Empirical Software Engineering (2020) 25:5295–5332
https://doi.org/10.1007/s10664-020-09884-x

Automated test reuse for highly configurable software

Stefan Fischer[1] · Gabriela Karoline Michelon[2] · Rudolf Ramler[1] · Lukas Linsbauer[3] · Alexander Egyed[2]

Published online: 16 September 2020

Abstract
Dealing with highly configurable systems is generally very complex. Researchers and practitioners have conceived hundreds of different analysis techniques 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. Moreover, tests themselves are rarely configurable and instead built for specific configurations. However, existing tests need to be adapted to run on a different configuration. In this paper, we report on an experiment about automatically reusing existing tests in configurable systems. We used manually developed tests for specific configurations of three configurable systems and investigated how changing the configuration affects the tests. Subsequently, we employed an approach for automated reuse to generate new test variants (by reusing from existing ones) for combinations of previous configurations and compared their results to the ones from existing tests. Our results showed that we could directly reuse some tests for different configurations. Nonetheless, our automatically generated test variants generally yielded better results. Our generated tests had a higher or equal success rate to the existing tests in most cases. Even in the cases the success rate was equal, our generated tests generally had higher code coverage.

Keywords Variability · Configurable software · Clone-and-own · Reuse · Testing

1 Introduction

To keep pace with the growing demand for custom tailored software products, companies develop configurable software systems. A range of techniques has 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, supporting a wide range of

Communicated by: Laurence Duchien, Thomas Thüm and Paul Grünbacher

This article belongs to the Topical Collection: *Configurable Systems*

✉ Stefan Fischer
stefan.fischer@scch.at

Extended author information available on the last page of the article.

different hardware from hand held devices (e.g. Android phones) to large supercomputer clusters (Berger et al. 2013). The customizability of software has the advantage of more flexibility to meet specific customer requirements.

However, 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, it is infeasible to test all combinations of configuration options, because the number of possible configurations is generally too high and usually increases exponentially with the number of configuration options. 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 (Krueger 2006). 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). Research has shown that tests from different configurations can sometimes be reused, but will potentially not work correctly anymore and risk for faults to remain undetected (Cohen et al. 2006). In order to more effectively test different configurations, the existing tests must be adapted for each specific configuration. However, in practice it is still a prevalent problem that the tests themselves are often not configurable (Mukelabai et al. 2018).

The main goal of the work presented in this paper is to investigate the possibility for reusing tests for changed configurations. Moreover, we focus on automatically reusing tests from previous configurations to new configurations that combine previously tested configuration options. For our experiments, we used three different configurable systems, using different mechanisms to configure them. Two of our systems use preprocessor directives that a preprocessor uses to adjust the implementation to correspond to a specific configuration before compiling the code. These two systems contain configurable tests, which are tests derived from an annotated template, that we used to compare the quality of our automatically reused tests to. The other system we used is 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 at runtime. We implemented test variants for several different Bugzilla configurations by copying and adapting the tests from previous configurations, for two versions of Bugzilla. Such a *clone-and-own* approach is often used in practice for developing and extending related software systems (Dubinsky et al. 2013).

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 (Krüger et al. 2018). Thus, we applied an existing approach for automated reuse, *ECCO (Extraction and Composition for Clone-and-Own)* to automatically generate new tests from existing ones written for other configurations (Fischer et al. 2014). In our previous work, we already found evidence to support the usefulness of *ECCO* to automatically reuse tests for pairwise combinations of configuration options in a configurable software system (Fischer et al. 2019).

We extent our previous work in this paper by:

- Additional systems: In our previous work, we only had one configurable system available to perform our experiments on. We extend this by applying *ECCO* to more systems, to recover more evidence of the general usefulness of automated reuse for testing.
- More combinations: In our previous work, we composed pairwise combinations of configuration options, to assess how often the automated reuse can combine tests and how common it fails to, which entails additional manual effort to fix the tests. In this paper, we extend this by also composing three-wise combinations. This allows us to better assess the interaction issues that might arise when applying automated reuse, to combine more configuration options.
- Manual baseline variants: Two of the systems used in the experiments described in this paper contain configurable tests. Therefore, we could derive tests for all pairwise and three-wise configurations automatically from an annotated template and use them as a baseline to compare our variants composed with *ECCO* against. This allows us further insight in the quality of the generated tests and of the manual effort to fix them to be equal to the manually developed test variants. Additionally, we were able to apply mutation testing to one of these two systems to provide a further evaluation of the quality of the tests.

Our evaluation showed promising results that the tests generated with *ECCO* could be generally executed on the configurations they were generated for, and yielded better results than the individual test variants that existed without automated reuse. Especially, for the two systems with configurable tests, all generated tests could be executed on their intended configurations. The code coverage for the composed tests for these two systems was very close to the configurable test variants and was on average about a 5% improvement over the existing individual variants. Moreover, the variants composed with *ECCO* for these two systems are very similar to the ones derived from the annotated template, with an average similarity of 95.41% and 98.88% respectively. The mutation score, for the one system we could apply mutation testing for, improved 14.2% on average with the *ECCO* test variants compared to the existing individual variants. For the other system, we found an improvement in terms of success rate of executable tests of on average 27.8% compared to individual test variants that were available without automated reuse. Similarly, we found an average 10.1% improvement in terms of code coverage.

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. The automated reuse approach is outline in Section 4. Section 5 presents the systems of our study and the experiments performed with them, as well as the metrics recorded during the experiments. Sections 6 and 7 summarize the results of our experiments and discuss their implications on our research questions. Finally, Section 8 describes related work to our study and Section 9 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 and existing approaches for testing them.

2.1 Highly Configurable Systems

Software systems are frequently developed with configuration options, so they can be better tailored to specific customer needs. These configuration options (a.k.a. features (Berger et al. 2015)) 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 (Mukelabai et al. 2018). 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*.

In the SPLE community, there are many different opinions of what constitutes an SPL, and the line between configurable systems and SPLs can be blurry. We define SPLs as a special kind of configurable software systems, that allow to derive independent variants from them.

As an example for a configurable system, consider the Stack SPL in Fig. 1. The SPL can be configured to support different functionalities. For instance, the method `clear` in Line 17 can be added or removed depending on the configuration. The same applies for method `size` in Line 20. Moreover, method `pop` in Line 6 can be configured to behave differently when called on an empty stack. It either throws an EmptyStackException or simply returns `null`, when there are no items in the stack. The variability model for the StackSPL is depicted in Fig. 8, with all configuration options and the constraints between them.

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* (Dubinsky et al. 2013). However, this leads to a set of similar variants that have to be maintained separately, which becomes more difficult, the more variants that have been developed.

2.2 Configurable Software Testing

There exists a substantial amount of research focusing on testing configurable software (do Carmo Machado et al. 2012; do Carmo Machado et al. 2014; da Mota SilveiraNeto et al. 2011; Engström and Runeson 2011).

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) (Lopez-Herrejon et al. 2015). CIT techniques applied to configurable systems commonly use a variability model from which they compute all valid t-wise configuration-option-combinations and select configurations that cover all these combinations. 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. Halin et al. showed in their work the usefulness of CIT and other sampling approaches to find faults in a configurable system (Halin et al. 2019). In their experiment, they tested all possible configurations in a configurable system. They found that

```
public class Stack<T> {
  private T[] items;
  private int size;
  public Stack() { ... }
  public T push(T x) { ... }
  public T pop(){
    if(size <= 0) {
      #ifdef $EmptyStackException
      throw new EmptyStackException();
      #elseif $ReturnNull
      return null;
      #end
    }
    return items[--size];
  }
  #ifdef $Clear
  public void clear() { size = 0; }
  #end
  #ifdef $Size
  public void size() { return size; }
  #end
}
```

Fig. 1 Code example of preprocessor annotations in Stack

CIT approaches with t = 3 detected all the faults that could be detected over all possible configurations, by only testing a small subset of them.

2.3 Mutation Testing

Mutation testing is generally used to evaluate the adequacy of test suites to detect faults in the code. Mutation testing makes small syntactic changes to the code using mutation operators. The underlying principle of mutation testing is to introduce small changes into the program that simulate common faults, called a mutant (Moghadam and Babamir 2014). The tests are evaluated on their capability of detecting these mutants in the code. Meaning, if a test that previously worked fails when running on the mutated code, the mutant was detected (i.e. killed). The result of mutation testing is generally a mutation score, which is the ratio of the number of mutants killed over the total number of mutants introduced. The mutated code has to lead to different behavior of the code, in order to be detectable during testing, otherwise it will decrease the mutation score (Holling et al. 2016). In summary, a higher mutation score signifies a more robust test suite, which is able to detect more of the introduced mutants.

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 conform 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 (Mukelabai et al.

```java
@Test(expected = EmptyStackException.class)
public void testPopEmptyStack() {
  Stack<Integer> stack = new Stack<>();
  stack.pop();
}
```

Fig. 2 Test example empty Stack with EmptyStackException

2018) 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.

For instance, in our running example, the Stack SPL, we need to adapt certain tests depending on how the stack behaves when calling `pop` when there are no items in the stack. In case it throws an EmptyStackException (see Line 9 in Fig. 1) we use the test in Fig. 2. This test simply calls `pop` on an empty stack and expects an EmptyStackException to be thrown. In case no EmptyStackException happens, the test will report a failure. Therefore, if the Stack SPL is configured to not throw the EmptyStackException, but instead return `null` (see Line 11 in Fig. 1) we also need to adapt the test to still work.

The adapted test is depicted in Fig. 3. It differs from the test before in that it no longer expects an EmptyStackException to be thrown, but instead expects the return value of the `pop` method call to be `null`.

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 (Skoglund and Runeson 2004). 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 (Dubinsky et al. 2013). 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 (Krüger et al. 2018). There are many benefits for migrating from clone-and-own to a more systematic approach on a reusable platform, like

```java
@Test
public void testPopEmptyStack() {
  Stack<Integer> stack = new Stack<>();
  assertNull(stack.pop());
}
```

Fig. 3 Test example empty Stack with ReturnNull

reduction in maintenance costs. A barrier preventing such a migration is often the fear of introducing new bugs during the migration (Krüger et al. 2018). 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 specific 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 after changing the configuration.

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 combinations of configuration options by reusing parts of the source code of existing configuration options' tests but untested together previously. Since the goal is automated reuse, and not to generate completely new tests, we require existing tests for every configuration option. Therefore, we investigate the use of the *ECCO* tool support for automatically composing such tests.

4 Automated Test Reuse with 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)* (Fischer et al. 2014, 2015). 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. 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 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 the 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 4 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. Figure 5 shows code snippets of two different test variants of our running example, testing the configurations with configuration option `isEmpty` and `size` activated, respectively. The configuration option `Base` is added to each variant, because *ECCO*

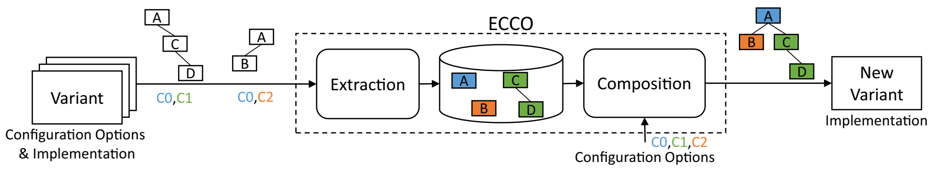

Fig. 4 Simplified *ECCO* Workflow

uses it to map the code parts to it that all variants have in common. We highlighted the lines that are different between these two variants in gray. The variant with configuration option `isEmpty` checks if the stack is empty in different parts of the test. Similarly, the variant with configuration option `size` checks the size the stack. Each of these test variants can not be executed on the source code of the others configuration, since they do not support the functionality of `isEmpty` or `size`, respectively. Trying to apply the tests to the wrong configuration would therefore lead to a compilation error.

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. Figure 6 shows snippets of the mapping that the *extraction* creates. Each of the code snippets is mapped to a presence condition, which is used to compute for which configurations the code should be included. Moreover, the *extraction* stores the order in which statements have appeared before (as hinted at in the line numbering in Fig. 6), so that they can be later sorted in that order again when composing a variant.

