Testing Configurable Software Systems: The Failure Observation Challenge

Fischer Ferreira
Federal University of Minas Gerais
Belo Horizonte, Minas Gerais, Brazil

Markos Viggiato
University of Alberta
Edmonton, Alberta, Canada

Maurício Souza
Federal University of Lavras
Lavras, Minas Gerais, Brazil

Eduardo Figueiredo
Federal University of Minas Gerais
Belo Horizonte, Minas Gerais, Brazil

ABSTRACT

Configurable software systems can be adapted or configured according to a set of features to increase reuse and productivity. The testing process is essential because configurations that fail may potentially hurt user experience and degrade the reputation of a project. However, testing configurable systems is very challenging due to the number of configurations to run with each test, leading to a combinatorial explosion in the number of configurations and tests. Currently, several testing techniques and tools have been proposed to deal with this challenge, but their potential practical application remains mostly unexplored. To encourage the research area on testing configurable systems, researchers and practitioners should be able to try out their solutions in common datasets. In this paper, we propose a dataset with 22 configurable software systems and an extensive test suite. Moreover, we report failures found in these systems and source code metrics to allow evaluating candidate solutions. We hope to engage the community and stimulate new and existing approaches to the problem of testing configurable systems.

KEYWORDS

Testing Configurable Systems; Software Product Line;

ACM Reference Format:
Fischer Ferreira, Markos Viggiato, Maurício Souza, and Eduardo Figueiredo. 2018. Testing Configurable Software Systems: The Failure Observation Challenge. In *SPLC '20: INTERNATIONAL SYSTEMS AND SOFTWARE PRODUCT LINE CONFERENCE, October 19–23, 2020, Montréal, Canada.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3302333.3302344

1 INTRODUCTION

Configurable systems are software systems that can be adapted or configured according to a set of features (configuration options). Configurable systems offer numerous options (or features) to fit specific customer needs [4, 32, 36], and developers may activate or deactivate options to address a diversity of deployment contexts and usages. To ensure that all configurations correctly compile, build, and run, developers usually spend considerable effort testing their systems because configurations that fail may hurt potential users and degrade the reputation of a project [15, 38].

Software testing is a key component for ensuring that all configurations work properly. However, testing configurable systems is more challenging than testing monolithic systems. While in monolithic software systems there is only one product/configuration (a combination of features) to be tested, for configurable systems we need to run all tests in several different configurations, which leads to a combinatorial explosion of configurations and tests. Therefore, testing thoroughly, against all configurations, is a costly practice. Alternatively, a popular strategy used in industry is to run the tests for a subset of default configurations. This approach is efficient, but it can miss bugs [14, 24].

Besides those two cases (testing only default configurations or exhaustively testing all configurations), several approaches for testing configurable systems have been proposed [7, 11, 20, 22, 26, 30, 31, 34]. Some of them consider only the feature model [7, 22, 26, 31] in order to define products to be tested. However, they may explore configurations not reached by tests. Other approaches [20, 30, 34] take the code (test or source) into account in addition to the feature model, and dynamically explore all reachable configurations from a given test. However, such dynamic techniques only explore configurations related to testing.

Even with a large number of testing techniques and tools for configurable systems [7, 22, 26, 30, 31, 34], their potential practical application remains mostly unexplored. We still lack a deeper understanding of the effectiveness of test strategies for configurable systems. In this context, we contribute to the research of testing configurable systems with a dataset composed of 22 configurable systems and an extensive test suite proposed in our prior work [12]. Therefore, we challenge the research community to use their testing strategies for configurable systems to find failures in the target systems of our proposed dataset.

We argue that the provided dataset is well suited as subjects for the challenge of finding failures due to the variety of configurable systems and because each configurable system has an extensive test suite. We expect participants to evaluate their solutions by measuring how strongly their testing strategies for configurable systems can be in finding failures in the systems of our dataset. Each solution could be assessed concerning how efficient and effective the solution is for testing configurable systems. We propose the use of the *recall* traditional metric and a new metric named *solution*

SPLC'20, October 19–23, 2020, Montréal, Canada

ACM ISBN 978-1-4503-6648-9/19/02…$15.00
https://doi.org/10.1145/3302333.3302344

efficiency to verify the effectiveness and efficiency of the proposed strategy. Through these metrics, it is possible to measure the number of priority test configurations and the number of failures found by the proposed testing strategies.

