Towards Automated Test Refactoring for Software Product Lines

Jacob Krüger
Harz University of Applied Sciences
Otto-von-Guericke-University
Wernigerode & Magdeburg, Germany
jkrueger@ovgu.de

Mustafa Al-Hajjaji
Otto-von-Guericke-University
pure-systems GmbH
Magdeburg, Germany
mustafa.alhajjaji@pure-systems.com

Sandro Schulze
Otto-von-Guericke-University
Magdeburg, Germany
sanschul@ovgu.de

Gunter Saake
Otto-von-Guericke-University
Magdeburg, Germany
saake@ovgu.de

Thomas Leich
Harz University of Applied Sciences
METOP GmbH
Wernigerode & Magdeburg, Germany
tleich@hs-harz.de

ABSTRACT

In practice, organizations often rely on the clone-and-own approach to reuse and customize existing systems. While increasing maintenance costs encourage some organizations to adopt their development processes towards more systematic reuse, others still avoid migrating to a reusable platform. Based on our experiences, a barrier preventing the adoption of software product lines is the fear of introducing new and more problematic bugs—during the migration or later on. We are aware of several works that automate software-product-line adoption, but they neglect the migration and maintenance of test cases. Automating the refactoring of tests can help to facilitate the adoption barrier, compare the quality after migrations, and support maintenance. In this vision paper, we *i)* discuss open research challenges that are based on our experiences and *ii)* sketch a first framework to develop automated solutions. Overall, we aim to illustrate our idea and initiate further research to facilitate the adoption and maintenance of software product lines.

CCS CONCEPTS

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

KEYWORDS

Software product line, testing, legacy system, extractive approach, migration, maintenance

ACM Reference Format:
Jacob Krüger, Mustafa Al-Hajjaji, Sandro Schulze, Gunter Saake, and Thomas Leich. 2018. Towards Automated Test Refactoring for Software Product Lines. In *SPLC '18: 22nd International Systems and Software Product Line Conference, September 10–14, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3233027.3233040

SPLC '18, September 10–14, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-6464-5/18/09…$15.00
https://doi.org/10.1145/3233027.3233040

1 INTRODUCTION

Organizations often develop a customized system by cloning an existing one to fulfill new customer requirements [4, 15, 38]. This *clone-and-own* approach is a simple and initially cheap reuse strategy [19, 39]. However, with an increasing number of clones, the efforts for development and maintenance can increase [4, 39], for example, due to propagating changes and bug fixes [35]. For these reasons, an organization may consider to migrate its cloned systems towards a *software product line* to reduce costs [11, 26]. Such migrations are referred to as *extractive approach* [25], which is the most common software-product-line adoption strategy in practice [8, 16].

During the extractive adoption, legacy systems are partitioned and migrated into reusable *features* [4, 25, 27]. These features implement commonalities and variabilities of the systems and are used to configure and instantiate a customized system [4, 5, 12]. Different surveys [18, 28] show that several approaches have been proposed to automate such extractions and, thus, reduce their risks and costs [11, 26]. Still, from our experience, some organizations do not initiate a migration process, because they fear to introduce new bugs into the resulting software product line. Form their perspective, the promised benefits do not outweigh the necessary investments to ensure correctness and testability.

To tackle this problem, we propose to not only rely on refactoring [20] to migrate the legacy systems and their test cases, but also to implement an automated mapping between them. Based on this mapping, we can identify cloned tests and those that may need to be updated after code is extracted. For instance, a feature's implementation from different legacy systems may be merged. Here, the question arises, which of the corresponding tests is the best suitable to test the merged source code and which adaptations are required? In this vision paper, we sketch a corresponding adoption strategy and its challenges, arguing that this can help to facilitate the adoption barrier towards software product lines [11]. As an existing software product line is also updated, this mapping can further support the maintenance phase. Thus, while our focus is on the extractive adoption, our idea can also facilitate software-product-line engineering in general. In detail, we describe:

- Using a running example, we explore the importance of refactoring tests while extracting a software product line.
- Based on the running example, we discuss arising challenges for test refactoring during migrations of legacy systems.

