




Model-based Testing for a Family of Mobile Applications: Industrial Experiences

Stefan Fischer 

Rudolf Ramler 

Software Competence Center
Hagenberg GmbH (SCCH), Austria

Wesley K. G. Assunção 

Alexander Egyed 

Johannes Kepler University Linz
Austria

Christian Gradl

Sebastian Auberger

hello again GmbH, Austria

ABSTRACT

Testing is a fundamental verification activity to produce high-quality software. However, testing is a costly and complex activity. The success of software testing depends on the quality of test cases but finding a good set of test cases is laborious. To make matters worse, when dealing with a family of systems (e.g., variants of a mobile applications), test cases must assure that a diversity of configurations in potentially many variants work as expected. This is the case of hello again GmbH, a company that develops mobile applications for customer loyalty (e.g., discounts, free products, rewards, or insider perks). The company targets several business domains, and currently supports about 700 application variants. Testing such applications including all their variability is a cumbersome task. Even simple test cases designed for one variant most likely cannot be reused for other variants. To support developers at hello again GmbH, we present a solution to employ a model-based testing approach to their family of mobile apps. Model-based testing focuses on automatizing the design and generation of test cases. We present results of applying model-based testing on 27 applications from hello again GmbH and report the challenges and lessons learned for designing a variable test model. Our expected contribution is to support companies and practitioners looking for solutions to test families of software products.





CCS CONCEPTS

• **Computer systems organization** → **Embedded systems**; *Redundancy*; Robotics; • **Networks** → Network reliability.

KEYWORDS

Software Product Lines, Variability Testing, Mobile Testing

ACM Reference Format:

Stefan Fischer , Rudolf Ramler , Wesley K. G. Assunção , Alexander Egyed , Christian Gradl, and Sebastian Auberger. 2023. Model-based Testing for a Family of Mobile Applications: Industrial Experiences. In *Proceedings of 27th International Systems and Software Product Line Conference (SPLC'23)*. ACM, New York, NY, USA, 12 pages. <https://doi.org/XXXXXXX.XXXXXXX>

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](https://permissions.acm.org).

SPLC'23, August 28–September 1, 2023, Tokyo, Japan

© 2023 Association for Computing Machinery.

ACM ISBN 978-1-4503-XXXX-X/18/06...\$15.00

<https://doi.org/XXXXXXX.XXXXXXX>

1 INTRODUCTION

Testing is a fundamental activity to verify the quality of software products. However, testing is also a complex and time-consuming activity, consuming up to 40% of the overall development cost [20]. One challenge for testing software properly is to find a good set of test cases [6]. To support engineers in this task, *model-based testing* automates the design, generation, and optionally the execution of test cases [43]. In model-based testing, a *test model* represents the behavior of a software system under test, which is the basis for the generation of test cases [16].

The use of model-based testing for testing a single software product is widely observed in the literature and in practice [1, 23, 29, 43]. However, companies rarely develop one single product. Instead, they usually develop *families of software products*, in which each variant is customized/configured for a different client of a market segment [8]. Additionally, companies frequently allow users to tailor their system variants to specific scenarios [34]. In the context of model-based testing, dealing with families of software products is challenging, as the test model has to express variability and adapt to the characteristics of the different variants under verification.

Our industry partner, hello again GmbH, experiences that scenario and challenges described above. The company operates in the customer loyalty segment, developing mobile applications (*apps* for short) that offer discounts, free products, rewards, or insider perks. Due to the success of their apps, the company currently supports app variants for about 700 business clients in different domains. However, the testing activity for the family of apps at hello again GmbH is challenging. Because of the variability among the app variants, test cases designed for one app most likely will not execute correctly on other variants, making a direct reuse of test cases between variants infeasible. Existing research has shown promising results in automatically reusing test variants for different configuration [17, 19]. However, no such test variants are available at hello again. Additionally, when test cases fail, developers have a hard time figuring out the variant-specific causes of the problem and which of the variants are affected. Due to these challenges most testing of the applications was done manually and automation only available to test some apps and use cases.

To overcome the challenges of testing a family of mobile apps at hello again GmbH, we proposed a model-based testing solution, which can automatically adapt the model for different variants without any further manual intervention. Model-based testing has shown promising results for testing families of software products [36]. The goal of this paper is to present our solution and report our experiences (i.e., challenges and lessons learned) when applying model-based testing mobile app variants.

Our solution relies on the page object pattern to represent app screens as classes [31], and the OSMO tool to design, generate, and execute the test cases [27]. However, we extended the OSMO tool to be flexible regarding the agility of mobile development, i.e., to cope with new screens in apps that are not yet defined in the test models. Additionally, we adapted OSMO's algorithms to explore the test model, improving the scalability when testing variants with many screens. Another benefit of our solutions is that, while most of the existing tools for mobile testing focus on Android and only few on iOS [42], our solution works for both platforms.

We performed an evaluation of our proposed solution to evaluate if our model works on different variants and especially the generated model parts can be used to navigate arbitrary screen links. Our evaluation relies on a set of 27 app variants developed at hello again GmbH. We computed several metrics of the test models and the variants under test. Then, we answer research questions related to: (i) how variable are the apps of our industry partner, (ii) whether our model-based testing solution works for the given set of app variants, (iii) how our model performs with different strategies for dealing with variability, and (iv) what improvement can be achieved by our adaptations to the OSMO tool.

The results of the evaluation show a high variability in the family of mobile applications regarding the number and type of screens. Nevertheless, we were able to develop a model-based test solution that works for most apps of the entire family (23 out of 27 app variants) without the need of any further adaptation. We were able to generate large parts of the test model from the configuration specification, saving engineers the effort and time needed to implement these parts manually. Finally, we were able to show the usefulness of our adaptations to OSMO's algorithms, which lead to an average improvement of 73.8% in the length of generated tests.

2 BACKGROUND AND RELATED WORK

This section overviews the main concepts used in our work, namely variability, model-based testing, and mobile app testing.

Software Variability. Variability is a property that enables a software system, software asset, or development environment to be configured, customized, or adapted for specific contexts [8]. Such a customization can be performed by developers during the software development. For instance, by using variability, engineers may delay design decisions to later stages of the development process, or to the runtime [24, 41]. Furthermore, end users can benefit from variability to make the software products suit their needs and preferences. Thus, the customization of software products is the basis for creating families of related software systems [7, 26].

Model-Based Testing. Software testing consists of three stages: (i) design/generation of test cases, (ii) execution of such test cases, and (iii) derivation of verdicts [38]. *Model-based testing* (MBT) can support the two first stages, by generating test cases from a model describing the system under test, or by generating and executing test cases simultaneously, in online MBT [44]. MBT has three key elements: (i) the model used to describe the software behavior, (ii) the test-generation algorithm, and (iii) the tools that generate supporting infrastructure for the tests. Different notations to

define a test model have been proposed, such as Activity-based notations (Flowcharts, BPMN, or UML activity diagrams), state-based (or pre/post) notations, transition-based notations (UML State Machines or labeled transition systems), decision tables, and stochastic notations [44]. Moreover, models can be expressed in different representations, like Graphs in GraphWalker¹ or Java code in OSMO².

MBT tools can improve testing practices by increasing the effectiveness of the tests, shortening the testing cycle, and reducing the cost of test generation [12]. However, as for many software engineering activities, adding variability to MBT increases complexity and introduces additional challenges [37]. For a test model to be able to test any arbitrary variant of an SPL, the model has to adapt to the tested variant. Such a model is often referred to as a 150% model in literature [15, 22].

