On the Challenges for Creating a Test Suite for Configurable Software Systems

Fischer Ferreira[1], João P. Diniz[1], Gustavo Vale[2], and Eduardo Figueiredo[1]

[1] Federal University of Minas Gerais (UFMG)
[2] Saarland University, Saarbrücken, Germany
{fischerjf, jpaulo, figueiredo}@dcc.ufmg.br, vale@cs.uni-saarland.de

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. Considering a gap of knowledge on creating tests for configurable software systems, we propose a list of ten challenges faced when performing test suites for configurable systems and dealing with a test suite for our dataset systems. Our list includes, for instance, the challenges of testing high coupled classes and of determining metrics for measuring the quality of the test suite. Our results can be seen as lessons learned on creating tests for configurable systems and they aim at supporting researchers and practitioners on this activity for configurable systems.

Keywords: Testing Configurable Systems, Software Faults, Feature Interaction, Challenges

1 Introduction

Configurable software systems (for short, *configurable systems*) are systems that can be efficiently extended, changed, or customized based on a set of features that fits specific customer needs [3]. Features allow configurable systems to have an optional or incremental unit of functionality [4]. The Web browser Firefox, Linux Kernel, and Android family are well-known examples of configurable systems.

Testing configurable systems is even more challenging than testing traditional software systems since instead of testing a product, configurable systems entail testing the set of all possible combinations of features. As the size of possible configurations to be tested grows exponentially, the testing effort becomes costly and often impractical [17, 21, 24]. Furthermore, dealing with all feature combinations may generate undesired feature interaction [31]. Unexpected feature interactions occurs when features that work correctly separately cause unexpected behavior when executing together with other features [31]. Given the recurring issues caused by unexpected feature interactions, several researchers have proposed approaches and techniques to deal with them [23, 26].

Although previous work concentrates on *the explosion of combinations* and *feature interactions* challenges, we present other challenges faced when testing configuration systems in practice. For instance, the challenges (1) on the creation of a test suite, (2) on measuring the test suite and its quality, and (3) on the identification of faults are often ignored. We defined ten challenges related to configurable software testing based through our experience when creating and extending the test suite for our dataset systems [11, 12]. Our main goal is to report the main challenges in the complete life-cycle of 30 open-source configuration systems [9–12]. Therefore, instead of focus on the well-known challenges present in the literature, we present challenges faced when creating, extending, assessing, and using test suites for 30 configurable systems. For brief, our list of challenges is:

1. *Creating from scratch and expanding test suites*
2. *Creating test cases in highly coupled classes*
3. *Dealing with the combinatorial explosion of configurations*
4. *Sampling configurations for test*
5. *Running the test suites*
6. *Assessing the quality of the automated test suites*
7. *Measuring the test suites*
8. *Dealing with false positives from tests*
9. *Tracking feature interaction faults*
10. *Finding technical debts in test cases of configurable systems*

We believe that presenting the faced challenges on the complete life-cycle of 30 configurable systems is just a starting point for researchers to deeply investigate each challenge and propose different ways to deal with them. Practitioners can also benefit from our work by looking at our solutions on each challenge and, hopefully, save time. At the end, we learned that testing configurable systems is related to the ability to automatically run test cases for all possible/desired configurations and identify faults efficiently. Furthermore, a high concentration of faults in a region of code (i.e., a method, class, or feature) might also be an indicative of technical debt in such region [10].

2 Background and Related Work

Configurable systems can be efficiently customized according to a set of features (configuration options) that satisfies different customer needs [32]. 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* [3].

In compile-time strategies, developers activate a set of features that is preprocessed in order to generate the final product [3]. This strategy can be divided into annotative and compositional approaches. Conditional compilation (CC) is a technique that uses the former approach [28], while Feature-oriented Programming (FOP) [4], Aspectual Feature Modules (AFM) [3], and Delta-oriented Programming (DOP) [29] are examples of techniques that implement the latter. 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, while in compositional approaches, developers implement each variation point in modularized components (e.g., features [4] in FOP, and deltas [29] in DOP). The Linux Kernel and the Firefox Web Browser are examples of configurable systems developed with compile-time strategies.

