Designing a Test Model for a Configurable System: An Exploratory Study of Preprocessor Directives and Feature Toggles

Stefan Fischer
Software Competence Center Hagenberg GmbH (SCCH), Austria

Gabriela K. Michelon
Institute for Software Systems Engineering & LIT Secure and Correct Systems Lab, Johannes Kepler University Linz, Austria

Rudolf Ramler
Software Competence Center Hagenberg GmbH (SCCH), Austria

Wesley K. G. Assunção
Alexander Egyed
Institute for Software Systems Engineering, Johannes Kepler University Linz, Austria

ABSTRACT

Testing is important in software development, but it has high cost. Thus, techniques to reduce the cost of software testing have been proposed. Model-based testing, one of such techniques, focuses on automatizing the generation of test cases. In the context of highly configurable systems, model-based testing must capture the system behavior and also encode the variability that exists among the variants. Previous research has shown promising results in applying model-based testing to configurable systems. Test models that encode variability into them directly improve the reasoning for faults from interactions. However, there is no study about the use of different variability mechanisms to encode variability in test models. In this paper, we investigate advantages and drawbacks of test model designs exploring the use of two variability mechanisms, namely preprocessor directives and feature toggles. The results are discussed in regard to run-time reasoning and re-configuration, alongside with metrics about complexity and maintainability. With this work, we contribute to the testing activity of highly configurable systems by providing engineers insights of comparing two well-known and widely used variability mechanisms, which can support informed decisions when choosing for which mechanisms to use for model-based testing.

CCS CONCEPTS

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

KEYWORDS

software testing, variability mechanism, test case generation, software product lines

ACM Reference Format:
Stefan Fischer, Gabriela K. Michelon, Rudolf Ramler, Wesley K. G. Assunção, 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 2023), January 25–27, 2023, Odense, Denmark*. ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3571788.3571795

VaMoS 2023, January 25–27, 2023, Odense, Denmark

ACM ISBN 979-8-4007-0001-9/23/01.
https://doi.org/10.1145/3571788.3571795

1 INTRODUCTION

Testing is a vitally-important and highly-cost activity in software development [8, 11]. To reduce the cost of software testing, *model-based testing* advocated for automatizing the design and generation of test cases [32], going further from the automation of only test cases execution. In model-based testing, a *test model* is designed to automatically create the test cases [32]. Model-based testing is widely investigated in the context of single systems [1, 12, 15]. However, software systems are rarely entirely defined during the development time. The large majority of software systems have configuration options to support variability [3], enabling engineers or users to tailor their systems to specific scenarios [25]. Thus, model-based testing must also take variability into account.

Highly configurable systems (HCSs) rely on the definition of configuration options (i.e., features), where a configuration option represents a certain functionality or behavior in a system [34]. HCSs are complex pieces of software in which variability is pervasive in the entire development process [7]. Variability can be implemented by using different approaches, in which annotative or compositional approaches are the main subdivision [14]. In industry, annotative HCSs are predominant [23], being used in programming languages like C (with `#ifdef` preprocessor directives) and also object-oriented languages, such as C++ and Java. Feature toggling is also an annotative approach recently investigated in the context of HCSs [19, 20]. Using feature toggles, there is no need for variability management libraries, since the annotations are based on common `if` and `else` clauses of the target programming language [23].

Applying model-based testing for HCSs means that variability must also be described in the test models, allowing the generation of test cases for the different configurations of the software system. Model-based testing has been investigated in the literature in the context of configurable cyber-physical systems [2], automated reuse of test variants [10], pairwise testing to verify a subset of all possible configurations [18], deriving test case scenarios [31], and quality assurance of a safety-critical system family [4]. However, these approaches rely mostly on pre-processor directives. Feature toggles have been discussed for testing of single systems, supporting beta testing or A/B testing [30]. The flexibility of toggles allow developers easily enabling/disabling certain features for testing and debugging tasks. It is easier to spread testing across different groups of users, e.g., some users could test one combination of features, while others are testing another combination. However, feature toggles for testing HCSs have not been explored yet.

Despite existing pieces of work, little is known about the use of different variability mechanism in test models. There are no

pieces of work on model-based testing that investigate the benefits and drawbacks of different variability mechanisms. Yet, there is no study exploring the use of feature toggles in model-based testing. As feature toggles have been recently investigated in HCSs [13], it might also be successfully applied in the testing activity. This lack of knowledge hinders the decision of engineers on which variability mechanism to use. Furthermore, researchers cannot propose approaches or solutions for this topic without knowing what are the problems each variability mechanism have.

The goal of this work is twofold: (i) to explore the use of feature toggles in model-based testing, and (ii) to investigate the advantages and drawbacks of two variability mechanisms used in industry (i.e., preprocessor directives and feature toggles) for model-based testing. For testing variants of a widely used industrial HCS, namely the Bugzilla web application, we developed two test models: (i) one using preprocessor directives at arbitrary abstraction levels, and (ii) one using feature toggles at fixed abstraction levels. We compare both test models quantitatively in terms of model size, complexity based on scattering and tangling degrees [29], code duplication, files and option locations [19]. Finally, we qualitatively discuss both models regarding flexibility, scope, efficiency, separation of concerns, traceability, modifiability/maintainability, and configurability [24].

The results of our study show that the design with toggles has some drawbacks compared to the design using preprocessor annotations. On the one hand, using the design with toggles duplicates test steps, the test model is more complex, more difficult to understand, and harder to maintain. On the other hand, the advantages of this design are related to the more disciplined design variability representation, allowing more sophisticated analysis (e.g., more straightforward application of coverage metrics), and runtime reconfiguration of the test model. The main contribution of this work is to support software engineers with empirical results about the use of two well-known variability mechanisms. When planning the testing activity of HCSs, such engineers can use our results to take informed decisions on which mechanisms and design to use for model-based testing, focusing on their needs and considering the trade-offs between the two mechanisms.