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. Figure 7 shows code snippets of a test generated with *ECCO* for the combination of the configuration options `isEmpty` and `size`. This new test variant contains the code to check both the

Variant (`BASE` + `ISEMPTY`):

```
@Test
public void testSingleItemStack() {
  Stack<Integer> stack = new Stack<>();
  assertTrue(stack.isEmpty());
  stack.push(0);
  assertFalse(stack.isEmpty());
  assertEquals((int) stack.pop(), 0);
  assertTrue(stack.isEmpty());
}
```

Variant (`BASE` + `SIZE`):

```
@Test
public void testSingleItemStack() {
  Stack<Integer> stack = new Stack<>();
  assertEquals(stack.size(), 0);
  stack.push(0);
  assertEquals(stack.size(), 1);
  assertEquals((int) stack.pop(), 0);
  assertEquals(stack.size(), 0);
}
```

Fig. 5 Source Code Snippets of Stack SPL Tests

(BASE)

```
1 @Test
2 public void testSingleItemStack() {
3   Stack<Integer> stack = new Stack<>();
5   stack.push(0);
7   assertEquals((int) stack.pop(), 0);
9 }
```

(ISEMPTY)

```
4   assertTrue(stack.isEmpty());
6   assertFalse(stack.isEmpty());
8   assertTrue(stack.isEmpty());
```

(SIZE)

```
4   assertEquals(stack.size(), 0);
6   assertEquals(stack.size(), 1);
8   assertEquals(stack.size(), 0);
```

Fig. 6 Example of extracted mapping for a model transition

`isEmpty` method and the `size`. The order of the calls to the `isEmpty` and `size` methods is arbitrary, because the corresponding statements have not appeared in a variant in the input. Therefore, these statements' order are only sorted in respect to the statements that have appeared together within a variant in the input. In this case, the ordering of these statements in respect with one another does not make a difference, because they only check if the stack has the expected status. *ECCO* creates hints to the positions that the ordering could not be determined, which we did not use in this work, because we wanted to generate new test variants fully automated.

The newly composed test can be executed on the combined configuration with `isEmpty` and `size` activated. In this case, we would be able to also run the original tests in Fig. 5. However, they only test the two configuration options in isolation from one another, while the *ECCO* generated test executed both in the same test. Therefore, the test variant generated with *ECCO* also achieves higher code coverage.

Variant (BASE + ISEMPTY + SIZE):

```
 1 @Test
 2 public void testSingleItemStack() {
 3   Stack<Integer> stack = new Stack<>();
 4   assertEquals(stack.size(), 0);
 5   assertTrue(stack.isEmpty());
 6   stack.push(0);
 7   assertEquals(stack.size(), 1);
 8   assertFalse(stack.isEmpty());
 9   assertEquals((int) stack.pop(), 0);
10   assertEquals(stack.size(), 0);
11   assertTrue(stack.isEmpty());
12 }
```

Fig. 7 Source Code Snippets of Composed Stack SPL Test

5 Experiment Design

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

5.1 Systems Under Test

We used three different configurable software systems in our experiments. In this section, we provide a short description for all of them.

5.1.1 Stack

The first system we used in our evaluation is our running example, the Stack SPL that we developed for illustration purposes. We showed the basic implementation in Section 2.1. The Stack SPL implements some of the most common functionalities of the stack configurable. Only the functionalities push and pop are not configurable, but exist in every configuration. Moreover, the behavior when using pop or peek on an empty stack is configurable. All of the configuration options are depicted in the variability model for the Stack SPL in Fig. 8.

The most common form of a variability model is the *feature model*, which is a tree-like structure where the nodes describe the configuration options (i.e. features) and the edges denote the different forms of relations among configuration options (Benavides et al. 2010). In our running example, for the sake of simplicity, we use three types of feature relations. The first type is *optional configuration options*, which may or may not be selected in a variant when their parent feature is selected. Examples are configuration options `Peek`, `IsEmpty`, `Size` and `Clear`. A second type is *alternatives*, where exactly one member of a group of configuration options must be selected when their parent is selected. Examples for this are configuration options `EmptyStackException` and `ReturnNull`. Finally, there are also two examples for *mandatory configuration options*, features `Operations` and `EmptyStackPop`, which are always selected if their parent is selected. In our case, these two configuration options exist for every configuration, because they are direct children of the root, which is always selected in a feature model. In total, the feature model allows for 32 different Stack configurations.

As mentioned above, we developed the Stack SPL ourselves. Moreover, we also developed tests for the Stack, as configurable JUnit tests, which can be configured according to the configuration they should test by preprocessor directives. In sum, we developed six tests that execute some standard use cases for a stack and perform different checks depending on

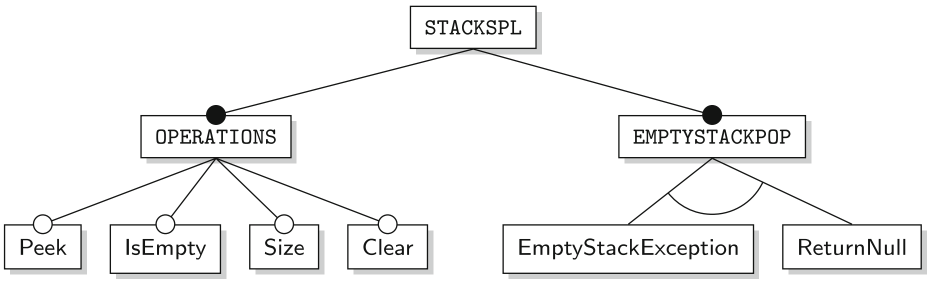

Fig. 8 Feature Model for Stack

the configuration. The test suite is configured by a preprocessor and can be used to test all 32 possible configurations.

5.1.2 ArgoUML

ArgoUML is an open source UML modeling tool that was refactored into a SPL (Couto et al. 2011). It has been widely used in the SPLE-community for different purposes, like evaluating feature location techniques (Martinez et al. 2018; Martinez et al. 2017). In the variability model (Fig. 9) we can see that it contains eight optional configuration options. Six of these configuration options add different UML diagram types, one adds logging functionality, and the configuration option `Cognitive` enables to turn on suggestions to improve the UML diagram during design. ArgoUML can be configured to 256 different configurations.

For ArgoUML, a test suite with 1198 unit tests exists. However, the tests for ArgoUML were not configurable from the start and only worked on a configuration with all configuration options turned on (i.e. all features selected). In order to be able to use the tests also on other configurations, we adapted them by annotating tests with preprocessor directives and created specification files with presence conditions to include or exclude entire classes. Therefore, we can use a preprocessor to generate a test suite for each of the 256 configurations. We did so following a process similar to the one performed in a previous paper about test case generation for SPLs (Fischer et al. 2018). In this process, we analyzed the source code and test code of ArgoUML and checked for dependencies from the test code to the source code (i.e. source code parts called by the tests). These dependencies point to code that would break, if we used specific configurations that did not contain the source code parts used by the test. If a piece of source code that a test depended on belonged to a specific configuration option (i.e. was annotated with the feature), then we annotated the test to also belong to this congratulation option. If a test included dependencies to multiple configurations options, the test was annotated with all of them in conjunction (i.e. logical AND relation). We did this until we did not find any more dependencies that would break the code in specific configurations. To confirm these annotations, we tested all 256 configurations with the corresponding variants derived from the annotated source code and found that all of them worked.

5.1.3 Bugzilla

The final configurable system we used in our experiments is the widely used, open-source bug tracker Bugzilla. Bugzilla is a Web-based application, so the front-end (user interface)

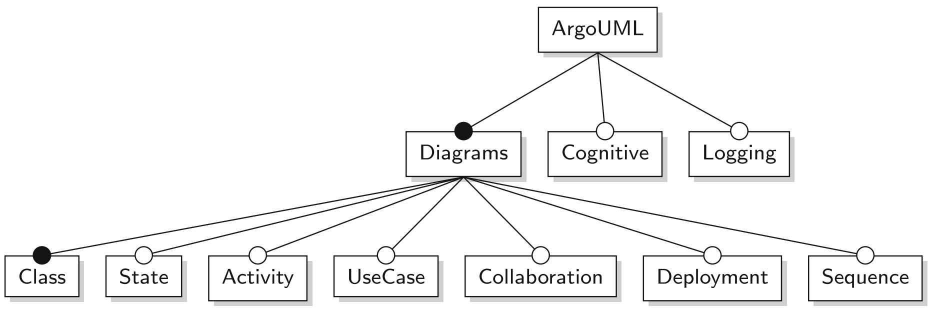

Fig. 9 Feature-Model for ArgoUML

of the Bugzilla server is accessed using a Web browser. For Bugzilla, we used two different versions.

The first version we used was the Virtual Bugzilla Server (version 3.4) provided by ALM Works[1] that we also used in our previous work (Fischer et al. 2019). The second version we used was the Bugzilla (version 5.1.1) provided by Turnkey[2]. Both versions can be downloaded as ready to use virtual machine image containing the Bugzilla Web application (in the respective version), an Apache Web server, and a MySQL database running on Debian Linux.

We started off with Bugzilla 3, and 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[3] to control the Chrome Web browser to interact with Bugzilla. These tests run on the default configuration of Bugzilla. Subsequently, we identified a range of different configuration options that can be used to change the default behavior of the use cases tested for Bugzilla. We arbitrarily selected a diverse set of 15 different options (listed in Table 1), which resulted in configuration changes that are directly observable in the Web front-end, in the navigation structure, or in the bug tracking workflow of Bugzilla. These configuration options where chosen because they change the behavior of one of the tested use cases and we therefore expected them to have an impact on our tests when turned on. Thus, these options will make adaptations to the existing tests necessary in order to run them after a different configuration of the selected configuration options. 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.

For Bugzilla 5, we followed a similar process. We also started by developing tests for the default configuration, however we reused the tests from version 3.4 and adapted them to work on Bugzilla 5. Subsequently, we checked the previously identified configuration option in the version 5.1.1 and again used a clone-and-own process to adapt the tests from the tests for the default configuration and checking the adaptations made for the different configurations in version 3.4. However, during this process we discovered that two configuration options used in version 3.4 no longer exist in version 5.1.1. Therefore, we only used the remaining 13 options in the newer version.

Since we do not have a variability model for Bugzilla, like for the other two systems, we list the 15 configuration options we used in our experiments in Table 1 along with the number of test cases developed in each of the variants, which only have this one configuration option changed from the default configuration. For comparison, we also list the number of tests cases for the default configuration in Table 1, which has all of the listed configuration options disabled. We report the numbers of tests per Bugzilla version and provide a description of the impact of changing a specific configuration option. The two configuration options (CO_{03} and CO_{15}) that only exist for version 3.4 are highlighted in gray. Note the difference in the number of tests between the two versions is due to the configuration option CO_{03} no longer existing in Bugzilla 5. In fact, Bugzilla 5 always allows the use of empty search values, and therefore the search always behaves like in Bugzilla 3 with configuration option CO_{03} turned on. This made two tests, which verified the behavior of the

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

Table 1 Configurations and Number of Tests

Option	Tests v3	Tests v5	Description
-	34	33	Default configuration of Bugzilla
CO_{01}	36	35	Enable status white board field for optional comments
CO_{02}	34	33	Disable priority selection on bug report submission
CO_{03}	33	-	Allow using empty values in bug search form
CO_{04}	35	34	Add an additional product to organize bug reports
CO_{05}	36	35	Add an additional component to a product
CO_{06}	36	35	Add an additional version to a product
CO_{07}	34	33	Configure bug status workflow to a minimum set of states
CO_{08}	34	34	Configure bug status workflow to a different entry state
CO_{09}	35	34	Require descriptions on creating a new bug entry
CO_{10}	34	33	Require descriptions on all bug status changes
CO_{11}	35	34	Require resolution description on setting a bug to resolved
CO_{12}	35	34	Enforce a comment when a bug is marked as duplicate
CO_{13}	35	34	Enforce dependencies to be resolved before bug can be fixed
CO_{14}	34	33	Set the default bug status of duplicates to verified
CO_{15}	34	-	Set the default bug status of duplicates to closed

search function in Bugzilla 3 without configuration option CO_{03} obsolete for Bugzilla 5. For the same reason, the variant with CO_{03} turned on, also has one less test in Bugzilla 3, because it removed the two obsolete tests and added one that tests the behavior with CO_{03} activated. We adapted this additional test, for the behavior with CO_{03}, for Bugzilla 5 and added it to all manually developed variants.

