Tool Demo: Testing Configurable Systems with FeatureIDE

Mustafa Al-Hajjaji,[1] Jens Meinicke,[1,2] Sebastian Krieter,[1] Reimar Schröter,[1]
Thomas Thüm,[3] Thomas Leich,[2] Gunter Saake[1]

[1] University of Magdeburg, Germany, [2] METOP GmbH, Germany, [3] TU Braunschweig, Germany

Abstract

Most software systems are designed to provide custom functionality using configuration options. Testing such systems is challenging as running tests of a single configuration is often not sufficient, because defects may appear in other configurations. Ideally, all configurations of a software system should be tested, which is usually not applicable in practice due to the combinatorial explosion with respect to the configuration options. Multiple sampling strategies aim to reduce the set of tested configurations to a feasible amount, such as T-wise sampling, random configurations, and user-defined configurations. However, these strategies are often not applied in practice as they require manual effort or a specialized testing framework. Within our tool FeatureIDE, we integrate all aforementioned strategies and reduce the manual effort by automating the process of generating and testing configurations. Furthermore, we provide support for unit testing to avoid redundant test executions and for variability-aware testing. With this extension of FeatureIDE, we aim to make recent testing techniques for configurable systems applicable in practice.

Categories and Subject Descriptors D.2.3 [*Software Engineering*]: Programming Environments; D.2.5 [*Software Engineering*]: Testing and Debugging

Keywords T-Wise Sampling, Prioritization, Testing

1. Introduction

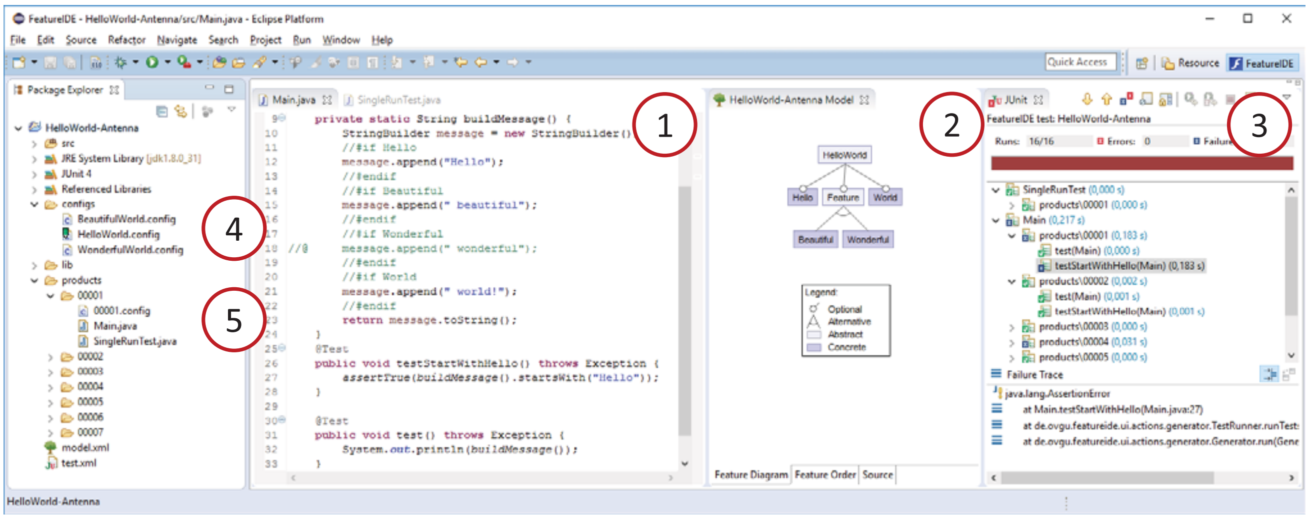
A configurable system enables customers to compose software systems from a large set of configuration options (also known as features). A feature is defined as an increment in functionality recognized by customers [7]. While configurable systems provide many advantages, such as reduction of development costs and time to market, they challenge quality assurance [23]. Usually, testing a configurable system includes creating configurations, building the corresponding products, testing them by executing their test cases, and using the results for debugging [28]. In practice, developers tend to test only one or a few configurations, as testing all valid configurations only scales to configurable systems with a small number of features [24].