In execution-time strategies, developers activate features that contain code blocks at execution-time [28]. Variability encoding is an example of execution-time strategy [3]. Similar to conditional compilation, developers should create conditional structures (e.g., `if/else` statements or ternary operator `?:`), and generate the so-called *meta-products* [34]. 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 [21, 24]. Android family is a (set of) configurable system(s) developed using variability encoding [24]. In this study, configurable systems are systems that the features code are enabled or disabled at runtime. We do not consider program parameters [3] and compile-time strategies to determine configurations.

2.1 Feature Interaction Problem

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 [31]. Unexpected behavior can introduce faults that manifest themselves in specific configurations. Feature interactions are among the greatest challenges in developing configurable systems [31] and enforces the need to create testing suites that cover all potential interactions [26]. These faults are usually caused by a problem with the interactions of two or more features.

```java
public class WeatherReport {
  private Date date;
  private String temperature;

  public WeatherReport(Date currentDate, String
      currentTemperature){
    this.date = currentDate;
    this.temperature = currentTemperature;
  }

  public String createText(String c) {
    if (Configuration.SMILEY)
      c = c.replace(":]", getSmiley(":]"));
    if (Configuration.WEATHER)
      c = c.replace("[:weather:]", temperature);
    return c;
  }
}
```

Listing 1.1. Variability encoding example adapted from [24]

Listing 1.1 presents a traditional and simple example of feature interaction faults [24]. In this example, the `createText` method edits and returns a string *c*. Case only feature SMILEY is active, the method replaces *:]* by ☺. Looking at Listing 1.1, we see that both features use a string *":]"*. It may cause a feature interaction fault. To illustrate, let us consider that *createText* method receives as input the string *"[:weather:]"* (i.e., *c* = *[:weather:]*) and both features SMILEY and WEATHER are enabled. This way feature SMILEY will change string *c* to *[:weather*☺ and the code of feature WEATHER will not find the expected string *c* anymore, even thought it was exactly the expected input string before.

2.2 Related Work

There are dozens of papers that describe the challenge of testing configurable systems [21, 23, 24, 32]. However, previous work has focused on reporting the combinatorial difficulty explosion of the number of configurations to test and feature interaction faults. This section discusses two studies related to these challenges. The first discusses the difficulty in dealing with the combinatorial explosion of the number of configurations. The second discusses feature interaction faults.

Generating test configurations. Souto et al. [32] discusses the generation of test configurations in two different approaches through the sound test technique that generates all configurations. However, they argue that the approach does not scale to a system with many features. In addition, they discuss an alternative approach to test all possible configurations through sample testing. Their work aims to demonstrate the costs to test configurable systems and present a sample test solution to reduce this cost. Different from them, we demonstrate our personal observations based on previous empirical studies on other aspects related to the test phase. For this, we have listed seven challenges and our initial ideas for dealing with the challenges listed.

Feature Interaction Faults. Soares et al. [31] discusses aspects related to challenges arising from feature interaction faults by observing 35 primary studies. Through the observations of their work, it is possible to note the community's concern for decades about the problems related to the interaction of features. Our work demonstrates seven challenges related to the test phase, which are ultimately related to this major objective of identifying feature interaction faults.

3 Our Experience on Testing Configurable Systems

Considering that the lack of studies reporting and investigating the challenges on the complete life-cycle of testing configurable systems, we aim at filling this gap by **reporting the challenges we faced when creating, extending, assessing, using, and identifying faults on the test suites of 30 configurable systems** [11]. These challenges can benefit both researchers and practitioners since the creation, use, and maintenance of test suites are a fundamental step for testing configurable systems and our lessons learned can be used, for instance, as a starting point for future research.

