.

Comparison of Testing Strategies Designed for Configurable Systems. Medeiros et al. (2016) investigated faults using ten sampling algorithms (SPLCATool, CASA, ACTS, Statement-coverage, Most-enabled–disabled and Random) and 135 configuration-related faults from 24 subject systems developed in C programming language and ten different types of faults such as Memory Leaks. The main difference from our study is that we evaluated the configurations recommended by t-wise strategies in Java-based configurable systems and we provide analyses investigating the dispersion of faults on classes and features. Furthermore, while in their study, researchers manually created a corpus of 135 faults, we identified faults automatically. Regarding the difference on results, Medeiros et al. (2016) showed that the recommended configurations of their selected sampling algorithms include at least 66% of the 135 faults. In our study, the t-wise strategies found around 1/3 of the faults found. Anyway, it is hard to compare these results because the set of systems, strategies used as well as the programming language are different and they may provide different results (see discussion presented in Section 7). In any event, similar to them, we found t-wise strategies with greater "*t*" have greater coverage. In our work, a 4-wise strategy found about 1/3 of the faults and in their work a 6-wise strategy found all 135 reported faults.

9. Conclusion and future work

In this work, we compare the performance of the configurations suggested by eighteen t-wise strategies (five t-wise strategies with sixteen variations plus two baselines). In this comparison, we used a dataset previously proposed with 30 configurable systems and with a test suite available for each of them. Regarding the performance of a t-wise strategy, we identified which testing strategies provide configurations that were often faster, more comprehensive (i.e., with greater coverage), more effective, more time-efficient, and more coverage-efficient. In addition, we investigated the dispersion of faults over classes and features and investigated if it is possible to distinguish failing classes and failing features from classes and features safe of faults.

As a result, we found that for each group t-wise: (i) *ICPL-T1*, *ICPL-T2*, *ICPL-T3* and *YASA-T4* are usually fast in relation to groups 1-, 2-, 3-, and 4-wise, respectively. (ii) *Chvatal-T1*, *IncLing-T2*, *Chvatal-T3*, and *Chvatal-T4* are the most comprehensive testing strategy. (iii) *Chvatal-T1*, *ICPL-T2*, *Chvatal-T3*, and *Chvatal-T4* are the testing strategies that recommends configurations able to find the greatest number of faults. (iv) *Chvatal-T1*, *ICPL-T2*, *Chvatal-T3*, and *YASA-T4* are the testing strategy that recommends configurations able to find the best balance among faults found and time, and (v) *Chvatal-T1*, *CASA-T2*, *Chvatal-T3*, and *Chvatal-T4* are the testing strategies that recommends configurations able to find the best balance among faults found and the number of recommended configurations in relation to groups 1-, 2-, 3-, and 4-wise, respectively. Moreover, we found that faults are usually concentrated in a few classes and features and these fault-prone components are distinguishable from components safe of faults only measuring their source code with metrics normally used in practice.

Our results can be used by practitioners to support their decision of which testing strategy to use, know characteristics of classes and features that normally fail on testing, and guide them on increasing test coverage on fault-prone components. Researchers and tool builders may also benefit from our study since we provide several directions for improving existing testing strategies. These directions include, for instance, using source code metrics to identify fault-prone components and use this information on existing testing strategies. Or, use a variant *t* on the testing strategy depending on the number of valid configurations or constraints provided by practitioners when using the tool.

As future work, we suggest the replication of this study with configurable systems developed with other programming languages, other set of configurable systems, or further versions of the subject configurable systems. In another direction, someone could evaluate how this study impacted on the evolution of the subject configurable systems. Finally, we also suggest a study that includes other testing strategies beyond t-wise strategies, such as statement-coverage and one-disable or one-enable.

CRediT authorship contribution statement

Fischer Ferreira: Conceptualization, Methodology, Software, Formal analysis, Investigation, Resources, Data curation, Writing - original draft, Writing - review & editing, Visualization, Project administration. **Gustavo Vale:** Methodology, Validation, Investigation, Writing - original draft, Writing - review & editing, Visualization. **João P. Diniz:** Conceptualization, Methodology, Validation, Review & editing. **Eduardo Figueiredo:** Conceptualization, Methodology, Validation, Supervision, Project administration, Funding acquisition.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

This research was partially supported by Brazilian funding agencies: CNPq (Grant 424340/2016-0), CNPq (Grant 290136/2015-6), CAPES, and FAPEMIG (grant PPM-00651-17).