Challenge: the participants must propose a testing strategy for configurable systems. The efficiency of the strategy needs to be higher than our baseline. However, its effectiveness must be maintained. In other words, the proposed testing strategies must maximize recall and minimize the number of tested configurations.

We propose two metrics as a measure of effectiveness and efficiency. The dataset of configurable systems with extensive test suites and reports of failures can be found at:

https://fischerjf.github.io/challenge/

2 BACKGROUND

This section presents an overview of variability encoding (Section 2.1) and presents an overview of testing on configurable systems (Section 2.2).

2.1 Variability Encoding Overview

The variations in configurable systems of our dataset use the variability encoding (execution-time) technique. Configurable systems have long been studied by the software product line engineering community [4, 32] and represent a configurable set of systems that share a common, managed set of options (or features) to address specific needs. Among the strategies to introduce variability in software systems, variability encoding has drawn practitioners' attention since developers only need to *annotate variation points* on their existing systems. Thus, developers only activate or deactivate features to address different deployment contexts. For short, while annotating variation points, developers should create a configuration file where they determine options that are going to be active in a target variation.

Listing 1 presents a fragment of code in a configurable system, named Companies. In this example, the method *getTotal* returns a string containing a calculated value. If the feature TOTAL_WALKER is active, the method returns the value calculated by the TotalWalker class. If the feature TOTAL_REDUCER is active, the method returns the value calculated by the TotalReducer class. On the other hand, if the features TOTAL_WALKER and TOTAL_REDUCER are not active, no value is calculated.

```
1  public String getTotal() {
2     String value = "";
3     if (Configuration.TOTAL_WALKER) {
4         TotalWalker walker = new TotalWalker();
5         walker.postorder(currentValue);
6         value = Double.toString(walker.getTotal());
7     } else if (Configuration.TOTAL_REDUCER) {
8         TotalReducer total = new TotalReducer();
9         double valueDouble = total.reduce(currentValue);
10        value = Double.toString(valueDouble);
11    }
12    return value;
13 }
```

Listing 1: Variability encoding example

2.2 Testing Configurable Systems

The fact that the number of possible product variants grows exponentially with the number of variation points makes such thorough testing infeasible. In other words, above a certain amount of features, it is infeasible to test all possible feature combinations (e.g., using brute force algorithms). This way, researchers and practitioners have to choose somehow the configurations they want to test. However, it is not trivial to know which priority configuration to test.

Over the years, various approaches have been developed to test configurable systems [9, 10, 23]. These approaches can be classified into: *configuration sampling* [16, 18, 34] and *variability-aware testing* [21, 27, 39]. Configuration sampling approaches sample a representative subset of all valid configurations of the system and test them individually. Variability-aware testing approaches instrument the testing environment to take variability information and reduce the test execution effort. For instance, among the sampling approaches, the incremental sampling method generates and tests products one at a time to enhance the sampling efficiency in terms of the interaction coverage rate [3].

Some test strategies consider only the feature model to generate products to be tested, often performed on a small subset of configurations, which presumably covers a sufficient amount of functionality of the system. Other approaches take the code (test or source) into account in addition to the feature model, and dynamically explore all reachable configurations from a given test [21, 34]. These dynamic techniques do not explore configurations unrelated to the tests. However, the quality of the test suite of the configurable system strongly influences the performance of these approaches. If the test suite is unable to observe failures, strategies that find all possible configurations or a representative group of them may not observe failures effectively. The configuration sampling or variability-aware testing strategies only establish the valid configurations that discover the configurations for testing.

In this context, we encourage the community to use our dataset as it provides a test suite for each configurable system available. As an advantage of our challenge, testing strategies can benefit from the available test suite. Moreover, the large dataset of configurable systems and failures is a unique opportunity for us to characterize them. For instance, a deep understanding of feature interaction failures in configurable systems may help practitioners to identify the reasons for failures that occur in their systems.

3 DATASET OVERVIEW

In this section, we provide an overview of the proposed dataset. Section 3.1 presents the metrics that characterize the proposed dataset. We provide a summary of the test-enriched dataset in Section 3.2. Section 3.3 shows an overview of the test suite creation process for configurable systems. Section 3.4 presents a motivating example of using the dataset. In Section 3.5, we present the failures found in the proposed dataset. Finally, Section 3.6 describes the dataset artifacts.