Figure 1 provides an overview of the steps of our research, highlighting the lessons learned at each step. We also use the knowledge on creating test cases to propose the ten challenges in this paper. To achieve our goal, we performed a systematic mapping study (step 1), created and/or extended test suites for 30 configurable systems (step 2), empirically investigated testing strategies to generate a set of configurations to be tested (step 3 and step 4), and summarize the observed challenges during the life-cycle of testing configurable systems (step 5). Note that, despite we learned lessons in the first four steps, the focus of this paper is the last step. We spent an average of one year to complete the studies of Steps 1, 2, 3, and 4.

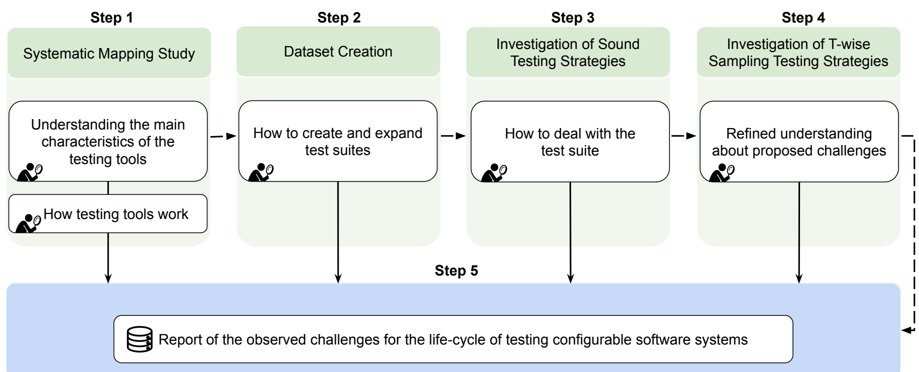

Fig. 1. Study steps

Step 1. We performed a systematic mapping study aiming at identifying the existing testing tools for configurable systems [9]. In this systematic mapping study, we (i) provided an overview of the state-of-the-art of testing strategies for configurable systems, (ii) analyzed the main characteristics of the testing tools for configurable systems, and (iii) presented the main strategies used to test configurable systems. We found a total of 34 tools and collected the main characteristics of testing tools for configurable systems. Through this step, we understand how the testing tools for configurable systems work and their main characteristics. In addition, Step 1 was essential to discover what are the main restrictions of testing configurable systems [9].

Step 2. We conduct an *ad hoc* literature review by analyzing survey papers on testing configurable systems and well-known datasets of configurable systems (e.g., SPL2go and SPLRepository) [11]. Despite we have found 60 configurable systems developed in Java-based programming languages, only 10 configurable systems have their test suite public available. Aiming at increasing this number, we create a test suite for further 20 systems. The creation of tests followed two stop criteria: (i) at least 70% of code coverage and (ii) at least 40% of the mutants killed. These two criteria aims at providing a high-quality testing dataset. At the end, we created test suites for 30 configurable systems. With the creation and expansion of the test suites for the dataset systems, we identified the main challenges faced. This the first dataset has been used by the community to research testing approaches in configurable systems [12].

Step 3. We conducted a comparative study [9], we evaluated the two main sound tools found in the systematic mapping studies: *VarexJ* [24] and *SPLat* [21]. The main goal of that study was to identify advantages and drawbacks on testing tools that use sound testing techniques. Hence, we measured the effectiveness and efficiency of *VarexJ* and *SPLat* to test seven systems acquired in the step 2. We only use seven configurable systems from our dataset due to the limitations of the non-scaling approach for systems with many features. We observed that, even though both tools use sound testing technique strategy, they generally did not present high intersection rates [9]. Alternatively, our experiments have shown that for configurable systems with up to 17 features, it was possible to use testing strategies that test all configurations.

Step 4. In this study [10], we compared variations of five t-wise sampling testing strategies, named *CASA* [16], *Chvatal* [18], *ICPL* [18], *IncLing* [2], and *YASA* [22]. We selected t-wise strategies because they ensure a degree of coverage for determining the set of configurations recommended by the strategy. At the end, we compared 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*).

Step 5. We summarize the main challenges observed in the previous steps. These steps are the main outcome of this study and are presented in the next section.