References

Abal, I., Brabrand, C., Wasowski, A., 2014. 42 Variability bugs in the linux kernel: a qualitative analysis In: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE),

Al-Hajjaji, M., Krieter, S., Thüm, T., Lochau, M., Saake, G., 2016. IncLing: Efficient product-line testing using incremental pairwise sampling. In: 15th Proceedings of the ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE), pp. 144–155.

Al-Hajjaji, M., Meinicke, J., Krieter, S., Schröter, R., Thüm, T., Leich, T., Saake, G., 2017. Tool demo: Testing configurable systems with featureide. In: International Conference on Generative Programming: Concepts & Experiences (GPCE). pp. 173–177.

Al-Hajjaji, M., Thüm, T., Meinicke, J., Lochau, M., Saake, G., Similarity-based prioritization in software product-line testing. In: Proceedings of the 18th International Software Product Line Conference (SPLC), pp. 197–206.
Apel, S., Batory, D.S., Kästner, C., Saake, G., 2013a. Feature-oriented software product lines – concepts and implementation.
Apel, S., Leich, T., Saake, G., 2008. Aspectual feature modules. IEEE Trans. Softw. Eng. (TSE) (2), 162–180.
Apel, S., Rhein, A.v., Wendler, P., Grosslinger, A., Beyer, D., 2013b. Strategies for product-line verification: Case studies and experiments. In: Proceedings of the 2013 International Conference on Software Engineering (ICSE), pp. 482–491.
Apel, S., Speidel, H., Wendler, P., Rhein, A.V., Beyer, D., 2011. Detection of feature interactions using feature-aware verification. In: 26th Proceedings of the International Conference on Automated Software Engineering (ASE), pp. 372–375.
Basili, V., Rombach, H.D., 1988. The TAME project: Towards improvement-oriented software environments. IEEE Trans. Softw. Eng. (TSE) 758–773.
Batory, D., 2005. Feature models, grammars, and propositional formulas. In: International Conference on Software Product Lines (SPLC). pp. 7–20.
Chidamber, S.R., Darcy, D.P., Kemerer, C.F., 1998. Managerial use of metrics for object-oriented software: An exploratory analysis. Trans. Softw. Eng. (TSE) 629–639.
Chvatal, V., 1979. A greedy heuristic for the set-covering problem. Math. Oper. Res. 233–235.
CK, 2020. CK. https://github.com/mauricioaniche/ck, 16-Aug-2020.
Cohen, M.B., Dwyer, M.B., Shi, J., 2008. Constructing interaction test suites for highly-configurable systems in the presence of constraints: A greedy approach. IEEE Trans. Softw. Eng. (TSE) 633–650.
Cohen, M.B., Gibbons, P.B., Mugridge, W.B., Colbourn, C.J., 2003. Constructing test suites for interaction testing. In: Proceedings of the International Conference on Software Engineering (ICSE), pp. 38–48.
Coles, H., Laurent, T., Henard, C., Papadakis, M., Ventresque, A., 2016. Pit: A practical mutation testing tool for java. In: 25th International Symposium on Software Testing and Analysis (ISSTA). pp. 449–452.
Couto, M.V., Valente, M.T., Figueiredo, E., 2011. Extracting software product lines: A case study using conditional compilation. In: 15th European Conference on Software Maintenance and Reengineering (CSMR). pp. 191–200.
Czarnecki, K., Eisenecker, U., 2000. Generative Programming: Methods, Tools, and Applications. Addison-Wesley.
Engström, E., Runeson, P., 2011. Software product line testing - a systematic mapping study. Inform. Softw. Technol. (IST) 2–13.
Ferreira, F., Diniz, J.P., Silva, C., Figueiredo, E., 2019. Testing tools for configurable software systems: A review-based empirical study. In: Proceedings of the 13th International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS), pp. 1–10.
Ferreira, F., Vale, G., Diniz, J.P., Figueiredo, E., 2020c. Community-wide dataset of configurable systems. https://fischerjf.github.io/Community-wide-Dataset-of-Configurable-Systems, Accessed 15-Aug-2020.
