Product Sampling for Product Lines: The Scalability Challenge

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

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

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

Sebastian Krieter
sebastian.krieter@ovgu.de
OVGU Magdeburg
Magdeburg

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

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

ABSTRACT

Quality assurance for product lines is often infeasible for each product separately. Instead, often only a subset of all products (i.e. a sample) is considered in testing such that at least the coverage of certain feature interactions is guaranteed. While pair-wise interaction sampling covers all interactions between two features, its generalization to t-wise interaction sampling ensures coverage for all interactions among t features. However, sampling for 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 approach this challenge, 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 the sampling is evaluated based on the time and memory consumed to retrieve a sample and the sample size for a given coverage (i.e. t value). A well-performing sampling algorithm achieves full t-wise coverage, while minimizing all three of these properties.

KEYWORDS

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

1 INTRODUCTION

Modern software engineering struggles with increasing requirements regarding finer customization and faster time to market. A common solution is, to build a collection of customized products based on a common core also known as software product line [4]. However, the variability contained within software product lines poses a potentially large number of possible products. Analyzing all products is infeasible in most cases [29]. 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. By doing so, most faults can be found [8]. A common technique to build a subset of product configurations is combinatorial interaction sampling [16]. It aims to build samples, which achieve t-wise interaction coverage between 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. Most of those algorithms use feature constraints, described in feature models, as single input artifact to calculate a sample [29] (e.g. CASA [7], Chvatal [7, 9], ICPL [10], MoSoPo-LiTe [21], and IncLing [3]). In our challenge, we focus on algorithms using a feature model as input only. Sampling algorithms drastically reduce the number of products to test, compared to testing all possible products. However, when applied to large product lines from real-world applications with hundreds or thousands of features, they often run out of memory, do not terminate, or produce too large samples to test. Facing those scalability issues is one major challenge of product line research [14]. Hence, researchers strive to develop and implement more efficient sampling algorithms.

To assure the quality of product lines used in industry is a challenging task by only considering one version of the product line. However, this task becomes even more complex by regarding the evolution of product lines. Over time, new product-line versions are created based on changing environments or newly required functionality [18]. To gain confidence, that changes did not introduce unwanted behavior, regression testing is used. To do so, already generated products of the product line need to be generated and retested again [20]. Hence, the challenge of generating samples adds up with every new product-line version.

We contribute to the research of product line sampling by providing a collection of feature model evolution histories from large product lines from real-world scenarios. Our collection includes a selection of feature models for the history of 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 usage in industrial environments.

We challenge the research community to calculate samples, which achieve feature-, pair-, or three-wise coverage, for our provided feature models. As solution for our challenge, we expect the calculated product sample as well as sampling statistics, containing time and memory consumption. Furthermore, the sampling algorithm used for the calculation must be provided as source code or

[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

executable file (Jar-file, Windows executable). For transparent comparison, we also expect a report about the procedure of generating samples, including a description of the used software 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 new developed sampling algorithms. All participant can choose for which feature models they create samples. We provide a challenge category for every combination of feature model and coverage criterion, we can find in the submitted solutions. A solution can compete in any category for which it provides samples. Furthermore, each solution competes automatically in the challenge to sample the whole product line history for a certain t-wise coverage. Even though, all solutions participate implicitly in this second challenge, we encourage all researchers to actively take part in this second challenge by providing a self developed algorithm, which considers the product-line evolution history. For both challenges we use the minimal values for time and memory consumption to retrieve the sample, as well as the minimum value of sample size. Hence, we ask all participants to provide those statistics for every generated sample in their solutions.

