Product Sampling for Product Lines: The Scalability Challenge

Tobias Pett
t.pett@tu-braunschweig.de
TU Braunschweig
Germany

Thomas Thüm
t.thuem@tu-braunschweig.de
TU Braunschweig
Germany

Tobias Runge
tobias.runge@tu-braunschweig.de
TU Braunschweig
Germany

Sebastian Krieter
sebastian.krieter@ovgu.de
University of Magdeburg
Germany

Malte Lochau
malte.lochau@es.tu-darmstadt.de
TU Darmstadt
Germany

Ina Schaefer
i.schaefer@tu-braunschweig.de
TU Braunschweig
Germany

ABSTRACT

Quality assurance for product lines is often infeasible for each product separately. Instead, only a subset of all products (i.e., a sample) is considered during testing such that at least the coverage of certain feature interactions is guaranteed. While pair-wise interaction sampling only covers all interactions between two features, its generalization to t-wise interaction sampling ensures coverage for all interactions among t features. However, sampling large product lines poses a challenge, as today's algorithms tend to run out of memory, do not terminate, or produce samples, which are too large to be tested. To initiate a community effort, we provide a set of large real-world feature models with up-to 19 thousand features, which are supposed to be sampled. The performance of sampling approaches is evaluated based on the CPU time and memory consumed to retrieve a sample, the sample size for a given coverage (i.e. the value of t) and whether the sample achieves full t-wise coverage. A well-performing sampling algorithm achieves full t-wise coverage, while minimizing the other properties as best as possible.

CCS CONCEPTS

• **Software and its engineering → Software product lines**.

KEYWORDS

software product lines, product line testing, product sampling, real-world feature models

ACM Reference Format:
Tobias Pett, Thomas Thüm, Tobias Runge, Sebastian Krieter, Malte Lochau, and Ina Schaefer. 2019. Product Sampling for Product Lines: The Scalability Challenge. In *23rd International Systems and Software Product Line Conference - Volume A (SPLC '19), September 9–13, 2019, Paris, France.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3336294.3336322

SPLC '19, September 9–13, 2019, Paris, France

ACM ISBN 978-1-4503-7138-4/19/09…$15.00
https://doi.org/10.1145/3336294.3336322

1 INTRODUCTION

Modern software engineering struggles with increasing requirements regarding finer customization and faster time to market. A common solution is to design systems as software product lines. This way a collection of customized products can be build based on a common core [3]. However, the variability contained in a software product line poses a potentially large number of possible products. Analyzing all products individually is infeasible in most cases [30]. Therefore, quality assurance is often performed on a small subset of product configurations, which presumably covers a sufficient amount of functionality of the product line.

A common technique to build a subset of products is *combinatorial interaction sampling (CIT)* [17]. The goal of combinatorial interaction testing is to build samples, which achieve t-wise interaction coverage between sets of features. T-wise interaction coverage requires that every possible combination of t features is covered by at least one product in the sample. Commonly used coverage criteria include feature-wise coverage (t = 1), pair-wise coverage (t = 2), or three-wise coverage (t = 3).

Over the years, many sampling algorithms have been developed to generate samples that achieve feature interaction coverage. To generate a sample different input artifacts such as a feature model, implementation artifacts [11, 23, 25] or expert knowledge [7, 9] can be used [30]. However, the majority of sampling algorithms uses a feature model as single input artifact [30]. Hence, our challenge focuses on sampling algorithms using a feature as single input for sampling. Sampling algorithms drastically reduce the number of products to be tested, compared to testing all possible products individually. However, when applied to large product lines from real-world applications with hundreds or even thousands of features, they often run out of memory, do not terminate, produce unnecessary large samples to test, or the calculation takes to much time [4, 8]. Facing those scalability issues is a recent challenge of product line research [15].

To assure the quality of real-world product lines is already a challenging task by considering only one version. However, a product line evolves throughout its whole life cycle, based on changing environments or newly required functionality [19]. To assure that no unwanted behavior was introduced by the changes, a new sample needs to be generated for each product line version. Hence, the sampling scalability challenge needs to be handled again for each product line version. Until now, no sampling algorithm considers the evolution of product lines or previously calculated samples to

speed up the sampling process for retesting product-line functionality [30].