Ferreira, F., Vale, G., Figueiredo, E., Diniz, J.P., 2020a. On the proposal and evaluation of a test-enriched dataset for configurable systems. In: Proceedings of the 14th International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS), pp. 1–10.
Ferreira, F., Viggiato, M., Souza, M., Figueiredo, E., 2020b. Configurable software systems: The failure observation challenge. In: 2020 24th ACM International Systems and Software Product Line Conference (SPLC).
FEST, 2020. FEST. http://code.google.com/p/fest/, Accessed 16-Aug-2020.
Fischer, S., Lopez-Herrejon, R., Egyed, A., 2018. Towards a fault-detection benchmark for evaluating software product line testing approaches. In: 33rd Annual ACM Symposium on Applied Computing (SAC). pp. 2034–2041.
Galindo, J.A., Turner, H., Benavides, D., White, J., 2016. Testing variability-intensive systems using automated analysis: An application to android. Softw. Q. J. (SQJ) 365–405.
Garvin, B., Cohen, M., 2011. Feature interaction faults revisited: An exploratory study. In: International Symposium on Software Reliability Engineering (ISSRE). pp. 90–99.
Garvin, B.J., Cohen, M.B., Dwyer, M.B., 2011. Evaluating improvements to a meta-heuristic search for constrained interaction testing . Empirical Softw. Eng. (ESE) (1), 61–102.
Greiler, M., Deursen, A.v., Storey, M.-A., 2012. Test confessions: a study of testing practices for plug-in systems. In: Proceedings of the International Conference on Software Engineering (ICSE), pp, 244–254.
Halin, A., Nuttinck, A., Acher, M., Devroey, X., Perrouin, G., Baudry, B., 2019. Test them all, is it worth it? Assessing configuration sampling on the jhipster web development stack. Empir. Softw. Eng. (ESE) 674–717.
Henard, C., Papadakis, M., Perrouin, G., Klein, J., Heymans, P., Le Traon, Y., 2014. Bypassing the combinatorial explosion: Using similarity to generate and prioritize t-wise test configurations for software product lines. IEEE Trans. Softw. Eng. (TSE) 650–670.
Hervieu, A., Baudry, B., Gotlieb, A., 2011. Pacogen: Automatic generation of pairwise test configurations from feature models . In: 22th International Symposium on Software Reliability Engineering (ISSRE). pp. 120–129.
Inozemtseva, L., Holmes, R., 2014. Coverage is not strongly correlated with test suite effectiveness. In: Proceedings of the 36th International Conference on Software Engineering (ICSE), pp. 435–445.
Ivanković, M., Petrović, G., Just, R., Fraser, G., 2019. Code coverage at google. In: Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), pp. 955–963.
JaCoCo, 2020. JaCoCo. https://www.eclemma.org/jacoco/, Accessed 16-Aug-2020.
JavaPP, 2021. Javapp. https://www.slashdev.ca/javapp/, Accessed 12-Mar-2021.
Johansen, M.F., Haugen, Ø., Fleurey, F., 2011. Properties of realistic feature models make combinatorial testing of product lines feasible. In: 14th International Conference on Model Driven Engineering Languages and Systems (MODELS). pp. 638–652.
Johansen, M.F., Haugen, Ø., Fleurey, F., 2012a. An algorithm for generating T-wise covering arrays from large feature model. In: 16th Proceedings of the International Software Product Line Conference (SPLC), pp. 46–55.
Johansen, M.F., Haugen, Ø., Fleurey, F., Eldegard, A.G., Syversen, T., 2012b. Generating better partial covering arrays by modeling weights on sub-product lines. In: International Conference on Model Driven Engineering Languages and Systems (MODELS). pp. 269–284.
JUnit, 2020. JUnit. https://junit.org/junit5/, Accessed 16-Aug-2020.
Just, R., Jalali, D., Inozemtseva, L., Ernst, M.D., Holmes, R., Fraser, G., 2014. Are mutants a valid substitute for real faults in software Testing? In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 654–665.