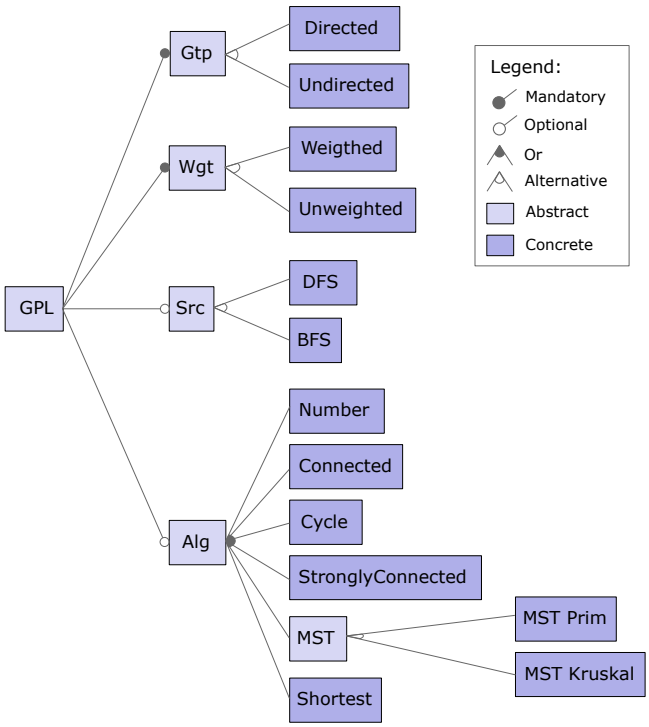
2 MOTIVATING EXAMPLE

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 by presenting current research results regarding this topic. As an example, we provide a small excerpt of the *Graph Library Product Line (GPL)*, introduced by Lopez-Herrejon et al. [15]. The *Graph Library Product Line* represents a product line 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 are only used as structuring tool, concrete features represent the actual functionality contained in the product line. Based on the structure described by the feature model, valid combinations of features can be built by selecting a subset of them. A valid combination of selected features is called a product configuration and represents a product in real applications. It is not reasonable to include abstract features into configurations, because they do not contribute to actual functionality of the product line. Hence, we require all submitted solutions to only include concrete features into their samples.

The 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*). Mandatory features such as *Wgt* must be selected to build valid product configurations. Optional features can either be selected or deselected.

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, such as *Directed*, and *Undirected*, require that exactly one feature must be selected. For example, *Edges* of a *Graph Library* can either be *Directed* or *Undirected*, but not both at the same time. Cross-tree constraints



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

Figure 1: Feature Model excerpt of the Graph Library Product Line

are used to model dependencies between features, which can not be expressed by structural means [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 a better understanding of the sampling scalabilty challenge, we need to take a look at how a sample is structured. A sample consists of a collection of valid product configurations. For testing purposes, samples are used to reduce the number of products that must be tested [27]. The main concern of a sampling algorithms is 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 [29].

Different procedures to generate samples with the given requirements were developed. One promising approach is Combinatorial Interaction Testing (CIT). This approach is based on the assumption that most faults can be found considering interactions between a few features [16]. In Table 1 we show an examples for the different CIT coverage criteria, based on our graph library 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*.

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}

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 / 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 feature are required.

Even though, pair-wise, and three-wise feature interaction coverage find more faults than feature-wise coverage or random sampling, they are not applicable for large product lines. The reason behind this, is the scalability problem of current sampling algorithms [2]. This scalability challenge for product line sampling 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. Evidence for this challenge can also be found in recent scientific work [14].

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 from industry, all feature names of the industrial product lines are obfuscated. Note that, this has no effect on the structure and dependencies of feature models. In case of Automotive02, we received the feature models in an obfuscated form. That means, the feature names were already replaced in the models by unique IDs. As for FinancialServices01, we used a self-developed obfuscator. The obfuscator uses a hashing algorithm to replace feature names with unique IDs. 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.

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. Therefore, Linux evolves fast and many product-line versions 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].

Variability for the Linux kernel is managed in KConfig [30], 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 [22] based on KConfigReader [11, 13] and FeatureIDE [17] 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 common Dimacs format. During the transformation from KConfig to Dimacs format, KConfigReader performs the so called Tseitin transformation [28], to save computation time. Using the Tseitin transformation leads to larger feature models. We transform the resulting Dimacs files into FeatureIDE XML format by using 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 from about 400 Linux versions as subject system for product sampling. Feature models of this collection are publicly available on our Git repository[6]. The feature models can be found in FeatureIDE XML or Dimacs format.