Listing 1 Basic buffer without restoring.

```
public class Buffer {
  int buf = 0;
  int get() {
    return buf;
  }
  void set(int x) {
    buf = x;
  }
}
```

Listing 2 Buffer extended with a single-value restore.

```
public class Buffer {
  int buf = 0;
  int back = 0;
  int get() {
    return buf;
  }
  void set(int x) {
    back = buf;
    buf = x;
  }
  void restore() {
    buf = back;
  }
}
```

Listing 3 Buffer with a stack to restore multiple values.

```
public class Buffer {
  int buf = 0;
  Stack<Integer> back = new Stack<Integer>();
  int get() {
    return buf;
  }
  void set(int x) {
    back.push(buf);
    buf = x;
  }
  void restore() {
    buf = back.pop();
  }
}
```

Figure 1: Cloned implementations of a buffer based on the example by Liu et al. [30].

- We sketch an initial framework to track code refactorings and map them towards the corresponding tests.

Based on this vision, we hope to encourage further research to investigate test refactoring for software-product-line engineering. We argue that such an approach can facilitate the practical adoption of software product lines by ensuring that the test suite can always be used to validate the code. While we solely focus on the extraction of test cases in this paper, the derived challenges also apply to software-product-line evolution. Thus, this work may additionally foster future research in this direction. Furthermore, while we only use unit tests [40] as examples, the challenges and proposed framework can also be adopted for other test artifacts.

2 PROBLEM STATEMENT

For a set of cloned systems, an organization may have different types of tests, for instance, unit tests [40]. These tests are used for all or only for a subset of the legacy systems. Due to migrating the legacy systems towards a software product line, these legacy tests may not be applicable anymore. For example, the code may be restructured, could require different input, or create changed output. Consequently, developers have to analyze refactorings and adopt existing tests accordingly—also considering replicated and customized tests. We argue that new analysis and adoption strategies for automatic test migration reduce the adoption barrier [11].

Listing 4: Unit tests for Listing 2.

```
import static org.junit.Assert.assertTrue;

public class Test {
  Buffer buf = new Buffer();

  @org.junit.Test
  public void bufTest() throws Exception {
    buf.set(42);
    assertTrue(buf.get() == 42);
  }

  @org.junit.Test
  public void backTest() throws Exception {
    buf.set(42);
    buf.set(24);
    buf.restore();
    assertTrue(buf.get() == 42);
  }
}
```

Extracting a software product line from cloned systems requires developers to analyze the existing code and migrate it into reusable features [4, 18, 25]. Still, this can be an error-prone and costly process [11, 26] and, based on our experience, organizations often fear the possibility to introduce new bugs. To address such issues, refactorings have been proposed that migrate clones towards a software product line while ensuring consistency and reducing the necessary effort [17, 23, 41]. However, refactoring cloned systems (or a software product line during maintenance) can easily yield to breaking the corresponding test cases. To this end, we propose to extend *code refactorings* with *test case mapping*. Especially for the extractive approach, we also have to avoid *test case replications*.

In the following, we will discuss these three aspects based on a running example: We display three customized implementations of a simple buffer manager in Figure 1 based on the work of Liu et al. [30]. The implementation in Listing 1 only stores a single value in a variable and can return it. Listing 2 and Listing 3 are extended versions that store a single or multiple former values, respectively.

Code Refactoring. The source code of legacy systems is migrated into a suitable domain implementation. During this integration of multiple systems into a single code base, several changes in the source code may be necessary, for example, due to the used variability mechanism or added glue code [4, 29, 42]. Still, already for a single system, refactoring source code can result in broken test cases [10]. Refactoring and integrating multiple legacy systems while introducing a variability mechanism in parallel is even more challenging, posing additional problems [31]. Thus, refactoring test cases is essential while extracting a software product line.

To exemplify this, we use the unit tests we display in Listing 4 that test *setting* the buffer value and *restoring* it for the implementation in Listing 2—cloned versions for the other two implementations are omitted. Assume we extract all three restoring capabilities (i.e., none, a variable, a stack) into different features. Based on the

variability mechanism, the test cases also require refactoring. For example, using preprocessors, all annotations in the base code should also be added to the test cases, which also have to be merged. Another possibility would be to use plug-ins or interfaces (e.g., similar to Java's comparable interface), when the feature implementation would heavily change. This would also require adapting the test cases to fit to the changed architecture.

