CaSPL-gen: a Context-aware Software Product Line benchmark generator

Magnus Hestvik
University of Oslo
Norway

Jacopo Mauro
University of Oslo
Norway

Ingrid Chieh Yu
University of Oslo
Norway

Abstract

Software Product Lines (SPLs) are a mechanism for large-scale reuse where families of related software systems are represented in terms of commonalities and variabilities, e.g., using Feature Models (FMs). Context-aware SPL have been proposed to model and deal with dynamic systems whose behavior and properties depend on the context where they are deployed and executed. Due to the novelty of approach, no existing benchmarks of context-aware SPL are available.

In this paper we overcome this limitation by introducing CaSPL-gen, i.e., the first benchmark generator tool able to generate random instances of context-aware SPL.

1 Introduction

Software Product Lines (SPLs) are an approach for structured large-scale reuse where families of related software systems are modeled in terms of commonalities and variabilities [20]. The set of all possible configurations is represented by a *variability model*, such as a Feature Model (FM) [12]. A FM structures features along a decomposition hierarchy where individual features represent configurable units of functionality, which may be enriched by additional *feature attributes*–variables with a type defined within the context of a feature. A *configuration* is a valid subset of features of a FM along with concrete values for the attributes of all selected features.

In the standard SPL approach, features are selected based on human desires only. Nowadays, however, this may not be enough: with the diffusion of the Internet of Things [2], a very large number of devices have to interact and adapt based on their surrounding. As a consequence, contextual information needs to also be considered in the configuration process. Usually, the development of such systems is performed by defining a context model and rules to specify which configurations and parameters to use for every interesting context [3, 6, 9, 14]. For example, considering the automotive industry, cars may change their navigation software according to their position. If a car is in Russia, instead of using as navigational system with the standard GPS, it may use the satellite navigation system GLONASS. Additionally, a cruise control should have a parameter, limiting the maximum selectable value based on the speed limit of the respective country.

This paper was presented at the NIK-2017 conference; see http://www.nik.no/.

Clearly, for systems having a very large number of context or features, the number of rules to write may increase drastically, which makes it very difficult (i) to list all the rules and (ii) to ensure that a valid configuration exists for every possible context. To mitigate this problem, different context extensions have been proposed within the SPL field [1, 7, 11, 18]. Usually these approaches represent the contextual information with additional data structures (e.g., trees, FM, ontologies) and relate the context with the features using external or cross-tree constraints. All these approaches are fairly recent, with a limited tool support for the analysis of the models, and no established benchmarks.

To support and foster the development of reasoners, configurators, and analyzers able to handle context-aware SPL, in this work we introduce CaSPL-gen, the first benchmark generator tool able to generate context-aware SPL. Among all the formalism for context-aware SPL proposed, we target HyVarRec [14]. The reasons behind this choices are twofold: i) HyVarRec formalism is one of the simplest context-aware extension since contexts are not represented by a complex data structure like trees or ontologies but are instead a name-value map, ii) HyVarRec is the only approach that comes with a reasoner [15] able to handle the full additional complexity brought by the addition of contexts.

The goal of CaSPL-gen is to allow the comparison of new tools both for functional testing[1] and for performance testing. CaSPL-gen will generate an extensive set of benchmarks that can be used to compare the results and the performance w.r.t. the HyVarRec reasoner [15]. Evaluating how the different tools run on the instances generated by CaSPL-gen is, however, beyond the scope of this paper.

This paper is structured as follows: In Section 2, we explain the background information needed to better understand context-aware SPL. In Section 3, we introduce and describe CaSPL-gen. Finally, in Section 4, we give an overview of related works, before drawing some concluding remarks.

The work presented here is based on the Master Thesis available at `http://urn.nb.no/URN:NBN:no-60166`.

2 Background

In this section we give a brief overview of the standard Feature Models (FMs) and their extension to capture contextual information.

Standard FM is a hierarchical representation of the variability of an SPL. A FM consist of features representing configurable functionality of a system [12]. Each feature can have several child features and a FM has exactly one root feature. As an example, Figure 1 exemplifies an instance of a FM for the SPL of a car manufacturer. In this case, the root feature is `Car` while `Emergency Call`, `Position Service` are two of its children. Features can be *mandatory* or *optional* and can be organized in *alternative-* or *or*-groups. In alternative groups, exactly one feature has to be selected, whereas in or groups, at least one feature has to be selected. For instance, in Figure 1, the feature `Distance Sensor` is optional and the feature `Emergency Call` is mandatory and has its children organized as an alternative group. Feature attributes are typed variables associated with a feature and are used to express more fine-grained variability [4]. It is possible to specify a domain for each attribute to limit its value range. For example, in Figure 1, the feature `Adaptive Cruise Control` owns an attribute `maxSpeed` of type `int` that has the interval

[1] With functional testing we intended the checking of whether a program satisfies its functional requirements, i.e. whether the program does what the user expects it to do.