Considering only a single configuration is not enough as some defects may only appear in some configurations [16]. Ideally, all valid configurations of a software system should be tested, especially for safety-critical systems. However, testing all valid configurations is often not possible due to the combinatorial explosion in the number of features and due to limited testing resources. Several strategies have been proposed to reduce the number of configurations that need to be tested, such as generating a reduced, yet sufficient subset of configurations [12, 18, 21, 31], random configurations, or user-defined configurations. These strategies require tedious manual effort, such as configuring and generating the software system as well as checking its validity. In addition, they may require a specialized testing framework. As a result, these strategies are often not applied in practice. Hence, automating the testing process is a necessary step to use the testing resources wisely.

To automate the testing process, numerous sampling tools have been introduced to create configurations while achieving a certain degree of coverage, such as MoSo-PoLiTe [29], CASA [12], Chvatal [17], ICPL [18], and IncLing [1]. Furthermore, Henard et al. introduce the PLEDGE tool to create and prioritize products based on their dissimilarity [14]. Moreover, Bürdek et al. present a tool that systematically explores similarities among products to enhance the testing efficiency [10]. While the aforementioned tools show promising results, each of them focuses only on testing rather than supporting the entire product-line development process.

We extended our tool FeatureIDE to automate the process of creating configurations, building products, and testing them by integrating the corresponding strategies. FeatureIDE is an Eclipse framework for feature-oriented software development [3, 32]. It incorporates tools for the implementation of configurable systems into an integrated de-

GPCE'16, October 31 – November 1, 2016, Amsterdam, Netherlands
© 2016 ACM. 978-1-4503-4446-3/16/10...
http://dx.doi.org/10.1145/2993236.2993254



```java
    private static String buildMessage() {
        StringBuilder message = new StringBuilder();
        //#if Hello
        message.append("Hello");
        //#endif
        //#if Beautiful
        message.append(" beautiful");
        //#endif
        //#if Wonderful
//@     message.append(" wonderful");
        //#endif
        //#if World
        message.append(" world!");
        //#endif
        return message.toString();
    }
    @Test
    public void testStartWithHello() throws Exception {
        assertTrue(buildMessage().startsWith("Hello"));
    }

    @Test
    public void test() throws Exception {
        System.out.println(buildMessage());
    }
```

```
FeatureIDE test: HelloWorld-Antenna
Runs: 16/16    Errors: 0    Failure

SingleRunTest (0,000 s)
  products\00001 (0,000 s)
Main (0,217 s)
  products\00001 (0,183 s)
    test(Main) (0,000 s)
    testStartWithHello(Main) (0,183 s)
  products\00002 (0,002 s)
    test(Main) (0,001 s)
    testStartWithHello(Main) (0,001 s)
  products\00003 (0,000 s)
  products\00004 (0,031 s)
  products\00005 (0,000 s)

Failure Trace
java.lang.AssertionError
    at Main.testStartWithHello(Main.java:27)
    at de.ovgu.featureide.ui.actions.generator.TestRunner.runTest
    at de.ovgu.featureide.ui.actions.generator.Generator.run(Gene
```

Figure 1. Support for testing with FeatureIDE: ① source code of a program including two unit tests, ② feature model defining valid combinations, ③ JUnit view, ④ user-defined configurations, and ⑤ a set of generated sample products.

velopment environment [32]. The contributions of this paper are as follows:

- Derivation of configurations using different techniques namely, (a) deriving all valid configurations, (b) generating random configurations, (c) using user-defined configurations, and (d) T-wise sampling.
- Generation of program variants independent of the programming paradigm, such as preprocessors and feature-oriented programming.
- Customization of product generation, such as adjusting the maximum number of created configurations and the order in which they are tested.
- Testing of generated program variants using JUnit and derivation of configuration-aware test results.
- Support to avoid redundant test executions and to achieve variability-aware testing.

2. Testing with FeatureIDE

In this section, we discuss the testing work-flow that we automated in FeatureIDE. We show a screen-shot of the FeatureIDE perspective in Eclipse in Figure 1. In the following, we explain each element and how it is integrated into FeatureIDE for testing purposes. The single elements are as follows: At ①, we show the source code of the program that we want to test, including two unit tests. At ②, we show the feature model that defines the variability of the program [6]. The folder `configs` at ④ shows three configurations that are user-defined. The folder `products` at ⑤ shows generated sample programs that are used for testing. The result of the tests for the configurations is shown in the JUnit view at ③.