We contribute to the research of product-line sampling by providing a collection of feature model evolution histories of real-world product lines. Our collection includes a selection of feature models for the history of the Linux kernel[1], including 400 versions[2] with over 19,000 features. Furthermore, we provide the evolution history of Automotive02[3] (four product-line versions with about 14,000 to 19,000 features), and FinancialServices01[4] (ten product-line versions with about 500 to 700 features). We argue that our provided product lines are well suited as subjects for the challenge of scalability of t-wise sampling algorithms, based on their size and use in different industrial environments.

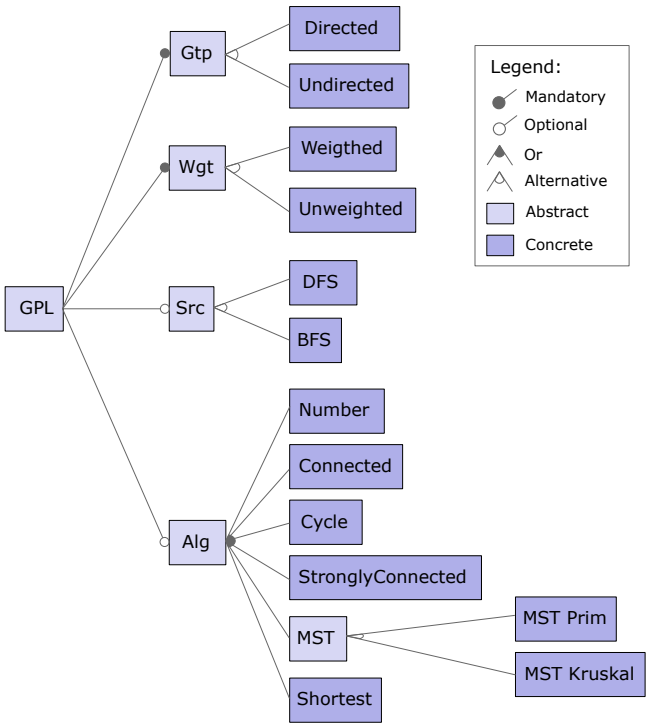
We challenge the research community to calculate samples, achieving a coverage measure for feature-, pair-, or three-wise coverage, with respect to our provided feature models. As solution for our challenge, we expect the calculated product sample as well as sampling statistics, containing CPU time and memory consumption. Furthermore, participants must provide, information about the achieved percentage of a coverage criteria (feature-, pair-, or three-wise). Beside those statistics, the sampling procedure used for the calculation must be provided as source code or executable file (Jar-file, Windows executable). For transparent comparison, we also expect a report about the procedure of sample generation, including a description of the used software libraries and hardware specification (e.g. operating system, memory size, and CPU).

To create a sample, participants are welcome to use already existing sampling algorithms and constraint solvers. In addition, we encourage all researchers to provide solutions containing newly developed sampling algorithms. All participants can choose the feature model(s) for which to generate samples. We provide a challenge category for every combination of feature model and coverage criterion, contained in submitted solutions. A solution can participate in any category for which it provides samples. Furthermore, a solution can participate in the challenge of sampling the whole product-line history, by generating samples for all versions contained in a product-line history. For both challenges, we consider as evaluation criteria the CPU time and memory consumption as well as the the sample size and the percentage of t-wise coverage for the given t. Hence, we ask all participants to provide those statistics for every generated sample in their solutions.

2 MOTIVATING EXAMPLE

In this section, we present a small example to illustrate basic concepts of feature modeling and sample creation. In addition, we illustrate the scalability challenge of sampling. As an example, an excerpt of the *Graph Product Line (GPL)*, introduced by Lopez-Herrejon and Batory [16] is provided. The *Graph Product Line* represents a family of graph applications.