The changes in the different configurations range from simply adding an optional comment field (CO_{01}) to completely changing the bug workflow and the states that can be assigned to a bug (CO_{07} and CO_{08}).

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, CO_{07} and CO_{08} both change the bug workflow and they can not both be configured at the same time. Configuration option CO_{12} can only be used when duplicates are allowed. A conflict also exists between configuration options CO_{14} and CO_{15}, 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 CO_{07}, because they change a bug status that might no longer be allowed in CO_{07}.

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 option 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 10 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[4] project that has been

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

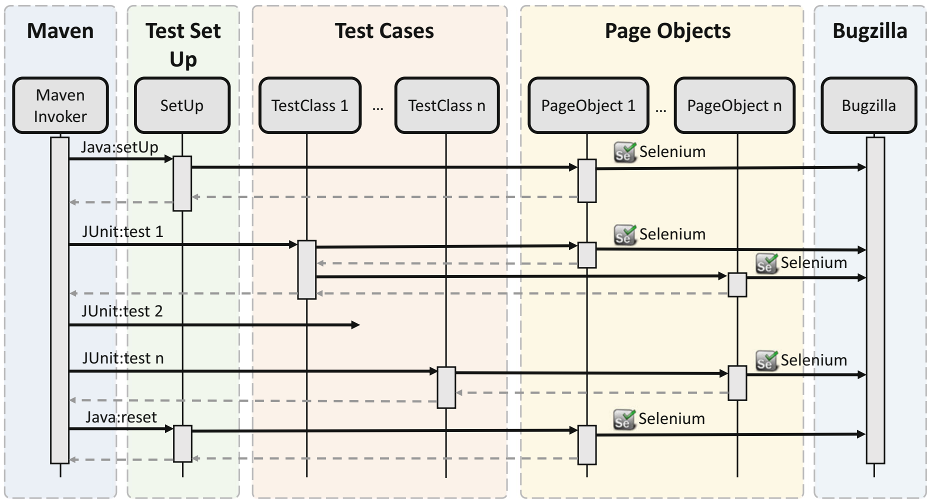

Fig. 10 Sequence of Executing a Bugzilla Test Variant

created by cloning and modifying the tests for the default configuration. We use the tool Maven Invoker[5] to automatically execute all test variants. As depicted in Fig. 10, 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.

5.2 Composing New Test Variants

We applied *ECCO* to create new test variants from existing tests. As existing tests, we used the Bugzilla tests that we developed manually and for the SPLs we generated a test variant for each optional configuration option enabled individually. This leaves us with 6 test variants for the Stack SPL, and 8 test variants for ArgoUML, each with one of the configuration options enabled. For Bugzilla, we used 16, respectively, 14 test variants as input to *ECCO*, each testing a configuration with one of the configuration options in Table 1 active. In practice, we have often observed that configuration options are tested in isolation like this, and combinations are often limited to only some selected configurations.

We used *ECCO* to generate tests for new configurations that are combinations of configuration options covered individually by their existing tests. It was used as described in Section 4 on the existing test variants along with the tested configuration options as input. Because the existing variants that we use as input to *ECCO* only contain configuration options in isolation, we are missing any information of code that might need to be added or removed to deal with combinations of configuration options. Such *interaction code* would

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

only exist if we included variants that contain combinations of configuration options already to the input to *ECCO*. Therefore, we expect this to be even more difficult for *ECCO* to combine the tests for different configuration options. *ECCO* would potentially perform even better if more such variants would be included in the input, but we apply it in the worst case we have observed in practice.

5.3 Experiment Execution

In this section, we describe the experiments that we performed to answer our research questions. We investigated how well tests for a specific configuration will still work, if we changed the configuration by changing one or two configuration options. Subsequently, we used *ECCO* to generate tests for the changed configuration and compared the test results with the ones from the manually developed tests.

Direct reuse of existing tests on changed configurations For our first research question *RQ1*, we want to investigate how well tests can be reused directly. Therefore, we used test cases that have been developed to test configurations with a single configuration option activated (called *individual* configurations), and executed them on configurations that combine this configuration option with one or two other options.

In particular, we executed the *individual* test variants on all pairwise and three-wise configurations that have the configuration option, the *individual* variant is supposed to test, activated. Therefore, we tested all pairwise and three-wise combinations of configuration options, and checked how well the tests developed for the *individual* configurations still worked. By analyzing how well the *individual* test variants work on the pairwise and three-wise combinations we can assess how common issues are for applying tests to a slightly different configuration and how much more severe these issues might become when more configuration options are combined.

We even combined configuration options for which we expected conflicts, like `EmptyStackException` and `ReturnNull` in the Stack SPL, to see how the system would behave in these cases. Note, that this combination would be excluded by the variability model for the Stack SPL in Fig. 8. Nonetheless, we included combinations like this, since in our experience in practice there often are no variability models employed and dependencies between configuration options are commonly only implicit knowledge of some domain experts. Therefore, such combinations might occur in practice, especially in combinations that might be more obscure than the example used here. This means for each pairwise configuration we executed the two *individual* test variants that test the *individual* configurations. Similarly, for each three-wise configuration, we executed the three *individual* test variants. This allowed us to asses to what degree the *individual* variants can be directly reused on changed configurations, and how much manual effort it would potentially take to adapt the tests.

Automated reuse by composing new tests To address *RQ2*, we applied *ECCO* to generate new test variants for all of the pairwise and three-wise configurations, by automatically reusing the *individual* test variants of each configuration option. We executed these new test variants on the pairwise and three-wise configurations they were generated for, and compared the results to the ones of the previous experiment. Moreover, for the SPLs we have configurable tests. Therefore, we also automatically derived the tests for the pairwise and three-wise configurations from the annotated test code of these two systems and compared how close the tests composed with *ECCO* are to these *original* ones.

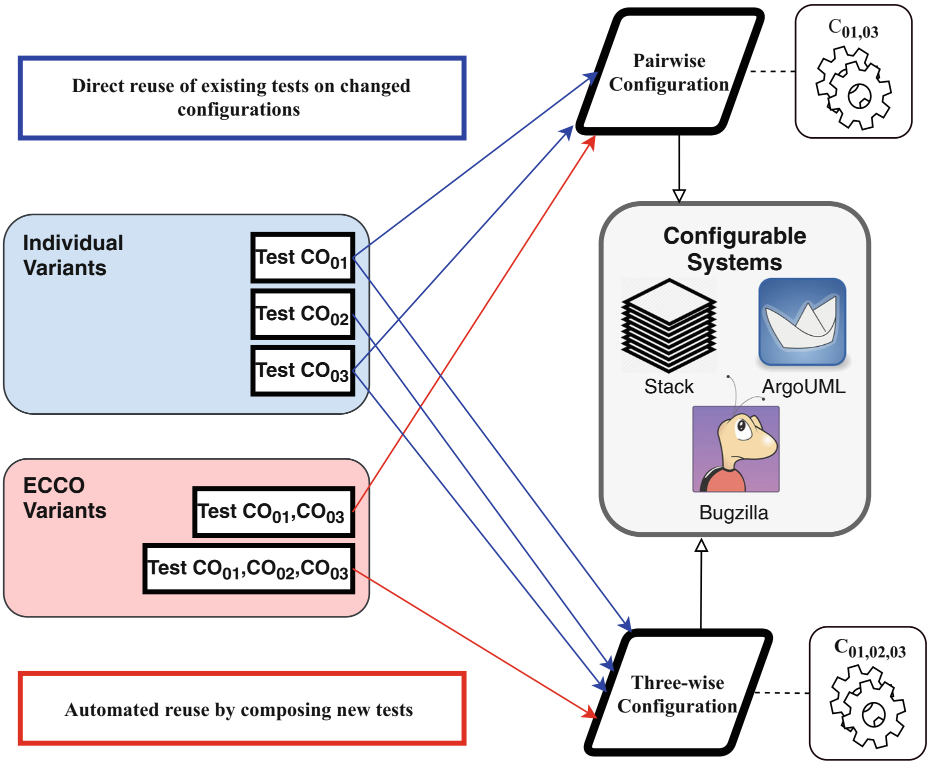

Fig. 11 Example Experiments on Pairwise and Three-wise combinations

Figure 11 illustrates the experiments on the pairwise and three-wise configurations. It shows the tests that we executed on the pairwise and three-wise configurations in both of our experiments. To setup the pairwise and three-wise configurations in the Stack SPL and ArgoUML, we automatically derived the source code from the SPLs, via preprocessor, and replaced the tests with the ones we wanted to execute at the time. Before, execution we made sure to clean the builds and recompile, to make sure that the correct tests would be executed. For Bugzilla we setup the configurations by performing the setups from the *individual* variants one after the other. 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 setups were executed.

All test variants of the Stack SPL and Bugzilla, original ones as well as those generated with *ECCO*, were realized as separate Maven projects. We used the Maven Surefire Plugin[6] to generate test reports and the JaCoCo[7] Maven Plugin to record code coverage. ArgoUML uses an Ant build system, which we executed with automated command line calls. We integrated JaCoCo in the build process to allow us to record the code coverage during testing.

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

5.4 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). By *failure* we refer to tests that fail because an assertion was violated, and by *error* we refer to cases where an exception led to the test failing. This terminology is also used in the test report, generated for each test execution.
- **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.

We measured the coverage of all our systems with JaCoCo. However, we were not able to measure the coverage of the Bugzilla Perl 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 Perl code itself and should be a feasible proxy of the code coverage.

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 the entire system (i.e. including all configuration options) that is executed during testing.

Mutation Metrics

- **Number of mutants killed** $KilledMut$. Number of mutants in a configuration that the tests detected.
- **Number of mutants created** $CreatedMut$. Number of mutants created for a configuration.
- **Mutation Score** $MutationScore$. Ratio of mutants in a configuration that were detected in relation to the number of mutants created.

$$MutationScore = KilledMut/CreatedMut$$

For mutation testing we used the PIT[8] mutation testing tool. Because of the minimum requirements of PIT, Java 5 or above and JUnit 4.6 or above, we were able to apply it only to the Stack SPL. We were not able to apply it to ArgoUML, since it uses JUnit 3.8.2 and has dependencies to it we could not port to a newer JUnit version. Bugzilla is developed in Perl, for which we were not able to perform mutation testing on. Nonetheless, we report the mutation score for Stack SPL, because we believe it provides a deeper insight in the quality of the tests.

To assess the manual effort required for finalizing the variants composed with *ECCO*, we compared them with the *original* variants derived from the annotated source code of the two SPLs (Stack and ArgoUML), and computed some metrics. For Bugzilla we do not have annotated code templates for the tests and therefore can not derive variants to compare our *ECCO* composed variants against. Therefore, we report the $Similarity$ only for the two systems Stack and ArgoUML.

Similarity Metrics

- **Number of Statements in a variant** $VarCount$. The number of statements that exist in a variant.
- **Number of surplus Statements** $Surplus$. The number of statements that exist in the *ECCO* variant but not in the variant derived from the annotated source code.
- **Number of missing Statements** $Missing$. The number of statements that exist in the *original* variant but are missing from the *ECCO* variant.
- **Similarity of two variants** $Similarity$. *Jaccard Similarity* of two variants, based on their statements.

$$Similarity = \frac{VarCount_{ECCO} - Surplus}{VarCount_{ECCO} + Missing}$$

To asses *ECCO*'s performance to extract the mapping and compose variants we measured the runtime and reports metrics. *ECCO* was executed on a HP EliteBook laptop with an Intel Core^{TM} i7-8650U processor (1.9 GHz, 4 cores) and 16GB of RAM and SSD storage running the Windows 10 operating system.