2.1 Developing Configurable Systems with FeatureIDE

Features in configurable software can have dependencies among each other (e.g., one feature might require or exclude another one). To specify the dependencies of features, FeatureIDE provides a feature model editor shown in Figure 1.②. The feature model is the central part of projects in FeatureIDE as it defines the variability of the systems that is used for configuration [9, 27] and analyses (e.g., to detect unused features or dead code) [30].

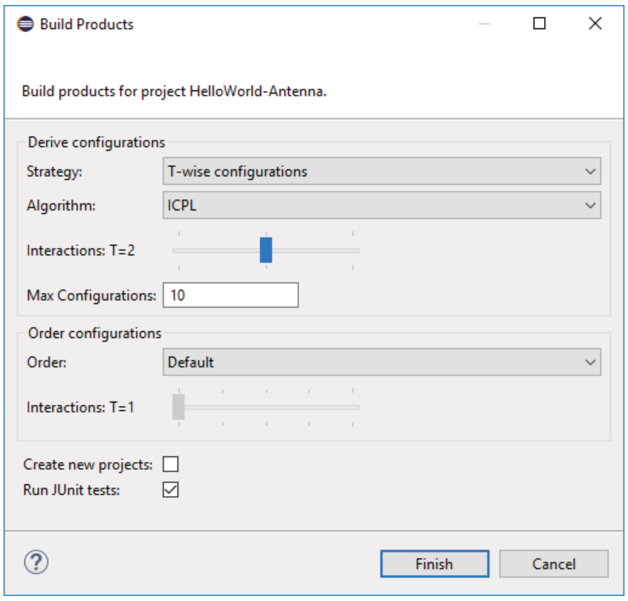
FeatureIDE supports the feature-oriented implementation of configurable software and is designed as an extensible framework [32]. In particular, it supports a variety of implementation mechanisms, such as feature-oriented programming [7], aspect-oriented programming [19], preprocessors [25], and runtime variability.

In our example, we show a configurable program using the integrated preprocessor Antenna (cf. ①). To configure the system, the user can manually define configurations using the configuration editor of FeatureIDE [27]. In the example of Figure 1, there are three *user-defined* configurations in the folder `configs` at ④. Only one configuration can be active at a time, which is then used to preprocess and compile the source code of the `src` folder.

2.2 Derive Configurations

Product-based testing is a technique which generates and tests individual products using an existing testing technique for single systems. For product-based testing and analyses, it is useful to automatically derive configurations from the feature model, because it defines the variability. To automatically derive configurations as well as to generate and test the products, we provide a dialog in which the user can choose how to derive the configurations. We show the dialog in Figure 2. In the following, we discuss the meaning of the options provided in the dialog.

Generation Strategy In FeatureIDE, we support several strategies to provide configurations for testing, namely us-


Build Products

Build products for project HelloWorld-Antenna.

Derive configurations

Strategy: T-wise configurations

Algorithm: ICPL

Interactions: T=2

Max Configurations: 10

Order configurations

Order: Default

Interactions: T=1

Create new projects: ☐

Run JUnit tests: ☑

Figure 2. Dialog to automatically derive and test products.

ing user-defined configurations, deriving all valid configurations, using T-wise sampling, and randomly generating configurations (cf. Figure 2).

User-defined configurations can be created manually using the integrated configuration editor [27]. These configurations are contained in the folder `configs` (cf.④). *All valid* configurations can be generated using an algorithm that exploits the tree structure of the feature model. Generating all valid configurations only scales for systems with a small configuration space. Alternatively, we provide a strategy to generate a fixed number of *random* configurations. Based on the satisfiability solver Sat4J [22], we provide a random configuration generator. The method can efficiently generate a large number of distinct random configurations.

Faults in configurable systems are often caused by interactions of features [21]. T-wise sampling algorithms aim to generate a minimal set of configurations that covers all interactions among T features.

T-Wise Sampling Algorithms In FeatureIDE, we currently integrate four T-wise sampling algorithms, namely CASA [12], Chvatal [11, 17], ICPL [18], and IncLing [1]. The *interaction*-bar can be used to specify the value for T.

- CASA [12] uses simulated annealing to derive configurations. Therefore, it is a non-deterministic algorithm where a different number of configurations may be created for the same configurable system in different runs.
- Chvatal [11, 17] is a heuristic algorithm proposed to create configurations for configurable systems.
- ICPL [18] is based on the Chvatal algorithm with several improvements, such as identifying the invalid feature combinations at an early stage. It generates the T-wise covering array efficiently as it makes use of multithreading.
- IncLing [1] is an algorithm that generates configurations incrementally for pairwise sampling. Instead of providing the complete solution at the end, the algorithm provides the calculated configuration as soon as possible. Thus, testing the configurations does not need to wait until the sampling algorithm has finished.