4 Ten Challenges for Testing Configurable Systems

This section presents the ten main challenges observed when creating, extending, assessing, using, and identifying faults on the test suites of 30 configurable systems in our previous studies [9–12]. For each challenge, we present our solution. Each following subsection describes a challenge.

4.1 Creating from Scratch and Expanding Test Suites

Both creating and expanding a test suite for configurable systems are challenging because, when designing test cases, the requirements of the configurable system, the variability of features, and their functionality should be taken into account. That is, it has to be adjusted in terms of which test cases should be performed according to the state of the features (active/inactive) since it is not feasible to select each test for the configuration to be tested manually.

The test suite has to be automatically called and only the tests related to the current configuration must be performed. In this way, in the design of the test cases, we have to consider active feature for the test case to be executed. For that, a mechanism must be made in each test case to test the requirements using the specific active features. On the other hand, such a mechanism is necessary to prevent the test cases related to inactive features from being executed. If this occurs, the test result will be a false positive.

Solution. Listing 1.2 shows a test class for the example presented in Listing 1.1. In addition to the environment set up, it shows a test case testing the `createText` method when the feature WEATHER is active. For short, taking *"[:weather:]"* as input, it should provide *"30.0° C"* as output. Note that as we have only 2 features, it is easy to create test cases to cover all 4 possible configuration scenarios (i.e., 2^2). However, as the number of related features increases, it is not feasible anymore. To deal with that, we inserted a general configuration file loaded before the execution of the test suite. A further observation is that automated testing tools are not helpful on creating test cases, specially when expanding a test suite. Initially, we tried to use tools to automatically generate test cases (e.g., Randoop [27] and EvoSuite [14]). However, after the generation of a small number of test cases, the subsequent new ones usually did not increase the test coverage as well as the percentage of killed mutants. As result, we suggest that testers use these tools carefully to only generate preliminary versions of their test suites.

```java
public class WeatherReportTest {
  WeatherReport wr;
  @Before
  public void setUp() {
    wr = new WeatherReport("2019-11-13", "30.0°C");
  }

  @Test
  public void weatherTest () {
   if (Configuration.WEATHER)
    assertEquals(wr.createText("[:weather:]"), "30.0°C");
  }
}
```

Listing 1.2. Test Cases for the WeatherReport Class

4.2 Creating Test Cases in Highly Coupled Classes

To have a deep understanding on the projects of our dataset [11], we first needed to read the documentation. After, we spent a considerable time understanding their architecture and source-code. We noted that high coupling classes made the understanding challenging. It also reflected on the creation of test cases because we had to take care of relationships that, in the first moment, were hidden.

Solution. To overcome this scenario, we had to simulate some particular setup that in the first moment was challenging to achieve given high coupling among classes. After creating test cases for several projects, it became easier because we could reuse some strategies, or find some pieces of code that helped with feature interaction problems. We use Mockito [25] to allow the creation of mock object simplifying the development of tests for classes with dependencies of other classes [33].

4.3 Dealing with the Combinatorial Explosion of Configurations

The main challenge researchers and software companies face when developing configurable software systems is to test them. Indeed, adding configuration options yields a combinatorial explosion on the number of configurations to test. Therefore, it is impractical to test them all. For example, a configurable system with 320 features can yield more possibilities (i.e., software variants) than the number of atoms in the universe. It is an insuperable challenge to test all possibilities of configurations for systems with many features. While in traditional software systems there is only one configuration (a combination of features) to be tested, for configurable systems we need to run all tests in several different configurations.

Configurable software testing can be divided into two main parts, generating the configurations and running the automated test suite for these configurations. To generate configurations, we may use strategies that have been developed to test configurable systems [9]. These strategies can be classified into: variability-aware testing [21, 24] and configuration sampling testing [2, 32]. *Variability-aware testing strategies* explore dynamically all reachable configurations from a given test, by monitoring feature variable accesses during test execution. Concerning running the test suite for the generated configurations, it is possible to make a configuration partition and test the configurations by groups.