Automotive02. The Automotive02 product line was originally derived from the collaboration with our industrial partner from the automotive industry. First, this product line was introduced by Schröter et al. [23] to prove the concept of feature model interfaces. Later, the Automotive02 product line was used in different studies as example product line from real-world applications [12, 19]. 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. Our third contributed product line is FinancialServices01. Originally, FinancialServices01 was obtained during

[5] https://github.com/torvalds/linux
[6] https://github.com/PettTo/Feature-Model-History-of-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

a project in collaboration with an industrial partner from the financial services domain. The data for this product line was introduced in [19]. 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.

4 EVALUATION OF SAMPLING ALGORITHMS

In the context of 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 for generating product samples. We challenge our participants to calculate a sample for at least one version (the more the better) from at least one of the provided product lines. For the calculation of samples, any sampling procedure can be used. This also includes any kind of constraint solver. We welcome any solution produced by already existing sampling procedures. In addition, all researchers are encouraged to present their own (newly) developed sampling procedures.

All feature models for this challenge are provided in the FeatureIDE-XML format. This way, they can be processed with FeatureIDE[7] [17]. 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). Hence, our participants can use FeatureIDE and its included functionality for this challenge, while they still have options to use other tools as well. Even though we recommend to use FeatureIDE to get started

[7]https://featureide.github.io/

Table 4: Solution Table for Graph Library

Coverage (t)	Sample Size	Time	Memory Usage
2	15	2,000	161,000

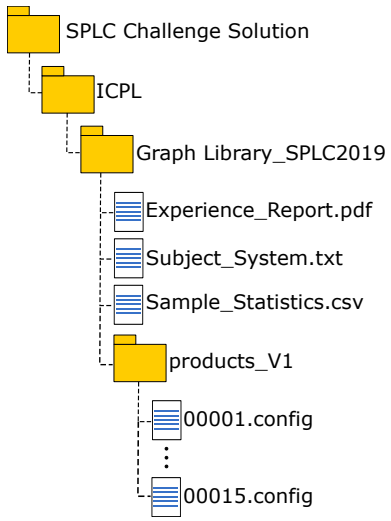


Figure 2: Example Folder Structure for Solutions

with this challenge, no participant is strictly bound to the usage of this tool.

Any sampling procedure, which fulfills feature interaction coverage for a certain value of t can be used as a solution for our challenge. This includes sampling algorithms as well as constraint solvers. We expect that the sampling procedure as well as all generated artifacts, such as the samples, 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. All artifacts, including the sampling procedure (e.g. source code of the algorithm, an executable, or solver library), samples, and reporting must be publicly available from the point of submitting on. If participants use an open-source framework like FeatureIDE to generate their samples, they do not need to include it in the solution artifacts. A link and description of the used version in the subject system report will be sufficient. If participants use self-implemented sampling procedures or closed-source tools, they must include the source code or an executable as solution artifact.

To encourage a uniform structure for submitting solutions, we provide a Git repository as an example structure[8]. 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 *ICPL*. Inside the algorithm folder, we expect a separate folder for each product line, which was sampled. The folder must be named after

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

the product line it contains samples for. In our example, this folder is named after our example product line *Graph Library_SPLC2019*. The participants need to store every required artifact for evaluation into the right product line folder. Fig. 2 shows that we expect one *Experience Report*, one *Subject System Report*, one file containing sample statics (CSV format), and folders for the generated samples. Even though our example contains only one product-line version, we expect a folder for every product-line version the participants generated samples for. 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[9].

The sample statistics must include the following artifacts:

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

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 example values contained in Table 4 are statistics for our *Graph Library* example. We used the ICPL algorithm of FeatureIDE to create a sample that covers pair-wise feature interaction coverage. The values for sample size and time consumption to retrieve the sample are shown in FeatureIDE after the sampling is finished. Regarding the values for memory consumption, we measured the maximal size of heap allocated during the sampling process using visualVM[10].