2 BACKGROUND AND RELATED WORK

This section describes the main concepts of model-based testing and variability mechanisms for HCSs, as well as related work.

2.1 Model-based testing

The software testing activity consists of three stages: (i) generation of test cases, (ii) execution of such test cases, and (iii) derivation of verdicts [28]. *Model-based testing* (MBT) supports the first stage, by providing a method for automatically generating test cases. According to Dalal et al. [6], the MBT approach depends on 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. Yet, MBT can be offline and online. On the offline MBT, test cases are generated for future execution, while on the online MBT, which is studied in this work, test cases are generated and executed simultaneously [33].

Models can be described by different paradigms and notations, for example, 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 [33]. In addition, models can also be expressed as Java programs [10], as we present in this work, using for example the framework OSMO MBT[1].

Although MBT tools can improve testing practices by increasing the effectiveness of the tests, shortening the testing cycle, and reducing the cost of test generation [6], it is challenging to apply MBT approaches to configurable systems [27]. In configurable systems, due to many potential products/configurations that can be derived, it might become infeasible to test all possible configurations [26]. In the following, we present the variability mechanisms to implement HCS in the context of our exploratory study.

2.2 Variability mechanisms

HCS are implemented with variability mechanisms. Each variability mechanism has its pros and cons. Zhang et al., [35] characterize and compare seven variability mechanisms, including preprocessor directives and feature toggles. In this paper, we focus on exploring MBT approaches for HCS implemented with *conditional compilation* (preprocessor technique with #ifdef directives) or *conditional execution* (conditional statements with feature toggles).

Conditional Compilation uses preprocessor directives such as #if, #elif, #ifdef, and #ifndef. The variability, i.e., features are realized at the construction time (processing and compilation). An advantage of this mechanism is that it can be applied to arbitrary granularity in the source files (e.g., classes, functions, type declarations, variables, or statements). Further, this mechanism uses explicit variation points in the source code with, e.g., #ifdef block, which can make it easier to find which parts of the source code belong to which feature. However, this also implies more integration of variability and code, which makes it potentially more complex. Therefore, #ifdef blocks tend to be complex and difficult to understand, when there are, for example, nested and scattered #ifdef blocks, and multiple macro constants tangled in #ifdef statements, as shown in existing work [21]. Another disadvantage of preprocessor-based systems is that variants are preprocessed, which means, their source code is copied and pasted according to selected macros, and this can lead to errors in the variants' code that cannot be easily found in the annotated code. Overall, this kind of mechanism tends to be hard to maintain because of its complexity [22].

Conditional Execution uses conditional statements, usually implemented as if-else or switch blocks. Unlike preprocessor directives, the variable features are realized at run-time (e.g., dynamic system adaptation) or are configured after compilation using a configuration file. In comparison to preprocessor directives, its granularity is coarser and limited because it is implemented in conditional code blocks. It is also hard to distinguish between the variation logic and code functionality because all code (for all variants) is always present. Further, the conditional execution compromises the compilation speed and efficiency at run-time due to the inclusion of all variant elements from code compilation till running [35].

[1] https://github.com/mukatee/osmo

2.3 Related work

There exist different studies taking into account variability for test cases [26]. However, in the context of our work, i.e., MBT for highly configurable systems, there is no study, to the best of our knowledge, exploring and comparing variability mechanisms applied to test models. For instance, Arrieta et al. [2] propose an MBT methodology for highly configurable cyber-physical systems. Their methodology semi-automatic generates test cases for a Simulink model from a Feature Model, which derives 100% of the test architecture.

Taking into account variability for generating test cases, Fischer et al. [10] investigates the difference between the automated reuse of test variants and models. However, their work does not focus on MBT, but rather on the reuse and adaption of existing tests for different configurations. In their work, they manually developed a test model of the Bugzilla web application, with variability in the test model realized by preprocessor directives. Our work thus differs from their work because we focus on presenting a comparison between MBT approach implemented with preprocessor directives and feature toggles.

Regarding MBT, there are studies focusing on testing a family of software products. For instance, Reuys et al. [31] present the ScenTED technique, which is a model-based and reuse-oriented technique for 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. A variation point is documented as a special decision within the activity diagram. Therefore, they do not use and compare different variability mechanisms to implement the test code/model, as we do in our work.

MBT activities such as defining test coverage, generation, and prioritization [16] can be used, for instance, for the quality assurance of a safety-critical system family of products as presented by Classen et al. [4]. They propose a model checking built on the *featured transitions systems* (FTSs) [5]. FTS is a compact mathematical model for representing the behavior of each product of a system family. This model is used to represent the behavior of a product during the domain engineering phase, i.e., when building the assets from which products are derived.

Differently from existing work, we thus focus on evaluating different aspects of using preprocessor directives and feature toggles as variability mechanisms for MBT [20]. There exist studies evaluating the structuring of the source code when implementing HCSs with feature toggles [19] and preprocessor directives [17, 21, 29], but they do not explore the use of feature toggles and preprocessor directives for model-based testing. Therefore, based on existing metrics from related work, we present an analysis of the pros and cons of both mechanisms, i.e., feature toggles and preprocessor directives, in the context of test models.

3 CONFIGURABLE TEST MODEL

As our example test model, we use a OSMO[2] model for testing the web application for the popular bug tracking system Bugzilla[3]. We originally implemented that model in our previous work [10], using conditional compilation as a variability mechanism. The model covers the main use cases of Bugzilla: (1) logging in and out of the system, (2) creating a new bug report, (3) editing the created report, (4) changing the status of a bug, and (5) searching for bug reports. Moreover, we identified 12 configuration options of Bugzilla (see Table 1), which impact these use cases and the model, that we annotated with preprocessor directives to be adaptable for these options in [10].