As shown in Fig. 1, our example includes 19 features in total, of which six are abstract and 13 are concrete. While abstract features



Connected ⇒ Undirected
StronglyConnected ⇒ Directed ∧ DFS
Cycle ⇒ DFS
MST ⇒ Undirected ∧ Weighted
Shortest ⇒ Directed ∧ Weighted

Figure 1: Feature Model excerpt of the Graph Product Line

are only used to group other features, concrete features are mapped to real implementation artifacts [28].

The combination of features is called product configuration. If a configuration conforms to all constraints, defined by the feature model, the configuration is valid. From valid product configurations real-world products are generated. Often abstract features are omitted in product configuration, because they do not represent actual implementation artifacts [28]

Features contained in our example are either organized as single feature or as a feature group. Single features can be mandatory (e.g. *Wgt*) or optional (e.g. *Src*). To build a valid configuration, a mandatory feature needs to be selected if its parent is selected. If the parent of an optional feature is selected, the optional feature itself does not need to be selected to build a valid product configuration.

The grouping of features describes their relation on the same hierarchy level. For example, the or-group formed by *Number*, *Connected*, *Cycle*, *StronglyConnected*, *MST*, and *Shortest*, defines that at least one of those features must be contained in a valid configuration. In contrast to or-groups, alternative-groups, require that exactly one feature must be selected, if the parent is selected. For example, *Edges* of a graph application can either be *Directed* or *Undirected*, but not both at the same time. Cross-tree constraints are propositional formulas over the features contained in a feature

[1] https://github.com/PettTo/Feature-Model-History-of-Linux
[2] Linux evolution history from 2013-11-06 to 2018-01-14
[3] https://github.com/PettTo/SPLC2019_The-Scalability-Challenge_Product-Lines/tree/master/Automotive02
[4] https://github.com/PettTo/SPLC2019_The-Scalability-Challenge_Product-Lines/tree/master/FinancialServices01

Table 1: Example for Coverage Criteria based on GPL

	t = 1	t = 2	t = 3
Conf. 1	{Directed}	{Directed; Weighted}	{Directed; Weighted; DFS}
Conf. 2	{Undirected}	{Undirected; Weighted}	{Undirected; Weighted; DFS}
Conf. 3		{Directed; Unweighted}	{Directed; Unweighted; DFS}
Conf. 4		{Undirected; Unweighted}	{Undirected; Unweighted; DFS}
Conf. 5			{Directed; Weighted}
Conf. 6			{Undirected; Weighted}
Conf. 7			{Directed; Unweighted}
Conf. 8			{Undirected; Unweighted}

model. They are used to express dependencies between features [12]. For example, our feature model contains the dependency that a *Cycle* algorithm can only be selected, if *Depth-First-Search* (*DFS*) is selected as search strategy.

For real-world product lines, it is infeasible to test all valid product configurations. Often, a smaller subset of all possible configurations is used. The subset of configurations is called sample. To generate samples different procedures exist. Main concern of sampling is to achieve a sufficient test coverage with lower test effort (smaller test suits). Hence, sampling procedures try to produce a sample that is as small as possible w.r.t. the number of products. However, another important requirement is to be efficient regarding time, and memory consumption while retrieving a sample [30].

A promising approach to generate samples, which achieve a certain amount of test coverage is Combinatorial Interaction Testing (CIT). This approach is based on the assumption that most faults can be found considering interactions between a few features [17]. In Table 1, we show an example for the different CIT coverage criteria, based on our example. Each column of the table represents a minimal sample to achieve a certain feature interaction coverage. The first column of Table 1 shows a sample required to fulfill feature-wise coverage for *Directed*. To fulfill feature-wise (t = 1) coverage, feature *Directed* must be selected and deselected in at least one configuration of a sample. The second column of Table 1 shows pair-wise (t = 2) coverage for the features *Directed* and *Weighted*. As shown in Table 1, we must build all possible feature combinations (selection and deselection) between *Directed* and *Weighted* to cover pair-wise coverage. In the third column of Table 1, example configurations required to achieve three-wise (t = 3) interaction coverage between *Directed*, *Weighted*, and *DFS* are shown. To fulfill three-wise coverage, all possible selection / deselection combinations between those three features are required.