Solution. To test the most significant number of configurations in our study, we run all valid configurations for configurable systems with up to 17 features. However, for the other systems, we have created a configuration list for testing that is the union set all configurations run by the sixteen testing strategies (see Step 4, Section 3). Using multiple sampling strategies allows us to achieve a wide range of configurations at a low cost [10].

4.4 Sampling Configurations for Test

Even if testing all possible configurations of a configurable system is possible, the test suite's quality positively impacts the efficiency on observing faults. Generating the configurations and running the test suite is time consuming. Recall that the time consumption can consist of the sum of the time to generate configurations (by sampling strategies) and the time to run the test suite for the configurations [10]. The ideal scenario is to test all valid configurations but, in practice, the explosion of configurations for testing makes exhaustive testing prohibitively expensive and practically infeasible. Alternatively, developers may test a sample of valid configurations [2]. Several strategies for choosing a sample of configurations to test have been proposed. Some of them use information only from the *feature model* [2, 18], while others also use information from the source code [21, 32].

Solution. Nowadays, several sampling test strategies are available [2, 16, 18, 22], but, the choice of a testing strategy should be adapted to the time available for testing. Even using a sample strategy testing, the total testing time can take hours [10]. In this way, during the configurable system life-cycle, different test strategies can be used to meet the constraints of efficiency and effectiveness. For instance, *1-wise* strategies (e.g., *Chvatal-T1* and *ICPL-T1*) prioritize a minimal number of configurations and are able to find faults. On the other hand, if developers needs a more robust check and can wait longer, they can use strategies that return a greater number of configurations, such as *Chvatal-T4* and *ICPL-T3*. Testing few configurations in the test phase of configurable systems can be an effective alternative as a preliminary and superficial evaluation of the configurable system. However, a preliminary assessment may not discover all feature interaction faults, as these types of faults are harder to discover. In this way, using an efficient strategy can support testing for configurable systems.

4.5 Running the Test Suites

The number of configurations for testing determines the number of times the test suite runs. Therefore, configurable systems with hundreds of configurations result in hundreds of runs of the test suite. Hence, the process of running the suite for each configuration must be automated. Faults in configurable systems can occur in the same test case, but with different configurations. Therefore, there is no point in running the test suite several times if it is impossible to track where the faults occurred.

Algorithm 1 Test suite execution manager

Require: List of configurations selected for testing
Ensure: Test reports
$totalConfigurations \leftarrow size(ListOfConfigurations)$
$config \leftarrow 1$
while $config \leq totalConfigurations$ **do**
    c=nextConfiguration(ListOfConfigurations)
    setFeatures(c);
    Execute the test suite with configuration c
    $config \leftarrow config + 1$
end while

Solution. We created in our studies a way to record the output of JUnit's execution [19], always using as a pair the configuration for testing and the line of the test case where the fault was triggered. We use the processor defined in previous work to record JUnit's execution [9]. In this way, in the subsequent analysis of the faults report, we can quickly identify which configuration fails and which part of the test case code triggered the faults. Algorithm 1 shows a pseudo-code demonstrating how to run the entire test suite for each configuration selected for testing strategy.

4.6 Assessing the Quality of the Automated Test Suites

Despite of some techniques proposed for traditional systems to evaluate the quality of tests (e.g., mutation analysis technique), we still face with the challenge of testing multiple configurations in configurable systems.

Mutation testing is a fault-based technique commonly used to evaluate the effectiveness of software testing [20]. Even though it is a self cost technique, it has been empirically shown to be one of the most robust test selection criteria when compared to other measures, such as control flow and data flow-based testing [20]. Mutation testing consists of introducing syntactical changes, called mutations, into the source code and check whether the test cases distinguish them [8]. The resulting programs are called mutants. A mutant is killed if a test case observes the modification of the mutant from the original code. If the test suite cannot see changes, the mutant remains alive indicating that the test suite is not good enough to identify faults and developers should improve their test suite to kill surviving mutants.