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, using the three described metrics. To establish a transparent evaluation, we will categorize the solutions based on which product line or product line versions is used as subject system for the sampling. Furthermore, we will consider, which kind of feature interaction coverage the solutions achieve, and on which system configuration the sampling was performed. After categorizing the solution as described, sample size, memory usage, and time consumption will be used as evaluation criteria. The aim for all participants should be to minimize those three criteria.

By submitting to the challenge of creating samples for one or more feature models for a product line, our participants automatically submit to a second challenge to sample the whole feature model history. We will consider how many feature models of the history were covered and categorize the solution accordingly. Furthermore, criteria like sample size, time consumption and memory consumption for generated single samples are summed up to create statistic values for the whole product line evolution. Even though every participant enters this challenge automatically, we encourage our participants to actively develop sampling algorithms, which consider the product line evolution history.

[9]https://github.com/PettTo/SPLC2019_The-Scalability-Challenge/blob/master/ICPL/Graph%20Library_SPLC2019/products_v1/00001.config

[10]https://visualvm.github.io/

5 SUMMARY

In this paper, we presented the process of sampling large software product lines as challenging task. 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 to 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 large product line evolution histories from industrial applications. Solutions can be generated by using newly developed, or currently available sampling algorithms. Submitted solutions will be evaluated based on their performance with regard to the size of produced samples and the time and memory consumption needed to retrieve the sample. Best performing algorithms minimize those values as much as possible. 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] Mustafa Al-Hajjaji, Sebastian Krieter, Thomas Thüm, Malte Lochau, and Gunter Saake. 2016. IncLing: Efficient Product-line Testing Using Incremental Pairwise Sampling. In *Proc. Int'l Conf. on Generative Programming: Concepts & Experiences (GPCE)*. ACM, 144–155.

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

[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] Brady J. Garvin, Myra B. Cohen, and Matthew B. Dwyer. 2011. Evaluating Improvements to a Meta-Heuristic Search for Constrained Interaction Testing. *Empirical Software Engineering (EMSE)* 16, 1 (2011), 61–102.

[8] Christopher Henard, Mike Papadakis, Gilles Perrouin, Jacques Klein, Patrick Heymans, and Yves Le Traon. 2014. Bypassing the Combinatorial Explosion: Using Similarity to Generate and Prioritize T-Wise Test Configurations for Software Product Lines. *IEEE Trans. Software Engineering (TSE)* 40, 7 (2014), 650–670.

[9] Martin Fagereng Johansen, Øystein Haugen, and Franck Fleurey. 2011. Properties of Realistic Feature Models Make Combinatorial Testing of Product Lines Feasible. In *Proc. Int'l Conf. on Model Driven Engineering Languages and Systems (MODELS)*. Springer, 638–652.

[10] Martin Fagereng Johansen, Øystein Haugen, Franck Fleurey, Anne Grete Eldegard, and Torbjørn Syversen. 2012. Generating Better Partial Covering Arrays by Modeling Weights on Sub-Product Lines. In *Proc. Int'l Conf. on Model Driven Engineering Languages and Systems (MODELS)*. Springer, 269–284.

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

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

[14] 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.

[15] 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.

[16] 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.

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

[18] 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.

[19] 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.

[20] Alessandro Orso, Nanjuan Shi, and Mary Jean Harrold. 2004. Scaling regression testing to large software systems. In *ACM SIGSOFT Software Engineering Notes*, Vol. 29. ACM, 241–251.

[21] Sebastian Oster, Ivan Zorcic, Florian Markert, and Malte Lochau. 2011. MoSo-PoLiTe - Tool Support for Pairwise and Model-Based Software Product Line Testing. In *Proc. Int'l Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, 79–82.

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

[23] 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.

[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, 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.

[26] 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.

[27] 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.

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

[29] 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.

[30] Roman Zippel. 2017. KConfig Documentation. Website. Available online at http://www.kernel.org/doc/Documentation/kbuild/kconfig-language.txt; visited on May 10th, 2017.