Mobile app testing. A systematic review reports studies on GUI testing of mobile apps [40]. The authors found that the most common testing approach for apps is model-based testing and the majority of approaches focused on functional testing, which is also the focus of our work. Over the past decade, numerous approaches to test mobile applications have been proposed. Most of them focus on automatically exploring the GUI of Android applications with different strategies such as search-based [33] or other systematic approaches [2, 5]. Other approaches explore the GUI and learn models from it, using such models for testing [3, 9, 32]. There are also approaches that generate tests using a record and replay approach [21, 25]. Another study reported experiences of model-based testing of mobile apps in the industry [28]. The authors of this study designed a test model to generate random test sequences that were applied to test apps. Similar to our work, they modelled test steps grouped by the test feature. However, variability and automatically generating model parts and adapting to different configurations was out of the scope of their work. There exist some freely available tools for automated mobile UI testing, like the Android Debug Bridge's UI/Application Exerciser Monkey³ or App Crawler⁴, or Robo Test⁵ by Google. These automated testing tools have the common issues, namely missing a test oracle and finding only limited faults, like for robustness testing or regression testing. Additionally, none of the tools or approaches described above target variability.

MTB for families of software products. There are studies taking into account variability in test cases [36]. For instance, Arrieta et al. [4] present an MBT methodology for highly configurable cyber-physical systems. Their methodology relies on a Feature Model to semi-automatically generate test cases for a Simulink model, which derives 100% of the test architecture. Fischer et al. [18] investigates the difference between the automated reuse of test variants and test model variants. However, their work does not focus on creating a reusable test model, but rather on the reuse and adaptation of existing tests for different configurations.

There are studies focusing on MBT for families of software products. For instance, Reuys et al. [39] propose a technique called ScenTED, which is a model-based and reuse-oriented technique for

¹<http://graphwalker.github.io/>

²<https://github.com/osmo-tool/osmo>

³<https://developer.android.com/studio/test/other-testing-tools/monkey>

⁴<https://developer.android.com/studio/test/other-testing-tools/app-crawler>

⁵<https://firebase.google.com/docs/test-lab/android/robo-ux-test>

test case derivation. Test case scenarios are derived from test models represented by activity diagrams, and the test cases are specified by sequence diagrams. Both diagrams have as prerequisite variation points encoded in them to allow deriving test case scenarios for different configurations. MBT activities such as defining test coverage, generation, and prioritization [30] can be used, for instance, for the quality assurance of a safety-critical system family of products as presented by Classen et al. [10]. They propose a model checking built on the *featured transition systems* [11], which is a compact mathematical model for representing the behavior of the family products. This model is used to represent the behavior of a product when building the assets from which products are derived. Fischer et al. [16] conducted a comparative study of different variability mechanisms, namely preprocessor directives and feature toggles, to encode variability in test models. As a result, the work presents the advantages and disadvantages of each mechanism.

In a recent mapping study, Petry et al. [36] analyzed 44 studies of MBT for SPLs. They found out that most studies take into account variability in models to be tested. For instance, UML models can be annotated with stereotypes that map a relationship between a given feature and its corresponding test models and implementation [14]. Although there are some studies about MBT for system families, to the best of our knowledge there is no study on MBT in the context discussed in this paper.

3 INDUSTRY CONTEXT

Our industry partner, hello again GmbH⁶, develops apps to improve customer loyalty for a wide range of businesses. The apps offer features like different vouchers or events to purchase products or services at discounts. Moreover, the apps can be used to track purchase frequency of customers and reward their loyalty with different bonuses. Currently, they support app variants for about 700 client businesses. To be able to server such a large customer base over different domains, the apps are developed with reuse and variability in mind. The configuration process is supported by a no-code *App builder*. Moreover, the company offers a control center in the background that allows businesses define events and rewards that are offered at a time. Besides that, this *Customer API* allows the businesses to configure parts of their apps. The apps are customized in different ways:

- (1) Selecting features like offering different vouchers/contests, or the login methods (e.g., via email, Google, Facebook, Twitter) using the *App builder* and the *Customer API*. This includes providing screens to access the features and configurations in the background, managing the type of rewards, which influences the interaction with associated screens.
- (2) Available screens in the app, selected from a large set of pre-defined screens, via the *App builder*. Furthermore, the screens themselves are configurable with different options, like the availability of a search feature in a long list of elements. This configuration is done on individual screens, not necessarily equal over all screens in the app variant.
- (3) Navigation between screens via tab bars, drawer menus, icon grid menus, or screens with lists of links which again supports

different list types that may have to be interacted with differently, via the *App builder*.

- (4) Dynamic screens that can be configured by the customer through a backend API to select pre-defined components that can be displayed, via the *App builder* and *Customer API*. These components include a configurable text displayed on the screen, a button to logout from the app, links to other screens, and many more.
- (5) General app design (i.e., look and feel) to fit with customer's corporate design, done manually for each app variant.

This extensive variability is challenging for testing the app variants, since test cases defined for one variant might not work for other variants. Thus, *the goal of our collaboration is to create a test model that can automatically test each app variant created from their family of apps built with the App builder and configuration through the Customer API*. Each app should be tested before deployment, to ensure all screens are reachable and properly displayed, and features are properly supported and working.

4 MODEL-BASED TESTING SOLUTION

The mobile applications are configured using different mechanisms, more static configuration with the no-code *App builder* and more dynamic configuration, including after deployment, using the *Customer API*. From the configuration with the *App builder* we get a *configuration file*, as a specification of the app variant, containing all app screens, components within these screens, and their configurations, like links between screens. One of the main parts of the configuration we are focusing on are *shortcuts*, which is how links to other screens are implemented and configured. We show an example snippet of such a *configuration file* in Listing 1. Each element has a *unique identifier* (see Line 2), as well as a *canonical name* that specifies which screen type the current instance is of. The example in Listing 1 shows part of the configuration of the main navigation mode through the app. In this case, the main navigation is done via a tab bar with buttons at the bottom of the screen that link to the configured screens. The links to other screens are configured in 'relationships'. 'children'. 'data' with the *unique identifier* of the target screens in the *configuration file*.

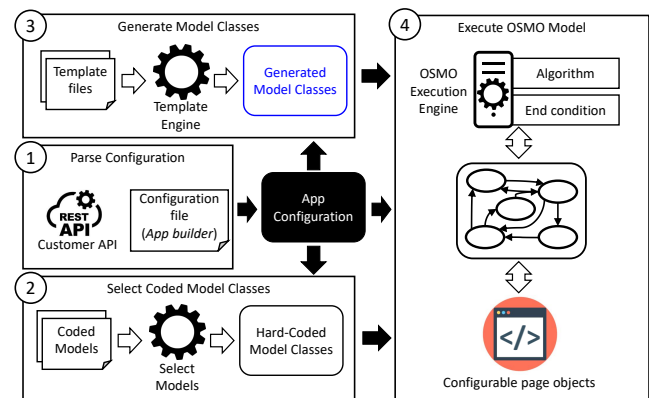


Figure 1: Workflow of generating, selecting, and executing the test model

⁶<https://www.hello-again.com/en/>

Listing 1: Snippet of main navigation configuration.

```

1 { 'type': "core.shortcuts",
2   'id': "5d5d118cd6b0440677556beb",
3   'attributes': {
4     'canonicalName': "navigation.icons",
5     'title': "Main navigation",
6     'screen': "navigation.TabBar",
7     'screens': [{
8       'canonicalType': "navigation.TabBar",
9       'canonicalName': "navigation.TabBar",
10      'settings': { ... }
11    }],
12    'settings': { ... }
13  },
14  'relationships': { 'children': { 'data': [ {
15    'type': "core.shortcuts",
16    'id': "5d5d124fd6b0440677556bed"
17  } ], {
18    'type': "core.shortcuts",
19    'id': "6364c133e94f113b71f79340"
20  } }, ...
21  ] } }
22 },

```

Due to the variety of configuration mechanisms used in the app building process, we likewise included different variability mechanisms in our test model. To test navigation between screens defined with the *configuration file* we generated model parts with a template engine, while for other configuration options we use run-time checks. Figure 1 shows the workflow from generating the model to executing it, calling page objects to interact with the app screens.