Maximal Number Of Configurations The user can specify a threshold n for the maximum number of configurations that should be tested. This option is available for all generation strategies. When generating all or a random set of configurations at most n configurations are calculated. For the strategy to use user-defined configurations, only the first n configurations will be tested. For all T-Wise sampling algorithms, we can limit the number of generated configuration to n as well. Thus, a 100% T-Wise interaction coverage might not be reached. As T-Wise algorithms typically cover most interactions in the first configurations, it is still reasonable to give a threshold for T-Wise sampling.

The generated configurations can be built into the products folder as shown in Figure 1.② or into a distinct Eclipse project using the option *create new project* in Figure 2.

2.3 Ordering of Generated Configurations

Optimizing the order of test cases is a good strategy to detect faults early. Therefore, the generated configurations can be *ordered* using one of three different techniques (cf.2). Each generation strategy outputs a list of configurations in a certain order.

To improve the order in which configurations are tested, we currently provide two greedy algorithms. The first one is to order configurations by *dissimilarity* [2]. The algorithm initially selects the configuration where the most features are selected. Then it selects the configuration that is most different to all previously tested configurations. This process is continued until all configurations are tested. The second technique aims to optimize feature *interaction coverage*. For this, the configuration that covers most feature interactions that are not already covered by previous configurations is selected. Again, this process is continued until all configurations are tested, or all interactions are covered (i.e., the number of configurations to test may be smaller). Note that the degree of the interaction coverage to order can be specified using the *interactions* scroll-bar, whereas higher T require more effort for ordering.

2.4 Test Configurations

The last step for testing the system is to execute the test cases. So far, we integrated JUnit to execute tests for Java. When selecting the check box called *Run JUnit tests*, test cases are executed after each product is generated. As shown in Figure 1.①, it is only necessary to annotate the test cases as known from JUnit.

To comprehend the results of testing multiple configurations (i.e., to associate the faults with configurations and to reproduce the fault), we provide a structured tree in the JUnit view as shown in ③. The root elements are the classes that are tested with the configurations as direct children. The leaf elements are the actual test cases. As known from the JUnit view the stack trace of the failing test is shown when selecting the element. Also, the location of the fault will be opened when selecting the entry in the stack trace.

Faults are associated with configurations that cause the defect. However, aggregated results that show, which tests fail under which condition (i.e., a minimal feature selection) would improve the comprehension of the faults [15, 26, 35]. For example, the test case *testStartsWithHello* in Figure 1 fails in all configurations where the feature *Hello* is not selected. The integration of aggregated results is usually non-trivial, especially when only a subset of all configurations is tested. Thus, this improvement is subject to future work.

3. Beyond Product-Based Testing

Product-based testing allows the execution of test cases on a set of configurations. In this section, we show how FeatureIDE provides further strategies to improve testing configurable systems by avoiding redundant test executions and with support for variability-aware testing.

3.1 Avoid Redundant Tests

Multiple testing approaches aim to reduce the number of tests to execute [20]. However, these approaches usually require a specialized infrastructure or domain knowledge. We propose a lightweight approach to improve the time to test multiple configurations.

Unit test cases are usually designed to test a small part of the program, such as single methods or classes. When running the test case on multiple products it is unlikely to get different results, especially if the test case is not affected by variability. Thus, for product-based testing the test case is executed redundantly multiple times causing unnecessary overhead. Instead, the test case should rather be executed only once.

To avoid redundant test executions, we provide a Java annotation for test classes called `@NonInteracting`. While testing multiple configurations as discussed in the previous section, FeatureIDE will execute tests of an annotated class only once. However, the user needs to decide manually whether the test cases do not interact. In the example of Figure 1, the test cases of the class `SingleRunTest` are only executed once as the JUnit view illustrates (cf. Figure 1.③).

3.2 Variability-Aware Testing

Product-based analyses are most common in practice as standard analysis techniques can be used for the analysis of configurable systems. However, this strategy is either unsound (i.e., it may miss faults that could be found by testing other configurations) or does not scale to the high amount of configurations to test. To analyze all configurations, variability-aware mechanisms have been proposed [8, 13, 26, 31, 33, 34]. Variability-aware analyses exploit the fact that the analysis of two similar configurations is also similar. Thus, these redundant calculations when analyzing multiple configurations can be avoided. Variability-aware analyses aim to execute these redundant parts only once to reduce the overall effort to execute all configurations.

