Automating Test Reuse for Highly Configurable Software

An Experiment

Stefan Fischer
Institute for Software Systems Engineering
Johannes Kepler University
Linz, Austria
stefan.fischer@jku.at

Rudolf Ramler
Software Competence Center Hagenberg GmbH
Hagenberg, Austria
rudolf.ramler@scch.at

Lukas Linsbauer
Institute for Software Systems Engineering
Johannes Kepler University
Linz, Austria
lukas.linsbauer@jku.at

Alexander Egyed
Institute for Software Systems Engineering
Johannes Kepler University
Linz, Austria
alexander.egyed@jku.at

ABSTRACT

Dealing with highly configurable systems is generally very complex. Hundreds of different analysis techniques have been conceived to deal with different aspects of configurable systems. One large focal point is the testing of configurable software. This is challenging due to the large number of possible configurations and because tests themselves are rarely configurable and instead built for specific configurations. Existing tests can usually not be reused on other configurations. Therefore, tests need to be adapted for the specific configuration they are supposed to test. In this paper we report on an experiment about reusing tests in a configurable system. We used manually developed tests for specific configurations of Bugzilla and investigated which of them could be reused for other configurations. Moreover, we automatically generated new test variants (by automatically reusing from existing ones) for combinations of previous configurations. Our results showed that we can directly reuse some tests for configurations which they were not intended for. Nonetheless, our automatically generated test variants generally yielded better results. When applying original tests to new configurations we found an average success rate for the tests of 81,84%. In contrast, our generated test variants achieved an average success rate of 98,72%. This is an increase of 16,88%.

CCS CONCEPTS

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

KEYWORDS

variability, configurable software, clone-and-own, reuse, testing

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SPLC '19, September 9–13, 2019, Paris, France
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-7138-4/19/09…$15.00
https://doi.org/10.1145/3336294.3336305

ACM Reference Format:
Stefan Fischer, Rudolf Ramler, Lukas Linsbauer, and Alexander Egyed. 2019. Automating Test Reuse for Highly Configurable Software: An Experiment. In *23rd International Systems and Software Product Line Conference - Volume A (SPLC '19), September 9–13, 2019, Paris, France.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3336294.3336305

1 INTRODUCTION

Companies develop configurable software systems to deal with the growing demand for custom tailored software products. A range of techniques have been devised for the development and maintenance of configurable software. Many large scale configurable systems, with thousands of configuration options, have been engineered. For instance, the Linux kernel has several thousands of configuration options, like supporting a wide range of different hardware from hand held devices (e.g. Android phones) to large supercomputer clusters [2].

A large number of configuration options means that there are often myriads of configurations that can be derived from the system. This variability is challenging for many tasks when working with configurable software. Not only do all the configuration options have to be considered in the development process, but also potential interactions between them. Broadly speaking, an interaction occurs when one configuration option changes the behavior associated with other options. When testing a configurable system, combinations of configuration options are of particular interest, as they may reveal undesired interactions. However, not all combinations can be tested, because the number of possible combinations usually increases exponentially with every configuration option. Krueger et al. discussed that already a system with more than 216 Boolean, non-constrained configuration options has a number of possible configurations comparable to the number of estimated atoms in the universe [11]. To handle this combinatorial explosion, commonly only subsets of possible configurations are selected for testing. For instance, Combinatorial Interaction Testing (CIT) selects configurations that cover combinations of n configuration options (therefore, often referred to as n-wise testing). A problem that still exists is that the tests for the system are themselves often not configurable [14]. In odrder to test different configurations, the existing tests must be adapted for each specific configuration.

The goal of this work is to investigate the possibility of reusing existing tests related to one configuration for another configuration in the context of a highly configurable software system. Our experiments are based on the widely used bug tracking system Bugzilla, which provides a large number of configuration options and can be adjusted to user needs in various ways. We implemented test variants for several different Bugzilla configurations by copying and adapting the tests from previous configurations. Such a *clone-and-own* approach is often used in practice for developing and extending related software systems [7]. Moreover, we used an automated reuse approach to generate new test variants for additional configurations that combine previously tested configuration options.

Automating the reuse of tests for configurable software can substantially reduce the effort for testing and it supports a more rigorous testing process. Krüger et al. discussed the need for automated test refactoring for the adoption of more systematic reuse approaches [12]. Thus, we applied *ECCO (Extraction and Composition for Clone-and-Own)* for automatically generating new tests from existing ones written for other configurations. *ECCO* is an approach for enhancing clone-and-own to systematically develop and maintain software variants [9]. We were able to show that the reuse support of *ECCO* is effective for automatically generating new test variants for the pairwise combination of configuration options. The approach produced 3565 executable test cases, which yielded better results than simply reusing the tests of the combined configurations in 66.34% of the new pairwise configurations and it was equally successful in 24.75%.

The remainder of this paper is structured as follows. Section 2 introduces the relevant background and Section 3 discusses the problems we aim to address and motivates our experiments. Section 4 presents the system of our study and the experiments performed with it, as well as the metrics recorded during the experiments. Sections 5 and 6 summarize the results of our experiments and discuss their implications on our research questions. Finally, Section 7 describes related work to our study and Section 8 summarizes the conclusions of our study and sketches our future work.

2 BACKGROUND

In this section, we discuss some of the necessary background for our work. We describe highly configurable systems, an automatic approach for reuse, and existing approaches for testing configurable software systems.

2.1 Highly Configurable Systems

Systems frequently offer configuration options to allow users to tailor them to their needs and preferences. These configuration options (a.k.a. features [1]) have different types of how they are expressed (e.g. Boolean options, Integers, ...) and can be realized in different forms in the system. For instance, using preprocessor directives (e.g. #IFDEFs), conditional execution (e.g. simple IFs), or build systems [14]. A large number of highly configurable systems are being maintained, ranging from just a few to thousands of configuration options (e.g. Linux kernel).

A wealth of research on highly configurable software is available in the field of Software Product Line Engineering (SPLE). Software product lines (SPLs) are families of related software systems distinguished by the set of configuration options (i.e. features) each one provides. SPLs are highly structured and they follow strict processes to deal with the contained variability. The available configuration options and dependencies between them are commonly expressed in a *variability model*.

Because SPLs typically entail a high upfront investment many practitioners use a more ad hoc approach of copying and adapting previous variants, known as *clone-and-own* [7]. However, this leads to a set of similar variants that have to be maintained separately, which becomes more difficult, the more variants being developed.