Even though, pair-wise, and three-wise feature interaction coverage find more faults than feature-wise coverage, often they are not applicable for large product lines. The reason behind this, is the scalability problem of current sampling algorithms [2]. The sampling scalability challenge cannot be shown on a small example product line such as *GPL*. However, our own experience with sampling research shows that time and memory consumption as well as the size of calculated samples increase drastically for complex product lines.

3 SUBJECT PRODUCT LINES

For this challenge we provide a collection of feature models of three large product lines from different domains. For each product line we provide multiple feature models, each representing a part of the product-line history. Every provided feature model contains hundreds or thousands of features. Our collection of product lines contains two closed-source product lines from industrial application areas (Automotive02 and FinancialServices01) and one open-source product line (Linux kernel).

To protect sensitive data, all feature names of closed source product lines are obfuscated. In order to do so a hashing algorithm to replace feature names with unique IDs is used. To assure that a feature name gets the same unique ID throughout the evolution history, the hashing function is constant for all versions of the product line. Therefore, the obfuscation has no effect on the structure and dependencies of feature models and their evolution. In case of Automotive02, we received the feature models in an obfuscated form. For FinancialServices01 the FeatureIDE obfuscation algorithm was used. To access the algorithm we used the FeatureIDE library [13].

We provide the following feature models in our challenge:

Linux. With over 19,000 features, the Linux kernel is one of the largest software product lines currently available. All product-line versions of Linux are open-source and highly-configurable. The open-source character of the product line invites a huge community, which frequently extends and improves the kernel's functionality and it's variability model. Because of those frequent updates, a long history of open-source variability models are available. The size and fast evolution of Linux poses challenges to the quality assurance of Linux. Hence, this product line is often used as benchmark to test new quality assurance procedures for product lines [1, 5, 6, 24, 26, 27].

Variability for the Linux kernel is managed in KConfig [31], which is a special file format created for the Linux kernel. To build a feature model for Linux the variability needs to be extracted from the KConfig files. We constructed a tool suite [21] based on KConfigReader [10, 14] and FeatureIDE library [13] to automatically extract Linux feature models from the Linux kernel Git repository[5]. KConfigReader is used to transform the variability model from KConfig into Conjunctive Normal Form (CNF) in the Dimacs format. During the transformation from KConfig to Dimacs format, KConfigReader performs the so called Tseitin transformation [29], to save computation time. Using the Tseitin transformation increases the number of literals contained in the CNF formulas. We transform the resulting Dimacs files into FeatureIDE XML format by using

[5] https://github.com/torvalds/linux

Table 2: Feature Model History: Automotive02

Version Name	Date	Features	Constraints
V1	2015-04-01	14,010	666
V2	2015-05-01	17,742	914
V3	2015-06-01	18,434	1,300
V4	2015-07-01	18,616	1,369

Table 3: Feature Model History: FinancialServices01

Version Name	Date	Features	Constraints
2017-05-22	2017-05-22	557	1,001
2017-09-28	2017-09-28	704	1,136
2017-10-20	2017-10-20	712	1,142
2017-11-20	2017-11-20	711	1,148
2017-12-22	2017-12-22	716	1,148
2018-01-23	2018-01-23	712	1,028
2018-02-20	2018-02-20	759	1,034
2018-03-26	2018-03-26	771	1,080
2018-04-23	2018-04-23	774	1,079
2018-05-09	2018-05-09	771	1,080

FeatureIDE import functionalities. As result, each of our provided Linux feature models contains over 60,000 features. This fact increases the scalability challenge for currently available sampling algorithms even more.

In this challenge, we offer a collection of feature models for 420 Linux versions as subject system for product sampling. Feature models of this collection are publicly available in our Git repository.[6] The feature models can be found in FeatureIDE XML and Dimacs format.