Table 1: Configuration options

Configuration option	Description
CO01	Enable status white board field for optional comments
CO04	Add an additional product to organize bug reports
CO05	Add an additional component to a product
CO06	Add an additional version to a product
CO07	Configure bug status workflow to minimum set of states
CO08	Configure bug status workflow to different entry state
CO09	Require descriptions on creating a new bug entry
CO10	Require descriptions on all bug status changes
CO11	Require description on setting a bug to resolved
CO12	Enforce a comment when a bug is marked as duplicate
CO13	Enforce dependencies to be resolved before bug is fixed
CO14	Set the default bug status of duplicates to verified

Figure 1 depicts how the model is executed to test Bugzilla, highlighting the parts that are implemented for the test model. The OSMO Tester takes the model to execute the defined *test steps*, in several model classes. A *test step* is a Java method in an OSMO model. A test case is generated by sequentially executing *test steps* until some end condition is met (e.g., all test steps have been executed at least once). Furthermore, OSMO allows defining *guards*, which define a condition under which the *test step* may be executed (e.g., the required web page is open). OSMO first checks the *guards* to determine which test steps are ready to be executed. Subsequently, it selects a *test step* to execute and invokes it. These steps are repeated until the tester meets the end condition. The model implementation uses an adapter class as an abstraction layer, with a collection of reusable methods, to interact with the page objects. This adapter offers methods that are used in *test steps* to hide the interactions with page objects and simplify the *test steps*. For instance, the adapter contains a method to perform the actions to log in to Bugzilla. There is a page object class for each Bugzilla web page the model interacts with. Page objects are a way of abstracting away details of pages and make the interaction with the UI more structured. This helps to avoid code duplication and changes in the UI can be limited to the page object, instead having to be addressed in any test step using them. The page objects use Selenium[4] to interact with the Bugzilla UI running in the browser.

3.1 Using preprocessors for test models

TestModel 1 shows a snippet of the OSMO model, taken without changes from [10]. The model is defined as Java code, with each test step in the model being a simple method with no parameters and Java annotation marking it as a test step with a name to identify it. The method `toConfirmed` implements a test step in the model that changes the status of a bug entry to "CONFIRMED". Depending on

[2]https://github.com/mukatee/osmo
[3]https://www.bugzilla.org/
[4]https://www.selenium.dev/

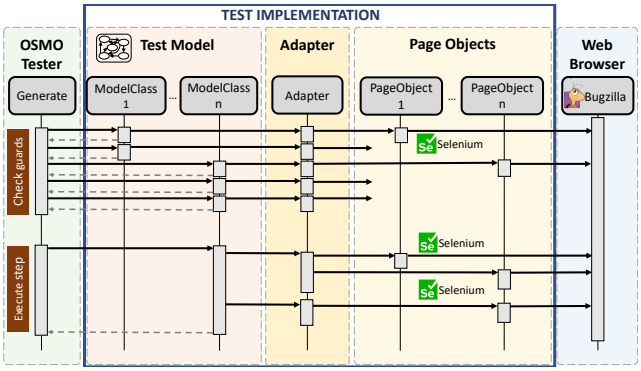

Figure 1: Test sequence for testing Bugzilla with our model

the configuration this step has to be performed slightly different, as when configuration option CO10 is active an additional comment is required as a description to change this bug status (see Line 3).

TestModel 1: Test model code snippet with preprocessor annotations.

```
1  @TestStep("toConfirmed")
2  public void toConfirmed() {
3    #if ($CO10)
4    adapter.editStateWithComment("CONFIRMED");
5    #else
6    adapter.editState("CONFIRMED");
7    #end
8    state.getCurrentlyOpenBug().setStatus("CONFIRMED");
9    ...
10   state.setCurrentlyOpenBug(null);
11 }

13 @Guard("toConfirmed")
14 public boolean toConfirmedGuard() {
15   ...
16   String status = bug.getStatus();
17   #if ($CO07)
18   return "RESOLVED".equals(status);
19   #else
20   return "UNCONFIRMED".equals(status) || "IN_PROGRESS".equals(
         status) || "RESOLVED".equals(status) || "VERIFIED".
         equals(status);
21   #end
22 }
```

The second method in TestModel 1 represents the guard condition of the test step, marked by the Java annotation containing the name of the test steps it applies to. Test steps can only be executed if all the guard methods linked to them return true, i.e., the guard conditions are fulfilled. This guard condition changes depending on the configuration CO07. If configuration option CO07 is active, it limits the bug statuses under which the status can be changed to "RESOLVED" (see Line 17). Variability in the page objects was implemented similarly, with annotations in the source code that allows us to derive variants from them.

The top part of Figure 2 depicts how we can use this annotated test model to test Bugzilla. From the test model with preprocessor directives, we can generate different model variants that allow us to test the corresponding Bugzilla configuration. In this notation, we show the system states in circles and the arrows between them represent the test steps. Designing the model with this variability mechanism made it convenient to quickly derive model variants for many configurations. However, when we want to test the configurable system in depth, first generating model variants and testing

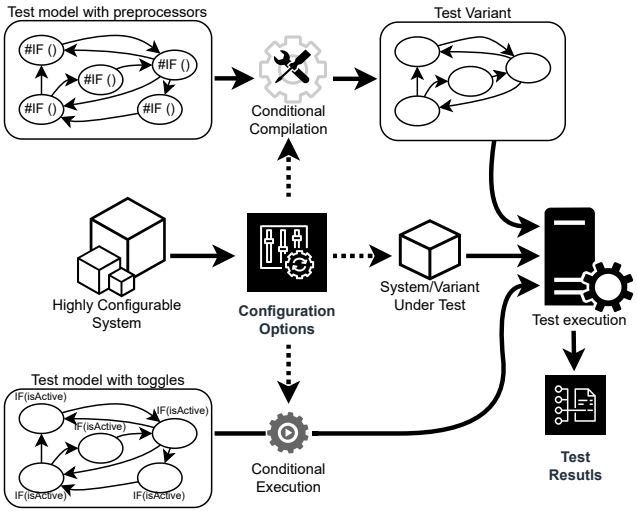

Figure 2: Process of testing with the models using the two variability mechanisms

selected configurations with them produces an overhead in effort. Testing with multiple variants makes the application of metrics like model coverage, which is the ratio of executed test steps to all steps, more complicated. For instance, in our example, the test step `toConfirmed` could have been reached in one variant, however, it is not immediately clear if we need to cover it again in another variant. Deciding this requires additional effort to decide if we need to execute this test step again in another variant. Having 100% model coverage for all variants might substantially increase the testing effort. Moreover, having the variability on arbitrary granularity levels further complicates reasoning over the model, to select which test steps should be executed on which configuration.