2.2 ECCO

In an effort to mitigate the problems associated with clone-and-own, we developed an approach called *ECCO (Extraction and Composition for Clone-and-Own)* [9, 10]. Its purpose is to support reuse in a clone-and-own context by analyzing commonalities and differences in existing variants and, subsequently, support the creation of new variants by automatically reusing relevant parts from existing variants. In a first step it *extracts* traceability information (i.e. mappings between configuration options and their implementation). The second step then allows to *compose* new variants by combining the relevant parts of the implementation using the extracted mapping information. In this paper we apply *ECCO* specifically for creating new variants of tests. First, we extract mappings between the different parts of the test source code and the configuration options that are tested by the analyzed tests. The test source code is analyzed at the level of the Abstract Syntax Tree (AST), which means that each individual element of a source code statement is considered. Then we compose new test variants by simply selecting a set of configuration options that shall be tested. The Abstract Syntax Tree (AST) of the new tests is automatically created as a *combination* of the relevant parts of the ASTs of the existing tests. Finally, the developer can manually adjust or extend the newly created variants if necessary, e.g., when the combination of existing source code parts is not sufficient to fully express new behavior of the system due to interactions or conflicts between configuration options. In context of testing, the newly generated test will likely fail when executed for the first time, indicating an unexpected and potentially erroneous interaction between configuration options.

Figure 1 shows the simplified *ECCO* workflow. Input is a set of existing variants. Each variant consists of its implementation and the information of which configuration options it encompasses. The *extraction* operation analyzes commonalities and differences in configuration options and the implementation of the variants, computes traceability information, and stores it in a repository. The *composition* operation then uses the information stored in the repository to compute the implementation of new variants given a set of desired configuration options.

2.3 Configurable Software Testing

There exists a substantial amount of research focusing on testing configurable software [4–6, 8]. A common thread among this research is the task to select variants for testing that are more likely to contain faulty interactions causing failures. The most prominent approaches use Combinatorial Interaction Testing (CIT) [13]. CIT

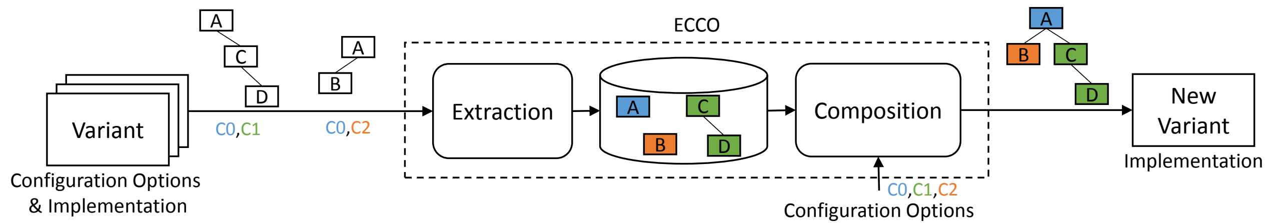

Figure 1: Simplified *ECCO* Workflow

techniques applied to configurable systems commonly use a variability model from which they calculate all valid t-wise configuration-option-combinations and, subsequently, covering arrays of a strength t. For instance, for t = 2, also known as pairwise testing, a CIT algorithm has to find a set of variants (i.e. covering array) to cover all combinations of two configuration option values that can be selected and which are allowed by the variability model.

3 PROBLEM STATEMENT

As discussed above, to test a highly configurable software system a subset of configurations has to be selected for testing. To test different configurations the tests also have to be adapted to these configurations. One solution to achieve this would be to develop the tests also as configurable software, so they can be automatically adjusted to the configurations they are supposed to test. However, in practice this is often not the case [14] as developing such tests would substantially raise development costs. Therefore, tests for a new configuration are usually created via cloning and manually adapting existing tests developed previously for a similar configuration. Following this process, practitioners end up with a set of test variants (i.e., partial clones), each to test a specific configuration.

However, covering all possible interactions that can occur in a configurable system is typically infeasible with such a manual process. Even only testing all pairwise combinations can quickly become infeasible due to combinatorics, when the tests have to be manually adapted for each configuration - not to mention the fact that these tests also have to be maintained throughout the evolution of the system [17]. In practice, testing is therefore usually focused on individual configurations (configuration options in isolation) and a few selected combinations. The limited coverage of combinations can lead to missing critical erroneous interactions between different configuration options. An approach to automatically generate tests for new configurations would help to reduce the effort of implementing and maintaining tests for a wide range of combinations and to find new interaction bugs. Given that the number of combinations is typically growing exponentially, the potential gain from using an automated approach can be huge in many projects.

Moreover, companies often use a clone-and-own process for developing new variants for their configurable system [7]. These variants are tested individually before deployment to the customers. Tests are reused from previous variants and are also adapted in a clone-and-own manner [12]. There are many benefits for migrating from clone-and-own to a more systematic approach on a reusable platform, like reduction in maintenance costs. A barrier preventing such a migration is often the fear of introducing new bugs during the migration [12]. Being able to automatically reuse tests from previous variants could help with this issue and it would allow to ensure that the system still behaves as expected after migration.

These practical problems motivated us to investigate systematic and automated reuse for software tests of configurable systems and to perform the experiments discussed in this paper. In particular we aim to answer the following research questions:

RQ1: To what degree can tests from configurations be reused directly? We analyze how many of the existing tests can be directly applied for testing other configurations and in how many cases modifications are required. Hence, this question allows us to assess the manual effort that would be required to adapt existing tests for different configurations.

RQ2: To what degree can we automatically generate test suites for new configurations from existing tests? The main goal of our experiments is to determine whether we can automatically compose test variants for new, previously untested configurations by reusing parts of the source code of existing tests. Therefore we investigate the use of the *ECCO* tool support for automatically composing such tests.

4 EXPERIMENT DESIGN

In this section, we discuss the methodology of our experiment. We start with explaining the system under test and the existing tests we have developed, followed by the setup used for our experiments, and the metrics measured during these experiments.

4.1 System Under Test

The configurable system we used in our experiments is the widely used, open-source bug tracker Bugzilla. Specifically, we used the Virtual Bugzilla Server (version 3.4) provided by ALM Works[1]. This is an virtual machine image containing a ready-to-use setup of the Bugzilla 3.4 Web application, an Apache Web server, and a MySQL database running on Debian Linux. Bugzilla is a Web-based application, so the front-end (user interface) of the Bugzilla server is accessed using a Web browser.

We initially implemented a suite of 34 automated test cases exercising the main functionality of Bugzilla via the Web front-end (e.g., submitting a bug report, searching and updating a report, changing the bug status). The tests are written in Java and use Selenium[2] to control the Chrome Web browser to interact with Bugzilla. The tests run on the default configuration of Bugzilla. Subsequently,

[1]https://almworks.com/archive/vbs
[2]https://www.seleniumhq.org/

we identified a range of different configuration options that can be used to change the default behavior of Bugzilla. We selected a diverse set of fifteen different options resulting in configuration changes that are directly observable in the Web front-end, in the navigation structure, or in the bug tracking workflow of Bugzilla. Thus, these options can be expected to impact our existing set of tests and make adaptations necessary in order to run them after a configuration change. Furthermore, some of the configuration options are expected to result in conflicts when activated in combination. We created tests for each of the additional configurations by manually performing clone-and-own starting from the test cases for the default configuration.