3.1 Evaluation Metrics

For a better comprehension of the subject configurable systems in our dataset, we collected static and variability metrics. We collected

metrics with five different tools. Metrics related to the size of the configurable systems (e.g., number of lines of code and number of packages) were computed by CK tool and Metrics. CK tool [6] is an open-source tool hosted in GitHub that contains a broad set of code metrics at class-level and method-level for Java projects. As the name suggests, CK tool computes all metrics in the well-known CK suite. CK suite includes object-oriented metrics, such as Coupling between Objects (CBO), Weighted Methods for Class (WMC), and Depth of Inheritance Tree (DIT) [5]. Similar to the CK tool, Metrics [28] is a Eclipse plugin that supports various size metrics, such as the number of lines of code (# LOC), the number of classes (# Classes) and methods (# Methods).

Metrics related to the variability were extracted with FeatureIDE. For instance, we collected the number of features and valid configurations from the feature model of each subject system. We used JaCoCo [17] and PIT [8] to retrieve metrics related to the test suite.

3.2 Test-enriched Configurable System Dataset

Table 1 presents an overview of the 22 configurable systems that compose our dataset. We provide additional information about the proposed dataset in our supplementary website [1]. These systems belong to several domains, such as games, text editor, media management, and file compression. The columns of Table 1 represent the systems' name, size, variability, and test suite metrics. We discuss each of these metrics next.

Size metrics. We selected systems of different sizes. We measure the number of lines of code (#LOC), packages (#Packages), classes (#Classes), and methods (#Methods). The configurable systems in our dataset vary from 189 lines of code (BankAccount) to more than 150 000 lines of code (ArgoUML). Similarly, while FeatureAMP8 has only 8 classes and ArgoUML has almost 2 000 classes.

Variability metrics. We selected systems with different amount of variability. For instance, while CheckStyle has 141 features, two systems (Chess and Paycard) have only four features. This variation can also be seen in the number of valid configurations (#VC). For instance, while Paycard has 6 valid configurations, FeatureAMP3 has 20 500 valid configurations.

Test suite metrics. The process of creating the test cases uses information from the systems documentation. For some systems, we only extend their original test suites. We rely on the code coverage metric through the test suite and the percentage of mutants killed (Section 3.3) to evaluate the quality of tests. We believe that participants in our challenge can use these metrics to support their analysis. We not only report the number of test cases each test suite has, but also the number of lines of code of each test suite. The variety of sizes and characteristics of the configurable systems of our dataset can be a challenge for candidate solutions. In this way, we encourage participants to apply their strategies to our dataset and report on which situations their test strategies provide the best results.

3.3 Test Suite Creation

The creation or extension of the test suite consists of two tasks: *creating test cases* and *generating mutants*

[1] https://fischerjf.github.io/challenge/.

Creating test cases. We use JUnit framework to create test cases. JUnit [19] is a popular open source test framework that provides an environment for handling automated testing. JUnit makes it possible to create and run a suite through plugins that can be coupled with the main integrated development environment. Unit tests are performed to verify if a piece of code related to each feature behaves following a target configurable system requirements. In addition to JUnit, we use Mockito [29] and *FEST* [13] to allow the creation of mock object simplifying the development of tests for classes with external dependencies and to write tests for systems with graphical user interfaces (GUI), respectively. The stop criterion for creating test cases is a code coverage of 70%. This threshold is related to the generation of mutants as explained next.

Generating mutants. We introduce mutations to configurable systems and check whether mutants are killed based on two tasks. First, we execute PIT to generate and execute test suit against the mutants, and analyze its report. PIT [8] is a mutation testing tool for Java integrated with Maven, which makes it easy to set up the environment to evaluate test suites through mutant analysis testing. Then, we manually create new test cases to kill some of the live mutants and run JaCoCo to retrieve code coverage. JaCoCo [17] is a free code coverage library that analyzes each line of code to check if it was executed or not by a test case and then returns code coverage information. If 40% of the mutants are not killed, we repeat mutation generation and test case creation until the test suites reach 70% of code coverage and 40% of mutants killed for each subject configurable system.