Runtime Metrics

- **Time to extract mappings** $ExtractTime$. The time for *ECCO*'s extraction to map configuration options to test code for a variant.
- **Time to compose a variant** $ComposeTime$. The time for *ECCO*'s compositions to retrieve traces and generate a variant.

[8]https://pitest.org/

6 Results

In this section, we summarize the results of our experiments on the different systems used.

6.1 Stack

We generated the test variants for pairwise and three-wise combinations of the Stack configuration options using *ECCO*. The extraction from the six *individual* Stack variants and the corresponding six configuration options with *ECCO* took 0.725 seconds. It took 1.8 seconds to generate all 15 pairwise and 1.8 seconds to generate the 20 three-wise variants.

Furthermore, we automatically derived the *original* test variants for all pairwise and three-wise configurations from the annotated test code of the SPL. This allows us to compare the results from the *ECCO* variants with the tests manually developed for a specific pairwise or three-wise configuration.

Figure 12 shows the $Similarity$ of the *ECCO* and the *original* variants. We can see that the *ECCO* variants are very close to the *original* variants, with an average $Similarity$ of 97.3% and 94% for pairwise and three-wise variants respectively. The majority of the differences are missing statements, which are on average 0.533 statements missing in six of the pairwise variants (four are missing one statement, two are missing two) and 1.45 statements missing in 15 of the three-wise variants (six variants are missing one statement, four are missing two statements, and five are missing three). On average, the *original* pairwise and three-wise variants include 23.1 tand 28.2 statements respectively. Additionally, one pairwise and four three-wise variants composed with *ECCO* contained a single surplus statement.

Figure 13 depicts the total number of tests executed for the different test variants. The number of tests in the *original* variants was always equal to the number of tests in the corresponding *ECCO* variant, ranging from four to six test cases. This is because none of the tests in the SPL are annotated with more than one configuration option, meaning that all tests are in at least one of the *individual* variants and can therefore be reused by *ECCO*. The *individual* variants generally have fewer tests, because they are missing some tests that are specific from another configuration option.

The $SuccessRate$ obtained from those test cases is always 100% for *ECCO* and *original* pairwise and three-wise tests. However, the *individual* variants achieved only a

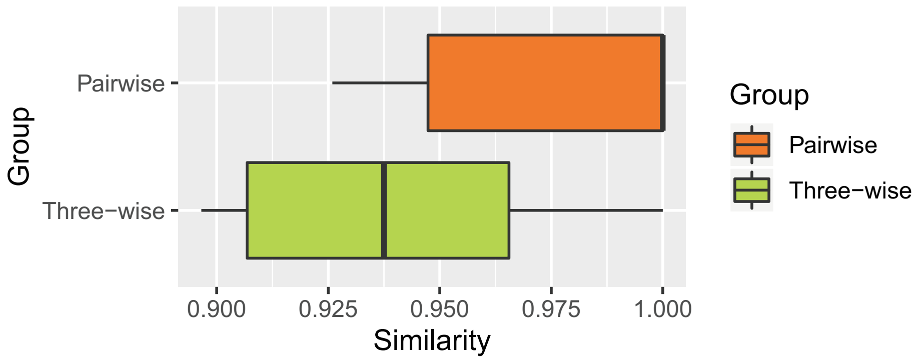

Fig. 12 $Similarity$ of the *ECCO* pairwise and three-wise test variants to the *original* variants of Stack

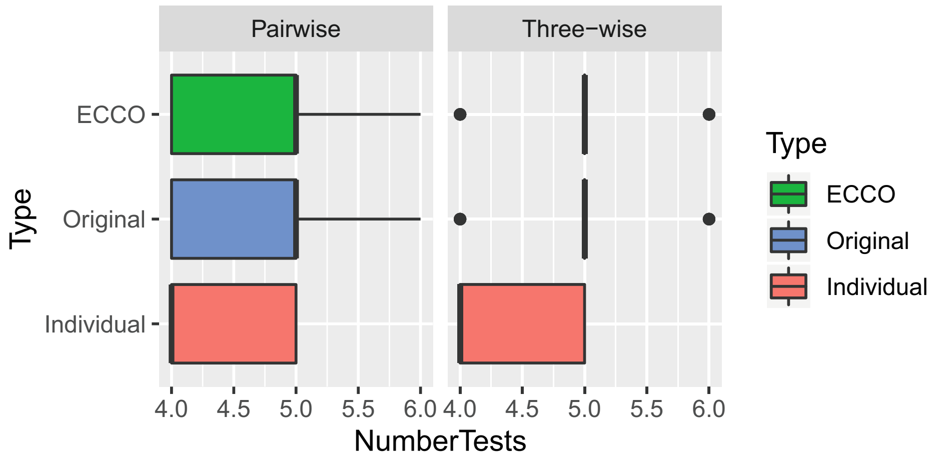

Fig. 13 Number of Test Cases ($Tests$) in the *ECCO* and *original* pairwise and three-wise test variants and the *individual* variants on pairwise and three-wise configurations of Stack

75% $SuccessRate$ in some cases. This happened in the pairwise combination of configuration options `EmptyStackException` and `ReturnNull`. When applying the tests from the *individual* variant for testing only configuration option `ReturnNull` an `EmptyStackException` occurs in a test that was not expected for this test. The same happens for all three-wise configurations that combine both configuration options `EmptyStackException` and `ReturnNull`, when running the *individual* variant for `ReturnNull`. This is an expected conflict, because the behavior of calling method `pop` on an empty stack can only be either `EmptyStackException` or `ReturnNull`, which is why these two configuration options are in an alternative in the feature model in Fig. 8. Nonetheless, we included this combination for completeness of the results and to see how the tests would behave. For all other cases, the $SuccessRate$ was also 100%.

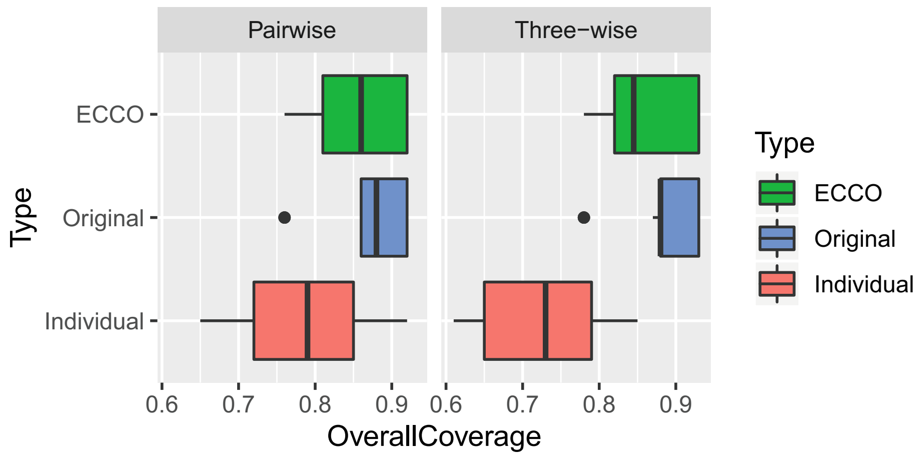

Fig. 14 $OverallCoverage$ of the *ECCO* and *original* pairwise and three-wise test variants and the *individual* variants on pairwise and three-wise configurations of Stack

The results for the $SuccessRate$ showed that *ECCO* can reliably generate test variants for the Stack, that can be executed on new configurations. Next, we depicted the $OverallCoverage$ of the different test variants in Fig. 14. *ECCO* test variants covered nearly the same number of statements as the *original* variants, in both pairwise and three-wise combinations. The cases where the *ECCO* variants have a lower coverage that the *original* ones come from statement inside the *original* test variants that only occur in combinations of different configuration options. For instance, there is a test to check the behavior of the stack when calling the `peek` method on an empty stack, which should behave similar to an empty stack `pop`, depending on configuration. However, the *individual* variants only test configuration options `peek`, `EmptyStackException` and `ReturnNull` in isolation. Therefore, none of the *individual* variants contain the code inside the test that evaluates the behavior correctly, which means *ECCO* could not reuse it and therefore has a lower coverage in some cases. This shows that, even though *ECCO* appears helpful for reusing tests, in some cases manual effort is necessary to complete the generated test variants. For this reason and because of the fewer tests, the $OverallCoverage$ of the *individual* variants was also lower than the other variants.

To further evaluate the quality of the tests we employed the mutation testing tool Pitest, to introduce small changes in the source code and check how well the tests could detect them. We depict the $MutationScore$ (the ratio of introduced mutants that were detected during testing) in Fig. 15. The *original* test variants are slightly more adequate to detect mutants than the *ECCO* generated tests for both pairwise and three-wise variants, which correlates with the results obtained for $OverallCoverage$.

Most of the *individual* test variants achieved a $MutationScore$ of 50 to 70%, for pairwise and three-wise configurations, while the *ECCO* merged test variants and the *original* variants got up a mutation score from 60% up to 80% in pairwise configurations and up to 70% to 80% in three-wise configurations. These results further underline the need to adapt the tests for changed configurations, and the usefulness of *ECCO* to support this process.

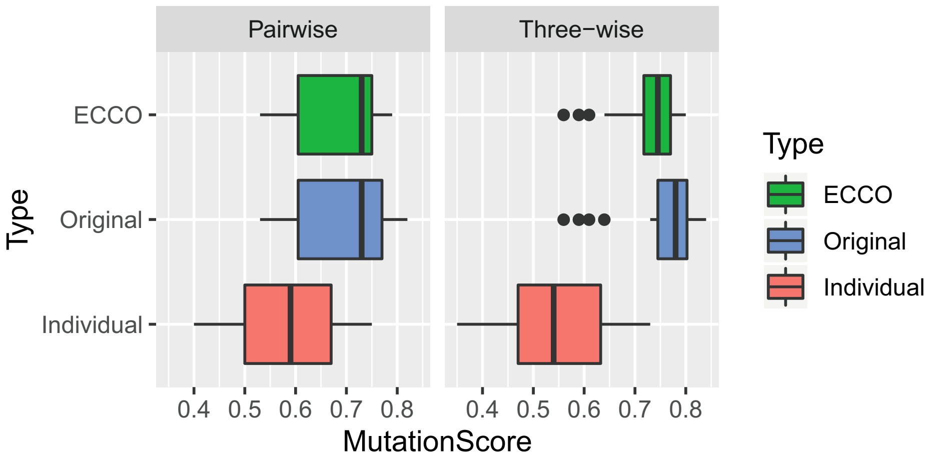

Fig. 15 $MutationScore$ of the *ECCO* and *original* pairwise and three-wise test variants and the *individual* variants on pairwise and three-wise configurations of Stack

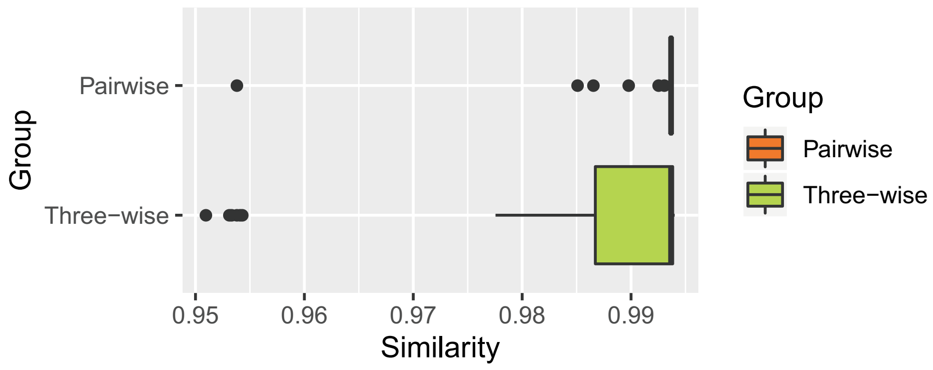

Fig. 16 *Similarity* of the *ECCO* pairwise and three-wise test variants to the *original* variants of ArgoUML

6.2 ArgoUML

ECCO took 13.8 seconds to extract the test code for the configuration options from the eight *individual* variants and about 23.8 seconds to generate the 28 pairwise variants. We also generate the three-wise combination for ArgoUML, which resulted in 56 variants in 48.3 seconds. Again, we created the pairwise and three-wise test variants with *ECCO* and automatically derived the *original* test variants for the pairwise and three-wise configurations from the annotated test code. All tests are executed on the source code for the pairwise or three-wise configurations, derived from the SPL.