Test Case Mapping. An issue closely related to the actual refactoring is the mapping of test cases to code and, thus, configurations. Different test cases may be merged, discarded, or changed depending on the extracted features. Thus, we have to map which test case belongs to which feature. Otherwise, the test cases will hardly be applicable on the extracted software product line. This may be done by implementing variability-aware test cases, meaning that the variability is also transferred into the test cases.

For example, the test case for the reversing operation we show in Listing 4 will only be applicable in some configurations. It will fail whenever the value cannot be restored. In addition, it does not cover the complete functionality of the stack implementation, wherefore an additional test may be necessary for that feature—to test the actual function of the stack to store multiple values. Depending on the used variability mechanism, the mapping can be closely related to the actual code refactoring. For instance, preprocessor annotations would already implement the mapping and only have to be used correctly [4].

Test Case Replications. The legacy test cases may not be the same for all variants, as customizations in one variant can require adapted or completely new test cases. Thus, for migrations we have to consider two possibilities: First, we have to identify and remove duplicated test cases to prevent unnecessary effort. Second, we have to analyze adapted test cases to identify and manage differences—preventing contradicting results and faulty test cases.

For instance, the first test case, which assesses whether the buffer is set, in Listing 4 is identical for all variants in Figure 1, wherefore only one of all potential duplicates must be kept. However, the test cases that check the restore functionality may be partly duplicated, but contain differences. Considering our example, for Listing 2 and Listing 3 we need to integrate the tests and implement variable testing, depending on whether a single value or a stack is used for restoring. Keeping all test cases may also cause errors if only specific implementations are kept, for example, because the variable `back` in Figure 1 is defined differently.

3 CHALLENGES

In the following, we describe six challenges that arise due to the aforementioned situation and discuss their implications.

CH_1 *How can we map test cases to features?*
As a set of legacy systems is merged into a single domain implementation, it is necessary to identify which test corresponds to which feature and, thus, configuration. Otherwise, it will hardly be possible to apply the tests correctly and avoid unnecessary test efforts. Approaching this issue will facilitate the extraction of product lines and ease the adoption of an appropriate quality assurance. Such a mapping can be based on the variability mechanism, for instance, by using the same preprocessor annotations (cf. Section 2).

CH_2 *How can we map code refactorings to the corresponding tests?*
When source code is refactored into a software product line, it is essential for developers to know which test cases are affected by these changes. This way, they can evaluate the impact of their changes and the necessity to adapt existing test cases. Thus, addressing this challenge helps developers to analyze and manage tests during the migration process. Considering our running example, we may decide to refactor Listing 1 and Listing 3 into features. Consequently, we have to identify that the corresponding test cases require updates, such as, removing duplicates (e.g., first test case in Listing 4), discarding inappropriate tests (e.g., second test case in Listing 4), and adapting variability for other test cases (e.g., clone of the second test case in Listing 4 for the stack implementation).

CH_3 *How can we measure similarity between test cases?*
To identify which test cases are replications or adaptations, it is necessary to measure their similarity. For this, suitable analysis techniques are necessary to support developers in identifying test cases, and their differences, that must be refactored. As a result, the effort of extracting software product lines can be reduced, lowering the adoption barrier [11]. Measuring similarity may be done based on clone detection (e.g., for the first test case in Listing 4) or execution traces, but requires certain criteria and metrics.

CH_4 *How and when do we derive new test cases?*
Identical to refactorings in a single system [34], the extraction of software product lines from legacy systems can result in new and changed implementations. Thus, additional test cases may be necessary to cover such changes. In this context, automated test case generation can support developers to facilitate the extraction process. For example, in Figure 1 the set method of Listing 2 and Listing 3 may be merged and receive a second variable to determine each restoring functionality at runtime. To this end, a new variable is introduced and requires additional test cases, next to the necessary updates.

CH_5 *How do we migrate test cases to suit a software product line?*
There are some works that address the refactoring of test cases for a single system [10, 14, 34]. However, when extracting a software product line, we may be able to benefit from analyzing existing test cases and use automation to some degree. Therefore, we have to investigate how we can merge and adapt similar test cases to the resulting software product line. For instance, based on the code refactorings and their mappings to the test cases, it may be possible to automatically merge them to some extent. In Listing 4, we could imagine that cloned test cases of the restoring function (e.g., for the stack implementation in Listing 3) are automatically merged.