Listing 2 demonstrates a unit test of the Companies test suite. For example, the code snippet from this test verifies if observers were added to the sub units (line 9) and checks if two observers were added to sample Company (line 10). We introduce a control condition to ensure that the test runs only if the features are active (`LOGGING` and `PRECEDENCE`), as shown in line 3 of Listing 2. This way, when this test runs, the features `LOGGING` and `PRECEDENCE` are set to TRUE. Therefore, through this instrumentation, test cases are modified to test configurable systems.

```
1  @Test
2  public void addObservers() {
3   if (Configuration.LOGGING && Configuration.PRECEDENCE) {
4     CompanyImpl sampleCompany = new CompanyImpl();
5     Logging log = new Logging();
6     sampleCompany.addObserver(log);
7     Precedence pre = new Precedence();
8     sampleCompany.addObserver(pre);
9     assertTrue(sampleCompany.observerAdded);
10    assertEquals(2, sampleCompany.countObservers());
11   }
12 }
```

Listing 2: Unit test example adapted from [34]

3.4 Example of Use

In this section, we present an example of configurable system in our dataset. We show our framework for running the test suite. With the support of our framework, participants can use the test suite for each configuration of features that their test strategies produce. Figure 1 shows the feature model of the configurable system called Companies. Companies is a human resource management system. Configurations enable various forms to calculate salary and give

Table 1: The Dataset

Name	Size metrics				Variability metrics			Test Suite metrics			
	#LOC	#Packages	#Classes	#Methods	#Features	#VC	#Var.	#Test	#LOCTest	#Coverage	#KM
ArgoUML[25]	153 977	92	1 812	13 034	8	256	1 388	1 326	17 014	17%	9%
ATM[33]	1 160	2	27	100	7	80	44	76	1 371	91%	79%
BankAccount[35]	189	3	9	22	10	144	13	42	539	92%	62%
Checkstyle[39]	61 435	14	78	719	141	$>2^{135}$	180	719	13 606	38%	5%
Chess[33]	2 149	7	22	162	3	8	20	77	1 296	72%	72%
Companies[34]	**2 477**	**16**	**50**	**244**	**10**	**192**	**255**	**42**	**1 850**	**70%**	**46%**
FeatureAMP1[35]	1 350	4	15	93	28	6 732	40	18	977	85%	46%
FeatureAMP2[35]	2 033	3	14	167	34	7 020	55	18	698	72%	43%
FeatureAMP3[35]	2 575	8	16	223	27	20 500	93	15	725	77%	42%
FeatureAMP4[35]	2 147	2	57	203	27	6 732	57	12	622	82%	40%
FeatureAMP5[35]	1 344	3	9	895	29	3 810	36	17	730	91%	49%
FeatureAMP6[35]	2 418	8	30	202	38	21 522	76	09	207	31%	43%
FeatureAMP7[35]	5 644	3	46	220	29	15 795	57	08	180	28%	40%
FeatureAMP8[35]	2 376	2	6	106	27	15 708	48	78	1 637	82%	42%
FeatureAMP9[35]	1 859	3	8	134	24	6 732	53	105	1 975	83%	63%
GPL[34]	1 235	3	17	78	13	73	59	51	1 162	83%	60%
MinePump[35]	244	2	7	26	7	64	4	34	459	91%	65%
Notepad [34]	1 564	4	17	90	17	256	24	25	1 790	59%	15%
Paycard[35]	374	2	8	27	4	6	10	13	453	88%	61%
Prop4J[35]	1 138	2	15	90	17	5 029	17	63	504	71%	67%
Sudoku[34]	949	2	13	51	6	20	53	35	650	80%	67%
Vending Machine[25]	472	2	7	21	8	256	7	37	297	97%	83%

#LOC, **#Packages**, **#Classes**, and **#Methods** stand for the number of lines of code, Packages, Classes, and Methods, respectively.
#Feat., **#VC**, and **#Var.** stand for the number of features, valid configurations and occurrences of variability in the source code, respectively.
#Test,**#LOCTest**, stand for the number of test cases, number of lines of code in the test suite, respectively.
Coverage and **#KM:** stand for percentage of coverage of the test suite, the percentage of killed mutants, respectively.

access to users according to their department. The configurable system Companies is composed of 13 features, 10 of which are concrete and three abstract. Figure 1 also shows one base and 6 optional features. Moreover, for the Companies, it is possible to find 192 valid configurations according to its feature model.