Automotive02. The Automotive02 product line was originally derived from the collaboration with our industrial partner from the automotive industry. We published those models and used them for the evaluation of feature model interfaces [22]. Later, the Automotive02 product line was used in different studies as subject product line from real-world applications [12, 20]. As shown in Table 2, we provide four feature models for this product line. They represent monthly snapshots from the evolution of Automotive02. The product line grew from 14,010 features and 666 constraints to a size of 18,616 features and 1,339 constraints, as shown in Table 2.

FinancialServices01. Originally, FinancialServices01 was obtained during a project in collaboration with an industrial partner from the financial services domain. We published and used the data to analyze feature-model anomalies [20]. We provide ten different versions of FinancialServices01 for our challenge. Each version represents a snapshot from the evolution history of the product line. Table 3 visualizes the statistics of each version. Throughout the evolution, FinancialServices01 grew from a product line with 557 features and 1,001 constraints to a product line with 771 features and 1,080 constraints.

[6] https://github.com/PettTo/Feature-Model-History-of-Linux

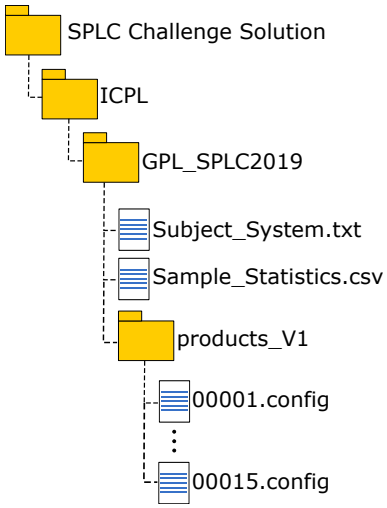


Figure 2: Example Folder Structure for Solutions

4 EVALUATION OF SAMPLING ALGORITHMS

For this challenge, we provide the feature models and their history for Linux, Automotive02, and FinancialServices01. All the provided feature models can be used as subject system to generate product samples. We challenge our participants to calculate a sample for at least one version from at least one of the provided product lines. For the calculation of samples, any sampling technique can be used. We welcome any solution produced by already existing sampling procedures. In addition, all researchers are encouraged to present their own or even newly developed sampling techniques.

All feature models for this challenge are provided in FeatureIDE's XML format. This way, they can be processed with FeatureIDE [18]. This includes showing a graphical representation of the feature model, generating products with certain sampling algorithms, and transforming the XML representation into a supported file format (GUIDSL, SXFM, Velvet, or Dimacs).

Even though we recommend to use FeatureIDE to get started with this challenge, no participant is strictly bound to the usage of this tool.

To produce samples for our challenge any sampling technique can be used, even if the technique does not cover 100% of t-wise coverage for a certain value t. We expect that the sampling technique as well as all generated artifacts, such as the samples, sampling statistics, and a report about the subject system configuration (hardware and operating system), are publicly available. Only this way, we can assure that the results of different participants could be evaluated and compared against each other. Furthermore, we ask our participants to include a report and discussion about their experiences and lessons learned.

To encourage a uniform structure for submitting solutions, we provide a Git repository as an example structure.[7] The expected structure of this Git repository is visualized in Fig. 2. We expect folders for all algorithms used for sampling. The folder name should reflect the used algorithms. In our example case, this is the folder

[7] https://github.com/PettTo/SPLC2019_The-Scalability-Challenge.git

Table 4: Solution Table for Graph Product Line

Coverage Criterion (t)	Percentage Covered (%)	Sample Size	CPU Time (ms)	Memory Usage (KB)
2	100	15	2,000	161,000

ICPL. Inside the algorithm folder, we expect a separate folder for each product line, which was sampled. The folder must be named as the product line it contains samples for. We generated samples for our running example (*GPL*). Hence, our example folder is named *GPL_SPLC2019*. Fig. 2 shows that we expect one *Experience Report*, one *Subject System Report*, one file containing sample statics in CSV format, and folders for the generated samples. Configurations of a submitted sample must be in FeatureIDE configuration format (*.config). This format is equal to a text file containing each selected feature in a separate line, as shown in the example Git repository.[8]