4.1 Page objects

For developing our tests, we used the page object pattern, where each screen in the applications is represented by a Java class [31]. This class contains the necessary logic to interact with the associated screen. An example of such a page object is depicted in Listing 2. In the page objects, we utilize Appium⁷ to interact with the mobile applications.

To be able to robustly identify widgets on the screens, we introduced IDs in the app source code. We use these IDs in the page objects to identify and interact with the widgets in any application variant. Moreover, we added IDs to each screen consisting of the *canonical name* and the *unique identifier* from the *configuration file* to enable a clear identification of screen instances. Similarly, for links that are specified in the *configuration file*, we added IDs containing their *unique identifier* in the *configuration file*. The apps should be consistent independent of the OS, and we use a class `Helper` (see Lines 10 and 13 in Listing 2) to retrieve elements from the UI, which takes care of different requests from the OS specific automation frameworks used in Appium. However, for some IDs we still have inconsistencies between Android and iOS, which we deal within the page objects (see Line 3 in Listing 2, where the screen ID is different for the two OS).

Each page object extends an abstract parent page object, which we pass an ID to the constructor that is used to verify if the correct screen is displayed in the app. We also use the page objects to verify whether all the expected elements are shown on the corresponding screen. Therefore, we can define a test oracle directly with the page objects for each screen. For this, the abstract parent class

contains the methods `verify` and `verifyOnce` to check if the screen is displayed correctly. The first method is called every time the screen is reached, for checks like making sure the screen ID is present. The second method is called only the first time we reach the screen in an execution, to avoid time intensive checks over and over, like checking if all links in a long list are present or costly screenshot comparisons. These two methods can perform different checks related to the screen and are part of the test oracle. If a check fails, then an exception is raised to alert an engineer about an issue.

Listing 2: Snippet of a page object implementation.

```

1 @Screen(canonicalName="seblau.auth.EmailLoginScreen")
2 public class EmailLoginPage extends AbstractPage {
3   final String SCREEN_ID = AppiumSetup.PLATFORM == OS.
4     IOS ? "screen.login.email" : "screen.login";
5
6   final String EMAIL_ID = "auth.input.email";
7   final String PASSWORD_ID = "auth.input.password";
8   final String LOGIN_BUTTON_ID = "auth.input.login";
9
10  public EmailLoginPage(AppiumDriver driver) {
11    this(driver, Helper.getResourceIdSelector(
12      SCREEN_ID));
13  }
14  public WebElement getEmailField(){
15    return driver.findElement(Helper.
16      getResourceIdSelector(EMAIL_ID));
17  }
18  ...
19  public AbstractPage login(User user){
20    getEmailField().sendKeys(user.getEmail());
21    getPasswordField().sendKeys(user.getPassword());
22    if(AppConfig.rememberMeCheckboxEnabled())
23      getRememberMeCheckbox().click();
24    getLoginButton().click();
25    return determineStartPage();
26  }
27
28  protected void verify();
29  protected void verifyOnce();
30 }

```

Additionally, we required a mapping of the page objects to the screen they are associated with. For this, we introduced a Java annotation for each page object (see Line 1 in Listing 2) that specifies the canonical name of the screen also used in the *configuration file*. In our example in Listing 2, we implement the interaction with the login screen via email address. The start screen that is displayed after login is specified in the *configuration file*. Therefore, we need to create an instance of the correct page object, corresponding to the start screen, after logging in. We use the Java annotation for the page object classes for this, to identify the correct class for the current screen. Then, we simply create an instance of this class using Java reflection and the page object subsequently verifies if the correct screen is displayed and checks in the `verify` methods are performed.

Finally, the page objects need to be able to interact with the screen in different configurations. This is even more complicated because there can be multiple instances of the same screen type in one app, all using different configurations. Therefore, configuration options are implemented in the page object, using condition execution with `if` statements. For example, Line 19 in Listing 2 checks if the screen in the current variant contains an optional checkbox to stay logged in to the app, even after restarting it, which the test should click. This variability mechanism allows us to dynamically react to

⁷<https://appium.io/>

the configuration on the current screen with one implementation of the page object.

4.2 The OSMO model

For model-based testing of the family of mobile apps, we chose OSMO.² Compared to other MBT tools, it does not require pre-defined system states to be modeled, like a state machine where the test execution transitions lead from one pre-defined state to another. Rather, OSMO uses guard conditions that specify if a test step can be executed at a certain app state (like in Listing 3 for the login via email). This guard can use any arbitrary data to determine if the associated steps can be executed or not. These more dynamic guards, which work more stateless than other MBT approaches, are the main reason for us to choose OSMO over other tools.

Listing 3: Snippet of a OSMO test model.

```

1 public class LoginLogoutModel extends AbstractModel {
2     public LoginLogoutModel(State state) {
3         super(state);
4     }
5
6     @TestStep("loginWithEmail")
7     public void loginWithEmail() {
8         User user = state.getRandomUser();
9         AbstractPage startPage ((EmailLoginPage)state.
10             currentPage()).login(user);
11         state.setCurrentPage(startPage);
12         state.setCurrentUser(user);
13     }
14     @Guard("loginWithEmail")
15     public boolean loginWithEmailGuard() {
16         return state.currentPage() instanceof
17             EmailLoginPage;
18     }
19 }

```

We implement these test steps over different model classes and group them by the executed feature or use case, like test steps to log in. Not only does this help with comprehending the model, but also allows us to exclude models for features that are not available in the tested variant, like logging in with Facebook. OSMO requires us to instantiate the model classes and pass them in a factory class. We can therefore simply avoid instantiating the classes not relevant for the current variant.

During test execution, OSMO checks the guards of all available test steps, and an execution algorithm selects the next test step to execute, from the ones with guards that returned true. To make these executions reproducible, we use seeds for all random parts of the test executions that can be configured to make all decisions the same way again. Moreover, we record all test steps that are executed in a test execution in a file, which we can replay again and reproduce the same test scenario and for debugging.

4.3 Generating model parts

The *configuration file* specifies the links between screens. For instance, screens which are reached via the tab bar at the bottom of the screen can be freely configured to any screen in the app (see Listing 1). Moreover, we can adjust the number of links on a screen. To test this unrestricted configurability, we opted to generate model parts for this behavior, based on the *configuration file*.

The test model generation process is depicted in Figure 1. It uses the *configuration file* and locates the configuration of certain

screen types and components from there. From this configuration, we generate unique test step names based on the titles of the source and target screens. Additionally, the generated test steps include, hard coded, the unique screen ID from the *configuration file*. We use these IDs to determine the screen that has to be shown after executing the test step and the corresponding page object that is loaded.

4.4 Dynamic screen components

The app variants contain configurable screens that can be customized by the businesses themselves. This is done via a backend *Customer API* that allows to select and configure components that should be displayed on the screens. Some of such components are customizable text, a button to logout, a list of rewards for customer loyalty, a list of links to other screen, and many more. When the app reaches such a configurable screen, our test model loads the corresponding page object. This page object dynamically checks which components are configured for the current screen. Then, the page object loads corresponding component objects that implement the interaction with the specific components, similar to a page object for a screen, as shown in Listing 4. The correct component object is selected by a component name specified in the class with a Java annotation, as in the canonical names of screens for page objects. Like the page objects these component objects contain a method `verify` that can be used to implement specific checks for this component and is used as part of the test oracle.