Kaltenecker, C., Grebhahn, A., Siegmund, N., Guo, J., Apel, S., 2019. Distance-based sampling of software configuration spaces. In: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). pp. 1084–1094.
Kastner, C., 2010. Virtual Separation of Concerns: Toward Preprocessors 2.0 (Ph.D. thesis). Otto-von-Guericke-Universitat Magdeburg.
Kim, C.H.P., Bodden, E., Batory, D., Khurshid, S., 2010. Reducing configurations to monitor in a software product line . In: 18th International Conference on Runtime Verification (RV). pp. 285–299.
Kim, C.H.P., Khurshid, S., Batory, D., 2012. Shared execution for efficiently testing product lines . In: 23th International Symposium on Software Reliability Engineering (ISSRE). pp. 221–230.
Kim, C.H.P., Marinov, D., Khurshid, S., Batory, D., Souto, S., Barros, P., d'Amorim, M., 2013. Splat: Lightweight dynamic analysis for reducing combinatorics in testing configurable systems. In: ESEC/FSE. pp. 257–267.
Krieter, S., Thüm, T., Schulze, S., Saake, G., Leich, T., 2020. YASA: Yet another sampling algorithm. In: Proceedings of the 14th International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS), pp. 1–10.
Kruse, P.M., Shehory, O., Citron, D., Condori-Fern'andez, N., Vos, T., Mendelson, B., 2014. Assessing the applicability of a combinatorial testing tool within an industrial environment. In: 11th Workshop on Experimental Software Engineering (ESELAW).
Kuhn, D.R., Kacker, R.N., Lei, Y., 2010. Practical Combinatorial Testing. Technical Report SP 800-142, National Institute of Standards & Technology, Gaithersburg, MD, United States.
Kuhn, D.R., Reilly, M.J., 2002. An investigation of the applicability of design of experiments to software testing. In: Proceedings of the 27th Annual NASA Goddard/IEEE Software Engineering Workshop, pp. 91–95.
Kuhn, D.R., Wallace, D.R., Gallo, Jr., A.M., 2004. Software fault interactions and implications for software testing. IEEE Trans. Softw. Eng. (TSE) 418–421.
Lamancha, P., Polo, M., Piattini, M., 2013. Systematic review on software product line testing. In: Software and Data Technologies (ICSOFT). pp. 58–71.
Lee, J., Kang, S., Lee, D., 2012. A survey on software product line testing. In: 16th International Software Product Line Conference (SPLC). pp. 31–40.
Lei, Y., Kacker, R., Kuhn, D.R., Okun, V., Lawrence, J., 2008. IPOG/IPOG-D: efficient test generation for multi-way combinatorial testing. In: Software Testing, Verification and Reliability (STVR). pp. 125–148.
Li, L., Martinez, J., Ziadi, T., Bissyandé, T.F., Klein, J., Traon, Y.L., 2016. Mining families of android applications for extractive SPL adoption. In: Proceedings of the 20th International Systems and Software Product Line Conference (SPLC), pp. 271–275.
Liebig, J., Von Rhein, A., Kästner, C., Apel, S., Dörre, J., Lengauer, C., 2013. Scalable analysis of variable software. In: Proceedings in 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE), pp. 81–91.
Linux, 2020. Linux kernel. http://www.kernel.org, Accessed 16-Aug-2020.
Lopez-Herrejon, R.E., Ferrer, J., Chicano, F., Haslinger, E.N., Egyed, A., Alba, E., 2014. Towards a benchmark and a comparison framework for combinatorial interaction testing of software product lines. CoRR arXiv:1401.5367.
Lopez-Herrejon, R.E., Fischer, S., Ramler, R., Egyed, A., 2015. A first systematic mapping study on combinatorial interaction testing for software product lines. In: 8th International Conference on Software Testing, Verification and Validation Workshops (ICSTW). pp. 1–10.
Lübke, D., Greenyer, J., Vatlin, D., 2019. Effectiveness of combinatorial test design with executable business processes. In: Empirical Studies on the Development of Executable Business Processes. pp. 199–223.