The sample statistics must include the following artifacts:

- Name of the algorithm or solver library used for sampling
- Feature interaction coverage criterion (t)
- Percentage of achieved feature interaction coverage
- Number of products generated (sample size)
- CPU Time consumption needed to retrieve the sample
- Memory consumption needed to retrieve the sample (maximal heap usage, during sampling)

We expect that the sample statistics are provided as CSV file. Table 4 shows an example of how we expect the CSV structure of submitted solutions. The values contained in Table 4 are statistics for our *Graph Product Line* example. We used the ICPL algorithm integrated in FeatureIDE to create a sample that covers pair-wise feature interaction coverage. To get the sample size for example, we counted the configurations contain in the generated sample. To retrieve the CPU time, we implemented a method to measure the sampling duration into FeatureIDE. Regarding the values for memory consumption, we measured the maximal size of heap allocated during the sampling process using visualVM[9]. Furthermore, we measured the percentage of t-wise coverage reached, by using the algorithm shown in Listing 1.

```
1  function claculatePercentage(FM, T, S)
2    List TWC ← calculateAllTcombinations(FM, T);
3    int CC ← calculateCovertCombination(TWC,S);
4    return (CC * 100) / |TWC|;
5  end function
```

Listing 1: Calculation: Percentage of Coverage Achieved

As input for our function we expect a feature model (*FM*), a t-wise coverage (*T*), and a sample (*S*). In line two, a list of all feature combinations theoretically needed to achieve t-wise coverage is calculated (*TWC*). Thereafter, the number of feature combinations (*CC*) from *TWC* contained in sample *S* is calculated. Thenceforth, the percentage of covered feature combinations is returned.

[8]https://github.com/PettTo/SPLC2019_The-Scalability-Challenge/tree/master/ICPL/GPL_SPLC2019/products_v1
[9]https://visualvm.github.io/

We expect that the CSV files provided as solutions conform to the following format:

- Use semicolon (";") as separator
- Use line feed ("\n") as line separator
- Provide a table header as shown in Table 4
- Use milliseconds (ms) as unit for sampling time
- Use kilobyte (KB) as unit for memory consumption
- Provide all numerical values as integer values

Submitted solutions will be evaluated based on the performance of their sampling algorithms. For the evaluation we use the provided values for sample size, CPU time consumption, memory consumption, and the percentage of t-wise coverage achieved. To establish a transparent evaluation, we will categorize the solutions based on which product line or product line version is used as subject system for the sampling process. Furthermore, we will consider, which kind of feature interaction coverage the solutions achieve. Another parameter for categorizing a solution is the subject system on which the sampling was performed. After categorizing the solution as described, sample size, memory usage, SPU time consumption, and the percentage of t-wise coverage achieved will be used as evaluation criteria. The aim for all participants should be to minimize sample size, memory consumption and CPU time consumption, while maximizing the percentage of achieved t-wise coverage.

Our participants can generate samples for all feature models of a product-line history to take part in a second challenge. In this challenge we evaluate how sampling techniques perform when applied to a product-line history. Even though every participant can use sampling techniques which sample each product-line version separately, we encourage our participants to actively develop sampling techniques, which consider the product line evolution explicitly. We ask our participants to provide the previously mentioned sample statics for each feature model of the product-line history separately. We will aggregate the values for each of our for metrics. This way, we get one result for each metric, which represents the performance for sampling the whole product-line evolution history. Solutions will be evaluated based on this results.

5 SUMMARY

In this challenge, we presented the process of sampling large software product lines. Common sampling algorithms run into different scalability issues when performed on large systems. Those issues include running out of memory, needing to much computation time, and producing too large samples. We challenge the research community to generate samples for large product lines and submit them as solutions. As subject systems for this challenge, we provide feature models of three real-world product line histories. Solutions can be generated by using newly developed, or currently available sampling algorithms. Submitted solutions will be evaluated based on their performance. As evaluation criteria we use CPU time and memory consumption as well as sample size and the percentage of t-wise coverage achieved by the sample. Best performing algorithms minimize CPU time and memory consumption, and calculate a small sample which still achieves a high percentage of t-wise coverage. We will provide a comparison of the performance of all submitted solutions.