3.2 Using toggles for test models

Because of the drawbacks listed above, we thought about a redesign of our test model. Since the OSMO model already includes the guard conditions in the model code, the goal is to use them also to encode the variability in the model. By defining the configuration options that need to be enabled or disabled for a test step in the guard condition, we can directly apply the model coverage metric, which shows us if we executed all test steps also with respect to the configurations required. Moreover, by checking the guard methods at run-time, one could determine which configurations to test while executing the model. We depict the testing process with the model using feature toggles at the bottom of Figure 2. Here we no longer have to generate model variants, but rather can use the model directly. Depending on the current system configuration, the test steps that are available change while executing the model.

TestModel 2: Test model code snippet with feature toggles.

```
@TestStep("toConfirmedC010NCO07")
public void toConfirmedC010NCO07() {
  adapter.editStateWithComment("CONFIRMED");
  state.getCurrentlyOpenBug().setStatus("CONFIRMED");
  ...
  state.setCurrentlyOpenBug(null);
}

@Guard("toConfirmedC010NCO07")
public boolean toConfirmedC010NC007Guard() {
  return ConfigurationOption.CO10.isActive() &&
         !ConfigurationOption.CO07.isActive();
}

@TestStep("toConfirmedNC010NCO07")
public void toConfirmedNC010NCO07() {
  adapter.editState("CONFIRMED");
  state.getCurrentlyOpenBug().setStatus("CONFIRMED");
  ...
  state.setCurrentlyOpenBug(null);
}

@Guard("toConfirmedNC010NCO07")
public boolean toConfirmedNC010NC007Guard() {
  return !ConfigurationOption.CO10.isActive() &&
         !ConfigurationOption.CO07.isActive();
}

@Guard({"toConfirmedC010NCO07","toConfirmedNC010NCO07"})
public boolean toConfirmedNC007Guard() {
  ...
  return "UNCONFIRMED".equals(status) || "IN_PROGRESS".equals(
       status) || "RESOLVED".equals(status) || "VERIFIED".
       equals(status);
}

@TestStep("toConfirmedC010CO07")
public void toConfirmedC010CO07() {
  adapter.editStateWithComment("CONFIRMED");
  state.getCurrentlyOpenBug().setStatus("CONFIRMED");
  ...
  state.setCurrentlyOpenBug(null);
}

@Guard("toConfirmedC010CO07")
public boolean toConfirmedC010CO07Guard() {
  return ConfigurationOption.CO10.isActive() &&
         ConfigurationOption.CO07.isActive();
}

@TestStep("toConfirmedNC010CO07")
public void toConfirmedNC010CO07() {
  adapter.editState("CONFIRMED");
  state.getCurrentlyOpenBug().setStatus("CONFIRMED");
  ...
  state.setCurrentlyOpenBug(null);
}

@Guard("toConfirmedNC010CO07")
public boolean toConfirmedNC010CO07Guard() {
  return !ConfigurationOption.CO10.isActive() &&
         ConfigurationOption.CO07.isActive();
}

@Guard({"toConfirmedC010CO07","toConfirmedNC010CO07"})
public boolean toConfirmedCO07Guard() {
  ...
  return "RESOLVED".equals(status);
}
```

We show a snippet for the same test step as before, for different configurations, in TestModel 2. We highlight the lines using the toggles implemented with Togglz[5], like used in [13]. Because the test step and its guard change depending on two configuration options, we require four instances of the test step and the corresponding guards to cover all combinations of the two options.

[5]https://www.togglz.org/

For each test step we added a separate guard method which only includes checks for the feature toggles to check the current configuration. This design makes it possible to directly use model-based coverage metrics, like model coverage, that will ensure we cover the test step in all relevant configurations. If we cover all test steps, all the combinations of options contained in the model had to be covered as well, which is more difficult to decide in the design with preprocessor directives. Additionally, we had to adapt the implementation of the adapter and the page objects (see Figure 1) to also use feature toggles instead of annotations. This was possible in a very straightforward manner, since the annotations were mostly at a level that conditional execution with toggles could also be applied, i.e., within methods. The source code for the two models is available at https://anonymous.4open.science/r/2023-VaMoS-Bugzilla_Models-647D/.

4 EXPLORATORY STUDY

In this section, we discuss the advantages and drawback of the two model designs with the different variability mechanisms (described in Section 3), and present metrics to substantiate our arguments.

4.1 Evaluation Metrics

We start with two size metrics to compare the two model designs:

(1) **Lines of Code (LoC):** The amount of lines in the test model code, computed with CLOC[6].
(2) **Number of test steps:** The amount of test steps that exist in a model version.

To assess the complexity of our two model implementations, we repurposed metrics from Queiroz et al. [29]. We slightly adapted them to analyze the complexity of test steps instead of annotations in the entire system, as follows:

(3) **Scattering degree (SD):** This degree is computed per configuration option of the system. It counts the number of test steps that refer to a given configuration option.
(4) **Tangling degree (TD):** This degree is computed per test step of the model. It counts the number of configuration options that occur in a given test step. Either the test step has preprocessor directives with the configuration option or uses it in its guard condition as a feature toggle.