Figure 16 depicts the *Similarity* of the *ECCO* and the *original* variants. The average *Similarity* for the pairwise and three-wise variants was 99.2% and 98.7% respectively. On average, the *original* pairwise and three-wise variants include 4616 and 5095 statements respectively. The pairwise *ECCO* variants included on average 15.2 statements surplus and where missing on average 28.9 statements. Similarly, the three-wise *ECCO* variants had on average 16.9 statements surplus and were missing 59 statements on average.

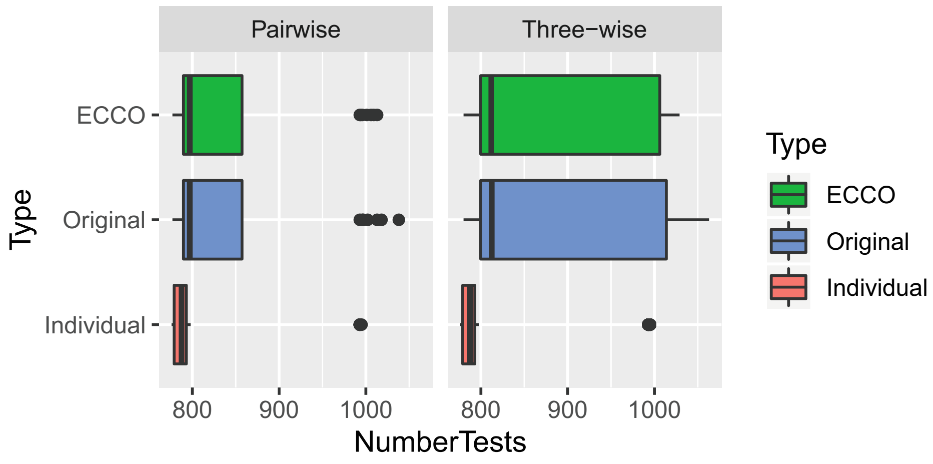

Fig. 17 Number of Test Cases (*Tests*) in the *ECCO* and *original* pairwise and three-wise test variants and the *individual* variants on pairwise and three-wise configurations of ArgoUML

Figure 17 shows the total number of tests executed for the pairwise and three-wise configurations for *ECCO* variants, the *original* test variants, and the *individual* variants. The number of tests in the *ECCO* variants and the *original* test variants are very close, ranging from 700 to 1100. The *ECCO* variants are missing some test cases that the SPL only adds to variants that have specific combinations of two or three configuration options. Therefore, none of the *individual* variants that *ECCO* used as input contain these tests. Since the *individual* variants only test configurations with individual configuration options, they also have fewer tests, about 770 to 800 tests for most variants. The only outlier is the *individual* variant testing configuration option `cognitive`, with nearly 1000 tests. This configuration option is linked to a larger number of tests than the others and is also responsible for many of the tests in the pairwise and three-wise variants, which explains the relatively low medians in the box plots *ECCO* and *original* variants in Fig. 17, because only combination with `cognitive` contain the most tests. Generally, there are large overlaps in the tests that different variants contain. This means many of the tests are included independent of the configuration that should be tested.

The $SuccessRate$ for all variant's tests was always 100%. Meaning no test raised any failures or errors. Therefore, *ECCO* was able to reliably merge the tests from different variants. Even though there seems not to be a clear advantage in the $SuccessRate$ when using *ECCO*, automatically being able to select tests for the corresponding configuration options is still useful. On the results for $OverallCoverage$ in Fig. 18, we can see the coverage for *ECCO* variants is always higher than the one for the *individual* variants, correlating with the number of tests in a variant. Moreover, the $OverallCoverage$ for *ECCO* was nearly as high as the $OverallCoverage$ for the *original* test variants.

Unfortunately, the $OverallCoverage$ of the tests used for ArgoUML was not very high, only up to about 14%. We confirmed this by also testing a variant, derived from the SPL, with all configuration options activated, and did not get much higher values. Therefore, this seems to be inherent to the tests for ArgoUML and not related to our experiment design. *ECCO* can by design not achieve a higher coverage than the *original* variant, since it only merges code that we provided as input.

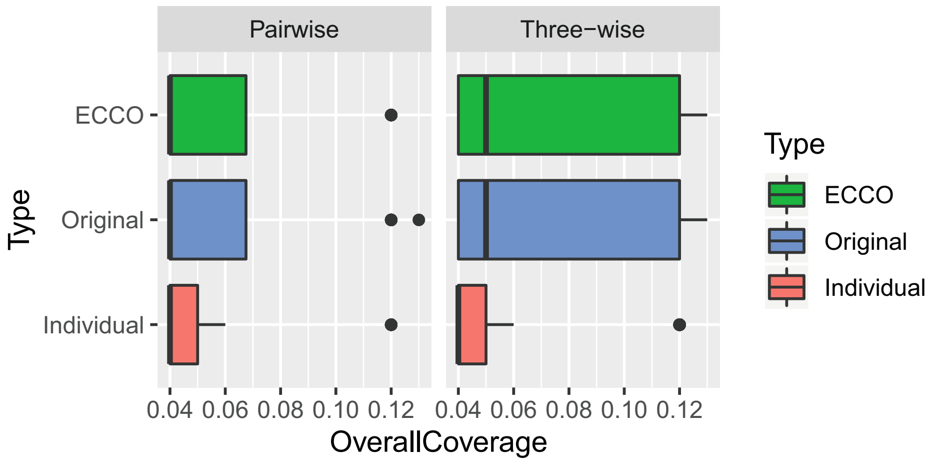

Fig. 18 $OverallCoverage$ of the *ECCO* and *original* pairwise and three-wise test variants and the *individual* variants on pairwise and three-wise configurations of ArgoUML

Generally, the tests for ArgoUML did not contain much variability in their code. Based on the configuration entire tests are added or removed when deriving the variants from the SPL, but the code inside the tests stays the same independent of configuration. This also explains why we always achieved a $SuccessRate$ of 100%.

6.3 Bugzilla

Next, we present the results for the two versions of Bugzilla that we used in our experiments. Since we did not have configurable tests for Bugzilla we can not compare our results to the *original* test results that could be achieved with tests devised specifically for the pairwise or three-wise configurations. We compare the results of our test variants generated with *ECCO* with the results of using the *individual* variants that were manually developed for the configurations with *individual* configuration options activated. To address specific configurations in Bugzilla we will use $C_{x,y,...}$ with the set of identifying numbers for the configuration options that have been activated (e.g. $C_{01,02}$ is the pairwise configuration that has both configuration options CO_{01} and CO_{02} from Table 1 activated). Similarly, we use $T_{x,y,...}$ with the set of identifying numbers for the configuration options that have been activated, for test variants that test a specific configuration (e.g. $T_{01,02}$ is the variant to specifically test configuration $C_{01,02}$).

6.3.1 Bugzilla 3.4

For Bugzilla 3, we could use all 15 configuration options, and therefore can compose 105 test variants for pairwise configurations with *ECCO* and 455 three-wise. It took 39.3 seconds to extract the mapping information from the 16 initial variants and 21.1 seconds to generate the 105 new pairwise variants and 85 seconds to generate all the 455 three-wise variants.

Four out of the 105 variants resulted in a compiler error and could therefore not be executed ($T_{04,05}$, $T_{04,09}$, $T_{09,11}$, and $T_{09,12}$). 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. Similarly, all 48 three-wise variants combining the tests for the same configuration options cause compiler errors. We executed the tests for the remaining 101 pairwise and 407 three-wise variants and computed our metrics. Moreover, during our experiment, we discovered that for four pairwise configurations ($C_{07,09}$, $C_{07,10}$, $C_{08,09}$, $C_{08,10}$), the setup did not work in one order of execution and in the other order we could not successfully reset the configuration. Nonetheless, we included the test results for the order that could be setup for these variants in our data, but we had manually override the image for the virtual machine running Bugzilla to do a hard reset and be certain that it was set back to the default configuration. Again, the same applied to all 42 three-wise configurations containing these combinations. The remaining configurations could be set up in both orders and the results for our metrics were the same in both orders for most configurations. The only exception were configurations combining CO_{14} and CO_{15}, due to the expected conflict between the two configuration options by setting different default statuses for duplicate bugs. Depending on the order of the setup, either the tests of the *individual* variant T_{14} or T_{15} had a higher $SuccessRate$. The results for the *ECCO* variants or other *individual* variants stayed the same, and the $SuccessRate$s for T_{14} and T_{15} did an exact flip with the order in which the configurations were set up and we included the results, because there was effectively no difference in the numbers.

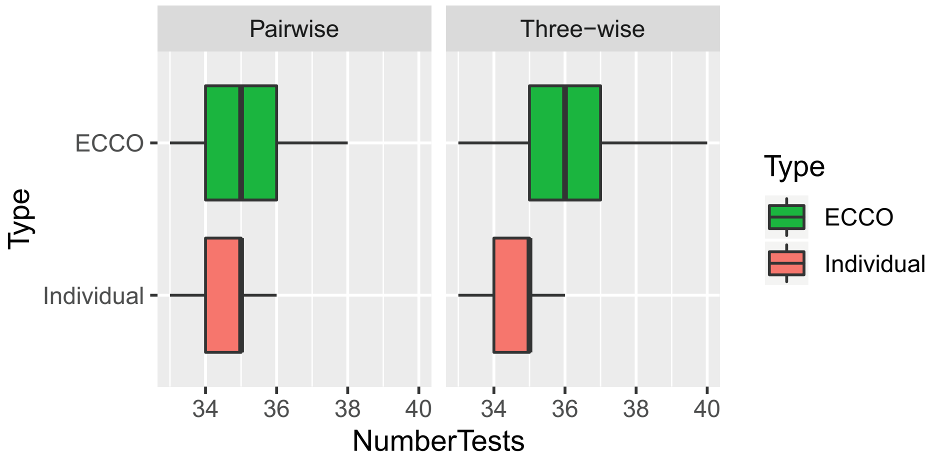

Fig. 19 Number of Test Cases ($Tests$) in the *ECCO* pairwise and three-wise test variants and the *individual* variants on pairwise and three-wise configurations of Bugzilla 3

Figure 19 depicts the number of tests executed from the *ECCO* and *individual* variants on pairwise and three-wise configurations. The number of tests in the *individual* variants ranges from 33 to 36 tests (see Table 1). For the *ECCO* variants, 33 to 38 tests are contained in the pairwise variants and up to 40 tests in three-wise variants, because tests associated with different configuration options are merged together in these variants.

The $SuccessRate$ for executing the tests from the different variants is depicted in Fig. 20. As we can see, the test variants generated with *ECCO* generally have a higher $SuccessRate$ than any of the *individual* variants. Moreover, there is a slight drop in $SuccessRate$ from pairwise to three-wise configurations, which is because there is a higher likelihood of combinations that cause certain tests to fail.