Config	Tests	Description
C00	34	Default configuration of Bugzilla
C01	36	Enable status white board field for optional comments
C02	34	Disable priority selection on bug report submission
C03	33	Allow using empty values in bug search form
C04	35	Add an additional product to organize bug reports
C05	36	Add an additional component to a product
C06	36	Add an additional version to a product
C07	34	Configure bug status workflow to a minimum set of states
C08	34	Configure bug status workflow to a different entry state
C09	35	Require descriptions on creating a new bug entry
C10	34	Require descriptions on all bug status changes
C11	35	Require resolution description on setting a bug to resolved
C12	35	Enforce a comment when a bug is marked as duplicate
C13	35	Enforce dependencies to be resolved before bug can be fixed
C14	34	Set the default bug status of duplicates to verified
C15	34	Set the default bug status of duplicates to closed

Table 1: Configurations and Number of Tests

We list the 16 configurations of Bugzilla used in our experiments in Table 1 along with the number of test cases developed in each of the variants and a description of the impact of changing a specific configuration option. The changes in the different configurations range from simply adding an optional comment field (C01) to completely changing the bug workflow and the states that can be assigned to a bug (C07 and C08). Some of these configuration options are related to the same functionality and we therefore expect conflicts if they are set simultaneously with one another. For instance, C07 and C08 both change the bug workflow and therefore they cannot both be configured at the same time. Configuration C12 can only be used when duplicates are allowed. A conflicting dependency also exists between configurations C14 and C15, because they both change the default status of duplicates to different states and therefore, they cannot be set simultaneously. Furthermore, we might run into another conflict if any of the two is activated together with C07, because they change a bug status that might no longer be allowed in C07.

Each of the test suite variants that were developed to test a specific configuration consists of a *(1) Test Set Up* that configures the Bugzilla server accordingly (i.e. activates the configuration and resets it to the default configuration after the tests were executed), *(2) Test Cases* that exercise the functionality of Bugzilla in various ways, and *(3) Page Objects* that use Selenium to access Bugzilla through its Web front-end. Figure 2 sketches the test execution cycle of one of the test variants targeting a specific configuration. All of the parts are implemented in Java and each test variant is an independent Maven[3] project that has been created by cloning and modifying the tests for the default configuration (C00). We use the tool Maven Invoker[4] to automatically execute all test variants. As depicted in Figure 2, we first call the *Test Set Up* to set Bugzilla to the desired configuration. Next, we use the JUnit test runner to execute the *Test Cases*, which call methods provided by the *Page Objects*. These are Java objects that use Selenium to interact with the Web pages realizing the Bugzilla front-end and to verify the expected outcome. Finally, we use the *Test Set Up* to reset the configuration back to its default in order to provide a clean basis for running other test variants using the same process.

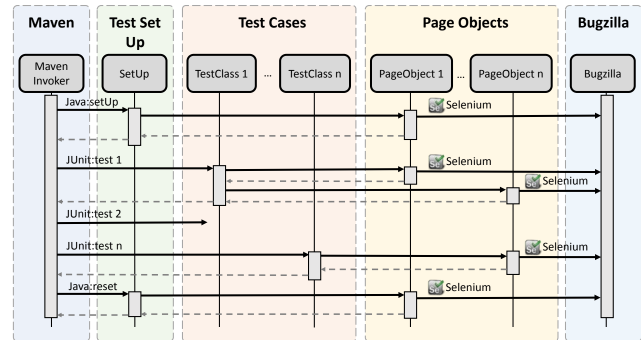

Figure 2: Sequence of Executing a Test Variant

4.2 Composing New Test Variants

We applied *ECCO* to create new test variants from existing tests. We generated the tests for new configurations that are combinations of configuration options covered individually by the existing tests. We did not adjust *ECCO* for this experiment. It was used as described in Section 2.2 on the existing test variants along with the covered configuration options as input.

Figure 3 shows code snippets of three different manually developed test variants, testing the configurations C00, C01, and C06 respectively. The configuration C00 is the variant testing the default configuration of Bugzilla. Configuration C01 adds an optional text field for commenting on the status of a bug, that is accessed in Line 24 in test variant T01. Although test T00 was written for the default configuration C00 it can still run successfully on C01, because the text field is optional and not accessed by the test. Test T01 can not successfully run on configuration C00 and will cause an error in Line 24, since the text field does not exist in this configuration.

Configuration C06 adds another product version to the Bugzilla default configuration, which then overrides the default selection of the version labeled *unspecified*. All test variants assert that the version of the created bug entry is *unspecified* (Lines 9, 22, and 38 respectively). However, if we execute the tests T00 or T01 on configuration C06 we would get a failure from this assertion, because the default version has been overridden and the new version is selected instead. The test variant T06 was therefore adapted by adding Line 33 that explicitly sets the version to *unspecified*.

[3]https://maven.apache.org/

[4]https://maven.apache.org/shared/maven-invoker/

Variant **T00** (BASE):

```
1  class CreateNewBugTest {
2    public void testCreateBugDefaultValues() {
3      ...
4      createBug.setSummary(summary);
5      BugCreatedPage created=createBug.commitBug();
6      ...
7      EditBugPage editBug=created.gotoCreatedBugPage();
8      assertEquals(summary, editBug.getSummary());
9      assertEquals("unspecified",editBug.getVersion());
10     ...
11   }
12 }
```

Variant **T01** (BASE + USESTATUSWHITEBOARD):

```
14 class CreateNewBugTest {
15   public void testCreateBugDefaultValues() {
16     ...
17     createBug.setSummary(summary);
18     BugCreatedPage created=createBug.commitBug();
19     ...
20     EditBugPage editBug=created.gotoCreatedBugPage();
21     assertEquals(summary, editBug.getSummary());
22     assertEquals("unspecified",editBug.getVersion());
23     ...
24     assertEquals("",editBug.getStatusWhiteboard());
25     ...
26   }
27 }
```

Variant **T06** (BASE + ADDVERSION):

```
29 class CreateNewBugTest {
30   public void testCreateBugDefaultValues() {
31     ...
32     createBug.setSummary(summary);
33     createBug.setVersion("unspecified");
34     BugCreatedPage created=createBug.commitBug();
35     ...
36     EditBugPage editBug=created.gotoCreatedBugPage();
37     assertEquals(summary, editBug.getSummary());
38     assertEquals("unspecified",editBug.getVersion());
39     ...
40   }
41 }
```

Figure 3: Source Code Snippets of Bugzilla Tests

We used the test variant for the default configuration C00 and the different variants created for the additional fifteen configurations C01-C15 as input for *ECCO*. Furthermore, we also provided the configuration option tested by each of the variants to allow *ECCO* to establish links between configuration options and the related source code parts of the tests. Based on the extracted knowledge, *ECCO* generates the source code of the tests for a new configuration, which is specified in terms of the activated configuration options.

Figure 4 shows code snippets of a test generated with *ECCO* for the combination of the configurations C01 and C06. This new test variant contains the code for the optional status text field in Line 53 and for setting the added bug version in Line 46. Therefore, this new test can be executed on the combined configuration with USESTATUSWHITEBOARD and ADDVERSION activated. In contrast, the tests T00 and T01 would fail on this combination due to the change caused by the added version. Test T06 would still pass, but it does not assert that the optional status text field has been activated. The test variant generated with *ECCO* also executes the assertion for the optional text field and therefore achieves higher code coverage.