Mahdavi-Hezaveh et al. [19] defined heuristics and metrics regarding the use of feature toggles. One of these heuristics is to avoid duplicates, which we used in our evaluation:

(5) **Number of duplicated test steps:** Our proposed design with toggles requires the duplication of test steps, therefore we report the number of test steps that we had to duplicate compared to the version of the model with preprocessor directives.

Additionally, Mahdavi-Hezaveh et al. [19] defined metrics regarding the locations in which feature toggles are used. We expand these metrics to not only apply to our feature toggles, but can be used for the design with preprocessor directives as well.

(6) **Number of files:** The number of files that contain a variation point (i.e., an #ifdef or a toggle), including configuration files, and code. Maintaining more files in connection with

[6]https://cloc.sourceforge.net/

a configuration option increases the likelihood of mistakes. We count the number of files for each configuration option and then average of the number of options.

(7) **Number of locations:** The number of different locations where a configuration option is used. We count each occurrence of an option, including multiples within a file. As for the last metric, we compute the average over all options.

4.2 Method

To compute the metrics presented above, we analyzed the source code of the two model designs. We implemented the analysis ourselves. This analysis parses the source code and generates the abstract syntax tree. From there, we can check which parts of the code are annotated with preprocessor directives and where feature toggles are used and map this to the methods that have Java annotations for test steps and guards.

4.3 Results & Discussion

Based on the metrics and method presented, we compared the two designs of our test model and report the results here.

Size: The different designs require different numbers of test steps, as shown in Section 3. Therefore, the model using preprocessor annotation contains *57 test steps*, whereas the model with feature toggles contains *78 test steps*. This translates to an increase from *1006 LoC* for the preprocessor directive versions to *1499 LoC* for the model with toggles. This is nearly a 50% increase in code, which can negatively affect the maintenance of the test model with toggles.

Scattering: Next we computed the scattering degree for the model with preprocessor directives and the model design using toggles. Figure 3 depicts the scattering degree over the model for each configuration option. We can see that for most configuration options, the scattering degree is worse for the model version using toggles. This is due to the additional test steps introduced in the model version with toggles. For instance, in our example in Section 3 we can see that CO07 and CO10 are each used in four test steps in the model using toggles compared to only in one test step in the model with preprocessor directives. On average, the scattering degree over the model is 4.33 for the model with preprocessor directives and 8.42 for the model with toggles. This means, on average, configuration options are used in nearly twice as many test steps in the model using toggles. A higher scattering degree implies that changes for such a configuration option can lead to changes in many more test steps.

Tangling: Similarly, we computed the tangling degree with the test steps of our two model designs. We present these results in Table 2, which shows for each tangling degree the number of test steps that reached it, per mechanism. In both models, there are 28 test steps that do not have any conditions with configuration options. Hence, they have *TD* of zero. For the following rows, the version with the toggles in the model guards always has more test steps for each tangling degree. This is further reflected in the average tangling degree (in the last row of Table 2) that is higher for the model design with feature toggles. A higher tangling degree generally implies more complex dependencies to configuration options that have to be taken into account when maintaining the test steps.

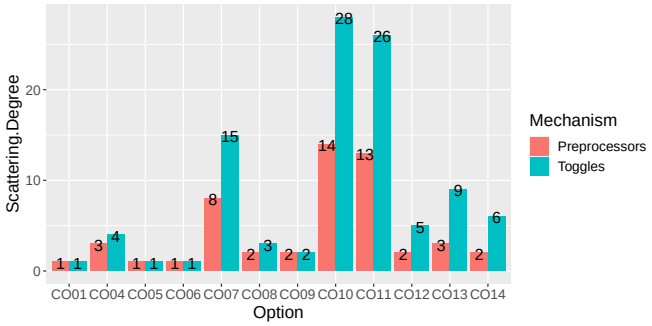

Figure 3: Scattering degree per variability mechanism

Table 2: Tangling degree per variability mechanism and number of test steps

TD	Mechanism	
	Preprocessors	Toggles
0	28	28
1	14	18
2	9	18
3	4	9
4	2	5
AVG TD	0.9122807	1.2948718

Following the threshold of Queiroz et al. [29] for our context of test steps, at least, 80% of the feature annotations in the test steps should have *TD* <= 1. As we can see in Table 2, the preprocessor mechanism has about 74% of the test steps with *TD* <= 1, whereas the toggles mechanism around 59% with *TD* <= 1.

The scattering degree and tangling degree are both worse due to the additional test steps. This is an expected result because the test steps with more variation points (usually correlated with more configuration options) have to be cloned more often to cover all combinations. Therefore, the configuration options used in more preprocessor directives for test steps are used in even more test steps with feature toggles, explaining the higher scattering degree for these options and the higher tangling degree for more test steps.

Duplicates: The number of test steps is higher in the model with feature toggles, because we duplicated the test steps that change, either in the step or the linked guard, depending on the configuration. Therefore, nine test steps from the annotated model with preprocessor directives have duplicates in the model with toggles, i.e., each has two test steps in the redesign. Additionally, four test steps required three duplicates in the redesigned model, i.e., each of these test steps is represented by four steps in the model with toggles. Finally, the remaining 44 test steps in the annotated model did not require any duplicates, but could simply be redesigned by adding the corresponding guard method where necessary or be reused without change (for the 28 steps without preprocessor directives). Additionally, for each duplicated test step, an additional guard method is added. Duplicates are a code smell and should be avoided if possible [19].