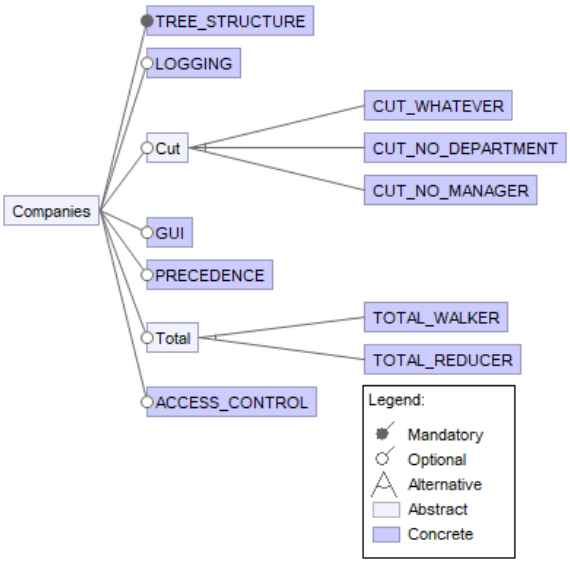

Figure 1: Feature Model of Companies

The challenged participants are expected to use our framework to run the test suite with their test solution. We present a small example to illustrate the use of our framework to call the test suite for each target system in our dataset. Figure 2 shows a valid configuration for the configurable system Companies. Each configuration must be in a file with only the active features listed, each feature in one line, as shown in Figure 2. Furthermore, each configurable system has a package called *"Experiment"*. The experiment package has the *"run"* method of the *" Challenge"* class that requires two parameters. The first is the target system (e.g., *"TargetSystem.COMPANIES"* in the given example) and the second one is the path of the configuration file. Using this structure, the entire test suite is called for each configuration provided. These procedures apply to all systems in our dataset.

```
TREE_STRUCTURE
CUT_WHATEVER
PRECEDENCE
ACESS_CONTROL
```

Figure 2: Configuration example

To exemplify the output, we report a feature interaction failure from the Companies configurable system. This system has a feature interaction problem related to the functionality of assigning a new salary to an employee of a company when the company expense cut is made. The related features are *PRECEDENCE* and *CUT_WHATEVER*. The business rule related to feature *PRECEDENCE* is to watch all salary changes in two situations: (i) an employee must have a lower salary than their direct manager and (ii) a manager of the upper department, if any, must have a higher salary than the manager of a target department. The business rule concerning feature *CUT_WHATEVER* refers to reduce salaries that may be in the whole company, a specific department, or a specific employee. For instance, when there is a salary reduction in a target department, the salary of each employee of this department is recalculated according to the established reduction. The failure we found (by the *"testTotalValueDepartment"*) happens when the feature *PRECEDENCE* is also active. Hence, another department may also have salary reduction. However, instead of retrieving the current salary, the recalculation is made on a previous salary, which makes the reduction larger than it should be.

Table 2: Failure Report

	Name	Baseline #Conf.	Baseline #FO	SE
DSA	ATM	80	80	1
	Bankaccount	144	6	0.041
	Chess	7	30	4.286
	Companies	**192**	**6**	**0.031**
	GPL	73	23	0.315
	MinePump	64	24	0.375
	Paycard	6	3	0.5
	Sudoku	20	4	0.2
DSB	ArgoUML	250	0	0
	Checkstyle	250	0	0
	FeatureAMP1	250	83	0.332
	FeatureAMP2	250	28	0.112
	FeatureAMP3	250	10	0.04
	FeatureAMP4	250	117	0.468
	FeatureAMP5	250	2	0.008
	FeatureAMP6	250	20	0.08
	FeatureAMP7	250	0	0
	FeatureAMP8	250	2	0.008
	FeatureAMP9	250	47	0.188
	Notepad	250	0	0
	Prop4J	250	0	0
	Vending Machine	250	0	0

#Conf. and **#FO** stand for the number of configurations analyzed and the number of failures occurred, respectively.
SE indicates the Solution Efficiency.

3.5 Failure Report

Table 2 presents a summary of failure reports for each configurable system in our dataset. We have divided our dataset into two parts: Dataset A (DSA) and Dataset B (DSB). Dataset A represents the configurable systems for which we exhaustively run all possible configurations. Dataset B refers to the systems for which we were unable to test all possible configurations, as shown in Table 2. As we can see, 16 systems present failures which represents 73% of the configurable systems in our dataset. In addition, we can see that 485 failures (#FO) was found. To illustrate this, we look at the data related to the Companies configurable system, which has 10 features (Table 1). We found 6 failures out of 192 analyzed configurations (Table 2).