Listing 4: Snippet of a component object implementation.

```

1 @ScreenComponent(name = "TextComponent")
2 public class TextComponent extends Component{
3     public TextComponent(AppiumDriver driver, String id,
4         GenericScreen screen) {
5         super(driver, id, screen);
6     }
7
8     @Override
9     public void verify(GenericScreen screen) {
10         // Compare the displayed and configured text.
11     }
12 }

```

4.5 Exploration of un-modeled parts

The development of the app family is ongoing and screens are being added. Since for new screens or very rare screens we may not have page objects developed yet, the test model could run into a dead-end from where it can not continue. If we reach such an unknown screen, no test step is defined to deal with it. To circumvent this, we added an automatic exploration mode that tries to get back to app parts that are modeled. This mode is entered by a test step with a guard method that checks if we could not recognize the current screen and if no other test step is available. During this automated exploration, elements on the screen are clicked randomly and the device specific navigate back functionality is tried randomly. After each such action, we try to identify the screen we are currently on, by checking if any of the screen IDs from the *configuration file* are present. If there is a page object implemented for this screen, identified by the Java class annotation, the automated exploration mode stops execution and the test step ends, which means OSMO can continue with other test steps available for the current screen.

For all test executions we implemented a configurable timeout that will end the test with an error message. Therefore, if the automated exploration does not find a way to enter a modeled screen the test will timeout and cannot get stuck in endless loops.

4.6 Other OSMO adaptations

We made some additional adaptations to OSMO to better meet our test goal and requirements.

Custom end conditions. OSMO allows setting end conditions for testing, which end the test if it is fulfilled. One goal for our tests is to reach all screens in any app variants through automated tests. The standard end conditions of OSMO could not guarantee this. Therefore, we developed our own end condition that checks if all screens reachable with a shortcut were reached in the current test execution. To recognize the screen we are currently on, we use the current page object and the screen IDs that were added in the apps. We implemented the identifier of the current screen as a OSMO coverage value, which can then easily be used in any location within our test runner implementation. To avoid endless test runs, if a screen cannot be reached, the end condition keeps track of the test step that can be executed from any screen and checks if all of them have been executed in the current test execution. If all test steps were executed and still screens have not been reached, the end condition fails the test with a corresponding error message.

Custom exploration algorithm. OSMO provides different algorithms to traverse the model and generate tests from it. These algorithms are different combinations of random traversals, some using weights of specific test steps and some try to balance which test steps are taken to avoid always executing the same ones. However, in our experiments, we quickly saw that the standard algorithms provided by OSMO took a long time to explore the entire app for some variants. To improve this, we developed our own algorithm better suited for our test goals. Our algorithm keeps track of the test steps already executed and avoids taking them again as long as other steps are available. Additionally, it remembers which test steps are available on which visited screen and a graph that stores from which screen we navigated to what other screen using which test step. It then uses this information to search for not yet executed test steps if no new step is available on the current screen. The algorithm computes the shortest path to all missing (i.e., not executed in the test) test steps using *Dijkstra's Shortest Path Algorithm* [13]. It then executes the test steps along the shortest path found to reach the missing test step.

5 EXPERIMENTS

To verify if our test model can be used to automatically test a family of mobile apps, we performed some experiments executing the model on different apps. This section describes the experiments.

5.1 Research goal

To assess the quality and usefulness of our solution, we applied it on a set of app variants from our industry partner. Figure 2 depicts the navigation through an app variant. We analyze whether the generated model parts can be used to navigate arbitrary screen links,

like shown in Figure 2, and how often the automated exploration is used. Therefore, we distinguish the three execution modes:

- (1) **Coded:** hard coded pre-defined test steps that are selected by class for inclusion in the variants. These steps are depicted as black arrows in Figure 2, in this case the login via email.
- (2) **Generated:** generated test steps from the *configuration file* that load the page objects dynamically, depicted as blue arrows in Figure 2.
- (3) **Auto Exploration:** the automated random exploration mode that recovers the model if it reaches an unknown / not modelled screen, depicted as red arrows in Figure 2.

Moreover, we want to investigate how well our implementation works and our algorithm compared to a default algorithm of OSMO. To investigate this we formulated the following research questions:

- **RQ0: How much do the different apps vary from one another?** This RQ aims at giving us a better understanding of the differences our test model needs to cover on app variants.
- **RQ1: Does our solution work for all app variants?** The goal for our collaboration is to create a test model that can be used to test any app variant. Therefore, we want to verify this by testing a set of app variants with different kinds of variability.
- **RQ2: How are the different execution modes distributed?** We are interested in knowing how much of the test execution uses *Coded* or *Generated* test steps in the model. This helps to show the variability in the different variants, since the generated steps are specific for an app variant. *Auto Exploration* is only used if the model reaches a screen that we did not model yet. Therefore, we are interested in how often that still happens, and how much is still missing from the model, and how long it takes to get back to the modeled part of the app variants.
- **RQ3: How much do our adaptations to OSMO's algorithms improve the test execution?** The reason we developed our own algorithm to execute the test model was that the standard algorithms in OSMO took too much time to test a single variant, which is an issue when testing many variants. With this RQ, we want to verify how much of an improvement we can get in time and screens visited with our algorithms.

5.2 Method

We performed experiments on 27 Android app variants from different domains, with different types of navigation between screens and variability in the available features. The set of variants was selected by our industry partner.

We executed OSMO with two different algorithms: the *standard OSMO weighted balancing algorithm* that randomly executes the model but takes weights and previously executed test steps into account to try to avoid executing the same test steps over and over again, and the *new exploration algorithm* we implemented (see Section 4.6). The OSMO algorithm was selected because it balances the execution to weigh not executed test steps higher and is therefore better suited to explore the entire model than other algorithms. Additionally, we performed experiments with two end conditions: *our own developed condition* to reach all screens according to the configuration file, and an end condition with a *fixed test length* of test steps. To answer our RQs, we used different data. For **RQ0**, the variability among variants. For **RQ1**, data from

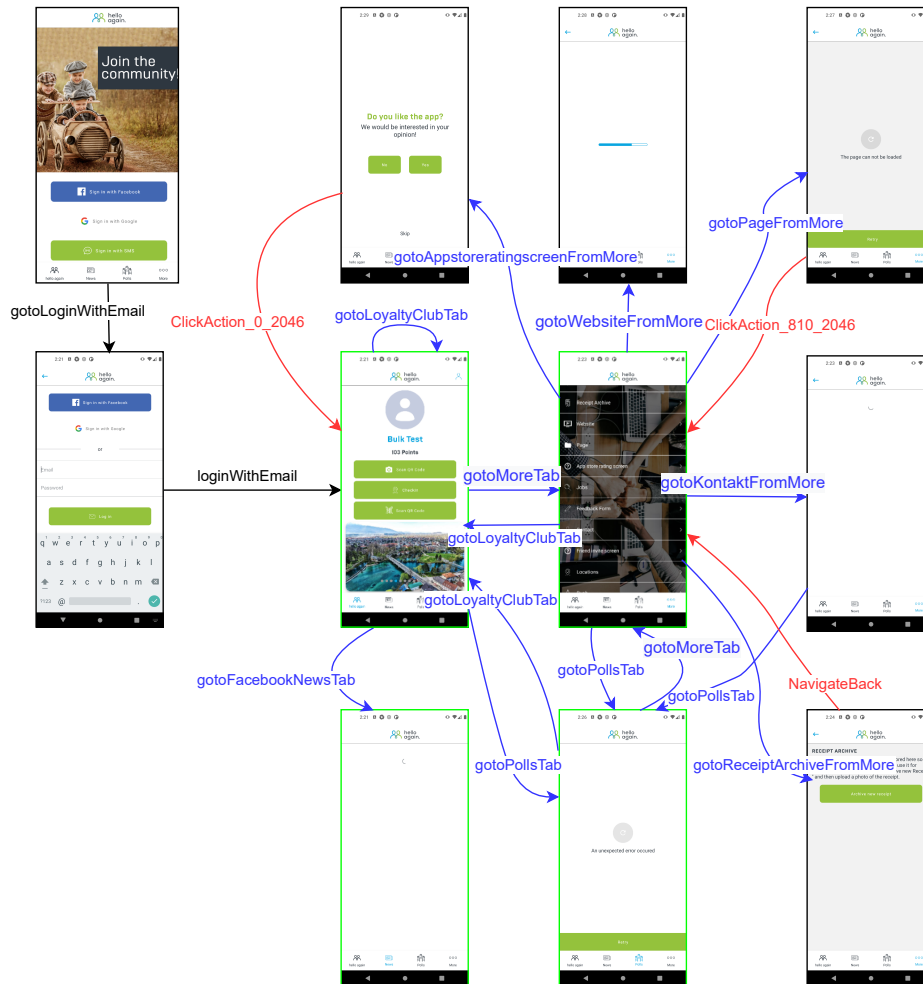


Figure 2: Example of testing an app using coded (black), generated (blue), auto exploration (red)

all experiments we performed. For **RQ2**, the OSMO’s weighted balancing algorithm with a fixed length of 100 test steps, with the automated exploration mode turned on. For **RQ3**, we executed both algorithms with both end condition, to see how long the different algorithms take to reach all screens. We also collected how many of the screens are reached by the algorithms for a fixed test length of 30 steps. For this experiment, we turned the automated exploration mode off, because it is not influenced by the algorithms and just adds additional noise for comparing the algorithms. Finally, for all experiments, we limited the time for each test execution with our model to two hours, after which the test ends with a failure.