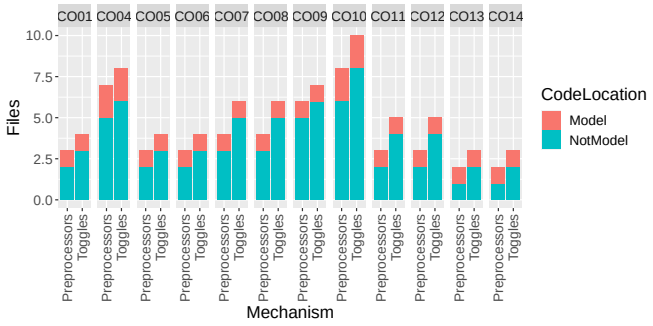

Figure 4: Number of files per variability mechanism

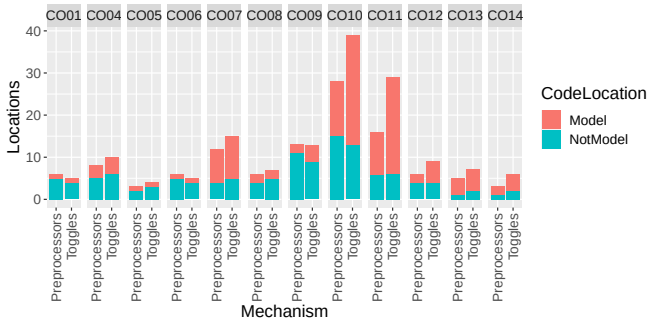

Figure 5: Number of locations per variability mechanism

Files and Locations: Figures 4 and 5 show the number of files and locations for configuration options, respectively. Here we further distinguish between usage in the models and the rest of the test code, i.e. adapter, and page objects. The number of files in which a configuration option is used is not much influenced by the mechanism used. Configuration option CO10 is the one used in the most files (ten). Some of the toggles are used more in one file because of the additional configuration file used by the Togglz framework. On the other hand, this is made up for most of the configurations options that are used in some additional preprocessor directives. For instance, to annotate entire methods in the adapter, so that method is only present in variants that call it. We cannot use feature toggles on an entire method like this, and simply overload such method in the adapter with toggles. On average, a configuration option is used in 4 files in the design with preprocessor directives and 5.42 files for the model with toggles. Similarly, for the locations a configuration option is used, we do not see much difference in the code outside the model. However, for some options we require more uses in the model with feature toggles, due to the duplication of test steps. Configuration options are used on average in 9.33 locations in the model with preprocessor directives and 12.42 locations in the model with toggles. According to Mahdavi-Hezaveh et al. [19], feature toggles should be used in as few locations as possible to improve maintainability.

Another heuristic defined by Mahdavi-Hezaveh et al. [19] is to use one shared method to check the value of all feature toggles. Because having fewer methods to check feature toggle value decreases the number of files and lines of code that need to be modified, which lowers the code complexity. We fulfill this since it is supported by the Togglz framework we use.

Table 3: Assessment of quality criteria for the variability mechanism in our test model designs

Quality Criterion	Mechanism	
	Preprocessors	Toggles
Flexibility	○	●
Scope	●	◓
Efficiency	◓	◓
Separation of Concerns	○	○
Traceability	◓	◓
Modifiability / Maintainability	◓	○
Configurability	◓	●

● well-supported, ◓ partially supported, ○ poorly/not supported

Qualitative analysis: The SPL literature describes several quality criteria for evaluating variability mechanisms [24]. We summarize and assess the quality criteria with respect to the variability mechanisms in our model designs in Table 3. Our assessment is classified in well-supported, partially supported, or poorly/not supported.

- **Flexibility** refers to the binding time of the variability with the SPL assets. Because feature toggles can be changed at runtime, i.e., conditional execution, they are more flexible than the conditional compilation with preprocessor directives.
- **Scope** is the granularity at which a mechanism supports variability, which is better supported with preprocessor directives that can be at an arbitrary level. Conditional execution, for instance with toggles, can only be used within the executable parts of code and not for changing the source code structure (e.g., removing entire methods or classes).
- **Efficiency** is the overhead required to support the variability in the system using the variability mechanism. Compared to preprocessor directives, feature toggles do not require additional build steps, but can be added using well known programming constructs. However, conditional execution has been known to decrease compilation speed and performance at run-time, because the entire code is present at execution and variation points are evaluated at run-time [35]. Therefore, we classified both designs as medium because their overheads come at different points in the life-cycle.
- **Separation of Concerns** refers to the decoupling of variable and common assets in the system. Neither of the two variability mechanisms particularly supports this.
- **Traceability** of the configuration options to the test assets is assessed very similar for the two designs. Both designs spread the implementation of the steps related to the configuration options through the model. To trace options in preprocessor directives we require additional tool support which is often not readily available to testers. The usage of toggles can be found with standard development tools, however the separation of guards and test steps implies some additional effort to trace all assets related to an option.

- **Modifiability / Maintainability** is worse for the model using toggles, because of the code duplication in the model. If one of the duplicated test steps changes, we will potentially have to change several at once. Maintaining the model with preprocessor directives does not have the issue of duplicates. However, the code annotated preprocessor directives is not necessarily compilable and common development tools lack support to deal with their added complexity [35]. Regarding the number of files and lines of code, feature toggles tend to lead to more complexity than annotated code to maintain the test model [19].
- **Configurability** is well-supported with both mechanisms. However, feature toggles have the added advantage of being able to re-configure the model at run-time, which makes it possible to test several configurations in one run of the model-based testing tool.

4.4 Lessons learned

From our experiment, we can see that designing the test model to define the variability entirely in the guards does have some drawbacks, compared to using preprocessor annotations. Using feature toggles requires many duplicates of test steps, the model is more complex, more difficult to understand, and harder to maintain. All of these drawbacks seem like a clear reason not to use this design. However, we still believe there are advantages in encoding the variability information in the test steps. Not only is the application of metrics like model coverage more straightforward, but also analysis over the model is supported when using toggles. For instance, the computation of which configurations should be tested can be decided at run-time, by checking for which configurations the guard conditions for test steps are fulfilled. The system could even be reconfigured at run-time, in our case, to enable different test steps. Additionally, a disciplined design of a test model, having a guard condition of configuration options for each test step, can enable more sophisticated analysis to improve the testing of feature interactions. For instance, we could improve the testing of configurable software with analysis similar as have been proposed for *featured transitions systems* [4].