We use a strategy provided by FeatureIDE, namely *All valid configurations* [2, 37] as a baseline for reporting failures in configurable systems. Since a thorough test against all configurations is a costly practice, we have selected the maximal number of up to 250 configurations because it provides the results in a feasible time for this study (up to 3 hours in a computer with 16 GB of RAM and an i7 3.60GHz processor). The last column of Table 2 presents a measurement of solution efficiency. We discuss this metric in Section 4. We make available on the dataset website the failure found, each configuration that failed, and the test cases that observed the reported failures.

3.6 Description of dataset artifacts

The challenge artifacts are available in the companion website of the dataset, organized into six items. We report the artifacts concerning the Companies configurable system. However, all other configurable systems in our dataset follow the same structure.

(1) **Feature Model**: We provide the feature model in two different file formats: Guidsl [2] and XML [3].
(2) **Metrics**: We provide a set of 14 metrics to characterize the dataset systems. We make these metrics available in a CSV file [4]. Each line of the file indicates a class of the configurable system, and the columns indicate the metrics. few examples of the available metrics are: i) CBO (Coupling between objects): it counts the number of dependencies a class has; and ii) WMC (Weight Method Class): it counts the number of branch instructions in a class. Participants can choose to use these metrics in their strategies if they wish.
(3) **Source Code**: We provide the source code and test suite for each configurable system [5]. These systems were encoded with the Java language using the variability encoding technique. Test approaches that prioritize test cases can choose a group of available test cases. However, new test cases should not be included in the challenge.
(4) **Found Failures**: We provide the found failures for the challenge systems [6]. We present these faults in a CSV file that contains the configuration in which the fault occurred, the stack trace, and the test case that observed the failure.
(5) **Analyzed Configurations**: We provide the settings that we run with our baseline [7]. The configuration file follows the model described in Figure 2.
(6) **Example configurations with IncLing tool**: We add some known sets of configurations already precalculated with the IncLing tool [8]. Participants can take this as an example of a possible solution to our challenge. **IncLing** [1] is an incremental sampling for pairwise interaction testing.

4 SOLUTION EVALUATION

This section presents the metrics that we use to measure how efficient and effective test strategies for configurable systems can be in observing failures.

4.1 Effectiveness Measurement for Dataset

We use the recall metric for the effectiveness of test strategies for configurable systems in Dataset A. The cases for which the strategy correctly found a failure are true positives (TP), also known as a hit. We consider an output correct when the analyzed tool agrees with the baseline in terms of the configuration that failed. We consider this case when the tool pointed out a failure that the baseline algorithm did not detect. Configurations that the brute force algorithm identified, but the candidate solution did not report are considered false negative (FN).

$$recall = \frac{TP}{(TP + FN)} \quad (1)$$

4.2 Measurement of Efficiency

For Dataset A and B, we provide the failures we could find by listing the first 250 valid configurations for each configurable system.

[2] https://github.com/fischerJF/challenge/blob/master/workspace_IncLing/companies/modified-model.m
[3] https://github.com/fischerJF/challenge/blob/master/workspace_IncLing/companies/model.xml
[4] https://github.com/fischerJF/challenge/blob/master/metrics/companies.csv
[5] https://github.com/fischerJF/challenge/tree/master/workspace_IncLing/companies
[6] https://github.com/fischerJF/challenge/blob/master/failuresFound/Bankaccount.csv
[7] https://github.com/fischerJF/challenge/tree/master/workspace_IncLing/Tools/All_valid_conf/companies/products
[8] https://github.com/fischerJF/challenge/tree/master/workspace_IncLing/Tools/IncLing/companies/products

However, it is possible to find other failures in unvisited configurations. In this way, we provide a metric that relates the number of failure occurrences to the number of configurations visited, as shown in Equation 2.

$$SE = \frac{FO}{(Conf.)} \quad (2)$$

5 CONCLUSION

We proposed a dataset with 22 configurable systems and an extensive test suite as a challenge for testing strategies for configurable software systems. We provide three groups of metrics (traditional, variability, and test suite) to characterize the proposed dataset for the challenge. Moreover, we found and reported a total of 485 feature interaction failures in 16 systems of the proposed dataset.