The experiments were executed on a Dell Latitude 5400 laptop with an Intel CoreTM i7-8665U processor (1.9 GHz, 4 cores), 32GB of RAM, SSD storage, and running the Windows 10 operating system. The app variants were executed on the Android Studio emulator version 32.1, on a Pixel 5 device running Android 11.

5.3 Metrics

From the experimental settings described above, we collected the following metrics from the app variants:

- *Number of shortcuts* in an app variant, to give an idea about the size of the apps (*RQ0*).
- *Number of screen links* in an app variant that can link to any screens and lead to generated test steps (*RQ0*).
- *Number of generated test steps* for a specific app variant (*RQ0*).

Moreover, we compute metrics from executing the test model against specific app variants:

- *Number of all executed test steps* in the entire test model (*RQ3*).
- *Number of executed hard coded test steps* in our test model (*RQ2*).
- *Number of executed steps among the generated ones* for each specific app of the family (*RQ2*).
- *Number of automated explorations* that were triggered during test execution (*RQ2*).
- *Number of executed interactions* with the automated exploration mode to recover to a state the model continue executing (*RQ2*).
- *Number of screens reached* in an app variant (*RQ3*).

Table 1: Tested mobile apps and metrics

App	Shortcuts	Link screens	Generated Test Steps	Main navigation
1	10	0	8	Icon grid
2	14	2	13	Drawer menu
3	9	0	8	Drawer menu
4	88	5	87	Drawer menu
5	17	1	15	Tab bar
6	9	1	8	Tab bar
7	11	1	9	Tab bar
8	10	1	9	Tab bar
9	13	1	11	Tab bar
10	8	1	7	Tab bar
11	13	2	12	Tab bar
12	9	1	8	Tab bar
13	14	2	12	Tab bar
14	10	1	9	Tab bar
15	8	1	7	Tab bar
16	17	2	16	Tab bar
17	13	1	11	Tab bar
18	12	2	10	Tab bar
19	12	1	11	Tab bar
20	12	2	11	Tab bar
21	17	1	12	Tab bar
22	6	5	17	Tab bar
23	17	2	14	Tab bar
24	17	2	11	Tab bar
25	7	2	13	Tab bar
26	17	2	13	Tab bar
27	14	1	10	Tab bar

5.4 Results

In this section, we present the results from our experiments, by research question.

RQ0: How much do the different apps vary from one another?

Table 1 lists the 27 app variant used in our experiment, with an app number in the first column for easier reference. This table also has the number of *shortcuts* in the app configuration, the number of screens with links to other screens via a shortcut, the number of generated test steps in our approach, and the main navigation type used by the app. The first observation we can make is that most app variants use the tab bar as the main navigation mode between screens. Moreover, we see that there is a great variety in the number of link screens and shortcuts between screens in the different apps, which results in a varying number of generated test steps. Our model currently contains 14 hard coded test steps, including the test step to enter the automated exploration mode.

Answering RQ0: The app variants have a high degree of variability, but there are commonalities in the core features that most apps support.

RQ1: Does our solution work for all app variants? In our experiments, all but one variant (App 5) could be executed with our test model. App 5 could not be executed because it has a custom login screen, only for this specific variant, that is not supported in our test model. For another two variants (Apps 22 and 25) we can run our test model, but both have some IDs missing on their link screens. These screens are of a subtype not encountered before the experiments. Thus, the source code was not yet extended to include the required IDs causing the tests to fail, because they could not access the links. Finally, we discovered issues with another app (App 1) that could not be executed with the automated exploration mode, because it was not able to re-enter the normal model execution. This happened because of issues how the automated exploration mode tries to recognize the currently displayed screen in combination with the different main navigation method of App 1 (i.e., the icon grid). With this method of navigation, the code for several screens gets stacked during navigation and only the top one is visible on the screen. However, this means that several of the IDs we use to identify the screens are available at the same time and the automated exploration mode can not recognize the actual screen as for the apps with the other navigation methods.

As described above, we were able to execute most of the 27 apps, except for those with missing IDs and an unsupported login screen. The remaining 23 apps could be executed as intended. However, some flakiness [35] remains in the current test model, which forced us to re-run the tests for some apps. The reason was issues with app permissions, like reading the current location, that sometimes caused a dialog to open in certain screens. Even though we configured Appium to automatically grant all permissions, this still kept happening for some apps in some executions. We were not able to reliably reproduce this issue. To deal with it, we coded steps to recognize the dialog and acknowledge it in the implementation of the affected page objects.

Answering RQ1: For most app variants (23 out of 27) our solution (i.e., test model) works as intended. However, there remain some open issues that we need to address in our ongoing work.

RQ2: How are the different execution modes distributed? We show the distribution of test steps for the three different modes in Figure 3. We excluded variant App 1 and App 5 because of the different main navigation method and the unsupported login screen. Finally, Apps 22 and 25 have missing IDs in their link screens and therefore fail before the 100 test steps could be executed.

In Figure 3 we see that most of the executed test steps are generated steps. For the variants that could execute all 100 test steps, on average, 71.5% of the steps were generated ones, 23.9% hard coded steps, and 4.6% ones that entered the automated exploration mode.

Figure 4 depicts the number of interactions the automated exploration mode required to go to a state that the test model could be executed on again, for every time the mode was entered. Apps 14, 22 and 25 never entered the automated exploration mode and are therefore omitted from Figure 4. For most occurrences, namely 53 times, a modeled screen was reached after just one automated interaction, by either navigating back through the android back button, by clicking on the in-app back button or by clicking on other

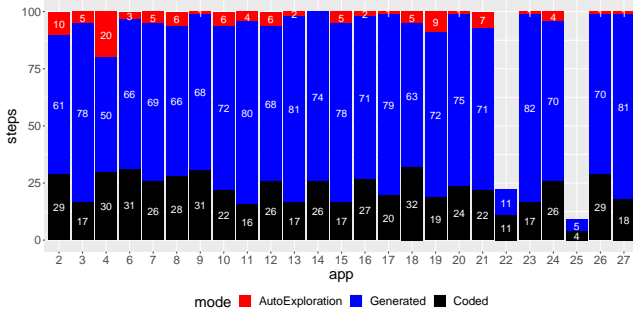


Figure 3: Distribution of execution modes

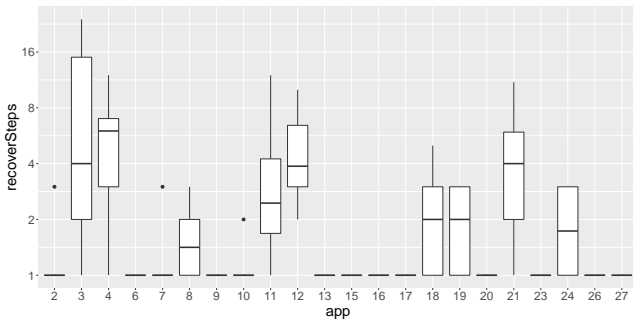


Figure 4: Number of interactions required to recover to a state that the model can keep executing

screen elements (a tab in the tab bar in most cases). On average, the automated exploration mode needed 3.1 interactions with the app to reach a modeled screen and continue with the normal model execution.