CH_6 *How can we evaluate the quality of refactored test cases?*
For each refactored test case, the question arises whether it is suitable or requires further adaptations. To address this point, evaluations for test cases are necessary. This way, a defined degree of quality for the extracted test cases can be guaranteed to also ensure the quality of the resulting software product line. Considering Listing 4, we may see that the second test case would fulfill the same requirements for Listing 2, but not for Listing 3. Thus, we could identify issues in the resulting code and also support the previous challenges.

Addressing the aforementioned challenges and combining their outcomes seems promising to facilitate the extraction and evolution of software product lines.

4 TOWARDS A FRAMEWORK

In this section, we sketch our initial idea on managing legacy test cases while extracting a software product line. We depict the corresponding process in Figure 2. To explain our idea, we use *abstract syntax trees* [44] as representations of the legacy systems' source code on a model-based level. We remark that other representations, such as *dependency graphs* or *execution traces*, or even other approaches may be better suited to address some of the aforementioned challenges. The benefits of abstract syntax trees are that they only require static analysis and allow to compare variants on code level. However, they do not comprise information about a variants behavior at execution time and can only be indirectly mapped to test cases—while dynamic representations allow to execute tests and, thus, map them to code. For now, we base our framework on abstract syntax trees, but we arguably have to explore other representations to evaluate them and enable automation.

The root of an abstract syntax tree can represent a whole system, with its children illustrating packages, import declarations, down to classes and statements. By adapting approaches to analyze these representations, for example by Baxter et al. [6] and Neamtiu et al. [33], we aim to collect the necessary information to automate and, thus, facilitate the migration of legacy test cases. In the following, we describe the different steps that we depict in Figure 2.

Extracting Abstract Syntax Trees. At the beginning of our process, we extract an abstract syntax tree from each existing legacy system. Then, we have to execute the existing test cases to map their executions onto the abstract syntax trees. Thus, we identify all relevant nodes and have an abstract representation of the test coverage [47] (i.e., based on statements) that is achieved for the legacy systems. We aim to use these information on test coverage to track refactorings from the legacy systems towards the extracted software product line and assess their impact. To this end, we assume that any refactoring of the source code can also be represented in the abstract syntax tree, for example as proposed by Behringer and Rothkugel [7], and nodes are either inserted, removed, or replaced. We track if such modifications change the coverage of the system or a specific test case (e.g., nodes are removed that belonged to its execution trace), for instance, if a feature is derived we add a new node representing this feature. This way, we address the first (CH_1) and second challenge (CH_2) we discussed in Section 3. Considering our example in Figure 1, we would map the first test case (and its clones) of Listing 4 to each of the legacy implementations. Thus, any change in these parts can be mapped accordingly.

Aggregating Abstract Syntax Trees. In the next step (cf. Figure 2), we aggregate the derived abstract syntax trees into a single tree, based on comparing and transforming the representations [13]. The purpose of this aggregation is to identify test cases that cover identical or similar parts (e.g., statements) of the legacy systems. Thus, we can reduce the number of considered test cases by removing replications. Additionally, we can also identify those tests that are affected by the same changes but still contain variability. These may cover specific features in the resulting software product line. As a result, we have a reduced set of test cases for the further process and identified relations between them, addressing our third research challenge (CH_3). For instance, in the previous step we mapped the first test case of Listing 4 and its clones to three different legacy implementations. Based on comparing the abstract syntax trees, we could identify that they actually cover identical code and remove two of them—and updating the mappings.

Mapping Refactorings. The third step of our framework is to map source code refactorings onto an abstract syntax tree that represents the resulting software product line. For this purpose, developers can either start from scratch or base it on the abstract syntax tree of a specific legacy system. By mapping refactorings with model transformations and using the previously merged abstract syntax tree, we can identify new, modified, and removed nodes that may affect test cases. In Figure 2, we exemplify such situations with red nodes. We remark that we consider legacy nodes that are only appended to the software product line's abstract syntax tree as unchanged. Based on the change, we can consider:

(1) If a test case covers only unchanged nodes, it can be applied without further adaptations.
(2) If a test case covers modified nodes, it must be adapted to the new structure.
(3) If a test case covered removed nodes, it must be adapted if possible or potentially discarded.
(4) If no test case covers a node, it might be necessary to derive a completely new one.