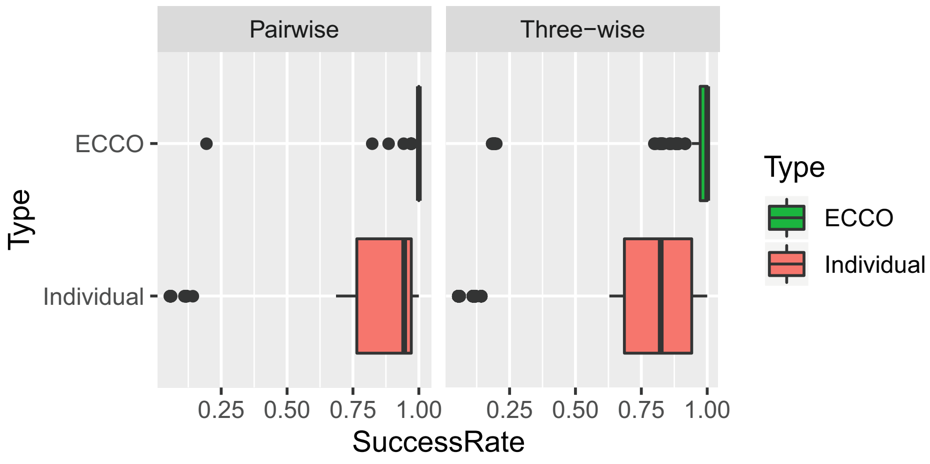

Fig. 20 $SuccessRate$ of the *ECCO* pairwise and three-wise test variants and the *individual* variants on pairwise and three-wise configurations of Bugzilla 3

Table 2 Contingency Tables of Successfully Passing Tests for *ECCO* vs. *individual* variants for Bugzilla 3

		individual variants Successful				Total
(a) Pairwise Configurations						
		None	One	Both		
ECCO	Success	67	24	1		92
variants	Fail	3	6	0		9
Total		70	30	1		101
(b)Three-wise Configurations						
		None	One	Two	All Three	
ECCO	Success	292	11	0	0	303
variants	Fail	93	11	0	0	104
Total		385	22	0	0	407

We additionally compared the test variants that could run on the pairwise and three-wise configurations without any further manual effort in the contingency Table 2a and b. The majority of the pairwise variants generated with *ECCO* could run on the pairwise configurations without any problem (92 out of 101). On the other hand, the *individual* variants only worked on 31 configurations (for 30 pairwise configurations one *individual* variant worked and for one configuration both *individual* variants worked without any errors or failures). For three-wise configurations, the results are somewhat worse, but still 74% (303 configurations) of the three-wise configurations could be tested with the *ECCO* variants without any problems. These results further supports the usefulness of *ECCO* to reuse tests for specific configurations.

Figure 21 depicts the $OverallCoverage$ for the different test variants. We found that generally the *ECCO* tests have higher coverage in most cases, and therefore executed more of the code of our page objects. However, since *ECCO* merges code from *individual* variants

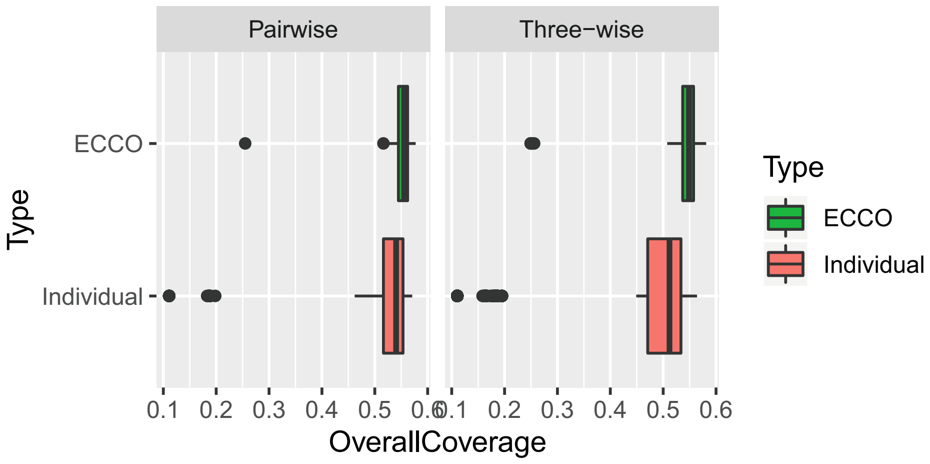

Fig. 21 $OverallCoverage$ of the *ECCO* pairwise and three-wise test variants and the *individual* variants on pairwise and three-wise configurations of Bugzilla 3

we would expect the generated variants to contain more code to execute than the *individual* variants.

6.3.2 Bugzilla 5.1.1

For Bugzilla 5 we can generate 78 pairwise and 286 three-wise combinations of the 13 configuration options. The extraction with *ECCO* took 14.7 seconds. The composition took 19.8 seconds for the pairwise variants and 67.8 seconds for the three-wise variants.

One of the pairwise combinations resulted in a compiler error, which was the variant $T_{04,09}$, that caused also a problem in Bugzilla 3. We excluded this variant from our experiments. Similarly, all eleven three-wise variants that merged T_{04} and T_{09} together caused a compiler error, which led us to exclude these eleven variants from the experiments. This leaves us with 77 pairwise and 275 three-wise test variants for the experiments.

For four pairwise and 38 three-wise combinations, the setup only worked in one of the two orders. This happened for combinations with $C_{07,09}$, $C_{07,10}$, $C_{08,09}$, $C_{08,10}$, which are the same combinations that caused this problem for Bugzilla 3. Again, the automatic reset of these configurations did not work, and we had to manually overwrite the virtual machine image to make sure Bugzilla was in the default configuration again after executing the tests. We included the results for the order that worked.

For the remaining configurations, for which the setup worked on both orders, there was no difference in the results, no matter which setup order was used. We show the number of tests executed on the pairwise and three-wise configurations for the *ECCO* variants and the *individual* variants in Fig. 22. The number of tests in the *ECCO* variants range from 33 to 37 in the pairwise variants and up to 39 in the three-wise variants. The *individual* variants generally have fewer tests, ranging from 33 to 35 test cases, because they are missing some tests that are specific from another configuration option.

Figure 23 depicts the $SuccessRate$ of the *ECCO* variants and the *individual* variants. For most configurations, the $SuccessRate$ of the *ECCO* variants (on average 97.3% and 94.8% for pairwise and three-wise respectively) was higher than for the *individual* variants (on average 78.7% and 63.3% for pairwise and three-wise respectively). These results support the usefulness of *ECCO* to reuse tests, even though the $SuccessRate$ was 100% only for 33 pairwise and 65 three-wise of the *ECCO* variants. We argue that the manual effort

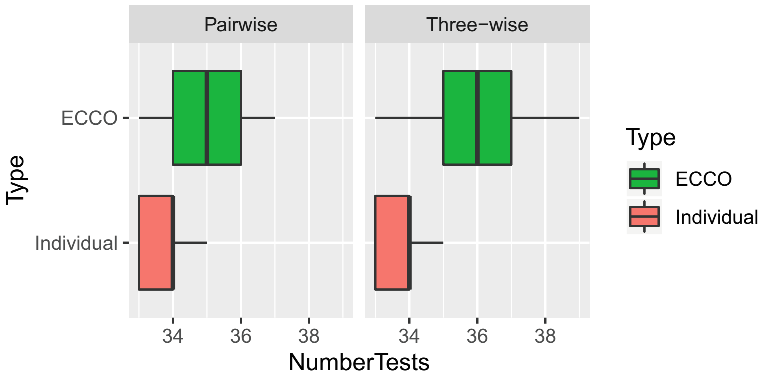

Fig. 22 Number of Test Cases ($Tests$) in the *ECCO* pairwise and three-wise test variants and the *individual* variants on pairwise and three-wise configurations of Bugzilla 5

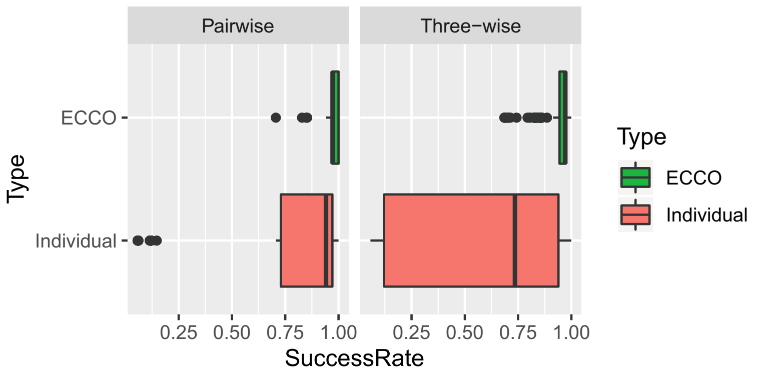

Fig. 23 *SuccessRate* of the *ECCO* pairwise and three-wise test variants and the *individual* variants on pairwise and three-wise configurations of Bugzilla 5

of fixing the few failing *ECCO* tests should be lower than adapting and manually combining the tests from the *individual* variants that appear to have even more problems in the execution.

Like for Bugzilla 3, we also checked how many of the variants could run without any error on Bugzilla 5 pairwise and three-wise configurations and show the results in Table 3a and b. However, the results for Bugzilla 5 are worse than for Bugzilla 3, in that fewer variants could be reused without a problem, implying a higher amount of manual effort to fix the test cases for the different configurations. For the pairwise configurations, only 33 out of the 77 *ECCO* variants worked on their intended configuration. Similarly, only 65 three-wise *ECCO* variants worked without any issue on their intended three-wise configuration. For the remaining 44 pairwise and 210 three-wise variants manual effort is required to fix them. Nevertheless, for both pairwise and three-wise configurations *ECCO* variants performed better in terms of *SuccessRate* than the *individual* variants. Only 23 of 275 the three-wise configurations could apply a single one of the *individual* variant without any adaption.

Consistent with the different numbers of test cases and the difference in *SuccessRate* the results for *OverallCoverage* show that the *ECCO* variants generally cover more of

Table 3 Contingency Tables of Successfully Passing Tests for *ECCO* vs. *individual* variants for Bugzilla 5

		individual variants Successful				Total
(a) Pairwise Configurations						
		None	One	Both		
ECCO Pair	Success	19	13	1		33
Variants	Fail	32	12	0		44
Total		51	25	1		77
(b)Three-wise Configurations						
		None	One	Two	All Three	
ECCO Pair	Success	59	6	0	0	65
Variants	Fail	193	17	0	0	210
Total		252	23	0	0	275

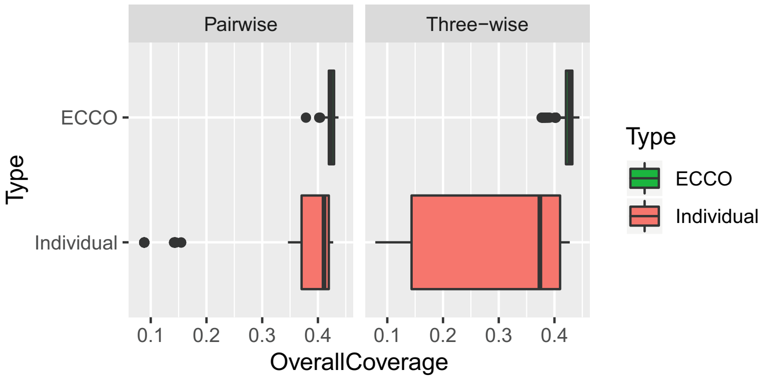

Fig. 24 *OverallCoverage* of the *ECCO* pairwise and three-wise test variants and the *individual* variants on pairwise and three-wise configurations of Bugzilla 5

the page objects' source code (see Fig. 24). The average *OverallCoverage* was 42.5% for the *ECCO* variants and 32.3% for the *individual* variants. The larger number of tests in the *ECCO* variants has a positive effect on the coverage, as well as the higher *SuccessRate*, since failing tests might have more test steps after the failure or error occurred that will not be executed anymore.

6.4 *ECCO* Runtime

The *ECCO ExtractTime* in seconds for each variant is depicted in Fig. 25. We observed that the extraction generally takes longer for larger systems, because more code and therefore larger ASTs have to be mapped to their configuration options. For the Stack, the extraction took less than one second for each variant. For the other systems, it took on average between one and three seconds to extract a variant. The first variant added to *ECCO* took slightly longer for all systems. This is because of the warm-up effect of the Java Virtual Machine (JVM), due to class loading and bytecode interpretation at startup. After this, the time to add a variant to the mapping is relatively constant for the systems, until we have added about twelve variants. As the number of variants that have been added to *ECCO*

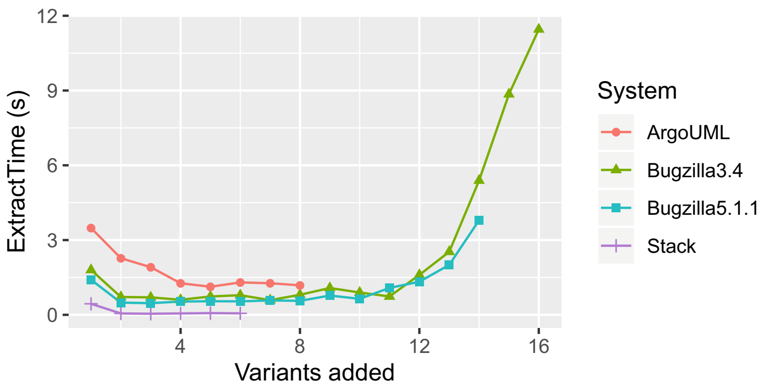

Fig. 25 *ExtractTime* of *ECCO* for individual variants per system

increases, the operations to update the existing mapping get more computational expensive. This is in part because the new variant needs to be compared to more code fragments that already exist in the mapping. The other reason for this increase in time is because of the increasing number of configuration options, which means the extraction needs to consider more combinations of configuration options that could be mapped to source code fragments. Generally, the number of configuration options is the largest contributor to the $ExtractTime$ of $ECCO$.