From our experience from this experiment, we believe that a deeper investigation into designing test models with such configuration guard conditions is a worthwhile endeavor. However, the drawback from our model design need to be addressed in order to be applicable in practice. One idea would be to develop an automatic transformation from the annotated model to one using feature toggles, so that developers can benefit from both mechanisms, depending on the test goals. Additionally, further quantitative investigation is needed to assess how efficiently the coverage of test models with feature toggles can be calculated.

4.5 Threats to validity

Internal: Although we used quality criteria and metrics that have been already used for evaluating variability mechanisms with respect to variable systems [17, 19, 21, 24, 29], they are not metrics already used to evaluate test models, which can be a source of bias. However, these metrics and criteria are used to measure the complexity and maintainability of the system code, we thus avoid varying definitions and metrics to not limit the applicability of our work. We just had to slightly adapt how we compute the complexity of the code based on test steps instead of annotations in the code.

Another internal threat to validity can be errors in the computation of metrics that can affect the results. Although we used our own tool to compute the metrics, we double-checked all results. Further, our redesigned annotated test model with feature toggles, is implemented with Togglz similar to Jézéquel et al. [13], and available at https://anonymous.4open.science/r/2023-VaMoS-Bugzilla_Models-647D/ for future work and comparisons.

External: The choice of the MBT framework to conduct our study can be a source of bias because we only looked at one model for one system. We analyzed only one system, Bugzilla. To generalize our results, we require studies with more systems in the future. However, it is hard to find publicly available configurable systems with tests that are also configurable, and Bugzilla is a popular and widely used configurable system, which is available and with tests annotated with preprocessor directives [9]. Regarding the test model, it is a java program, which enables easier adoption of a variability mechanism in the code with preprocessor directives. We thus could quickly generate different model variants, which allow us to test many Bugzilla's configurations. Further, this model was already used and originally implemented in a previous research [10] and covers the main use cases of Bugzilla.

Another external threat to validity is that there exist different mechanisms for implementing an SPL or highly configurable software, and we only investigate two variability mechanisms for MBT. However, preprocessor directives are widely used in the present day to implement HCS [17] and are predominant in industry [23]. In addition, the feature toggle is increasingly used by software practitioners for many kinds of applications [13]. Therefore, it is worth exploring the advantages of both mechanisms to find a way to suppress their drawbacks for MBT.

5 CONCLUSIONS AND FUTURE WORK

We discussed advantages and drawbacks of designing a model for MBT of an HCS with two different variability mechanisms, preprocessor directives (conditional compilation) and feature toggles (conditional execution). Our analysis showed advantages for the design using conditional compilation in terms of complexity and maintainability. However, we discuss advantages in other areas for designing the model using conditional execution. We hope that our analysis can help future design decisions for test models for HCS.

We are currently working on applying MBT to an HCS in the industry and want to investigate more designs to find the optimal trade-off for creating a configurable model that is easy to maintain and apply. Another direction for future work could be conducting a survey with practitioners that use MBT approaches for HCSs to obtain a practical comparison of existing variability mechanisms in the context of test code. This would enable us to expand our analysis and provide recommendations to practitioners. Furthermore, we plan to perform experiments with the model using feature toggles to utilize its advantages. For instance, for a run-time configurable system, as in our study, we plan to develop an algorithm in OSMO that automatically re-configures the system while testing with the model. By doing this, we can create test runs that test all relevant

configurations in one go while using the minimum of required test steps. Finally, to utilize the advantages of both our designs and mitigate their drawbacks, we plan to develop an automatic transformation from the model using preprocessor directives to the one using feature toggles.

ACKNOWLEDGMENTS

The research reported in this paper has been funded by BMK, BMDW, and the State of Upper Austria in the frame of SCCH, part of the COMET Programme managed by FFG; the LIT Secure and Correct System Lab funded by the State of Upper Austria; and the Austrian Science Fund (FWF), grand no. P31989.

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] Aitor Arrieta, Goiuria Sagardui, and Leire Etxeberria. 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.

[3] Rafael Capilla, Jan Bosch, Kyo-Chul Kang, et al. 2013. Systems and software variability management. *Concepts Tools and Experiences* 10 (2013), 2517766.

[4] 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 Trans. Software Eng.* 39, 8 (2013), 1069–1089. https://doi.org/10.1109/TSE.2012.86

[5] Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, Axel Legay, and Jean-Francois 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

[6] 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

[7] Clemens Dubslaff, Kallistos Weis, Christel Baier, and Sven Apel. 2022. Causality in Configurable Software Systems. In *44th International Conference on Software Engineering*. Association for Computing Machinery, New York, NY, USA, 325–337. https://doi.org/10.1145/3510003.3510200

[8] Fischer Ferreira, Gustavo Vale, João P. Diniz, and Eduardo Figueiredo. 2021. Evaluating T-wise testing strategies in a community-wide dataset of configurable software systems. *Journal of Systems and Software* 179 (2021), 110990. https://doi.org/10.1016/j.jss.2021.110990

[9] 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

[10] 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 2021, Taipei, Taiwan, December 6-9, 2021*. IEEE, 421–430. https://doi.org/10.1109/APSEC53868.2021.00049

[11] Vahid Garousi and Junji Zhi. 2013. A survey of software testing practices in Canada. *Journal of Systems and Software* 86, 5 (2013), 1354–1376.

[12] 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.

[13] Jean-Marc Jézéquel, Jörg Kienzle, and Mathieu Acher. 2022. From feature models to feature toggles in practice. In *26th ACM International Systems and Software Product Line Conference-Volume A*. 234–244.