By extending this analysis, we can address our fourth research challenge (CH_4) and decide for which nodes we need new or adapted test cases. For our example, we can imagine that the getter, for which we identified one test case, may be updated to return the whole buffer object instead of the last value. Thus, corresponding nodes in the abstract syntax tree would be modified and the second scenario would apply to the mapped test case.

Refactoring Test Cases. Depending on the decisions a developer derives from the previous assessment, test cases must be derived or refactored. Besides utilizing existing work on regression testing [46], approaches for single systems [10, 14] can be adopted to address the fifth challenge (CH_5). In particular, we may be able to extend them based on semantic knowledge of the applied software-product-line refactorings to automatically extend the test suite. For instance, on a formal level of pre- and post-conditions, if these conditions are derived automatically for new program elements (i.e., a feature) during the refactoring, it may be possible to automatically generate new tests for this particular element. Another point to address is, whether we can automatically join and combine legacy test cases to cover more extensive refactorings. Currently, we envision a less automated approach to support developers in this step of the process. Again, assuming that the previously exemplified test case requires an update, adapting it to test the object may be partly possible. Nonetheless, while we may be able to define rules that replace the equality operators in Listing 4 with testing for an instance, this may not be enough to derive a proper test case.

Evaluating Test Cases. One way to assess the quality of test cases, especially new and refactored ones, of the extracted software product line is to use mutation testing [22]. By using conventional mutation operators and operators for software product lines [1,

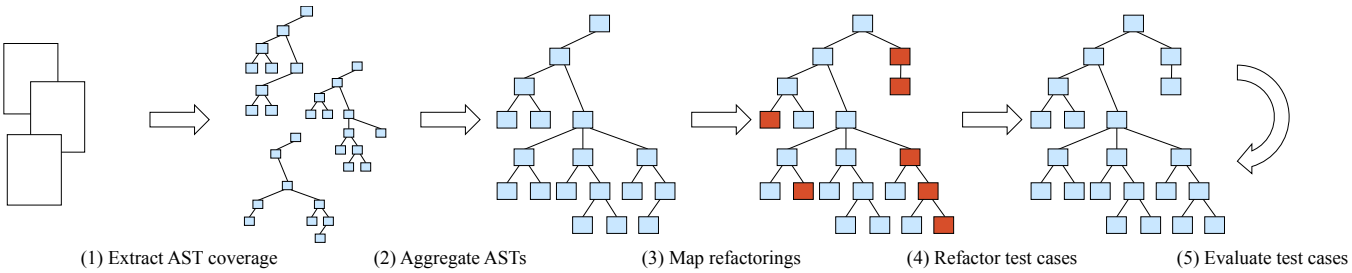

Figure 2: Sketch of the envisioned framework. The abstract syntax tree structures represent the source code of the legacy systems that are then migrated to a software product line.

2], we execute test cases against generated mutants that contain faults [9]. We can then ensure a certain quality of these test cases based on two assessments: How well do the test cases perform on the extracted software product line and is their quality comparable to the legacy counterparts (i.e., if they have been adapted)? Still, while we can automatically provide some information on the quality of test cases, it is also an expensive means [2, 22] and cannot fully replace a manual assessment. Nonetheless, mutation testing is one possible way to address our sixth challenges (CH_6). Considering our previous example, a return values mutant for the getter in Listing 2 would return a zero instead of the variable, which would be revealed by the legacy test. For our refactored code, the same operator would return `null` instead of the buffer object. A refactored test case can also reveal this by checking the instance of the object. Consequently, we could assume the same quality of the test cases for this method, as they find the same mutant. Still, due to the high costs of mutation testing, ensuring correctness based on the refactorings is important.

Summary. We presented an initial framework to manage test-case refactoring during software-product-line extraction and exemplified it based on our running example. While we are focusing on abstract syntax trees and, thus, a representation close to the source code, other representations also seem necessary. For instance, using model-based software testing [13, 43] may help to improve automation on test case analysis and creation.