Solution. To use this technique in a test suite for a configurable system, a viable alternative is to choose among the valid configurations, a single configuration that exercises a greater number of test cases. Then, it is possible to use a test case generation tool, such as PIT [6]. Through the generated mutants it is possible to check the quality of the test cases to kill the introduced mutants. New test cases can be created to kill living, non-equivalent mutants [8]. We use the mutant generation process described in the previous work [10].

4.7 Measuring the Test Suites

You cannot control something if you cannot measure it [7]. Hence, we needed to find ways to measure what the test suite covers. Metrics like code coverage and the number of test cases are not enough to know if the test suite covers a particular feature. The testers can prioritize the test for specific features because they have more accessible testing features and do not cover with testing other features in the same way. Therefore, we need to measure how the test suite covers the features.

Solution. Specific metrics to identify test suite coverage can also help the tester creating new test cases. For example, if a particular feature has higher priority for users of the configurable system, this feature should be heavily tested. In our studies, we measured the distribution of the test suite for each feature by calculating the variability points for each class and identifying which features reach the variability point. After, we checked which test cases can test each code fragment of the variability point. This way, we were able to estimate how the test suite tested the features.

4.8 Dealing with False Positives from Tests

The identification of false positives requires a manual verification effort. False positives occur mainly due to three problems: (i) poor test case design, (ii) lack of memory, and (iii) timeout. *Poor test case design.* The project of the test suite must consider all the features that are part of the tested functionality. Otherwise, the test case may fail for some configurations on which the target feature is not active. The problem is that this false positive can occur in only a few configurations, giving the impression of the feature interaction faults when a problem occurs in the test case design. Tests can also fail regardless of the configuration. These false positives are easier to identify, but they can cause problems when many faults have to be analyzed. A viable alternative is to correct the defects of these tests first since they fail in all configurations. Thus, with any sample configuration for testing, it is possible to identify these tests.

Lack of memory and Timeout. We analyzed the systems with several sample sizes. We noticed that when running the test suite for many configurations, the execution sometimes crashes due to a lack of memory. To identify problems with lack of memory, we identified the test reports provided by JUnit. To deal with this, we ran the test suite for batches of configuration. *Timeout.* Test cases can have problems with timeout definitions that can cause the test case to fail improperly. However, this type of fault is easily identified through the faults report provided by JUnit.

Solution. A viable alternative for identifying false positives in tests of configurable systems is to identify the failing configurations reported by the tests. Then, set the features according to the failed configuration, to the configurable system and execute only the failed test case. The analysis must occur initially to verify that all features are in the correct state. Otherwise, we checked if the computational state is correct. If no error is found in the configurable system code, the test case scenario should be reviewed.

4.9 Tracking Feature Interaction Faults

Identifying the origin of faults is challenging in configurable systems because the feature interaction must be considered. We perceived that the source code implementing some features are often tangled. We noticed two constraints needed to discover feature interaction faults. First, the sampling of configurations for testing must include at least one with feature interaction faults.

Second, at least one test case must identify the feature interaction faults. Failure to comply with these two constraints results in the tests' inability to observe faults. The first constraint takes up the challenge listed in Section 4.3 but, as discussed, it is impracticable to test all possible configurations for systems with several features. Features with feature interaction faults can appear in several valid configurations. However, if only one faulty configuration is found, it is sufficient to solve the problem reflected in several other configurations.

Solution. We observed in our study that a test case can fail several times in the same line of code but with different configurations. The developer solving the feature interaction faults of the configuration found will be correcting the faults in several other configurations that have the same feature combination. In this way, a viable alternative to identify feature interaction faults is to use testing strategies by sampling and creating compelling test suites to find faults.

Furthermore, we suggest to perform, for the list of configurations that caused faults, in each test case, the *frequent itemset mining analysis* [1]. A frequent itemset is defined as a set of items that occur together in at least a support threshold value of all available transactions . Therefore, we expect to identify which features and their respective states are most present in the faults. It is important to note that this technique can provide itemsets of different sizes, facilitating the analysis.