Variant **T01-06** (BASE + USESTATUSWHITEBOARD + ADDVERSION):

```
42 class CreateNewBugTest {
43   public void testCreateBugDefaultValues() {
44     ...
45     createBug.setSummary(summary);
46     createBug.setVersion("unspecified");
47     BugCreatedPage created=createBug.commitBug();
48     ...
49     EditBugPage editBug=created.gotoCreatedBugPage();
50     assertEquals(summary, editBug.getSummary());
51     assertEquals("unspecified", editBug.getVersion());
52     ...
53     assertEquals("",editBug.getStatusWhiteboard());
54     ...
55   }
56 }
```

Figure 4: Source Code Snippets of Composed Test

4.3 Experiment Execution

We performed several different experiments to answer our research questions stated above.

Direct reuse of existing tests on other configurations: To answer *RQ1*, we investigated how many of the existing tests of the original variants could be direly reused for testing other configurations. We first executed all tests on the individual configuration they were created for to ensure they work correctly. Then we executed each of the tests also on all other configurations to find out how much the individual configurations influence the test runs. From the results we analyzed to what degree the tests are influenced by the different configurations.

Direct reuse of existing tests on new pairwise configurations: We created new configurations by building pairwise combinations of existing configurations, i.e., by activating the Bugzill options related to two individual configurations simultaneously. Then, we executed the tests for each of the two configurations to evaluate the reuse of the existing tests on these new pairwise configurations. To activate the configuration options in Bugzilla, we used the existing setup code from both of the involved test variants and executed them in sequence, so both options would be set. In order to mitigate the possibility that the setup of the first configuration influences the other one (e.g., one masking the other and corrupting the outcome), we performed the experiment twice and changed the order in which the two setups were executed. This allowed us to asses to what degree the original variants can be directly reused on pairwise configurations.

Automated reuse by composing new tests: To address *RQ2*, we also executed newly composed tests on the pairwise combinations of existing configurations. We applied *ECCO* to generate new test variants for all possible combinations of any two existing configurations. The goal of this experiment was to assess the usefulness of automatically composing new tests from existing tests for new configurations. Figure 5 illustrates the experiment on pairwise configurations for the example of testing configuration C01-06. The pairwise configuration C01-06 represents the combination of the individual configurations C01 and C06. Instead of testing this new configuration with the exsiting tests T01 (developed for the configuration C01) and T06 (developed for C06), Please note that *ECCO* is

deterministic and therefore we only had to perform this experiment once.

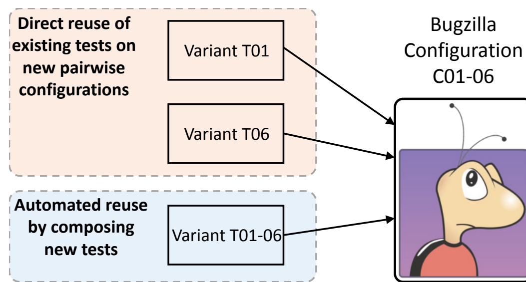

Figure 5: Example Experiments on Pairwise Configurations

Automatic reuse of setup code: We can use *ECCO* not only to generate test code for the new configurations but also for generating the setup and reset code to configure Bugzilla accordingly. In the previous experiment, we executed the setup of each individual configuration in sequence to activate all required configuration options. To investigate if the *ECCO* approach is also applicable for composing setup code, we repeated the previous experiment, using the *ECCO* generated setup and reset code instead of the original ones. We compared the results to those from the previous experiment in order to assess how well the *ECCO* setup code worked.

All test variants used in the experiments, original ones as well as those generated with *ECCO*, were realized as separate Maven projects. We used the Maven Surefire Plugin[5] to generate test reports and the JaCoCo[6] Maven Plugin to record code coverage.

4.4 Metrics

Next, we will discuss the metrics we recorded for our experiments. From the test reports we can extract the first set of metrics.

Test Result Metrics.

- **Number of Test Cases** $Tests$. The number of test cases that exists for a variant.
- **Number of Successful Tests** *Succ*. The number of test cases that were executed in a variant without any problem (i.e. no failures or errors).
- **Test Success Rate** *SuccessRate*. The rate in which test cases could be executed without problem (i.e. no failures or errors).

$$SuccessRate = Succ/Tests$$

Furthermore, as the existing tests may still pass but yield less coverage than the composed ones, we also analyze and compare the coverage achieved by the tests. However, we were not able to measure the actual coverage of the Bugzilla code. Instead, we measured the coverage of the Java code for the page objects of the executed variants. Our reasoning for doing this was that each page object represents an actual page of Bugzilla, and the code that was executed uses parts of the page. Therefore, we argue that coverage of the page object should logically be correlated with the coverage of Bugzilla itself. The coverage report from JaCoCo includes lines, methods, and classes that have been executed during testing. However, the line coverage metric is influenced by the formatting (i.e. a statement can be in one line or split into several lines). Because *ECCO* may format some statements different than they were in the original variants we computed statement coverage instead, which allows a better comparison. We did this by iterating over the Abstract Syntax Tree (AST) (generated with the Eclipse Java development tools (JDT)) and checking the JaCoCo report for each statement if the corresponding line was executed. Moreover, from this AST we computed the number of statements that exist in each variant and over all variants combined.

Coverage Metrics.

- **Number of Statements** *FullCount*. The number of unique statements that exist combined over all variants.
- **Number of executed Statements** $VarCovered$. The number of statements that have been executed when testing a variant.
- **Statement Coverage on all code** *OverallCoverage*. Coverage of statements in an individual variant in relation to all statements of all variants.

$$OverallCoverage = VarCovered/FullCount$$

Therefore, *OverallCoverage* is the proportion of Bugzilla that we can access with our page objects and which is executed during testing.

5 RESULTS

In this section, we present the results of our experiments.

5.1 Direct Reuse of Existing Tests on Other Configurations

First, we executed all test variants on all configurations. In Figure 6 we depict the *SuccessRate* at which test cases from existing test suite variants (columns) could be applied to existing configurations (rows). As is expected, the diagonal is filled with the values 1.0, meaning 100% of the test cases could be applied to the configurations they were developed for. Moreover, some variants can apply all their test cases to some other configurations, for example tests from C10 (i.e. T10) can be applied to six configurations besides C10 itself. Most of the variants can run a fairly high number of their test cases on other configurations (between 70% and 90%). The exception for this are the configurations C04 and C05, on which only the most basic tests from other configurations could be executed. Similarly, the test suite for configuration C04 (i.e. T04) could not run many of its tests on other configurations than C04 itself.