Answering RQ2: In our current test model, the majority of the test steps executed are the ones generated from the configuration file, followed by hard-code ones. The automated exploration was used the least, on average, and in most cases recovered in only one or very few interactions.

RQ3: How much do our adaptations to OSMO’s algorithms improve the test execution? In the first experiment to answer this research question, we executed the two algorithms with the end condition to reach all screens that are linked to with a shortcut. Figure 5 presents the number of test steps required to fulfill the end condition, with the test steps on the y-axis in logarithmic scale. We also excluded App 5, as the number of steps is zero. The results show that in general our exploration algorithm reached all relevant screens in fewer test steps than the standard OSMO algorithm. There are only two exceptions: App 10, where the weighted balancing algorithms randomly generated a test with fewer steps that fulfills our end condition; and App 25, where the weighted balancing algorithm reached the screen with the missing IDs quicker and then failed because of them. Moreover, for App 4 the weighted balancing algorithm took more than two hours to generate the

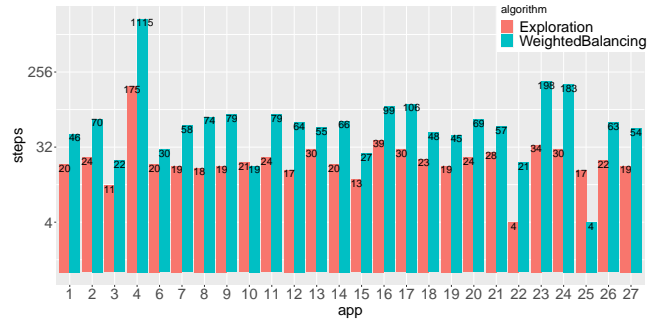


Figure 5: Number of test steps required to reach all screens

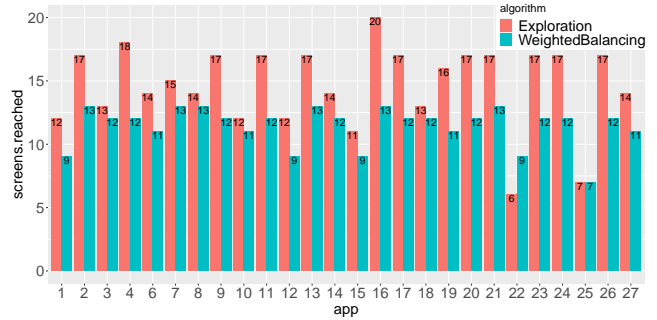


Figure 6: Number of screens reached in 30 test steps

test and therefore failed before reaching the end condition. App 4 has the highest number of shortcuts and includes five different link screens, which are more difficult to reach in a random testing scenario. In such a case, the advantage of our algorithm prevails.

For the second experiment, we used the end condition to include a fixed number of 30 test steps in the generated tests. In this scenario, the number of test steps is the same for both algorithms, but we measure the number of screens that are reached with only a limited number of steps. Because our exploration algorithm tries to exclude test steps that have previously already been executed, we would assume that it reaches more screens in the same number of test steps. Figure 6 shows the number of screens reached. For all app variants but two, the exploration algorithm reached more screens. The two apps where this is not the case are Apps 22 and 25, which contain the link screens with the missing IDs.

Answering RQ3: From our experiments, we found that with our algorithm we could improve the test length, to reach all screens, by 73.8% on average. In other words, our algorithm can reduce the number of test steps required to meet the coverage criteria. Moreover, with a fixed test length of 30 steps, our algorithm reaches on average 28.3% more screens.

6 DISCUSSION

In this section we discuss the challenges we faced in our work on testing the family of mobile app variants and present lessons learned during our work and the analysis of our experiments.

6.1 Challenges

During the conduction of this work, we faced several challenges related to the characteristics of the system under test, limitations of our approach, or due to shortcomings of the used tooling. These challenges are described next.

Huge variability space. The app family is highly configurable. Some of the variability comes from features that can be optionally selected for a variant. There exist many configuration options for the features themselves and how they need to be interacted with. Additionally, to this already very large configuration space, the apps can contain nearly unconstrained links between screens, which has to be dealt with when testing the apps.

Making the app family testable. At the start of our collaboration, the app variants did not include many IDs to identify screens or their contained elements and widgets. Appium allows us to identify elements with multiple attributes, like x-path or text. However, the apps support switching between languages, which make the identification via text impossible. Moreover, there are parts where the text can be configured by the customer business. Identifying elements using x-path has similar issues when elements are not fixed to a location. In our case the elements' location changes between variants and some can be configured by the customer to be in a different location. A lot of effort was invested to make all elements required for testing identifiable. We identified all required elements and added IDs to them in the app source code. And still every time we extend the model to new screens or features we have to keep adding IDs, which slows down the rollout of a new version of the test model that then only works on the newest builds that contain all the IDs.

Keeping track of the app state. To interact with the app variants correctly, we need to keep their current state in mind. Some parts of the app state, like currently available rewards and vouchers, can be retrieved through an API during testing. Other parts, like the user currently logged in, have to be stored during the test execution. We keep track of the current screen by storing the page object associated with the screen. However, when using the automated exploration of the app and subsequently resuming with the normal test model execution, identifying the current screens is challenging. We use IDs that identify the current screen instance and check if we can generate a page object instance for it. However, we found that there are still issues with this approach for app variants that have a very large number of screens since the identification of the current screen can take very long, because the number of IDs that have to be checked. More issues can come from apps that navigate to different screens by overlaying the new screen over the last one, as happened in our experiments with App 1. In such a case, our current approach is not able to identify the current screen and will fail to create the correct page object, which can fail the test.

Testing apps for different OS with the same model. Despite not having apps for different operating systems (OSs) in our experiments, we had this discussion with developers at our industry partner. The challenge comes from the need of the same apps being available for different mobile OS. We tried to restrict differences in the interaction between the different OS to our Helper class that

selects elements with the OS specific UI automation framework used by Appium. This requires that the IDs in the apps are consisted between OS and the apps behave the same independent of the OS. However, to this day we were not able to fully realize this. There are remaining differences in the IDs for elements that we have to deal with in the page objects, which could not be consistently be propagated in the app build.

Moreover, there are distinct challenges when testing on different OS. For instance, for iOS, Appium requires passing the nesting depth of elements in the screen that are loaded for interaction by the automation framework. If we set this nesting depth too low, we can not access certain elements. If we set it too high, the app execution becomes very slow and tests run longer. On the other hand, when testing with Android, we encountered issues with app permissions. For instance, some screens require permissions to access the phone location. Appium allows passing arguments to grant all permissions at initialization. Nevertheless, in our experiments, the dialog to ask for the permissions would randomly appear and lead to tests failing that had to be re-executed.

Appium field annotations could not be used. Appium provides Java annotations that can be used for fields in the page object classes to automatically initialize them to address certain elements on the screen. These are usually recommended to use for the implementation of page objects. However, due to the variability of the app family, we could not use these annotations, because some elements might only be available for some variants. If we test a variant that does not have the element and use these field annotations, the test would fail as soon as we instantiate the page object. To support the variability of the app screens, we loaded elements during the execution of certain interactions with the app and made their execution conditional with if or switch statements.

6.2 Lessons Learned

Do not rely on one variability mechanism. At the beginning of the project, we investigated the best way to model our variable test model, and documented our thoughts in our previous work [16]. We first wanted to model the configurable links between screens, with a separate test step for each link. To achieve this, we had to generate the test steps from the configuration files and templates, which corresponds to the variability mechanism of conditional compilation. However, we then dove into the configurability of the individual screens and learned that there can be instances of the same screen with different configurations within a variant. Therefore, we implemented the page objects to be able to correspond to the different configurations dynamically with conditional execution. For other parts, like adding model classes if a feature is present for a variant, we also use conditional execution, because it is better supported by existing programming tools.