Machado, I., McGregor, J., Cavalcanti, Y., Almeida, E., 2014. On strategies for testing software product lines: A systematic literature review. Inform. Softw. Technol. (IST) 1183–1199.
Martinez, J., Assunção, W.K., Ziadi, T., 2017. ESPLA: A catalog of extractive SPL adoption case studies. In: 21st International Systems and Software Product Line Conference (SPLC). pp. 38–41.
Medeiros, F., Kästner, C., Ribeiro, M., Gheyi, R., Apel, S., 2016. A comparison of 10 sampling algorithms for configurable systems. In: 38th International Conference on Software Engineering (ICSE). pp. 643–654.
Meinicke, J., Thüm, T., Schröter, R., Benduhn, F., Saake, G., 2014. An overview on analysis tools for software product lines. In: 18th Proceedings of the International Software Product Line Conference: Companion Volume for Workshops, Demonstrations and Tools (SPLC), pp. 94–101.
Meinicke, J., Wong, C.-P., Kästner, C., Thüm, T., Saake, G., 2016. On essential configuration complexity: Measuring interactions in highly configurable systems. In: 31st Proceedings of the International Conference on Automated Software Engineering (ASE), pp. 483–494.
Metrics, 2020. Metrics. http://metrics.sourceforge.net/, Accessed 16-Aug-2020.
Mockito, 2020. Mockito. https://site.mockito.org/, Accessed 16-Aug-2020.
da Mota, P.A., Carmo Machado, I.d., McGregor, J.D., de Almeida, E.S., de Lemos Meira, S.R., 2011. A systematic mapping study of software product lines testing. Inform. Softw. Technol. (IST) 407–423.
Mozilla, 2020. Firefox web browser. http://hg.mozilla.org, Accessed 16-Aug-2020.
Nguyen, H.V., Kästner, C., Nguyen, T.N., 2014. Exploring variability-aware execution for testing plugin-based web applications. In: Proceedings of the International Conference on Software Engineering (ICSE), pp. 907–918.
Nguyen, S., Nguyen, H., Tran, N., Tran, H., Nguyen, T., 2019. Feature-interaction aware configuration prioritization for configurable code. In: 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 489–501.
Nie, C., Leung, H., 2011. A survey of combinatorial testing. Comput. Surv. 11:1–11:29.
Oster, S., Zink, M., Lochau, M., Grechanik, M., 2011. Pairwise feature-interaction testing for SPLs: Potentials and limitations. In: 15th Proceedings of the International Software Product Line Conference, Volume 2 (ISPLC), pp. 6.
Pohl, K., Metzger, A., 2006. Software product line testing. Inform. Softw. Technol. (IST) 78–81.
Post, H., Sinz, C., 2008. Configuration lifting: Verification meets software configuration. In: Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 347–350.
Puoskari, E., Vos, T.E., Condori-Fernandez, N., Kruse, P.M., 2013. Evaluating applicability of combinatorial testing in an industrial environment: A case study. In: Proceedings of the 2013 International Workshop on Joining AcadeMiA and Industry Contributions To Testing Automation, pp. 7–12.
Sánchez, A.B., Segura, S., Parejo, J.A., Cortés, A.R., 2017. Variability testing in the wild: the drupal case study. Softw. Syst. Model. (SoSyM) 173–194.
Santos, A., Alves, P., Figueiredo, E., Ferrari, F., 2016. Avoiding code pitfalls in aspect-oriented programming. Sci. Comput. Program. (SCP) 119, 31–50.
Schaefer, I., Bettini, L., Bono, V., Damiani, F., Tanzarella, N., 2010. Delta-oriented programming of software product lines. In: International Conference on Software Product Lines. pp. 77–91.
Schaefer, I., Bettini, L., Damiani, F., 2011. Compositional type-checking for delta-oriented programming. In: Proceedings of International Conference on Aspect-Oriented Software Development (AOSD), pp. 43–56.
Schuster, S., Schulze, S., Schaefer, I., 2014. Structural feature interaction patterns: Case studies and guidelines. In: Proceedings of the Eighth International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS), pp. 1–8.
Sheskin, D.J., 2020. Handbook of Parametric and Nonparametric Statistical Procedures. crc Press.
Siegmund, N., Kolesnikov, S., Kästner, C., Apel, S., Batory, D., Rosenmüller, M., Saake, G., 2012. Predicting performance via automated feature-interaction detection. In: Proceedings 34th International Conference on Software Engineering (ICSE), pp. 167–177.
Siegmund, J., Schumann, J., 2015. Confounding parameters on program comprehension: A literature survey. Empir. Softw. Eng. (ESE).
Soares, L., Meinicke, J., Nadi, S., Kästner, C., de Almeida, E., 2018a. Varxplorer: Lightweight process for dynamic analysis of feature interactions. In: Proceedings of the 12th International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS), pp. 59–66.
Soares, L.R., Schobbens, P., do Carmo Machado, I., de Almeida, E.S., 2018b. Feature interaction in software product line engineering: A systematic mapping study. Inform. Softw. Technol. (IST) 98, 44–58.
Souto, S., d'Amorim, M., Gheyi, R., 2017. Balancing soundness and efficiency for practical testing of configurable systems. In: 39th Proceedings of the International Conference on Software Engineering (ICSE), pp. 632–642.
SPL2go, 2020. SPL2go. http://spl2go.cs.ovgu.de/projects/, Accessed 16-Aug-2020.
Svahnberg, M., Van Gurp, J., Bosch, J., 2005. A taxonomy of variability realization techniques. Softw. - Pract. Exp. 705–754.
Thüm, T., Kästner, C., Benduhn, F., Meinicke, J., Saake, G., Leich, T., 2014. Featureide: An extensible framework for feature-oriented software development. Sci. Comput. Program. (SCP) 70–85.
Thüm, T., Schaefer, I., Apel, S., Hentschel, M., 2012. Family-based deductive verification of software product lines. In: Proceedings of the 11th International Conference on Generative Programming and Component Engineering (GPCE), pp. 11–20.
Vale, G., Albuquerque, D., Figueiredo, E., Garcia, A., 2015. Defining metric thresholds for software product lines: A comparative study. In: Proceedings of the 19th International Conference on Software Product Line (SPLC), pp. 176–185.
Varshosaz, M., Al-Hajjaji, M., Thüm, T., Runge, T., Mousavi, M.R., Schaefer, I., 2018. A classification of product sampling for software product lines. In: Proceedings of the 22nd International Systems and Software Product Line (SPLC), pp. 1–13.
Wong, C., Meinicke, J., Lazarek, L., Kästner, C., 2018. Faster variational execution with transparent bytecode transformation. In: Proceedings of the ACM on Programming Languages (OOPSLA).
Xiang, Y., Huang, H., Li, M., Li, S., Yang, X., 2021. Looking for novelty in search-based software product line testing. IEEE Trans. Softw. Eng. (TSE) 1.
Yu, L., Lei, Y., Nourozborazjany, M., Kacker, R.N., Kuhn, D.R., 2013. An efficient algorithm for constraint handling in combinatorial test generation. In: Proceedings of the 6th International Conference on Software Testing, Verification and Validation (ICST), pp. 242–251.