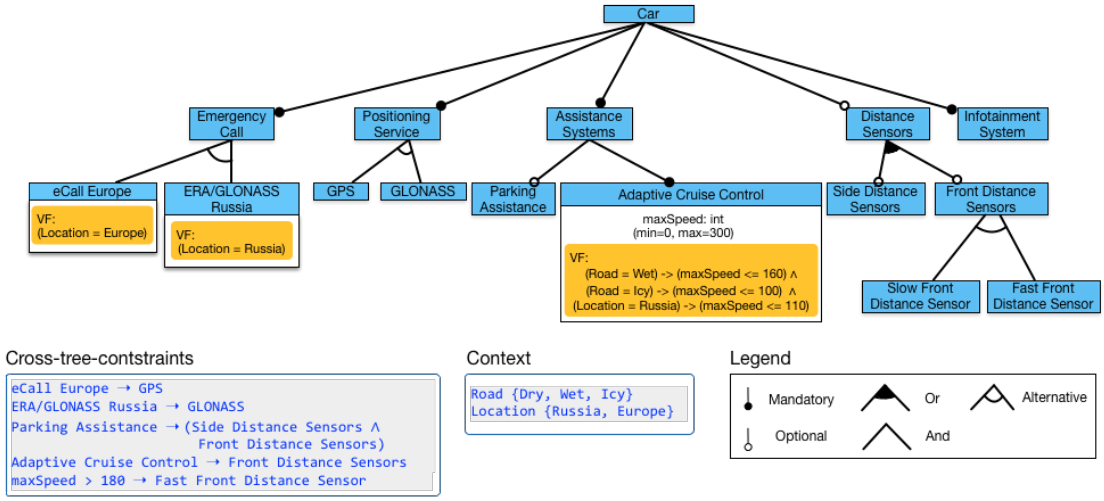

Figure 1: Example of context-aware FM for a car manufacturer SPL.

$[0, 300]$ as domain. In FMs, Cross Tree Constraints (CTC) are used to specify additional dependencies between features which are not specified via the hierarchical structure of the FM. CTCs can be defined as Boolean formulas on features and expressions on feature attributes. A *configuration* of an FM is a set of selected features, feature versions and values for the relevant feature attributes. A configuration is *valid* if it conforms with the hierarchy of the FM and the CTCs. A FM is void if it does not allow any valid configuration.

In [14, 19], an extension of standard FM was presented to capture contextual information allowing to model the dependencies of the selected features or attributes with respect to some external or environmental parameter. In the following we denote with Context-aware Feature Models (CFMs) this extended version of FM. Contextual information is captured with identifier-value pairs where a context is associated with a value. For instance, in Figure 1, two contexts are defined: `Road` and `Location` to represent the status of the road used by the car and the car location. The contextual information is used to force the selection of features. This is done by enforcing so called Validity Formulas (VF), i.e., propositional formulas associated with a feature that can relate features and attributes with context values. For instance, in Figure 1, the feature `Ecall Europe` has associated the VF `Location = Europe` that forces the feature to be selectable only if the value of the context `Location` is `Europe`.

As a configuration of an SPL may be invalidated due to changed values for contextual information and the violation of VFs, the product needs to be reconfigured. To do so reconfigurators such as HyVarRec [14, 19] may be used.

3 CaSPL-gen

In this section we introduce the tool CaSPL-gen that we implemented for the generation of random CFMs.

CaSPL-gen was developed to create a fixed number of CFMs that are restricted by certain pre-defined parameters. Since CFMs are an extension of standard FM, instead of generating CFMs from scratch we exploit an available benchmark generation tool for standard FM. In particular, internally CaSPL-gen uses BeTTy [22], i.e., a Java framework that supports the benchmarking and the analysis of FM. With BeTTy it is possible to

Name	Description	Default value	Source
sizeDataSet	# instances to generate	10	New
numberOfFeatures	# of features	50	BeTTy
percentageCTC	max % of CTCs relative to # of features	30	BeTTy
probMand	Percentage of mandatory relationships	Random	BeTTy
probOpt	Percentage of optional relationships	Random	BeTTy
probAlt	Percentage of or-relationships	Random	BeTTy
probOr	Percentage of alternative relationships	Random	BeTTy
maxBranchingFactor	Maximum branching factor	10	BeTTy
maxSetChildren	Maximum # of sub-features in sets	5	BeTTy
minAttrValue	min value for attribute domain	0	New
maxAttrValue	max value for attribute domain	100	New
contextMaxSize	max # of contexts	10	New
contextMaxValue	max domain size for context	10	New
maxPercentageVFs	max % of VFs relative to # of features	20	New
simpleMode	selects the simple execution modality	false	New

Table 1: CaSPL-gen settings. The last column distinguishes the parameters used internally to call BeTTy from the new ones used only by CaSPL-gen.

create large numbers of FM, with or without attributes in a short amount of time. The FMs generated by using Betty are then extended with VFs and contexts, finally exported in the HyVarRec JSON format [14].