Tweaking the MBT algorithm can pay off. We found that creating our own algorithm to decide the next test step improved the test execution and explores the entire test model in less time than the standard OSMO algorithms. To achieve this, we had to provide the algorithm with additional data to represent the current app state, which the standard algorithms do not have. Thus, the advantages of our algorithm might be limited to our specific model design. For

instance, if instead of generating a test step for each link in the tab bar, we simply used one test step that randomly clicks on a tab and loads the corresponding page object, we could not learn the following state and the advantage our algorithm brings might disappear. Therefore, for different cases, other algorithms might have more advantages, but nonetheless we believe that considering optimizations to the algorithms can have a beneficial impact in test generation.

Maintain separation of concerns for verifying app parts. We developed model classes to contain test steps for the same features or use cases. This helps to better understand the model and the tests that can be generated for specific use cases. Moreover, we kept the tasks of verifying the screens and interacting with them directly in the page objects. We believe this design support a better comprehension of the model and helps to avoid side effect between configurations.

Invest in testability from the start of the development. As mentioned above, we had to invest a lot of effort into making the apps testable by introducing IDs to screens and elements within them. Therefore, companies should invest in these measure from the start. This is a known issues in practice, but nonetheless we saw again in this collaboration that this well-known best practice was not applied. In the future, when adding new feature and screens, such IDs should be introduced already in new developments and testability should be kept in mind. This can also help in reducing inconsistencies between Android and iOS.

7 CONCLUSIONS AND FUTURE WORK

We presented in this work a solution for testing a family of mobile applications using a model-based testing approach. This solution emerged from our collaboration with hello again GmbH. To adhere to the configuration of the current app variants in general, our test model can be adjusted in several ways. We select required model parts for available features, generate model parts from a configuration specification file, and dynamically adjust the interactions with the application screens with configurable page objects. To assess how well our solution works on the app family, we applied it to a set of 27 app variants provided by hello again GmbH. Our findings showed that the model works in most cases, with some remaining issues that we need to further improve in our future work. Moreover, we investigate how often which model parts are used during execution, and the effect a custom algorithm has on the test execution, showing a reasonable improvement in our case.

The model-based test approach described in this paper is in use to test production apps at our partner company. To date about 300 apps have been tested with it and the manual effort for testing could be reduced about 50% according to our industry partner. As part of our future work, we could do a more in depth evaluation of the results from applying the testing approach at the company. Moreover, we plan to run our model on more variants on device farms with many different devices and device configurations. This will allow us to further evaluate the impact of different devices and other aspects in test execution. We are also interested in investigating the use of the graphs stored from our algorithm for offline computation of test sequences, to reach certain test goals, such as navigating to

all screens. This could speed up the generation and execution of tests. However, the problem is very similar to the traveling sales person problem, which is known to be \mathcal{NP} -complete. Another item for our future work is to support the maintenance of the test model, by using data from operation and identify parts that are missing from the model or where to model execution deviates from real operation. Finally, we are interested in increasing the level of automation for our testing process and increase the proportion of the model that can be covered with automated exploration, with providing predefined test sequences only for certain parts (e.g., login to the app), and to provide an oracle for more difficult to verify parts.

ACKNOWLEDGMENTS

The research reported in this paper has been supported by BMK, BMAW, and the State of Upper Austria in the frame of the SCCH competence center INTEGRATE (FFG grant 892418) part of the FFG COMET Competence Centers for Excellent Technologies Programme; and by the Austrian Science Fund (FWF) grant P31989-N31.

REFERENCES

- [1] Tanwir Ahmad, Junaid Iqbal, Adnan Ashraf, Dragos Truscan, and Ivan Porres. 2019. Model-based testing using UML activity diagrams: A systematic mapping study. *Computer Science Review* 33 (2019), 98–112.
- [2] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Salvatore De Carmine, and Atif M. Memon. 2012. Using GUI ripping for automated testing of Android applications. In *IEEE/ACM International Conference on Automated Software Engineering, ASE'12, Essen, Germany, September 3-7, 2012*. ACM, 258–261. <https://doi.org/10.1145/2351676.2351717>
- [3] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Bryan Dzung Ta, and Atif M. Memon. 2015. MobiGUITAR: Automated Model-Based Testing of Mobile Apps. *IEEE Softw.* 32, 5 (2015), 53–59. <https://doi.org/10.1109/MS.2014.55>
- [4] Aitor Arrieta, Goiuria Sagardui, and Leire Etxebarria. 2014. A model-based testing methodology for the systematic validation of highly configurable cyber-physical systems. In *6th International Conference on Advances in System Testing and Validation Lifecycle*. IARIA XPS Press, 66–72.
- [5] Tanzirul Azim and Iulian Neamtiu. 2013. Targeted and depth-first exploration for systematic testing of android apps. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*. ACM, 641–660. <https://doi.org/10.1145/2509136.2509549>
- [6] Antonia Bertolino. 2007. Software testing research: Achievements, challenges, dreams. In *Future of Software Engineering (FOSE'07)*. IEEE, 85–103.
- [7] J. Bosch, R. Capilla, and R. Hilliard. 2015. Trends in Systems and Software Variability. *IEEE Software* 32, 3 (2015), 44–51.
- [8] Rafael Capilla, J. Bosch, and K. C. Kang. 2013. *Systems and Software Variability Management: Concepts, Tools and Experiences*. Springer.
- [9] Wontae Choi, George C. Necula, and Koushik Sen. 2013. Guided GUI testing of android apps with minimal restart and approximate learning. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*. ACM, 623–640. <https://doi.org/10.1145/2509136.2509552>
- [10] Andreas Classen, Maxime Cordy, Pierre-Yves Schobbens, Patrick Heymans, Axel Legay, and Jean-François Raskin. 2013. Featured Transition Systems: Foundations for Verifying Variability-Intensive Systems and Their Application to LTL Model Checking. *IEEE Transactions on Software Engineering* 39, 8 (2013), 1069–1089. <https://doi.org/10.1109/TSE.2012.86>
- [11] Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, Axel Legay, and Jean-François Raskin. 2010. Model checking lots of systems: efficient verification of temporal properties in software product lines. In *2010 ACM/IEEE 32nd International Conference on Software Engineering*, Vol. 1. 335–344. <https://doi.org/10.1145/1806799.1806850>
- [12] S.R. Dalal, A. Jain, N. Karunanithi, J.M. Leaton, C.M. Lott, G.C. Patton, and B.M. Horowitz. 1999. Model-based testing in practice. In *International Conference on Software Engineering*. 285–294. <https://doi.org/10.1145/302405.302640>
- [13] Edsger W Dijkstra. 2022. A note on two problems in connexion with graphs. In *Edsger Wybe Dijkstra: His Life, Work, and Legacy*. 287–290.