5.2 Direct Reuse of Existing Tests on New Pairwise Configurations

Next, we executed the tests on pairwise configurations. There are 105 possible pairs in total ($\binom{15}{2} = 105$). The order in which we executed the setup made no difference for the results of most of the configurations. For configuration C14-15 we found a difference in the *SuccessRate* depending on the order, due to the expected conflict between the two configurations. The *SuccessRate*s did an exact flip with the order in which the configurations was set up and we included the results, because there was effectively no difference in the numbers. Furthermore, we also discovered that the combined

[5]https://maven.apache.org/surefire/maven-surefire-plugin/
[6]https://www.eclemma.org/jacoco/

	T00	T01	T02	T03	T04	T05	T06	T07	T08	T09	T10	T11	T12	T13	T14	T15
C00	1,00	0,81	1,00	0,84	0,06	0,89	0,89	0,85	0,91	0,97	1,00	0,89	0,97	0,97	0,97	0,97
C01	1,00	1,00	1,00	0,84	0,06	0,89	0,89	0,85	0,91	0,97	1,00	0,89	0,97	0,97	0,97	0,97
C02	0,97	0,78	1,00	0,81	0,06	0,89	0,89	0,82	0,91	0,94	0,97	0,86	0,94	0,94	0,94	0,94
C03	0,94	0,75	0,94	1,00	0,06	0,83	0,83	0,79	0,85	0,91	0,94	0,83	0,91	0,91	0,91	0,91
C04	0,06	0,06	0,06	0,06	1,00	0,06	0,06	0,06	0,06	0,06	0,06	0,06	0,06	0,06	0,06	0,06
C05	0,12	0,11	0,12	0,12	0,06	1,00	0,11	0,12	0,12	0,11	0,12	0,11	0,11	0,11	0,12	0,12
C06	0,94	0,78	0,94	0,78	0,06	0,89	1,00	0,79	0,91	0,91	0,94	0,83	0,91	0,91	0,91	0,91
C07	0,74	0,67	0,74	0,69	0,06	0,67	0,67	1,00	0,71	0,71	0,74	0,71	0,71	0,74	0,71	0,71
C08	0,82	0,69	0,82	0,69	0,06	0,78	0,81	0,85	1,00	0,80	0,82	0,80	0,80	0,80	0,79	0,79
C09	0,97	0,81	0,97	0,81	0,06	0,89	0,89	0,82	0,91	1,00	1,00	0,86	0,94	0,94	0,94	0,94
C10	0,74	0,69	0,74	0,69	0,06	0,67	0,67	0,74	0,71	0,77	1,00	0,69	0,74	0,71	0,74	0,74
C11	0,94	0,75	0,94	0,78	0,06	0,83	0,83	0,85	0,85	0,91	1,00	1,00	0,91	0,94	0,91	0,91
C12	0,97	0,78	0,97	0,81	0,06	0,86	0,86	0,82	0,88	0,94	1,00	0,86	1,00	0,94	0,97	0,97
C13	1,00	0,81	1,00	0,84	0,06	0,89	0,89	0,85	0,91	0,97	1,00	0,89	0,97	1,00	0,97	0,97
C14	0,97	0,78	0,97	0,81	0,06	0,86	0,86	0,82	0,88	0,94	0,97	0,86	0,94	0,94	1,00	0,97
C15	0,97	0,78	0,97	0,81	0,06	0,86	0,86	0,82	0,88	0,94	0,97	0,86	0,94	0,94	0,97	1,00

Figure 6: *SuccessRate* for Reusing the Test Cases of each Test Variant (Columns) on all Configurations (Rows)

setup and reset did not work for four configuration pairs (C07-09, C07-10, C08-09, and C08-10) when running our experiments. We were able to run the setup for these variants in at least one order so we were able to retrieve the data for the experiment, but we had to manually reset the virtual machine image of Bugzilla to re-establish the default configuration.

Figure 7 depicts the *SuccessRate* for executing the original variants on the 105 pairwise configurations. We executed the two variants corresponding to the two combined configurations. Subsequently, we classified them in the best and the worst of the two variants and printed them sorted by the Worst*SuccessRate*. Moreover, we depict the quantiles for 25%, 50%, and 75% of the entire *SuccessRate* data.

For the majority of the 105 configurations none of the original test variants could be applied successfully (74 times). In 30 cases, we were able to reuse one test variant completely and in one case (C01-13) both of them worked.

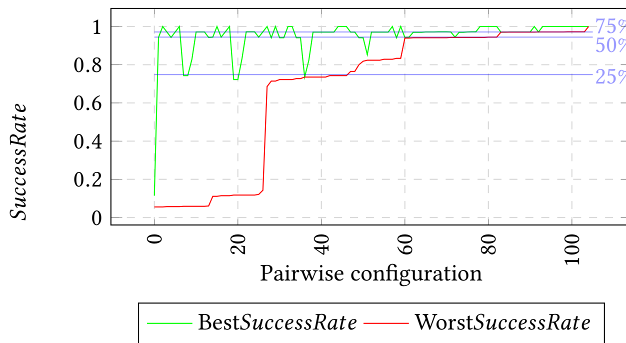

Figure 7: *SuccessRate* of Original Variants on Pairwise Combinations

5.3 Automatic Reuse by Composing New Tests

Next, we generated the pairwise test variants using *ECCO* for each of the 105 pairwise configurations. It took 33.1 seconds to extract the mapping information from the 16 initial variants and 49.3 seconds to generate all 105 new variants. We used a system with an Intel i7-3610QM CPU @2.3 GHz and 16GB of RAM. Four out of these 105 variants resulted in a compiler error and could therefore not be executed (T04-05, T04-09, T09-11, and T09-12). These errors occurred at positions where the AST merge lead to merge conflicts that *ECCO* can not automatically decide. For instance, when two different return statements appear at the end of a method or when the same variable is defined in a method twice due to the merge. We executed the tests for the remaining 101 variants and computed our metrics. The order of the setup did not affect the results of the tests generated by *ECCO*.

Figure 8 shows the number of test cases part of the 101 working *ECCO* generated variants. They range from 33 to 38 test cases, whereas the number of test cases for the original variants only ranged from 33 to 36 test cases. These results can be expected as *ECCO* merged test cases from two different variants into one variant.

Figure 8: Number of Test Cases (*Tests*) in the *ECCO* Variants

Figure 9 depicts the *SuccessRate* of the 101 working generated *ECCO* variants on the pairwise configurations. We observed a very high *SuccessRate* for the *ECCO* generated test variants. In fact, in 92 of the 101 variants all tests passed successfully (i.e. *SuccessRate* = 1.0), which is also why all the quantiles are at 1.0.

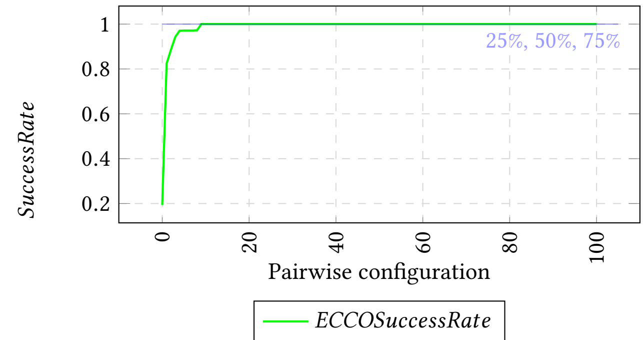

Figure 9: *SuccessRate* of *ECCO* Variants on Pairwise Combinations

We compared these results with the ones from using the original variants on the pairwise combinations. Table 2 shows the number of times that our with *ECCO* generated tests could all be executed successfully compared to when executing the original test variants on the pair-wise configurations. We can see that for the majority of the 101 configurations none of the original test variants could be applied successfully (70 times). In contrast, the *ECCO* generated test variants passed successfully for 67 of these 70 configurations. In 30 cases it was possible to reuse one test variant completely, and

		Original Variants Successful			Total
		None	One	Both	
ECCO Pair Variants	Success	67	24	1	92
	Fail	3	6	0	9
Total		70	30	1	101

Table 2: Contingency Table of Successfully Passing Tests for *ECCO* Variants vs. Original Variants

in only one configuration we were able to apply both original test variants.