5 RELATED WORK

Several approaches have been proposed to extract systems and their different artifacts into software product lines [18, 28]. For example, Alves et al. [3], Rubin and Chechik [37], and Xue [45] all propose to extract a software product line from similar legacy systems based on analyzing corresponding models. These approaches mainly utilize the idea of combining different variability models (e.g., feature models) and partly extend their approaches with, for instance, dependency graphs and information retrieval. Similarly, Koschke et al. [24] compose the architecture of a software product line by combining architectural models of legacy systems. Our idea to utilize abstract syntax trees to model the execution of test cases and the affected program elements builds on similar concepts. However, in contrast to existing approaches, we focus on extracting test cases rather than the systems themselves and, thus, complement these works. In addition, it is possible to automatically extract abstract syntax trees, while most other approaches require experts to derive the necessary models.

Other authors propose approaches to identify and extract commonalities of legacy systems. For example, Mende et al. [32], Duszynski et al. [16], and Krüger et al. [27] utilize code-clone detection [36] rather then model comparison. This way, they identify and measure similarity and variability in legacy systems solely based on the source code. Moreover, several authors [17, 23, 41] propose refactorings to extract source code into a software product line. These approaches focus on analyzing and refactoring source code—including test cases, but not on the mapping between those. We complement these works and may use similar ideas, for example, using code-clone detection on abstract syntax trees [6], to derive additional information and integrate our own approach into these.

For single system development, Guerra and Fernandes [21] observe that refactoring test cases is different from refactoring source code. To this end, Passier et al. [34] propose to maintain unit tests during refactorings by tracking source code changes and reflecting these to the test cases. Similarly, Chu et al. [10] describe an approach to guide the refactoring of test cases for design patterns. In contrast to these approaches, we focus on the refactoring of multiple legacy tests to be suitable for an extracted software product line. Thus, we can use, but have to extend and refine, such works.

6 CONCLUSION

Several approaches have been proposed to extract artifacts of cloned legacy systems into a software product line. Most focus on modeling the systems and source code, but rarely consider other artifacts. In particular, we argue that it is necessary to improve the management and refactoring of legacy test cases, which is already challenging in single system development. Otherwise, ensuring the quality and testability of an extracted software product line can hardly be ensured. In this paper, we discussed the problems and challenges we identified in the context of refactoring legacy test cases. Furthermore, we proposed an initial concept for managing and automating this process based on abstract syntax trees.

In future work, we plan to implement the sketched framework. To this end, we especially have to develop a suitable mapping from legacy source code (i.e., abstract syntax trees) to the test cases and an update mechanism for refactorings. Additionally, we aim to investigate further automation and to perform case studies.

Acknowledgments This research is supported by the German Research Foundation (DFG; LE 3382/2-1, SA 465/49-1), the German Federal Ministry of Education and Research (BMBF; 01|S16043N), and Volkswagen Financial Services AG.

REFERENCES

[1] Mustafa Al-Hajjaji, Fabian Benduhn, Thomas Thüm, Thomas Leich, and Gunter Saake. 2016. Mutation Operators for Preprocessor-Based Variability. In *International Workshop on Variability Modelling of Software-Intensive Systems*. ACM, 81–88.

[2] Mustafa Al-Hajjaji, Jacob Krüger, Fabian Benduhn, Thomas Leich, and Gunter Saake. 2017. Efficient Mutation Testing in Configurable Systems. In *International Workshop on Variability and Complexity in Software Design*. IEEE, 2–8.

[3] Vander Alves, Rohit Gheyi, Tiago Massoni, Uirá Kulesza, Paulo Borba, and Carlos Lucena. 2006. Refactoring Product Lines. In *International Conference on Generative Programming: Concepts and Experiences*. ACM, 201–210.

[4] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines*. Springer.

[5] Sven Apel and Christian Kästner. 2009. An Overview of Feature-Oriented Software Development. *Journal of Object Technology* 8, 5 (2009), 49–84.

[6] Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. 1998. Clone Detection Using Abstract Syntax Trees. In *International Conference on Software Maintenance*. 368–377.

[7] Benjamin Behringer and Steffen Rothkugel. 2016. Integrating Feature-Based Implementation Approaches Using a Common Graph-based Representation. In *ACM Symposium on Applied Computing*. ACM, 1504–1511.

[8] Thorsten Berger, Ralf Rublack, Divya Nair, Joanne M. Atlee, Martin Becker, Krzysztof Czarnecki, and Andrzej Wąsowski. 2013. A Survey of Variability Modeling in Industrial Practice. In *International Workshop on Variability Modelling of Software-Intensive Systems*. ACM, 7:1–7:8.