Several datasets for the configurable systems have been used. However, this dataset is the first dataset for configurable systems with an extensive test suite [12]. Furthermore, it is an excellent opportunity to share knowledge on test strategies for configurable systems because we use the same test suite towards an unbiased comparison of effectiveness and efficiency of testing strategies for configurable systems. We believe that our dataset can be a common point of comparison for configurable system testing strategies, and we encourage you to submit your solutions to the proposed challenge.

ACKNOWLEDGMENTS

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

REFERENCES

[1] M. Al-Hajjaji, S. Krieter, T. Thüm, M. Lochau, and G. Saake. 2016. IncLing: Efficient Product-line Testing Using Incremental Pairwise Sampling. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences* (Amsterdam, Netherlands) *(GPCE 2016)*. ACM, New York, NY, USA, 144–155. https://doi.org/10.1145/2993236.2993253

[2] M. Al-Hajjaji, J. Meinicke, S. Krieter, R. Schröter, and T. Leich T. Saake G. Thüm. 2017. Tool demo: testing configurable systems with FeatureIDE. *ACM SIGPLAN Notices* 52, 3 (2017), 173–177.

[3] M. Krieter S. Thüm T. Lochau M. Saake G Al-Hajjaji. 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)*. 144–155.

[4] S. Apel, S. Batory, C. Kästner, and G. Saake. 2013. *Feature-Oriented Software Product Lines: Concepts and Implementation.* Springer.

[5] S. R. Chidamber, D. P. Darcy, and C. F. Kemerer. 1998. Managerial use of metrics for object-oriented software: An exploratory analysis. *Transactions on software Engineering (TSE)* (1998), 629–639.

[6] CK. [n.d.]. CK. https://github.com/mauricioaniche/ck, Accessed 16-nov-2019.

[7] Myra B. Cohen, Peter B. Gibbons, Warwick B. Mugridge, and Charles J. Colbourn. 2003. Constructing test suites for interaction testing. In *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE, 38–48.

[8] H. Coles, T. Laurent, C. Henard, M. Papadakis, and A. Ventresque. 2016. Pit: a practical mutation testing tool for java. In *In 25th International Symposium on Software Testing and Analysis (ISSTA)*. 449–452.

[9] P. A. D. M. S. Neto, I. do Carmo Machado, J. D. McGregor, E. S. de Almeida, and S. R. de Lemos Meira. 2011. A Systematic Mapping Study of Software Product Lines Testing. *Information and Software Technology (IST)* (2011), 407–423.

[10] E. Engström and P. Runeson. 2011. Software Product Line Testing - A Systematic Mapping Study. *Information and Software Technology (IST)* (2011), 2–13.

[11] F. Ferreira, J. P. Diniz, C. Silva, and E. Figueiredo. 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)*.

[12] F. Ferreira, G. Vale, E. Figueiredo, and J. P. Diniz. 2020. On the Proposal and Evaluation of a Test-enriched Dataset for Configurable Systems. In *Proceedings of the 4th International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS)*.

[13] Fest. [n.d.]. Fest. http://code.google.com/p/fest/, Accessed 16-nov-2019.

[14] M. Greiler, A. Deursen, and M. Storey. 2012. Test Confessions: A Study of Testing Practices for Plug-in Systems. In *Proceedings of the International Conference on Software Engineering*. IEEE, 244–254.

[15] A. Halin, A. Nuttinck, M. Acher, X. Devroey, G. Perrouin, and B. Baudry. 2019. Test them all, is it worth it? Assessing configuration sampling on the JHipster Web development stack. *Empirical Software Engineering (ESE)* (2019), 674–717.

[16] IncLing. [n.d.]. IncLing Tool. http://www.tinyurl.com/IncrementalSampling, Accessed 15-nov-2019.

[17] JaCoCo. [n.d.]. Metrics. https://www.eclemma.org/jacoco/, Accessed 16-nov-2019.

[18] M. F. Haugen Ø. Fleurey F. Johansen. 2012. An Algorithm for Generating T-wise Covering Arrays from Large Feature Models. In *16th Proceedings of the International Software Product Line Conference - Volume 1 (SPLC)*. 46–55.