For some configurations our results showed a very low *SuccessRate*, as we can see in the outliers at the start in Figure 9. Moreover, Table 2 shows that for three configurations none of the variants worked without problems. These three configurations are also the ones where our *ECCO* variants performed the worst in terms of the *SuccessRate*. The worst case in the *ECCO SuccessRate* stems from configuration C05-10 (i.e. the combination of C05 and C10), with a *SuccessRate* of only 19.4%. We found that C05 changes the process for many tested use cases of Bugzilla and, therefore, most tests failed in the pairwise combinations containing C05 and in the *direct reuse* results shown in Figure 6. Nonetheless, the *ECCO* variants for combinations with C05 worked without a problem. The combination with C10 seems to work specifically poorly, because it also requires a comment on all bug status changes that the tests from T05 do not include.

The next configuration that did not work with any variants and which resulted in the second worst *SuccessRate* for the generated *ECCO* variant was C07-08. However, we expected conflicts for the combination of these configurations, as discussed in Section 4.1.

The third and final configuration that each applied variant encountered problems on was C07-11. Test from T07 failed because C11 requires a comment on bug resolution change that is not implemented in the tests, and T11 failed because C07 restricts the Bugzilla workflow and bug status values that are possible and therefore bugs have another status then expected by the tests.

Other interesting results were for combinations that we actually expected conflicts. For instance, we found that for the combination of C09 and C10 we indeed found a conflict when applying the tests generated with *ECCO* (i.e. 1 failed test in T09). However, the tests of T10 worked on the combination without a problem, because C10 requires comments on all bug status changes including the one required by C09. Moreover, we expected conflicts for configurations C07-14 and C07-15, but found the *ECCO* tests worked without problems, because the status changes tested for duplications still worked in the minimal workflow configuration. In contrast, all original variants encountered some problems during testing, like T07 that expected a different status for bug duplicates and the rest did not work on the minimal workflow. Finally, we also expected a conflict for combining C14 and C15 and indeed found that we can not configure both of them at the same time, because they configure the same configuration option. Hence, the *ECCO* test variant did not work and which one of the original variants that worked depended on the order of configuration (i.e. for the configuration that was configured last the tests in the corresponding variant worked).

5.4 Automatic Reuse of Setup Code

Next, we performed the experiment again with the setup code generated by *ECCO*. For most configurations and test variants the results did not change compared to the previous experiment above. The four variants that could not setup/reset the configuration correctly also could not reset Bugzilla to the default configuration with the *ECCO* generated setup code. Therefore, we had to stop the virtual machine running Bugzilla and reset the image manually, before continuing the experiments in the same way we did in the previous experiment.

When further investigating the differences in our results we found two configurations that behaved slightly different with the *ECCO* setup. The most severe differences in the results occurred in variant T07-08 that already confirmed our expected conflicts in the previous experiment. However, with the *ECCO* setup the results are even worse and each variant caused 32 errors out of 34 test cases, which translates to a *SuccessRate* of 5.9%. The second configuration for which we found differences in our test results was C09-10 where variant T09-10 caused one error in the previous experiment. Surprisingly, variant T09-10 works without any problem when we used the *ECCO* setup to establish configuration C09-10. We investigated why this is the case and found that the setup code in variant T09 is a subset of the code in T10, and therefore the merged variant T09-10 had the equal setup code as T10. This is also why tests from T10 work on configuration C09. Finally, for configurations C14-15 we found in the previous experiment that the order of the setup mattered for the test outcomes, and therefore the results for this experiment matched only the results of the setup order that matched the order that *ECCO* generated.

6 DISCUSSION

In this section we discuss the implications of the results on our research questions.

RQ1: To what degree can tests from configurations be reused directly? In our first experiment we found that some test variants work without any failures or errors on other configurations (see Figure 6). This was the case because some configurations enabled a subset of configuration options on other configurations, like C09 where a comment is only required in a subset of the cases of C10 and therefore the tests of C10 (i.e. T10) also work on C09. For other configurations (i.e. C04 and C05) we found that only some of the basic tests worked and most others failed, because these configurations change many aspects of the main use cases of Bugzilla that even involve new Web pages that the other tests were not designed to interact with. However, the majority of the remaining variants could execute between 67% and 97% of their test cases on other configurations without problems.

The second experiment showed that the *direct reuse* works in 31 of the 105 configurations for at least one variant. However, this leaves the majority of test variants to be adapted in terms of fixing failing tests, which requires considerable manual effort.

RQ2: To what degree can we automatically generate test suites for new configurations from existing tests? Our results suggest that *ECCO* is useful for automatically generating new test variants. The data of our *automatic reuse* experiment showed a significantly higher *SuccessRate* for test variants generated with

ECCO compared to directly reusing existing variants on the new configurations. We measured an average *SuccessRate* of 98,72% for tests we generated with *ECCO*, compared to a *SuccessRate* of 81,84% for using the two original test variants. Even if we always select the original variants with the highest *SuccessRate* for every pairwise configuration, the average *SuccessRate* would be less (95.8%). However, the information required to make this selection for a specific configuration options is unknown before execution. If we would instead always choose the worse of the two variants, the average *SuccessRate* drops to 67.9%.

Figure 10 depicts the rates in which tests cases were successful for the generated tests (i.e. *ECCO*), and for the two variants testing the two configurations that were merged. We can see that the test variants generated with *ECCO* have a significantly higher *SuccessRate* than any of the other variants. For 31 configurations we were able to successfully execute an existing test variant of the previous configurations, which means 32 initial variants could be applied (since for one configuration we could reuse both variants fully). We confirmed the statistical significance of the results using the Wilcoxon-Rank-sum test (p-value:2.2 exp(−16)) [16]. Additionally, we computed the effect size measure $\hat{A}_{12}$: 0.886, which means our generated test variants lead to a higher *SuccessRate* than the original two variants in 88.6% of the cases.