Variability-aware testing requires a product simulator (a.k.a. metaproduct) that represents all configurations [5, 33]. This simulator is a transformation of the system into a program with runtime variability, which can simulate all configurations. Currently, FeatureIDE only supports variability-aware analysis for feature-oriented programming with FeatureHouse [4, 5, 33]. The metaproduct can be generated via the project's contextmenu (i.e., *FeatureIDE* → *FeatureHouse* → *Build Metaproduct*). When building the project the metaproduct will be generated instead of a single configuration.

Different tools for variability-aware analysis require special types of feature model classes (i.e., a standard Java class that defines all features and valid combinations thereof). To support these different tools, FeatureIDE enables the user to choose between different model files. We support variability-aware testing with VarexJ [26], model checking with JavaPathfinder [13] and JPF-BDD [34], and theorem proving with KeY [8]. The type of the model class can be selected via the project's properties (i.e., *FeatureIDE* → *Feature Project* → *Metaproduct Generation*). In the future, we aim to provide further support for other implementation techniques, especially for runtime variability, to ease the application of variability-aware testing and analyses.

4. Conclusions

Product-based testing of a configurable system involves creating configurations, generating the corresponding products, and executing their test cases. Within FeatureIDE, we integrate several strategies to automate the testing process. In particular, we enable the user to define configurations, generate a random set of configurations, or generate a subset of configurations using T-wise sampling algorithms. In addition, the user can optimize testing by reordering the generated configurations to detect faults earlier. Furthermore, FeatureIDE provides support to avoid redundant execution of test cases and variability-aware testing. With those integrated strategies and algorithms, we aim to ease the testing of configurable systems regardless the size and the nature of those systems.

Acknowledgments

This work is partially funded by the BMBF grant (01IS14017A, 01IS14017B).

References

[1] M. Al-Hajjaji, S. Krieter, T. Thüm, M. Lochau, and G. Saake. IncLing: Efficient Product-Line Testing Using Incremental Pairwise Sampling. In *GPCE*, 2016. To appear.

[2] M. Al-Hajjaji, T. Thüm, J. Meinicke, M. Lochau, and G. Saake. Similarity-Based Prioritization in Software Product-Line Testing. In *SPLC*, pp. 197–206. ACM, 2014.

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

[4] S. Apel, C. Kästner, and C. Lengauer. Language-Independent and Automated Software Composition: The FeatureHouse Experience. *TSE*, 39(1):63–79, 2013.

[5] S. Apel, A. von Rhein, P. Wendler, A. Größlinger, and D. Beyer. Strategies for Product-Line Verification: Case Studies and Experiments. In *ICSE*, pp. 482–491. IEEE, 2013.

[6] D. Batory. Feature Models, Grammars, and Propositional Formulas. In *SPLC*, pp. 7–20. Springer, 2005.

[7] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *TSE*, 30(6):355–371, 2004.

[8] B. Beckert, R. Hähnle, and P. Schmitt. *Verification of Object-Oriented Software: The KeY Approach*. Springer, 2007.

[9] D. Benavides, S. Segura, and A. Ruiz-Cortés. Automated Analysis of Feature Models 20 Years Later: A Literature Review. *Information Systems*, 35(6):615–708, 2010.

[10] J. Bürdek, M. Lochau, S. Bauregger, A. Holzer, A. von Rhein, S. Apel, and D. Beyer. Facilitating Reuse in Multi-goal Test-Suite Generation for Software Product Lines. In *FASE*, pp. 84–99. Springer, 2015.

[11] V. Chvatal. A Greedy Heuristic for the Set-Covering Problem. *MOR*, 4(3):233–235, 1979.

[12] B. J. Garvin, M. B. Cohen, and M. B. Dwyer. Evaluating Improvements to a Meta-Heuristic Search for Constrained Interaction Testing. *EMSE*, 16(1):61–102, 2011.

[13] K. Havelund and T. Pressburger. Model Checking Java Programs Using Java PathFinder. *STTT*, 2(4):366–381, 2000.

[14] C. Henard, M. Papadakis, G. Perrouin, J. Klein, and Y. L. Traon. PLEDGE: A Product Line Editor and Test Generation Tool. In *SPLC*, pp. 126–129. ACM, 2013.