[19] JUnit. [n.d.]. JUnit. https://junit.org/junit5/, Accessed 16-nov-2019.

[20] P. Kim, S. Khurshid, and D. Batory. 2012. Shared Execution for Efficiently Testing Product Lines. In *Proceedings of the International Symposium on Software Reliability Engineering*. IEEE, 221–230.

[21] P. Kim, D. Marinov, S.z Khurshid, D. Batory, S. Souto, P. Barros, and M. d'Amorim. 2013. SPLat: lightweight dynamic analysis for reducing combinatorics in testing configurable systems. In *9th Proceedings of the Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. 257–267.

[22] D. R. Kuhn, R. N. Kacker, and Y. Lei. 2010. *Practical Combinatorial Testing.* Technical Report SP 800-142. National Institute of Standards & Technology, Gaithersburg, MD, United States.

[23] I. Machado, J. McGregor, Y. Cavalcanti, and E. Almeida. 2014. On strategies for testing software product lines: A systematic literature review. *Information and Software Technology (IST)* (2014), 1183 – 1199.

[24] I. C. Machado, J. D. Mcgregor, Y. C. Cavalcanti, and E. S. de Almeida. 2014. On Strategies for Testing Software Product Lines: A Systematic Literature Review. *Information and Software Technology (IST)* 56, 10 (2014), 1183–1199.

[25] J. Martinez, W. KG Assunção, and T. Ziadi. 2017. ESPLA: A catalog of Extractive SPL Adoption case studies. In *In 21st International Systems and Software Product Line Conference (SPLC)*. 38–41.

[26] F. Medeiros, C. Kästner, M. Ribeiro, R. Gheyi, and S. Apel. 2016. A Comparison of 10 Sampling Algorithms for Configurable Systems. In *Proceedings of the International Conference on Software Engineering*. ACM, 643–654.

[27] J. Meinicke, C. Wong, C. Kästner, T. Thüm, and Gunter S. 2016. On essential configuration complexity: measuring interactions in highly-configurable systems. In *31st Proceedings of the International Conference on Automated Software Engineering (ASE)*. 483–494.

[28] Metrics. [n.d.]. Metrics. http://metrics.sourceforge.net/, Accessed 16-nov-2019.

[29] Mockito. [n.d.]. Mockito. https://site.mockito.org/, Accessed 16-nov-2019.

[30] H. Nguyen, Christian K., and N. Nguyen. 2014. Exploring Variability-aware Execution for Testing Plugin-based Web Applications. In *Proceedings of the International Conference on Software Engineering (ICSE)*. ACM, 907–918.

[31] C. Nie and H. Leung. 2011. A survey of combinatorial testing. *ACM Computing Surveys* 43, 2 (2011), 11:1–11:29.

[32] K. Pohl and A. Metzger. 2006. Software Product Line Testing. *Information and Software Technology (IST)* (2006), 78–81.

[33] A. Santos, P. Alves, E. Figueiredo, and F. Ferrari. 2016. Avoiding code pitfalls in aspect-oriented programming. *Science of Computer Programming (SCP)* 119 (2016), 31–50.

[34] S. Souto, M. d'Amorim, and R. Gheyi. 2017. Balancing soundness and efficiency for practical testing of configurable systems. In *In 39th Proceedings of the International Conference on Software Engineering (ICSE)*. 632–642.

[35] spl2go. [n.d.]. SPL2go. http://spl2go.cs.ovgu.de/projects/, Accessed 10-nov-2019.

[36] M. Svahnberg, J. Van Gurp, and J. Bosch. 2005. A taxonomy of variability realization techniques. *Software: Practice and experience (SPE)* (2005), 705–754.

[37] T. Thüm, C. Kästner, F. Benduhn, J. Meinicke, G. Saake, and T. Leich. 2014. FeatureIDE: An extensible framework for feature-oriented software development. *Science of Computer Programming (SCP)* (2014), 70–85.

[38] G. Vale, D. Albuquerque, E. Figueiredo, and A. Garcia. 2015. Defining metric thresholds for software product lines: a comparative study. In *Proceedings of the 19th International Conference on Software Product Line (SPLC)*. 176–185.

[39] C. Wong, J. Meinicke, L. Lazarek, and C. Kästner. 2018. Faster variational execution with transparent bytecode transformation. *Proceedings of the ACM on Programming Languages (OOPSLA)* (2018).