REFERENCES

[1] Iago Abal, Jean Melo, Stefan Stănciulescu, Claus Brabrand, Márcio Ribeiro, and Andrzej Wąsowski. 2018. Variability Bugs in Highly Configurable Systems: A Qualitative Analysis. *Trans. Software Engineering and Methodology (TOSEM)* 26, 3, Article 10 (2018), 10:1–10:34 pages.

[2] Bestoun S. Ahmed, Kamal Z. Zamli, Wasif Afzal, and Miroslav Bures. 2017. Constrained Interaction Testing: A Systematic Literature Study. *IEEE Access* 5 (2017), 25706–25730.

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

[4] Andrea Arcuri and Lionel Briand. 2011. Formal analysis of the probability of interaction fault detection using random testing. *TSE* 38, 5 (2011), 1088–1099.

[5] Christian Dietrich, Reinhard Tartler, Wolfgang Schröder-Preikschat, and Daniel Lohmann. 2012. Understanding Linux Feature Distribution. In *Proc. of Workshop on Modularity in Systems Software (MISS)*. ACM, 15–20.

[6] Nicolas Dintzner, Arie van Deursen, and Martin Pinzger. 2018. FEVER: An Approach to Analyze Feature-Oriented Changes and Artefact Co-Evolution in Highly Configurable Systems. *Empirical Software Engineering (EMSE)* 23, 2 (2018), 905–952.

[7] Alireza Ensan, Ebrahim Bagheri, Mohsen Asadi, Dragan Gasevic, and Yevgen Biletskiy. 2011. Goal-Oriented Test Case Selection and Prioritization for Product Line Feature Models. In *Proc. Inte'l Conf. on Information Technology: New Generations (ITNG)*. IEEE, 291–298.

[8] Mats Grindal, Jeff Offutt, and Sten F Andler. 2005. Combination testing strategies: a survey. *STVR* 15, 3 (2005), 167–199.

[9] Christopher Henard, Mike Papadakis, Gilles Perrouin, Jacques Klein, and Yves Le Traon. 2013. Multi-Objective Test Generation for Software Product Lines. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 62–71.

[10] Christian Kästner. 2017. Differential Testing for Variational Analyses: Experience from Developing KConfigReader. *arXiv preprint arXiv:1706.09357* (2017).

[11] Chang Hwan Peter Kim, Don Batory, and Sarfraz Khurshid. 2011. Reducing Combinatorics in Testing Product Lines. In *Proc. Int'l Conf. on Aspect-Oriented Software Development (AOSD)*. ACM, 57—68.

[12] Alexander Knüppel, Thomas Thüm, Stephan Mennicke, Jens Meinicke, and Ina Schaefer. 2017. Is There a Mismatch Between Real-World Feature Models and Product-Line Research?. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*. ACM, 291–302.

[13] Sebastian Krieter, Marcus Pinnecke, Jacob Krüger, Joshua Sprey, Christopher Sontag, Thomas Thüm, Thomas Leich, and Gunter Saake. 2017. FeatureIDE: Empowering Third-Party Developers. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 42–45.

[14] Christian Kästner. 2018. KConfigReader. Website. Available online at https://github.com/ckaestne/kconfigreader; visited on August 29th, 2018.

[15] Jörg Liebig, Alexander von Rhein, Christian Kästner, Sven Apel, Jens Dörre, and Christian Lengauer. 2013. Scalable Analysis of Variable Software. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*. ACM, 81–91.

[16] Roberto E. Lopez-Herrejon and Don Batory. 2001. A Standard Problem for Evaluating Product-Line Methodologies. In *Proc. Int'l Symposium on Generative and Component-Based Software Engineering (GCSE)*. Springer, 10–24.