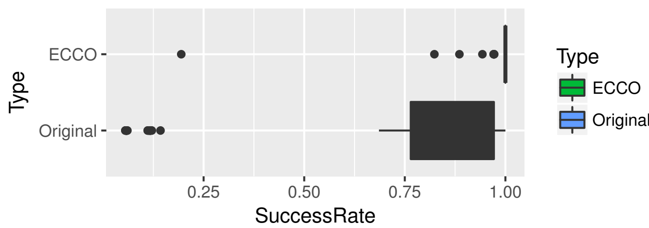

Figure 10: *SuccessRate* **of** ***ECCO*** **Tests vs. Original Tests**

Figure 11 depicts the *OverallCoverage* for the different test variants. We found that generally the *ECCO* tests have higher coverage in most cases and, therefore, they executed more of the code of our page objects. However, since *ECCO* merges code from original variants we would expect the generated variants to contain more code to execute than the original variants.

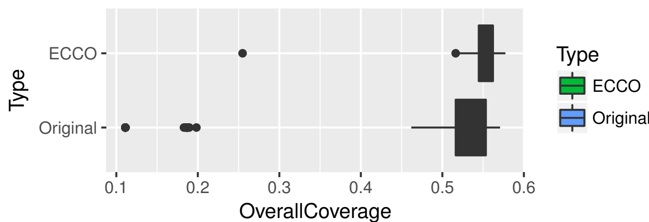

Figure 11: *OverallCoverage* **of** ***ECCO*** **Tests vs. Original Tests**

Finally, we investigated the relations between *SuccessRate* and *OverallCoverage* of our results. To do this we classified the results depending on the *ECCO* variants results compared to the results for the metrics of other variants. In Table 3 we show the contingency table of this comparison. We can see that for 34 configurations the *ECCO* variants were best in both metrics. Moreover, for the configurations in which the *ECCO* variants *SuccessRate* was equal to or worse than for other variants, the *ECCO* variant still had the highest *OverallCoverage* in the majority of cases.

		OverallCoverage					Total
		ECCO	*ECCO*+1	*ECCO*+2	*ECCO*-1	*ECCO*-2	
SuccessRate	*ECCO*	34	20	0	12	1	67
	ECCO+1	18	4	0	2	0	24
	ECCO+2	1	0	0	0	0	1
	ECCO-1	6	0	2	1	0	9
	ECCO-2	0	0	0	0	0	0
Total		59	24	2	15	1	101

ECCO: *ECCO* results were the best,
ECCO+X: *ECCO* results were equally best with 1 or 2 other variants,
ECCO-X: *ECCO* results were worse than 1 or 2 other variants

Table 3: Contingency Table *SuccessRate* **vs.** *OverallCoverage*

These results support the general usefulness of *ECCO* to automatically generate new test variants. Moreover, we found that *ECCO* can also be used to generate the test setup code of our test variants. Reducing the effort for reusing existing test code for new configurations is beneficial for testing highly configurable software. Furthermore, *ECCO* could be used for refactoring tests when moving variants that were developed using clone-and-own to a platform for systematic reuse.

6.1 Limitations

In our experiments we only measured the rate in which test cases succeeded and the code coverage. We did not have any fault data, so we were not able to investigate if the tests would actually be able to discover faults in the system. To further study the usefulness of an automated reuse approach for tests and to evaluate the generated test quality, measuring the fault detection capabilities is the next logical step. However, for now this has to remain an item on our future work agenda. Nonetheless, demonstrating that we can generate working test variants for new configurations is an important step into the direction of automated test reuse.

Another limitation of our work we want to point out is that we only generate test variants for configurations which are pairwise combinations of previously tested configurations. We do not generate test suites for entirely new configurations. With *ECCO* we can automatically reuse tests by mapping configuration options to test code that already exists, but the approach cannot be used to generate entirely new test code.

6.2 Threats to Validity

External validity: Our study only includes one configurable system. Studies on more systems are required to determine the degree to which results may be generalized. However, the system that we used is well known and should be representative for this type of configurable systems. A possible source for bias might be the configuration options we selected to use in our experiment. This choice was based on selecting arbitrary options from the Bugzilla configuration pages that have an observable impact on the user interface. Another possible source for bias might be that the tests were also created by the authors. We developed the tests for the

default configuration and then for the other 15 configurations using a clone-and-own process, all before the experiments. We did not alter the tests at all for our experiments, so they are more realistic and even led to compiler errors in four of the variants generated by *ECCO*.

Internal validity: We required several tools to perform the experiments and for data analysis. Errors in these tools might bias our results. To reduce this possibility, we validated all used tools and our code on smaller examples and subsets of the data. Another possible source for bias might come from the automatic setup of the configurations. To reduce this possibility, we randomly checked the configurations in Bugzilla and ensured it was configured as intended. Furthermore, we performed the experiments on pairwise configurations with the setup in both orders and with the setup generated by *ECCO*. To proof the applicability of *ECCO* for our test code we used it to reconstruct the original variants. Hence, we used all 16 variants as input for *ECCO* and regenerate all of them. We compared the Abstract Syntax Tree (AST) of the original variants with *ECCO*'s reconstruction of the variants and found no difference. Moreover, we executed all the tests on the reconstructed variants like in our first *direct reuse* experiment and compared the results and found no difference in the test results. Similarly, we found no difference in the coverage data we recorded. These results confirm the basic usefulness of *ECCO* for our experiment and showed that it successfully can identify parts specific to configuration options from the initial variants.

Construct validity: We measured test success and code coverage on the Java page objects. Instead of only measuring which tests run without problem it would also be interesting to test for faults in the system and compare which tests are able to detect faults in different configurations. However, we did not have any faults for our system and were also not able to perform mutation testing on the virtual machine that runs the tested system. Moreover, we measured the code coverage only on the Java page objects, because we could not measure coverage within the virtual machine. We argue that the coverage of the page objects is likely to correlate with the coverage of the corresponding Bugzilla page, but this might be a source for bias in our results.

7 RELATED WORK

Cohen et al. performed an experiment to show the effect of executing tests for different configurations of a highly configurable system [3]. They found small differences in fault detection and code coverage across configurations. This experiment is similar to our first experiment for *direct reuse*, were we also found that many test cases still worked on different configurations. However, we did not have fault data available, nor could we inject mutants like Cohen et al. did in their experiments, which is a limitation of our work. The main difference of this work and the work from Cohen et al. is that our main goal was to assess our capabilities to automatically generate new variants to test combinations of previous configurations, which was out of the scope of the experiments of Cohen et al.

Ramler et al. reported their experience for automatically reusing tests across configurations and versions to increase code coverage [15]. They were able to increase coverage by directly reusing test cases from other configurations. We found similar results in our first experiment for *direct reuse* that showed that we can reuse some test cases for one configuration on other configurations without problems. Additionally, we performed experiments for automatically generating new test variants.

As we have mentioned before, Krüger et al. discuss the need for automatic refactoring of tests to reduce barriers of moving from variants developed with a clone-and-own process to a more systematic SPL platform [12]. They discuss challenges linked to such a refactoring and outline their own ideas for such a refactoring approach. Our experiments support the usefulness of using *ECCO* for reusing tests and we argue that with *ECCO* we can address several of the challenges discussed by Krüger et al.