4.10 Finding Technical Debts in Test Cases of Configurable Systems

Patterns and catalogs have been used in software engineering to deal with challenges. As examples of catalogs, we can cite *design patterns* [15], *bad smells* [13], and *architectural patterns* [5]. We performed this last challenge to encourage the creation of standards and catalogs for testing configurable system. There are dozens of testing strategies that test by sampling [23, 10, 9]. However, there is still no consensus on which strategy to use under some constraints, such as time, the criticality of configurable systems, and the number of features.

In addition, a failure taxonomy could be created. In this sense, Soares et al. [30] proposed a list of types of feature interaction faults including Bad Annotation (the expression used in the annotation is incorrect), Wrong Object (an object used in place of another), Misplaced Variable Overwrite (the value of a variable is overwritten in a wrong place with a wrong value), Conditional Statement (incorrect implementation of conditional statements, using wrong or incomplete conditions), and Spread Code (piece of code spread over different features leading to unnecessary dependencies and problems related to code modularity).

Solution. Aiming at comparing sampling testing strategies and understanding the location of faults, we use our dataset and compare suggested configurations from variations of five t-wise testing strategies (see Step 4, Section 3). This comparison aims to find which strategies are faster, more comprehensive, effective on identifying faults, time-efficient, and coverage-efficient in this dataset and the reasons why a strategy fared better in one investigated property. As a result, we had a first step toward the identification of patterns and problems in code that may easy or hinder testing.

5 Final Remarks

This paper discussed ten challenges related to the life-cycle of tests for configurable systems. We summarized the challenges through our observations based on the background acquired from previous empirical studies. We believe that researchers and practitioners can benefit from the described challenges and can propose solutions for them making our challenges just a starting point for future research. As future work, we plan to investigate other challenges and different solutions related to tests of configurable systems.

ACKNOWLEDGMENTS

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

References

1. Agrawal, R., Srikant, R., et al.: Fast algorithms for mining association rules. In: Proceedings. 20th International Conference Very Large Data Bases, VLDB. vol. 1215, pp. 487–499. Citeseer (1994)
2. Al-Hajjaji, M., Krieter, S., Thüm, T., Lochau, M., Saake, G.: 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 (2016)
3. Apel, S., Batory, D.S., Kästner, C., Saake, G.: Feature-Oriented Software Product Lines – Concepts and Implementation (2013)
4. Batory, D.: Feature Models, Grammars, and Propositional Formulas. In: International Conference on Software Product Lines (SPLC). pp. 7–20 (2005)
5. Buschmann, F., Henney, K., Schmidt, D.: Pattern-oriented software architecture, on patterns and pattern languages, vol. 5. John wiley & sons (2007)
6. Coles, H., Laurent, T., Henard, C., Papadakis, M., Ventresque, A.: Pit: A Practical Mutation Testing Tool for Java. In: In 25th International Symposium on Software Testing and Analysis (ISSTA). pp. 449–452 (2016)
7. DeMarco, T.: Controlling software projects: Management, measurement, and estimates. Prentice Hall PTR (1986)
8. DeMillo, R.A., Lipton, R.J., Sayward, F.G.: Hints on test data selection: Help for the practicing programmer. Computer **11**(4), 34–41 (1978)
9. Ferreira, F., Diniz, J.P., Silva, C., Figueiredo, E.: Testing Tools for Configurable Software Systems: A Review-based Empirical Study. In: Proceedings of the 13th International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS). pp. 1–10 (2019)
10. Ferreira, F., Vale, G., Diniz, J.P., Figueiredo, E.: Evaluating T-wise Testing Strategies in a Community-wide Dataset of Configurable Software Systems. Journal of Systems and Software (JSS) (2021)
11. Ferreira, F., Vale, G., Figueiredo, E., Diniz, J.P.: 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 (2020)
12. Ferreira, F., Viggiato, M., Souza, M., Figueiredo, E.: Configurable Software Systems: The Failure Observation Challenge. In: 2020 24th ACM International Systems and Software Product Line Conference (SPLC) (2020)
13. Fowler, M.: Refactoring: improving the design of existing code. Addison-Wesley Professional (2018)
14. Fraser, G., Arcuri, A.: Evosuite: automatic test suite generation for object-oriented software. In: Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering. pp. 416–419 (2011)
15. Gamma, E.: Design patterns: elements of reusable object-oriented software. Pearson Education India (1995)
16. Garvin, B.J., Cohen, M.B., Dwyer, M.B.: Evaluating Improvements to a Metaheuristic Search for Constrained Interaction Testing. Empirical Software Engineering (ESE) (1), 61–102 (2011)
17. Halin, A., Nuttinck, A., Acher, M., Devroey, X., Perrouin, G., Baudry, B.: Test Them all, is it Worth it? Assessing Configuration Sampling on the JHipster Web Development Stack. Empirical Software Engineering (ESE) pp. 674–717 (2019)