[17] Roberto E. Lopez-Herrejon, Stefan Fischer, Rudolf Ramler, and Aalexander Egyed. 2015. A First Systematic Mapping Study on Combinatorial Interaction Testing for Software Product Lines. In *Proc. Int'l Workshop on Combinatorial Testing (IWCT)*. IEEE, 1–10.

[18] Jens Meinicke, Thomas Thüm, Reimar Schröter, Fabian Benduhn, Thomas Leich, and Gunter Saake. 2017. *Mastering Software Variability with FeatureIDE*. Springer.

[19] Paulo Anselmo da Mota Silveira Neto, Ivan do Carmo Machado, Yguarata Cerqueira Cavalcanti, Eduardo Santana de Almeida, Vinicius Cardoso Garcia, and Silvio Romero de Lemos Meira. 2010. A regression testing approach for software product lines architectures. In *Software Components, Architectures and Reuse (SBCARS), 2010 Fourth Brazilian Symposium on*. IEEE, 41–50.

[20] Michael Nieke, Jacopo Mauro, Christoph Seidl, Thomas Thüm, Ingrid Chieh Yu, and Felix Franzke. 2018. Anomaly Analyses for Feature-Model Evolution. In *Proc. Int'l Conf. on Generative Programming and Component Engineering (GPCE)*. ACM, 188–201.

[21] Tobias Pett. 2018. *Stability of Product Sampling under Product-Line Evolution*. Master's thesis. Braunschweig.

[22] Reimar Schröter, Sebastian Krieter, Thomas Thüm, Fabian Benduhn, and Gunter Saake. 2016. Feature-Model Interfaces: The Highway to Compositional Analyses of Highly-Configurable Systems. In *Proc. Int'l Conf. on Software Engineering (ICSE)*. ACM, 667–678.

[23] Jiangfan Shi, Myra B. Cohen, and Matthew B. Dwyer. 2012. Integration Testing of Software Product Lines Using Compositional Symbolic Execution. In *Proc. Int'l Conf. on Fundamental Approaches to Software Engineering (FASE)*. Springer, 270–284.

[24] Julio Sincero, Horst Schirmeier, Wolfgang Schröder-Preikschat, and Olaf Spinczyk. 2007. Is the Linux Kernel a Software Product Line?. In *Proc. Int'l Workshop on Open Source Software and Product Lines (OSSPL)*. IEEE, 9–12.

[25] Reinhard Tartler, Christian Dietrich, Julio Sincero, Wolfgang Schröder-Preikschat, and Daniel Lohmann. 2014. Static Analysis of Variability in System Software: The 90,000 #Ifdefs Issue. In *Proc. USENIX Annual Technical Conference (ATC)*. USENIX Association, 421–432.

[26] Reinhard Tartler, Daniel Lohmann, Julio Sincero, and Wolfgang Schröder-Preikschat. 2011. Feature Consistency in Compile-Time-Configurable System Software: Facing the Linux 10,000 Feature Problem. In *Proc. Europ. Conf. on Computer Systems (EuroSys)*. ACM, 47–60.

[27] Reinhard Tartler, Julio Sincero, Wolfgang Schröder-Preikschat, and Daniel Lohmann. 2009. Dead or Alive: Finding Zombie Features in the Linux Kernel. In *Proc. Int'l Workshop on Feature-Oriented Software Development (FOSD)*. ACM, 81–86.

[28] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. 2014. A Classification and Survey of Analysis Strategies for Software Product Lines. *Comput. Surveys* 47, 1 (2014), 6:1–6:45.

[29] Grigori S Tseitin. 1983. On the complexity of derivation in propositional calculus. In *Automation of reasoning*. Springer, 466–483.

[30] Mahsa Varshosaz, Mustafa Al-Hajjaji, Thomas Thüm, Tobias Runge, Mohammad Reza Mousavi, and Ina Schaefer. 2018. A Classification of Product Sampling for Software Product Lines. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 1–13.

[31] Roman Zippel. 2017. KConfig Documentation. Website. Available online at http://www.kernel.org/doc/Documentation/kbuild/kconfig-language.txt; visited on June 21st, 2019.