[14] Christian Kästner, Sven Apel, and Martin Kuhlemann. 2008. Granularity in software product lines. In *2008 ACM/IEEE 30th International Conference on Software Engineering*. IEEE, 311–320.

[15] Stefan Kriebel, Matthias Markthaler, Karin Samira Salman, Timo Greifenberg, Steffen Hillemacher, 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.

[16] 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*, Bernhard Steffen, Christel Baier, Mark van den Brand, Johann Eder, Mike Hinchey, and Tiziana Margaria (Eds.). Springer International Publishing, Cham, 453–463.

[17] Jorg Liebig, Sven Apel, Christian Lengauer, Christian Kästner, and Michael Schulze. 2010. An analysis of the variability in forty preprocessor-based software product lines. In *32nd International Conference on Software Engineering*, Vol. 1. 105–114. https://doi.org/10.1145/1806799.1806819

[18] Malte Lochau, Sebastian Oster, Ursula Goltz, and Andy Schürr. 2012. Model-based pairwise testing for feature interaction coverage in software product line engineering. *Softw. Qual. J.* 20, 3-4 (2012), 567–604. https://doi.org/10.1007/s11219-011-9165-4

[19] Rezvan Mahdavi-Hezaveh, Nirav Ajmeri, and Laurie Williams. 2022. Feature toggles as code: Heuristics and metrics for structuring feature toggles. *Information and Software Technology* 145 (2022), 106813. https://doi.org/10.1016/j.infsof.2021.106813

[20] Rezvan Mahdavi-Hezaveh, Jacob Dremann, and Laurie Williams. 2021. Software development with feature toggles: practices used by practitioners. *Empirical Software Engineering* 26, 1 (2021), 1–33.

[21] Gabriela K. Michelon, Wesley K. G. Assunção, David Obermann, Lukas Linsbauer, Paul Grünbacher, and Alexander Egyed. 2021. The Life Cycle of Features in Highly-Configurable Software Systems Evolving in Space and Time. In *20th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*. Association for Computing Machinery, New York, NY, USA, 2–15. https://doi.org/10.1145/3486609.3487195

[22] Gabriela Karoline Michelon, David Obermann, Wesley K. G. Assunção, Lukas Linsbauer, Paul Grünbacher, Stefan Fischer, Roberto E. Lopez-Herrejon, and Alexander Egyed. 2022. Evolving software system families in space and time with feature revisions. *Empir. Softw. Eng.* 27, 5 (2022), 112. https://doi.org/10.1007/s10664-021-10108-z

[23] Rodrigo André Ferreira Moreira, Wesley KG Assunção, Jabier Martinez, and Eduardo Figueiredo. 2022. Open-source software product line extraction processes: the ArgoUML-SPL and Phaser cases. *Empirical Software Engineering* 27, 4 (2022), 1–35.

[24] Erika Mir Olimpiew. 2008. *Model-Based Testing for Software Product Lines*. Ph. D. Dissertation. Fairfax, VA, USA,. Advisor(s) Hassan Gomaa.

[25] Sebastian Oster, Andreas Wübbeke, Gregor Engels, and Andy Schürr. 2011. A Survey of Model-Based Software Product Lines Testing. In *Model-Based Testing for Embedded Systems*, Justyna Zander, Ina Schieferdecker, and Pieter J. Mosterman (Eds.). CRC Press. https://doi.org/10.1201/b11321-14

[26] Kleber L. Petry, Edson OliveiraJr, 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

[27] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. 2005. *Software Product Line Engineering: Foundations, Principles and Techniques* (1 ed.). Springer.

[28] Alexander Pretschner. 2005. Model-Based Testing in Practice. In *FM 2005: Formal Methods*, John Fitzgerald, Ian J. Hayes, and Andrzej Tarlecki (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 537–541.

[29] Rodrigo Queiroz, Leonardo Teixeira Passos, Marco Túlio Valente, Claus Hunsen, Sven Apel, and Krzysztof Czarnecki. 2017. The shape of feature code: an analysis of twenty C-preprocessor-based systems. *Softw. Syst. Model.* 16, 1 (2017), 77–96. https://doi.org/10.1007/s10270-015-0483-z

[30] Md Tajmilur Rahman, Louis-Philippe Querel, Peter C. Rigby, and Bram Adams. 2016. Feature Toggles: Practitioner Practices and a Case Study. In *13th International Conference on Mining Software Repositories*. ACM, New York, NY, USA, 201–211. https://doi.org/10.1145/2901739.2901745

[31] Andreas Reuys, Erik Kamsties, Klaus Pohl, and Sacha Reis. 2005. Model-Based System Testing of Software Product Families. In *Advanced Information Systems Engineering, 17th International Conference, CAiSE 2005, Porto, Portugal, June 13-17, 2005, Proceedings (Lecture Notes in Computer Science, Vol. 3520)*, Oscar Pastor and João Falcão e Cunha (Eds.). Springer, 519–534. https://doi.org/10.1007/11431855_36

[32] Mark Utting and Bruno Legeard. 2010. *Practical model-based testing: a tools approach*. Elsevier.

[33] 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

[34] Alexander von Rhein, Alexander Grebhahn, Sven Apel, Norbert Siegmund, Dirk Beyer, and Thorsten Berger. 2015. Presence-Condition Simplification in Highly Configurable Systems. In *37th IEEE/ACM International Conference on Software Engineering, ICSE - Volume 1* (Florence, Italy), Antonia Bertolino, Gerardo Canfora, and Sebastian G. Elbaum (Eds.). IEEE Computer Society, 178–188. https://doi.org/10.1109/ICSE.2015.39

[35] Bo Zhang, Slawomir Duszynski, and Martin Becker. 2016. Variability mechanisms and lessons learned in practice. In *1st International Workshop on Variability and Complexity in Software Design*. ACM, 14–20. https://doi.org/10.1145/2897045.2897048