[9] Luiz Carvalho, Marcio Augusto Guimarães, Márcio Ribeiro, Leonardo Fernandes, Mustafa Al-Hajjaji, Rohit Gheyi, and Thomas Thüm. 2018. Equivalent Mutants in Configurable Systems: An Empirical Study. In *International Workshop on Variability Modelling of Software-Intensive Systems*. ACM, 11–18.

[10] Peng-Hua Chu, Nien-Lin Hsueh, Hong-Hsiang Chen, and Chien-Hung Liu. 2012. A Test Case Refactoring Approach for Pattern-Based Software Development. *Software Quality Journal* 20, 1 (2012), 43–75.

[11] Paul C. Clements and Charles W. Krueger. 2002. Point/Counterpoint: Being Proactive Pays Off/Eliminating the Adoption Barrier. *IEEE Software* 19, 4 (2002), 28–30.

[12] Paul C. Clements and Linda M. Northrop. 2001. *Software Product Lines: Practices and Patterns*. Addison-Wesley.

[13] Krzysztof Czarnecki and Simon Helsen. 2006. Feature-Based Survey of Model Transformation Approaches. *IBM Systems Journal* 45, 3 (2006), 621–645.

[14] Arie van Deursen, Leon Moonen, Alex van den Bergh, and Gerard Kok. 2002. *Extreme Programming Perspectives*. Addison-Wesley, Chapter Refactoring Test Code, 141–152.

[15] Yael Dubinsky, Julia Rubin, Thorsten Berger, Slawomir Duszynski, Martin Becker, and Krzysztof Czarnecki. 2013. An Exploratory Study of Cloning in Industrial Software Product Lines. In *European Conference on Software Maintenance and Reengineering*. IEEE, 25–34.

[16] Slawomir Dus-zynski, Jens Knodel, and Martin Becker. 2011. Analyzing the Source Code of Multiple Software Variants for Reuse Potential. In *Working Conference on Reverse Engineering (WCRE)*. IEEE, 303–307.

[17] Wolfram Fenske, Jens Meinicke, Sandro Schulze, Steffen Schulze, and Gunter Saake. 2017. Variant-Preserving Refactorings for Migrating Cloned Products to a Product Line. In *International Conference on Software Analysis, Evolution and Reengineering*. IEEE, 316–326.

[18] Wolfram Fenske, Thomas Thüm, and Gunter Saake. 2013. A Taxonomy of Software Product Line Reengineering. In *International Workshop on Variability Modelling of Software-Intensive Systems*. ACM, 1–8.

[19] Stefan Fischer, Lukas Linsbauer, Roberto Erick Lopez-Herrejon, and Alexander Egyed. 2014. Enhancing Clone-And-Own with Systematic Reuse for Developing Software Variants. In *International Conference on Software Maintenance and Evolution*. IEEE, 391–400.

[20] Martin Fowler and Kent Beck. 1999. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley.

[21] Eduardo Martins Guerra and Clovis Torres Fernandes. 2007. Refactoring Test Code Safely. In *International Conference on Software Engineering Advances*. IEEE, 44–49.

[22] Yue Jia and Mark Harman. 2011. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering* 37, 5 (2011), 1–31.

[23] Jongwook Kim, Don Batory, and Danny Dig. 2017. Refactoring Java Software Product Lines. In *International Systems and Software Product Line Conference*. ACM, 59–68.

[24] Rainer Koschke, Pierre Frenzel, Andreas P. J. Breu, and Karsten Angstmann. 2009. Extending the Reflexion Method for Consolidating Software Variants into Product Lines. *Software Quality Journal* 17, 4 (2009), 331–366.

[25] Charles W. Krueger. 2002. Easing the Transition to Software Mass Customization. In *International Workshop on Software Product-Family Engineering*. Springer, 282–293.

[26] Jacob Krüger, Wolfram Fenske, Jens Meinicke, Thomas Leich, and Gunter Saake. 2016. Extracting Software Product Lines: A Cost Estimation Perspective. In *International Systems and Software Product Line Conference*. ACM, 354–361.