- [14] Ivan do Carmo Machado. 2014. *Fault model-based variability testing*. Ph.D. Dissertation. Federal University of Bahia, Salvador, Brazil. <http://repositorio.ufba.br/ri/handle/ri/22826>
- [15] Christian Dziobek and Jens Weiland. 2009. Variantenmodellierung und -konfiguration eingebetteter automotive Software mit Simulink. In *Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme V, Schloss Dagstuhl, Germany, 2009, Tagungsband Modellbasierte Entwicklung eingebetteter Systeme (Informatik-Bericht, Vol. 2009-01)*. TU Braunschweig, Institut für Software Systems Engineering, 36–45. http://www.sse-tubs.de/mbees-dagstuhl/MBEES2009_Proceedings_online_small.pdf
- [16] Stefan Fischer, Gabriela Karoline Michelon, Wesley K. G. Assunção, Rudolf Ramler, and Alexander Egyed. 2023. Designing a Test Model for a Configurable System: An Exploratory Study of Preprocessor Directives and Feature Toggles. In *17th International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, 31–39. <https://doi.org/10.1145/3571788.3571795>
- [17] Stefan Fischer, Gabriela Karoline Michelon, Rudolf Ramler, Lukas Linsbauer, and Alexander Egyed. 2020. Automated test reuse for highly configurable software. *Empir. Softw. Eng.* 25, 6 (2020), 5295–5332. <https://doi.org/10.1007/s10664-020-09884-x>
- [18] Stefan Fischer, Rudolf Ramler, and Lukas Linsbauer. 2021. Comparing Automated Reuse of Scripted Tests and Model-Based Tests for Configurable Software. In *28th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 421–430. <https://doi.org/10.1109/APSEC53868.2021.00049>
- [19] Stefan Fischer, Rudolf Ramler, Lukas Linsbauer, and Alexander Egyed. 2019. Automating test reuse for highly configurable software. In *Proceedings of the 23rd International Systems and Software Product Line Conference, SPLC 2019, Volume A, Paris, France, September 9-13, 2019*. ACM, 1:1–1:11. <https://doi.org/10.1145/3336294.3336305>
- [20] Wahid Garousi and Junji Zhi. 2013. A survey of software testing practices in Canada. *Journal of Systems and Software* 86, 5 (2013), 1354–1376.
- [21] Lorenzo Gomez, Iulian Neamtii, Tanzirul Azim, and Todd D. Millstein. 2013. RERAN: timing- and touch-sensitive record and replay for Android. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*. IEEE Computer Society, 72–81. <https://doi.org/10.1109/ICSE.2013.6606553>
- [22] Hans Grönniger, Holger Krahn, Claas Pinkernell, and Bernhard Rumpe. 2014. Modeling Variants of Automotive Systems using Views. *CoRR* abs/1409.6629 (2014). [arXiv:1409.6629](http://arxiv.org/abs/1409.6629) <http://arxiv.org/abs/1409.6629>
- [23] Havva Gulay Gurbuz and Bedir Tekinerdogan. 2018. Model-based testing for software safety: a systematic mapping study. *Software Quality Journal* 26, 4 (2018), 1327–1372.
- [24] Günter Halmans and Klaus Pohl. 2003. Communicating the Variability of a Software-Product Family to Customers. *Software and System Modeling* 2, 1 (2003), 15–36.
- [25] Yongjian Hu, Tanzirul Azim, and Iulian Neamtii. 2015. Versatile yet lightweight record-and-replay for Android. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*. ACM, 349–366. <https://doi.org/10.1145/2814270.2814320>
- [26] Kyo Kang, Sholom Cohen, James Hess, William Novak, and A. Peterson. 1990. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Technical Report CMU/SEI-90-TR-021. Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA. <http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=11231>
- [27] Teemu Kanstrén and Olli-Pekka Puolitaival. 2012. Using Built-In Domain-Specific Modeling Support to Guide Model-Based Test Generation. In *Proceedings 7th Workshop on Model-Based Testing, MBT 2012, Tallinn, Estonia, 25 March 2012 (EPTCS, Vol. 80)*. 58–72. <https://doi.org/10.4204/EPTCS.80.5>
- [28] Stefan Karlsson, Adnan Causevic, Daniel Sundmark, and Mårten Larsson. 2021. Model-based Automated Testing of Mobile Applications: An Industrial Case Study. In *14th IEEE International Conference on Software Testing, Verification and Validation Workshops, ICST Workshops 2021, Porto de Galinhas, Brazil, April 12-16, 2021*. IEEE, 130–137. <https://doi.org/10.1109/ICSTW52544.2021.00033>
- [29] Stefan Kriebel, Matthias Markthaler, Karin Samira Salman, Timo Greifenberg, Steffen Hillemecher, Bernhard Rumpe, Christoph Schulze, Andreas Wortmann, Philipp Orth, and Johannes Richenhagen. 2018. Improving model-based testing in automotive software engineering. In *40th International Conference on Software Engineering: Software Engineering in Practice*. 172–180.
- [30] Axel Legay, Gilles Perrouin, Xavier Devroey, Maxime Cordy, Pierre-Yves Schobbens, and Patrick Heymans. 2017. On Featured Transition Systems. In *SOFSEM 2017: Theory and Practice of Computer Science*. Springer International Publishing, Cham, 453–463.
- [31] Maurizio Leotta, Matteo Biagiola, Filippo Ricca, Mariano Ceccato, and Paolo Tonella. 2020. A Family of Experiments to Assess the Impact of Page Object Testing in Web Test Suite Development. In *13th IEEE International Conference on Software Testing, Validation and Verification, ICST 2020, Porto, Portugal, October 24-28, 2020*. IEEE, 263–273. <https://doi.org/10.1109/ICST46399.2020.00035>
- [32] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. 2013. Dynodroid: an input generation system for Android apps. In *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*. ACM, 224–234. <https://doi.org/10.1145/2491411.2491450>
- [33] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: multi-objective automated testing for Android applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*. ACM, 94–105. <https://doi.org/10.1145/2931037.2931054>
- [34] Sebastian Oster, Andreas Wübbecke, Gregor Engels, and Andy Schürr. 2011. A Survey of Model-Based Software Product Lines Testing. In *Model-Based Testing for Embedded Systems*. CRC Press. <https://doi.org/10.1201/b11321-14>
- [35] Owain Parry, Gregory M Kapfhammer, Michael Hilton, and Phil McMinn. 2021. A survey of flaky tests. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 1 (2021), 1–74.
- [36] Kleber L. Petry, Edson Oliveira Jr, and Avelino F. Zorzo. 2020. Model-based testing of software product lines: Mapping study and research roadmap. *J. Syst. Softw.* 167 (2020), 110608. <https://doi.org/10.1016/j.jss.2020.110608>
- [37] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. 2005. *Software Product Line Engineering: Foundations, Principles and Techniques* (1 ed.). Springer.
- [38] Alexander Pretschner. 2005. Model-Based Testing in Practice. In *FM 2005: Formal Methods*. Springer Berlin Heidelberg, Berlin, Heidelberg, 537–541.
- [39] Andreas Reuys, Erik Kamsties, Klaus Pohl, and Sacha Reis. 2005. Model-Based System Testing of Software Product Families. In *17th International Conference Advanced Information Systems Engineering (CAiSE) (Lecture Notes in Computer Science, Vol. 3520)*. Springer, 519–534. https://doi.org/10.1007/11431855_36
- [40] Kabir S. Said, Liming Nie, Adekunle Akinjobi Ajibode, and Xueyi Zhou. 2020. GUI testing for mobile applications: objectives, approaches and challenges. In *Internetware'20: 12th Asia-Pacific Symposium on Internetware, Singapore, November 1-3, 2020*. ACM, 51–60. <https://doi.org/10.1145/3457913.3457931>
- [41] Klaus Schmid and Isabel John. 2004. A customizable approach to full lifecycle variability management. *Science of Computer Programming* 53, 3 (2004), 259–284.
- [42] Porfirio Tramontana, Domenico Amalfitano, Nicola Amatucci, and Anna Rita Fasolino. 2019. Automated functional testing of mobile applications: a systematic mapping study. *Softw. Qual. J.* 27, 1 (2019), 149–201. <https://doi.org/10.1007/s11219-018-9418-6>
- [43] Mark Utting and Bruno Legeard. 2010. *Practical model-based testing: a tools approach*. Elsevier.
- [44] Mark Utting, Bruno Legeard, Fabrice Bouquet, Elizabeta Fourneret, Fabien Peureux, and Alexandre Vernotte. 2016. Chapter Two - Recent Advances in Model-Based Testing. *Advances in Computers*, Vol. 101. Elsevier, 53–120. <https://doi.org/10.1016/bs.adcom.2015.11.004>