[15] K. J. Hoffman, P. Eugster, and S. Jagannathan. Semantics-aware Trace Analysis. In *PLDI*, pp. 453–464. ACM, 2009.

[16] M. Jackson and P. Zave. Distributed Feature Composition: A Virtual Architecture for Telecommunications Services. *TSE*, 24(10):831–847, 1998.

[17] M. F. Johansen, Ø. Haugen, and F. Fleurey. Properties of Realistic Feature Models Make Combinatorial Testing of Product Lines Feasible. In *MODELS*, pp. 638–652. Springer, 2011.

[18] M. F. Johansen, Ø. Haugen, and F. Fleurey. An Algorithm for Generating T-Wise Covering Arrays from Large Feature Models. In *SPLC*, pp. 46–55. ACM, 2012.

[19] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *ECOOP*, pp. 220–242. Springer, 1997.

[20] C. H. P. Kim, D. Marinov, S. Khurshid, D. Batory, S. Souto, P. Barros, and M. d'Amorim. SPLat: Lightweight Dynamic Analysis for Reducing Combinatorics in Testing Configurable Systems. In *ESEC/FSE*, pp. 257–267. ACM, 2013.

[21] D. R. Kuhn, D. R. Wallace, and A. M. Gallo Jr. Software Fault Interactions and Implications for Software Testing. *TSE*, 30(6):418–421, 2004.

[22] D. Le Berre and A. Parrain. The sat4j Library, Release 2.2, System Description. *JSAT*, 7:59–64, 2010.

[23] J. McGregor. Testing a Software Product Line. In *Testing Techniques in Software Engineering*, pp. 104–140. Springer, 2010.

[24] F. Medeiros, C. Kästner, M. Ribeiro, S. Nadi, and R. Gheyi. The Love/Hate Relationship with the C Preprocessor: An Interview Study. In *ECOOP*, pp. 495–518. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2015.

[25] J. Meinicke, T. Thüm, R. Schröter, S. Krieter, F. Benduhn, G. Saake, and T. Leich. FeatureIDE: Taming the Preprocessor Wilderness. In *ICSE*, pp. 629–632. ACM, 2016.

[26] J. Meinicke, C.-P. Wong, C. Kästner, T. Thüm, and G. Saake. On Essential Configuration Complexity: Measuring Interactions in Highly-configurable Systems. In *ASE*, pp. 483–494. ACM, 2016.

[27] J. A. Pereira, S. Krieter, J. Meinicke, R. Schröter, G. Saake, and T. Leich. FeatureIDE: Scalable Product Configuration of Variable Systems. In *ICSR*, pp. 397–401. Springer, 2016.

[28] G. Perrouin, S. Sen, J. Klein, B. Baudry, and Y. Le Traon. Automated and Scalable T-Wise Test Case Generation Strategies for Software Product Lines. In *ICST*, pp. 459–468. IEEE, 2010.

[29] M. Steffens, S. Oster, M. Lochau, and T. Fogdal. Industrial Evaluation of Pairwise SPL Testing with MoSo-PoLiTe. In *VaMoS*, pp. 55–62. ACM, 2012.

[30] R. Tartler, D. Lohmann, J. Sincero, and W. Schröder-Preikschat. Feature Consistency in Compile-Time-Configurable System Software: Facing the Linux 10,000 Feature Problem. In *EuroSys*, pp. 47–60. ACM, 2011.

[31] T. Thüm, S. Apel, C. Kästner, I. Schaefer, and G. Saake. A Classification and Survey of Analysis Strategies for Software Product Lines. *CSUR*, 47(1):6:1–6:45, 2014.

[32] T. Thüm, C. Kästner, F. Benduhn, J. Meinicke, G. Saake, and T. Leich. FeatureIDE: An Extensible Framework for Feature-Oriented Software Development. *SCP*, 79(0):70–85, 2014.

[33] T. Thüm, J. Meinicke, F. Benduhn, M. Hentschel, A. von Rhein, and G. Saake. Potential Synergies of Theorem Proving and Model Checking for Software Product Lines. In *SPLC*, pp. 177–186. ACM, 2014.

[34] A. von Rhein, S. Apel, and F. Raimondi. Introducing Binary Decision Diagrams in the Explicit-State Verification of Java code. In *JavaPathfinder Workshop*, 2011.

[35] B. Xin, W. N. Sumner, and X. Zhang. Efficient Program Execution Indexing. In *PLDI*, pp. 238–248. ACM, 2008.