[27] Jacob Krüger, Louis Nell, Wolfram Fenske, Gunter Saake, and Thomas Leich. 2017. Finding Lost Features in Cloned Systems. In *International Systems and Software Product Line Conference*. ACM, 65–72.

[28] Miguel A. Laguna and Yania Crespo. 2013. A Systematic Mapping Study on Software Product Line Evolution: From Legacy System Reengineering to Product Line Refactoring. *Science of Computer Programming* 78, 8 (2013), 1010–1034.

[29] Frank van der Linden, Klaus Schmid, and Eelco Rommes. 2007. *Software Product Lines in Action*. Springer.

[30] Jia Liu, Don Batory, and Christian Lengauer. 2006. Feature Oriented Refactoring of Legacy Applications. In *International Conference on Software Engineering*. ACM, 112–121.

[31] John D. McGregor, Prakash Sodhani, and Sai Madhavapeddi. 2004. Testing Variability in a Software Product Line. In *International Workshop on Software Product Line Testing*. Avaya Labs, 45–50.

[32] Thilo Mende, Felix Beckwermert, Rainer Koschke, and Gerald Meier. 2008. Supporting the Grow-and-Prune Model in Software Product Lines Evolution Using Clone Detection. In *European Conference on Software Maintenance and Reengineering*. IEEE, 163–172.

[33] Iulian Neamtiu, Jeffrey S. Foster, and Michael Hicks. 2005. Understanding Source Code Evolution Using Abstract Syntax Tree Matching. *ACM SIGSOFT Software Engineering Notes* 30, 4 (2005), 1–5.

[34] Harrie Passier, Lex Bijlsma, and Christoph Bockisch. 2016. Maintaining Unit Tests During Refactoring. In *International Conference on Principles and Practices of Programming on the Java Platform*. ACM, 18:1–18:6.

[35] Tristan Pfofe, Thomas Thüm, Sandro Schulze, Wolfram Fenske, and Ina Schaefer. 2016. Synchronizing Software Variants with Variantsync. In *International Systems and Software Product Line Conference*. ACM, 329–332.

[36] Chanchal K. Roy, James R. Cordy, and Rainer Koschke. 2009. Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach. *Science of Computer Programming* 74, 7 (2009), 470–495.

[37] Julia Rubin and Marsha Chechik. 2012. Combining Related Products into Product Lines. In *International Conference on Fundamental Approaches to Software Engineering*. Springer, 285–300.

[38] Julia Rubin and Marsha Chechik. 2013. A Framework for Managing Cloned Product Variants. In *International Conference on Software Engineering*. IEEE, 1233–1236.

[39] Julia Rubin, Andrei Kirshin, Goetz Botterweck, and Marsha Chechik. 2012. Managing Forked Product Variants. In *International Systems and Software Product Line Conference*. ACM, 156–160.

[40] Per Runeson. 2006. A Survey of Unit Testing Practices. *IEEE Software* 23, 4 (2006), 22–29.

[41] Sandro Schulze, Malte Lochau, and Saskia Brunswig. 2013. Implementing Refactorings for FOP: Lessons Learned and Challenges Ahead. In *International Workshop on Feature-Oriented Software Development*. ACM, 33–40.

[42] Mikael Svahnberg, Jilles van Gurp, and Jan Bosch. 2005. A Taxonomy of Variability Realization Techniques. *Software: Practice and Experience* 35, 8 (2005), 705–754.

[43] Mark Utting, Alexander Pretschner, and Bruno Legeard. 2012. A Taxonomy of Model-Based Testing Approaches. *Software Testing, Verification and Reliability* 22, 5 (2012), 297–312.

[44] David S. Wile. 1997. Abstract Syntax from Concrete Syntax. In *International Conference on Software Engineering*. ACM, 472–480.

[45] Yinxing Xue. 2013. *Reengineering Legacy Software Products Into Software Product Line*. Ph.D. Dissertation. University of Singapore.

[46] Shin Yoo and Mark Harman. 2012. Regression Testing Minimization, Selection and Prioritization: A Survey. *Software Testing, Verification and Reliability* 22, 2 (2012), 67–120.

[47] Hong Zhu, Patrick A. V. Hall, and John H. R. May. 1997. Software Unit Test Coverage and Adequacy. *Comput. Surveys* 29, 4 (1997), 366–427.