18. Johansen, M.F., Haugen, Ø., Fleurey, F., Eldegard, A.G., Syversen, T.: 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 (2012)
19. JUnit: JUnit (Accessed 16-Aug-2020), https://junit.org/junit5/
20. Just, R., Jalali, D., Inozemtseva, L., Ernst, M.D., Holmes, R., Fraser, G.: 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 (2014)
21. Kim, C.H.P., Marinov, D., Khurshid, S., Batory, D., Souto, S., Barros, P., d'Amorim, M.: SPLat: Lightweight Dynamic Analysis for Reducing Combinatorics in Testing Configurable Systems. In: ESEC/FSE. pp. 257–267 (2013)
22. Krieter, S., Thüm, T., Schulze, S., Saake, G., Leich, T.: YASA: Yet Another Sampling Algorithm. In: Proceedings of the 14th International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS). pp. 1–10 (2020)
23. Medeiros, F., Kästner, C., Ribeiro, M., Gheyi, R., Apel, S.: A Comparison of 10 Sampling Algorithms for Configurable Systems. In: In 38th International Conference on Software Engineering (ICSE). pp. 643–654 (2016)
24. Meinicke, J., Wong, C.P., Kästner, C., Thüm, T., Saake, G.: 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 (2016)
25. Mockito: Mockito (Accessed 16-Aug-2020), https://site.mockito.org/
26. Oster, S., Zink, M., Lochau, M., Grechanik, M.: Pairwise Feature-interaction Testing for SPLs: Potentials and Limitations. In: 15th Proceedings of the International Software Product Line Conference, Volume 2 (ISPLC). p. 6 (2011)
27. Pacheco, C., Ernst, M.D.: Randoop: feedback-directed random testing for java. In: OOPSLA Companion. pp. 815–816 (2007)
28. Post, H., Sinz, C.: Configuration Lifting: Verification Meets Software Configuration. In: Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 347–350 (2008)
29. Schaefer, I., Bettini, L., Bono, V., Damiani, F., Tanzarella, N.: Delta-oriented Programming of Software Product Lines. In: International Conference on Software Product Lines. pp. 77–91 (2010)
30. Soares., L.: Feature Interactions in Highly Configurable Systems: A Dynamic Analysis Approach with VarXplorer. Ph.D. thesis, Federal University of Bahia (UFBA) (2019)
31. Soares, L.R., Schobbens, P., do Carmo Machado, I., de Almeida, E.S.: Feature Interaction in Software Product Line Engineering: A Systematic Mapping Study. Information and Software Technology (IST) **98**, 44–58 (2018)
32. Souto, S., d'Amorim, M., Gheyi, R.: Balancing Soundness and Efficiency for Practical Testing of Configurable Systems . In: In 39th Proceedings of the International Conference on Software Engineering (ICSE). pp. 632–642 (2017)
33. Spadini, D., Aniche, M., Bruntink, M., Bacchelli, A.: Mock objects for testing java systems. Empirical Software Engineering (3), 1461–1498 (2019)
34. Thüm, T., Schaefer, I., Apel, S., Hentschel, M.: 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 (2012)