Fischer Ferreira is Ph.D. student in Computer Science at the Federal University of Minas Gerais - UFMG (2016). He holds a Master's in Informatics from Pontifical Catholic University of Rio de Janeiro (2015) and a degree in Analysis and Development of Systems from Federal University of Rio Grande (2011). Fischer is a research scholar in Software Engineering Laboratory (LabSoft).

Gustavo Vale is bacharel of Information Systems by Federal University of Lavras (UFLA) and Master in Computer Science by Federal University of Minas Gerais (UFMG). He is a member at team PqES and Lab-Soft from UFLA and UFMG, respectively. Furthermore, he is currently Ph.D. student in the department of Computer Science at Saarland University.

João P. Diniz Ph.D. Student of Department of Computer Science (DCC), at UFMG. M.Sc. of Computer Science by Federal University of Minas Gerais. Bachelor of Computer Science by Federal University of Minas Gerais. Sun Certified Programmer for the Java 2 Platform (SCJP), Standard Edition 5.0.

Eduardo Figueiredo is currently a lecturer at Federal University of Minas Gerais (UFMG) and head of the Software Engineering Laboratory (LabSoft). He holds a Ph.D. degree in Computing at Lancaster University, U.K. (2009), a M.Sc. degree in Informatics at Pontifical Catholic University of Rio de Janeiro (2006), and a graduate degree in Computer Science at Federal University of Ouro Preto (2003). His research interest includes: software metrics, empirical software engineering, software reuse, software product lines, and aspect-oriented software development. Eduardo has also co-organized a number of workshops, such as ACoM, ESCOT, and LA-WASP.