8 CONCLUSIONS AND FUTURE WORK

In this paper we performed experiments on the reusability across configurations of a highly configurable software system. Furthermore, we used an approach for automatic reuse to generate tests for new configurations by reusing previously developed test variants. Our experiments showed that for most configurations a large proportion of around 70% to, in some cases, 100% of tests cases could be applied to other configurations without problems. The main goal of our study was to assess the usefulness of automatically generated test variants, for which we found an average success rate of 98.7%, compared to 81.8% when directly reusing previous variants. These results suggest a considerable advantage of our approach to automatically generate tests over the direct reuse approach, which requires additional manual effort for adapting the failing tests.

However, more experiments are required to confirm these findings. In our future work, first, we plan to use also other configurable systems to replicate our results. Ideally, these systems would have fault data available or allow to use mutation testing, so we could also assess the fault detection capabilities of different test variants. Additionally, we would be interested in performing further experiments with new configurations and other variants (e.g. 3-wise combinations). Finally, we plan to investigate if we can use the results from testing different configuration combinations to infer the existence of unknown interactions among configuration options.

ACKNOWLEDGMENTS

The research reported in this paper has been supported by the Austrian Ministry for Transport, Innovation and Technology, the Federal Ministry for Digital and Economic Affairs, and the Province of Upper Austria in the frame of the COMET center SCCH, grant no. FFG-865891. Furthermore, this research was in part funded by the JKU Linz Institute of Technology (LIT) by the state of Upper Austria, grant no. LIT-2016-2-SEE-019.

REFERENCES

[1] Thorsten Berger, Daniela Lettner, Julia Rubin, Paul Grünbacher, Adeline Silva, Martin Becker, Marsha Chechik, and Krzysztof Czarnecki. 2015. What is a feature?: a qualitative study of features in industrial software product lines. In *Proceedings of the 19th International Conference on Software Product Line, SPLC 2015, Nashville, TN, USA, July 20-24, 2015*, Douglas C. Schmidt (Ed.). ACM, 16–25. https://doi.org/10.1145/2791060.2791108

[2] Thorsten Berger, Steven She, Rafael Lotufo, Andrzej Wasowski, and Krzysztof Czarnecki. 2013. A Study of Variability Models and Languages in the Systems Software Domain. *IEEE Trans. Software Eng.* 39, 12 (2013), 1611–1640. https://doi.org/10.1109/TSE.2013.34

[3] Myra B. Cohen, Joshua Snyder, and Gregg Rothermel. 2006. Testing across configurations: implications for combinatorial testing. *ACM SIGSOFT Software Engineering Notes* 31, 6 (2006), 1–9. https://doi.org/10.1145/1218776.1218785
[4] Paulo Anselmo da Mota Silveira Neto, Ivan do Carmo Machado, John D. McGregor, Eduardo Santana de Almeida, and Silvio Romero de Lemos Meira. 2011. A systematic mapping study of software product lines testing. *Information & Software Technology* 53, 5 (2011), 407–423.
[5] Ivan do Carmo Machado, John D. McGregor, Yguaratã Cerqueira Cavalcanti, and Eduardo Santana de Almeida. 2014. On strategies for testing software product lines: A systematic literature review. *Information & Software Technology* 56, 10 (2014), 1183–1199.
[6] Ivan do Carmo Machado, John D. McGregor, and Eduardo Santana de Almeida. 2012. Strategies for testing products in software product lines. *ACM SIGSOFT Software Engineering Notes* 37, 6 (2012), 1–8.
[7] Yael Dubinsky, Julia Rubin, Thorsten Berger, Slawomir Duszynski, Martin Becker, and Krzysztof Czarnecki. 2013. An Exploratory Study of Cloning in Industrial Software Product Lines. In *17th European Conference on Software Maintenance and Reengineering, CSMR 2013, Genova, Italy, March 5-8, 2013*, Anthony Cleve, Filippo Ricca, and Maura Cerioli (Eds.). IEEE Computer Society, 25–34. https://doi.org/10.1109/CSMR.2013.13
[8] Emelie Engström and Per Runeson. 2011. Software product line testing - A systematic mapping study. *Information & Software Technology* 53, 1 (2011), 2–13.
[9] Stefan Fischer, Lukas Linsbauer, Roberto Erick Lopez-Herrejon, and Alexander Egyed. 2014. Enhancing Clone-and-Own with Systematic Reuse for Developing Software Variants. In *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014*. IEEE Computer Society, 391–400. https://doi.org/10.1109/ICSME.2014.61
[10] Stefan Fischer, Lukas Linsbauer, Roberto E. Lopez-Herrejon, and Alexander Egyed. 2015. The ECCO Tool: Extraction and Composition for Clone-and-Own. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 2*, Antonia Bertolino, Gerardo Canfora, and Sebastian G. Elbaum (Eds.). IEEE Computer Society, 665–668. https://doi.org/10.1109/ICSE.2015.218
[11] Charles W. Krueger. 2006. New methods in software product line practice. *Commun. ACM* 49, 12 (2006), 37–40. https://doi.org/10.1145/1183236.1183262
[12] Jacob Krüger, Mustafa Al-Hajjaji, Sandro Schulze, Gunter Saake, and Thomas Leich. 2018. Towards automated test refactoring for software product lines. In *Proceeedings of the 22nd International Systems and Software Product Line Conference - Volume 1, SPLC 2018, Gothenburg, Sweden, September 10-14, 2018*, Thorsten Berger, Paulo Borba, Goetz Botterweck, Tomi Männistö, David Benavides, Sarah Nadi, Timo Kehrer, Rick Rabiser, Christoph Elsner, and Mukelabai Mukelabai (Eds.). ACM, 143–148. https://doi.org/10.1145/3233027.3233040
[13] Roberto Erick Lopez-Herrejon, Stefan Fischer, Rudolf Ramler, and Alexander Egyed. 2015. A first systematic mapping study on combinatorial interaction testing for software product lines. In *Eighth IEEE International Conference on Software Testing, Verification and Validation, ICST 2015 Workshops, Graz, Austria, April 13-17, 2015*. IEEE Computer Society, 1–10. https://doi.org/10.1109/ICSTW.2015.7107435
[14] Mukelabai Mukelabai, Damir Nesic, Salome Maro, Thorsten Berger, and Jan-Philipp Steghöfer. 2018. Tackling combinatorial explosion: a study of industrial needs and practices for analyzing highly configurable systems. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, Marianne Huchard, Christian Kästner, and Gordon Fraser (Eds.). ACM, 155–166. https://doi.org/10.1145/3238147.3238201
[15] Rudolf Ramler and Werner Putschögl. 2013. Reusing Automated Regression Tests for Multiple Variants of a Software Product Line. In *Sixth IEEE International Conference on Software Testing, Verification and Validation, ICST 2013 Workshops Proceedings, Luxembourg, Luxembourg, March 18-22, 2013*. IEEE Computer Society, 122–123. https://doi.org/10.1109/ICSTW.2013.21
[16] David J. Sheskin. 2007. *Handbook of Parametric and Nonparametric Statistical Procedures* (4 ed.). Chapman & Hall/CRC.
[17] Mats Skoglund and Per Runeson. 2004. A case study on regression test suite maintenance in system evolution. In *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.* IEEE, 438–442.