Figure 26 depicts the $ComposeTime$ to compose the pairwise and three-wise variants for each system with $ECCO$. As the smallest systems, the Stack took the least amount of time to compose. ArgoUML, the largest system in terms of lines of code, took the longest with up to four seconds to compose a variant. Nevertheless, for ArgoUML it took on average only 0.8 seconds to compose a variant. For Bugzilla, the composition took on average 0.2 seconds to generate a single variant. We did not observe any difference in composition times of pairwise and three-wise variants. The outliers in Fig. 26 can be attributed to the JVM warm-up effect again, as they happened when composing the first variants. The main driver for the composition time for variants was the size of the code base.

7 Discussion

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

RQ1: To what degree can tests from specific configurations be reused directly? In our experiments on the first two systems, we found that in most cases the tests could be applied with a high $SuccessRate$ even on different configurations. This is because we activated additional configuration options which in these two systems mainly have the effect of adding functionality and in most cases did not change the behavior of existing functions, with the exception of the behavior of the Stack when trying to pop without pushing anything before. Especially, in the case of ArgoUML, the test cases are unit tests that test different functionality, which is either added or removed depending on the configuration, but its internal behavior stays the same regardless of other configuration options that are activated.

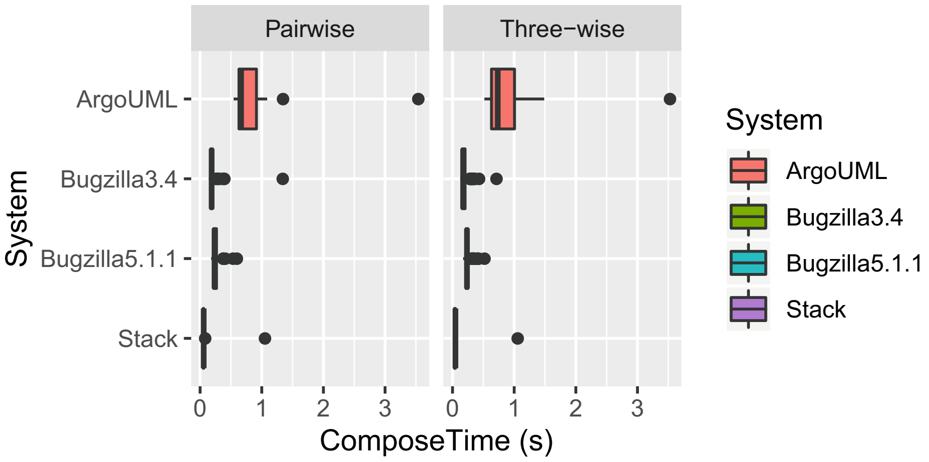

Fig. 26 $ComposeTime$ of $ECCO$ for pairwise and three-wise variants per system

The tests for Bugzilla however, are system tests that had to be adapted to run on different configurations. Therefore, we could observe that tests for use cases that changed according to a configuration option would no longer work after changing the configuration. Nonetheless, there were still many tests that were not affected by changing the configuration, because specific configuration options only changed certain of the tested use cases, leading to an average $SuccessRate$ for the two versions of 69.9% and 65.7%, respectively. This still shows the need to manually adapt most test variants for the majority of new configurations, which requires considerable manual effort.

RQ2: To what degree can we automatically generate test suites for new configurations from existing tests? Our results support the general usefulness of *ECCO* to automatically generate new test variants, by merging existing tests for the configuration options. For the first two configurable systems we used in our experiments, Stack and ArgoUML, we found that the test generated with *ECCO* had always a $SuccessRate$ of 100%. However, we could achieve the same by reusing the *individual* variants that we had from the start. Nonetheless, our *ECCO* generated tests achieved a higher $OverallCoverage$, and in most cases came very close to the results of the *original* variants, derived from the SPLs to test the pairwise or three-wise configurations. Especially, in the case of the Stack, where we were able to perform mutation testing, we found that the *ECCO* tests were able to kill nearly as many mutants as the *original* variants, and clearly outperformed the *individual* variants.

For the two versions of Bugzilla, we found that tests generated by *ECCO* generally outperformed the *individual* test variants in terms of $SuccessRate$ and $OverallCoverage$. Considering the results for pairwise combinations for Bugzilla 3, we found an average $SuccessRate$ of 98.72% for tests we generated with *ECCO*, compared to a $SuccessRate$ of 81.84% for using the two *individual* 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 is unknown before execution. If we would instead always choose the worse of the two variants, the average $SuccessRate$ drops to 67.9%. Similarly, for the three-wise combinations for Bugzilla 3 we found a found a average $SuccessRate$ of 95.9% for the *ECCO* test variants, while the *individual* variants achiever a $SuccessRate$ of only 68.8%.

The results for Bugzilla 5 turned out similar to the ones from Bugzilla 3. We found an average $SuccessRate$ of 95.3% for *ECCO* test variants (97.3% for pairwise and 94.8% for three-wise), while the average $SuccessRate$ for *individual* variants was 65.7% (78.7% for pairwise and 63.3% for three-wise).

Generally, the results for testing three-wise configurations could be predicted from results of testing the pairwise configurations before, since combinations that caused problems in pairwise would also cause these problems when adding another configuration option to three-wise combinations. If any problems existed in pairwise variants, we would find the same ones for three-wise variants and therefore the results for three-wise variants were slightly worse because more interactions between configuration options can occur there. Nonetheless, for Bugzilla some problems came up for the three-wise *ECCO* variants that did not exist in pairwise, which could imply interactions that only appear in three-wise combinations of configurations options, or that *ECCO* failed to generate the tests as well in these cases. If we would want to use *ECCO* to generate test variants for combinations of even more configuration options, for instance tests for Combinatorial Interaction Testing (CIT), we would most likely run into more problems like these. By evaluating *ECCO* on pairwise and three-wise combinations, as we did in this paper, we can better assess how common the problems for combinations are. Moreover, research suggests that most faults in a system can be detected with only covering all pairwise and three-wise combinations (Halin et al. 2019; Kuhn et al. 2004). Any configurations in a CIT covering array that includes one of these problem combinations would therefore also have issues that need to be manually fixed, when we composed it with *ECCO*. CIT is used because faults in configurable software can occur due to interactions of multiple configuration options. However, these interactions also make the reuse of tests more difficult. When using *ECCO*, we would expect some manual effort from developers to fix these interaction problems in the tests. Adding these fixed variants to *ECCO*s input would then help it in learning how to deal with these interactions and be able to reuse the fixed code in the future. Even with these problems, we still expect the overall manual effort for reusing tests to be lower when using *ECCO* than performing the entire reuse manually.

We believe that *ECCO* can help to reduce the effort for reusing existing test code for new configurations, and is therefore beneficial for testing highly configurable software. Nevertheless, it can not eliminate the need for manually inspecting and when necessary completing the generated tests, to avoid errors from different behavior of configurations that *ECCO* could not predict. The similarity of the *ECCO* variants to the *original* variants, for the two systems we could compute it for, shows that the variants can be generate with a high accuracy. Therefore, the changes that have to be made to finalize a composed variant should be manageable and we argue that the effort to fix an *ECCO* variant should be lower than to develop a variant from scratch or by applying manual clone-and-own. To show that this assumption is true in practice more research in needed and a user study to proof this is an item of our future work.

Another use case for which *ECCO* could be useful is the refactoring of individual test variants when moving variants that were developed using clone-and-own to a platform for more systematic reuse. We believe our results support the usefulness of *ECCO* for such a refactoring.

7.1 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. Moreover, it turned out that mutation testing was unfortunately only feasible on the smallest system, the Stack SPL, because of technological limitations. To further study the usefulness of our 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, and three-wise combinations of previously tested configuration options. 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 can not be used to generate entirely new test code.

To enable the replication and comparison of our study, we opted to choose open-source systems. However, publicly available configurable systems with tests are hard to come by. This limited us to perform our experiments on only a few configurable systems we could find, and even there we had to create the Stack SPL ourselves and annotated the tests of ArgoUML to make them useful for our experiment.

7.2 Threats to Validity

External validity Our study includes three configurable systems. Two of the systems are SPLs and the other one (Bugzilla) is a configurable software system for which we used two different versions. To determine the degree to which results may be generalized, studies on more systems may be needed in the future. However, we used systems that implement configurations in a diverse manner to represent different types 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 the variability models of the SPLs and by 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 respectively annotated by the authors. The Stack SPL was entirely developed by the authors, and the annotations for the ArgoUML tests were also done by us in a process with automatic support with source code analysis. For Bugzilla, we developed the tests for the default configuration and then for the other 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 some 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 in Bugzilla. 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 and three-wise configurations with the setup in two orders to further reduce the potential of bias because of it.

To show the applicability of *ECCO* for our test code we used it to reconstruct the individual variants we provided it as input. Hence, we used all individual variants as input for *ECCO* and regenerate all of them. We compared the Abstract Syntax Tree (AST) of the individual variants with *ECCO*'s reconstruction of the variants and found no difference. Moreover, we executed all tests on the reconstructed individual variants 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. 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 systems and were also not able to perform mutation testing for two of our three systems. We argue that comparing the code coverage of different variants should be a good proxy for comparing their mutation score, because the variants generally perform the same checks if they reach certain code parts. Moreover, code coverage and mutation score are usually correlated with one another, since a mutant can only be detected if it was covered (Aaltonen et al. 2010).

Moreover, for Bugzilla we measured the code coverage only on the Java page objects, because we could not find a feasible way to record the code coverage of Bugzilla's Perl code. 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.

Due to the difficulty to find tests for open-source configurable systems to use in our experiments, we had to analyze the tests from ArgoUML and insert the annotations (following the Fischer et al. methodology (Fischer et al. 2018)) of which tests should belong to which configuration option. Since, we are not the SPL developers we could not make sure that our tests were defined sufficiently well to test each configuration option. Moreover, the tests for ArgoUML achieved a fairly low coverage of the system, even in a configuration with all configuration options enabled, which is an indicator that the test we used may be of poor quality. However, the goal of our experiment was to reuse existing tests, and we showed that *ECCO* was useful for this task and achieve similar results as the original tests directly derived from the SPL.

8 Related Work

8.1 Test Reuse for Configurable Software

Cohen et al. performed an experiment to show the effect of executing tests for different configurations of a highly configurable system (Cohen et al. 2006). 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. In their experiments, they injected mutants to simulate faults in the system. However, we did not have fault data available, and could only inject mutants in one of our systems, 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 (Ramler and Putschögl 2013). 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 a changed configuration 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 (Krüger et al. 2018). 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.

8.2 Configuration Aware Testing

There exists a considerable amount of research on techniques for testing configurable systems. Various approaches have been devised that sample configurations to test or to select tests that should be executed on specific configurations to avoid re-executing redundant tests or variants. We summarize some of them here. Kim et al. performed static code analysis on

SPLs to reduce the number of configurations to test, by identifying configuration options that have no effect on a given test (Kim et al. 2011). Further work by Kim et al. and Souto et al. improves this by using dynamic analysis to determine what configuration options can be reached by a test and therefore reduce the test effort (Kim et al. 2013; Souto et al. 2017). Similarly, Nguyen et al. proposed Varex, a variability aware interpreter that identifies code that is common when testing multiple configurations and subsequently only executing it for one configuration (Nguyen et al. 2014). Reisner et al. use symbolic execution to evaluate how configuration options affect the coverage of a configurable system for a given test suite and can therefore reduce the configuration space that needs to be considered for testing (Reisner et al. 2010).

Employing similar analysis techniques in combination with *ECCO* could help a developer in identifying test cases that require manual fixing, by identifying configuration options that affect a test that was previously not executed on those options. Therefore, a developer could be directed to the tests that need fixing and even be informed of the configuration options that should be considered when altering the test code.

9 Conclusions and Future Work

In this paper, we performed experiments on test reusability across configurations of highly configurable software systems. Furthermore, we used an approach for automated reuse to generate tests for new configurations by reusing previously developed test variants. Our experiments showed that for most configurations a large proportion of test cases could be applied to changed configurations without problems. For our first two systems, nearly all *individual* test variants could be reused, and for the two versions of Bugzilla around 70% to, in some cases, 100% of tests cases could be directly reused. Automatically reusing tests yielded even better results in success rate and code coverage. Moreover, we were able to apply mutation testing to one of the systems. The results showed that the number of mutants detected during testing was higher with the automatically generated tests than with the existing *individual* test variants that were used as input for the automated reuse. These results suggest a considerable advantage for using automated test reuse over the direct reuse approach, which requires additional manual effort for adapting the failing tests.

In our future work, we ideally could apply the automated reuse to industrial systems with a high amount of variability to confirm the results of our experiments. Ideally, we could get access to configurable systems with fault data available to better assess the quality of reused tests in future experiments. Furthermore, we would like to investigate if we could use *ECCO* to support the evolution of tests for different version. For instance, we could manually evolve some of the test variants for Bugzilla 3.4 to 5.1.1 and use *ECCO* to check if the changes made during evolution could be reused to evolve the remaining variants automatically. We could compare the results in similar experiments as in this paper. Another direction we believe that would be valuable to research is the application of *ECCO* to different testing techniques, like for instance model-based testing. Moreover, 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. Finally, we want to perform a user study to better evaluate the effort of manual clone-and-own reuse versus for applying *ECCO*.

Acknowledgements The research reported in this paper has been funded by the Austrian Ministry for Transport, Innovation and Technology (BMVIT), the Federal Ministry for Digital and Economic Affairs (BMDW), and the Province of Upper Austria in the frame of the COMET - Competence Centers for Excellent

Technologies Programme managed by Austrian Research Promotion Agency FFG. This research was in part funded by the Linz Institute of Technology (LIT) Secure and Correct Systems Lab and the Austrian Science Fund (FWF), grant no. P31989.

References

Aaltonen K, Ihantola P, Seppälä O (2010) Mutation analysis vs. code coverage in automated assessment of students' testing skills. In: Cook WR, Clarke S, Rinard MC (eds) Companion to the 25th annual ACM SIGPLAN conference on object-oriented programming, systems, languages, and applications, SPLASH/OOPSLA 2010, october 17-21, 2010, Reno/Tahoe, Nevada, USA, pp 153–160, ACM. https://doi.org/10.1145/1869542.1869567

Benavides D, Segura S, Cortés AR (2010) Automated analysis of feature models 20 years later: A literature review. Inf. Syst. 35(6):615–636

Berger T, Lettner D, Rubin J, Grünbacher P, Silva A, Becker M, Chechik M, Czarnecki K (2015) What is a feature?: a qualitative study of features in industrial software product lines. In: Schmidt DC (ed) Proceedings of the 19th International Conference on Software Product Line, SPLC 2015, Nashville, TN, USA, July 20-24, 2015, pp 16–25, ACM. https://doi.org/10.1145/2791060.2791108

Berger T, She S, Lotufo R, Wasowski A, Czarnecki K (2013) A study of variability models and languages in the systems software domain. IEEE Trans. Software Eng. 39(12):1611–1640. https://doi.org/10.1109/TSE.2013.34

Cohen MB, Snyder J, Rothermel G (2006) Testing across configurations: implications for combinatorial testing. ACM SIGSOFT Software Engineering Notes 31(6):1–9. https://doi.org/10.1145/1218776.1218785

Couto MV, Valente MT, Figueiredo E (2011) Extracting software product lines: A case study using conditional compilation. In: Mens T, Kanellopoulos Y, Winter A (eds) 15th European Conference on Software Maintenance and Reengineering, CSMR 2011, 1-4 March 2011, Oldenburg, Germany, pp 191–200, IEEE Computer Society. https://doi.org/10.1109/CSMR.2011.25

Dubinsky Y, Rubin J, Berger T, Duszynski S, Becker M, Czarnecki K (2013) An exploratory study of cloning in industrial software product lines. In: Cleve A, Ricca F, Cerioli M (eds) 17th European Conference on Software Maintenance and Reengineering, CSMR 2013, Genova, Italy, March 5-8, 2013, pp 25–34, IEEE Computer Society. https://doi.org/10.1109/CSMR.2013.13

da Mota SilveiraNeto PA, doCarmoMachado I, McGregor JD, deAlmeida ES, deLemosMeira SR (2011) A systematic mapping study of software product lines testing. Information & Software Technology 53(5):407–423

do Carmo Machado I, McGregor JD, Cavalcanti YC, deAlmeida ES (2014) On strategies for testing software product lines: A systematic literature review. Information & Software Technology 56(10):1183–1199

do Carmo Machado I, McGregor JD, deAlmeida ES (2012) Strategies for testing products in software product lines. ACM SIGSOFT Software Engineering Notes 37(6):1–8

Engström E, Runeson P (2011) Software product line testing - A systematic mapping study. Information & Software Technology 53(1):2–13

Fischer S, Linsbauer L, Lopez-Herrejon RE, Egyed A (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, pp 391–400, IEEE Computer Society. https://doi.org/10.1109/ICSME.2014.61

Fischer S, Linsbauer L, Lopez-Herrejon RE, Egyed A (2015) The ECCO tool: Extraction and composition for clone-and-own. In: Bertolino A, Canfora G, Elbaum SG (eds) 37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 2, pp 665–668, IEEE Computer Society. https://doi.org/10.1109/ICSE.2015.218

Fischer S, Lopez-Herrejon RE, Egyed A (2018) Towards a fault-detection benchmark for evaluating software product line testing approaches. In: Haddad HM, Wainwright RL, Chbeir R (eds) Proceedings of the 33rd Annual ACM Symposium on Applied Computing, SAC 2018, Pau, France, April 09-13, 2018, pp 2034–2041, ACM. https://doi.org/10.1145/3167132.3167350

Fischer S, Ramler R, Linsbauer L, Egyed A (2019) Automating test reuse for highly configurable software. In: Berger T, Collet P, Duchien L, Fogdal T, Heymans P, Kehrer T, Martinez J, Mazo R, Montalvillo L, Salinesi C, Tërnava X, Thüm T, Ziadi T (eds) Proceedings of the 23rd International Systems and Software Product Line Conference, SPLC 2019, Volume A, Paris, France, September 9-13, 2019, pp 1:1–1:11, ACM. https://doi.org/10.1145/3336294.3336305

Halin A, Nuttinck A, Acher M, Devroey X, Perrouin G, Baudry B (2019) Test them all, is it worth it? assessing configuration sampling on the jhipster web development stack. Empir Softw Eng 24(2):674–717. https://doi.org/10.1007/s10664-018-9635-4

Holling D, Banescu S, Probst M, Petrovska A, Pretschner A (2016) Nequivack: Assessing mutation score confidence. In: 2016 IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops (ICSTW), pp 152–161. https://doi.org/10.1109/ICSTW.2016.29

Kim CHP, Batory DS, Khurshid S (2011) Reducing combinatorics in testing product lines. In: Borba P, Chiba S (eds) Proceedings of the 10th International Conference on Aspect-Oriented Software Development, AOSD 2011, Porto de Galinhas, Brazil, March 21-25, 2011, pp 57–68, ACM. https://doi.org/10.1145/1960275.1960284

Kim CHP, Marinov D, Khurshid S, Batory DS, Souto S, Barros P, d'Amorim M (2013) Splat: lightweight dynamic analysis for reducing combinatorics in testing configurable systems. In: Meyer B, Baresi L, Mezini M (eds) Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013, pp 257–267, ACM. https://doi.org/10.1145/2491411.2491459

Krueger CW (2006) New methods in software product line practice. Commun. ACM 49(12):37–40. https://doi.org/10.1145/1183236.1183262

Krüger J, Al-Hajjaji M, Schulze S, Saake G, Leich T (2018) Towards automated test refactoring for software product lines. In: Berger T, Borba P, Botterweck G, Männistö T, Benavides D, Nadi S, Kehrer T, Rabiser R, Elsner C, Mukelabai M (eds) Proceeedings of the 22nd International Systems and Software Product Line Conference - Volume 1, SPLC 2018, Gothenburg, Sweden, September 10-14, 2018, pp 143–148, ACM. https://doi.org/10.1145/3233027.3233040

Kuhn DR, Wallace DR, Gallo AM (2004) Software fault interactions and implications for software testing. IEEE Trans. Software Eng. 30(6):418–421. https://doi.org/10.1109/TSE.2004.24

Lopez-Herrejon RE, Fischer S, Ramler R, Egyed A (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, pp 1–10,IEEE Computer Society. https://doi.org/10.1109/ICSTW.2015.7107435

Martinez J, Assunção WKG, Ziadi T (2017) Espla: A catalog of extractive spl adoption case studies. In: SPLC, ACM

Martinez J, Ordoñez N, Tërnava X, Ziadi T, Aponte J, Figueiredo E, Valente MT (2018) Feature location benchmark with argouml SPL. In: SPLC, ACM

Moghadam MH, Babamir SM (2014) Mutation score evaluation in terms of object-oriented metrics. In: 2014 4th International Conference on Computer and Knowledge Engineering (ICCKE), pp 775–780. https://doi.org/10.1109/ICCKE.2014.6993419

Mukelabai M, Nesic D, Maro S, Berger T, Steghöfer J-P (2018) Tackling combinatorial explosion: a study of industrial needs and practices for analyzing highly configurable systems. In: Huchard M, Kästner C, Fraser G (eds) Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018, pp 155–166, ACM. https://doi.org/10.1145/3238147.3238201

Nguyen HV, Kästner C, Nguyen TN (2014) Exploring variability-aware execution for testing plugin-based web applications. In: Jalote P, Briand LC, vander Hoek A (eds) 36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014, pp 907–918, ACM. https://doi.org/10.1145/2568225.2568300

Ramler R, Putschögl W (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, pp 122–123, IEEE Computer Society. https://doi.org/10.1109/ICSTW.2013.21

Reisner E, Song C, Ma K-K, Foster JS, Porter AA (2010) Using symbolic evaluation to understand behavior in configurable software systems. In: Kramer J, Bishop J, Devanbu PT, Uchitel S (eds) Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010, pp 445–454, ACM. https://doi.org/10.1145/1806799.1806864

Skoglund M, Runeson P (2004) A case study on regression test suite maintenance in system evolution. In: 20th IEEE International Conference on Software Maintenance, 2004. Proceedings, pp 438–442, IEEE

Souto S, d'Amorim M, Gheyi R (2017) Balancing soundness and efficiency for practical testing of configurable systems. In: Uchitel S, Orso A, Robillard MP (eds) Proceedings of the 39th International

Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017, pp 632–642, IEEE / ACM. https://doi.org/10.1109/ICSE.2017.64

Affiliations

Stefan Fischer[1] · Gabriela Karoline Michelon[2] · Rudolf Ramler[1] · Lukas Linsbauer[3] · Alexander Egyed[2]

Gabriela Karoline Michelon
gabriela.michelon@jku.at

Rudolf Ramler
rudolf.ramler@scch.at

Lukas Linsbauer
l.linsbauer@tu-braunschweig.de

Alexander Egyed
alexander.egyed@jku.at

1 Software Competence Center Hagenberg GmbH, Hagenberg, Austria

2 Institute for Software Systems Engineering, Johannes Kepler University, Linz, Austria

3 Institute of Software Engineering and Automotive Informatics, Technische Universität Braunschweig, Braunschweig, Germany