CaSPL-gen can be invoked by setting different parameters, summarized in Table 1.[2] These parameters are used to define the size of the benchmark and the size of every CFM to be generated. Several parameters affect the structure of the CFMs, in particular the parameters that define the percentage of different relation types, the amount of CTC, and the amount of mandatory features are shared with the BeTTy tool and used to generate the basic FM structure.[3] We remark that these settings have an impact on the complexity of the final CFMs and on the running times of CaSPL-gen.

CaSPL-gen offers two execution modalities: a simple one that lets BeTTy do most of the work, and a default one that tries to avoid the creation of CFMs that do not admit valid configurations or that have a lot of dead features, i.e., features that due to the hierarchy and the CTC can never be selected. Before discussing the default execution modality and the usage of CaSPL-gen, we detail the simple execution modality.

Simple Execution Modality

The simple execution modality heavily relies on BeTTy for the generation of the benchmarks and is intended to be used only for the generation of large FMs where the additional checks performed in the default modality may take too long time.

The execution flow followed by CaSPL-gen to generate a CFM when executed in the simple modality is depicted in Figure 2. First BeTTy is run, setting its parameters according to the parameters given in input to CaSPL-gen. BeTTy outputs a FM according

[2] For more information and additional advance settings please see `https://github.com/magnurh/CaSPL/blob/master/README.md`.

[3] For more information related to these parameters and their behavior we refer the interested reader to [22].

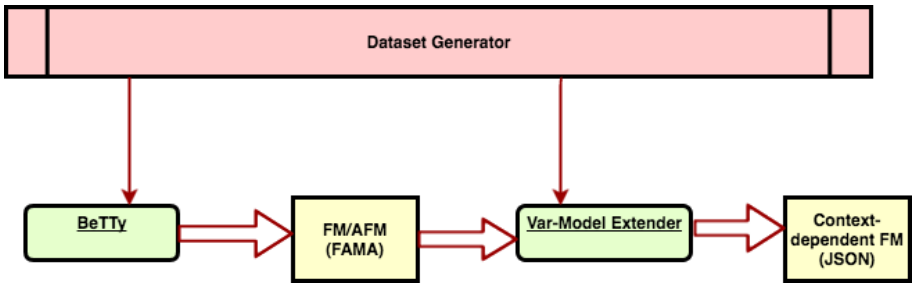

Figure 2: CaSPL-gen process flow for creating CFMs.

to the FAMA format [4]. Then, CaSPL-gen executes a process, here denoted as Var-ModelExtender, that reads the FM, adds one or more contexts and extends the FM with VF. The obtained CFM is then converted and saved in a JSON file, according to the HyVarRec format [14]. CaSPL-gen repeats this process for all the required number of CFMs to generate.

The contexts are created randomly based on the restrictions set by the user. Every context domain is generated randomly and then a random value in that range is assigned to the context. After the generation of the contexts, Var-ModelExtender proceeds in the generation of the VFs. Following the approach of BeTTy for the attributes, for simplicity, VFs are generated only for leaf features.[4] A VF is generated as a conditional between a feature and a context, or between a feature and an expression involving a context and an attribute. In the first case the VF formula has the form $F = 1 \rightarrow C \circ v$, where $F = 1$ means that feature F is selected, $\circ$ is a comparison operator and v is a value in the range of the C domain. In the second case the VF has the form $F = 1 \rightarrow (C \circ v \rightarrow a \diamond w)$, where $\diamond$ is a comparison operator, a is an attribute and w is a value of the domain of a. The Var-Model Extender makes sure that the feature F is the owner of the attribute a, and that the values v and w are within the defined ranges of C and a. The comparison operators used are $=, \neq, >, <, \geq$ and $\leq$.

To generate a VF, one feature and one context are picked randomly among all the ones available. Which VF form is used is then selected randomly, except in one case: if the selected feature is in a path from the root of the FM that is composed of only mandatory features, the second form is selected (otherwise the first form would make the CFM likely void for some instances of the context). If the VF is of the second form, an attribute is generated and added to both F and to the VF. [5]

Default execution modality

While the simple execution modality of CaSPL-gen just enriches the output of BeTTy with contexts and VF, the default execution modality tries to minimize the number of void CFMs, i.e., CFMs that do not admit any valid configuration. Moreover, CaSPL-gen also applies strategies to avoid the generation of some dead features and dead trees, i.e., features and subtrees that can not be selected.

To reduce the number of void CFMs, the SAT-reasoner used within the BeTTy framework is applied to evaluate the FM generated with BeTTy, establishing whether

[4]Note that this is only a syntactic limitation: exploiting the expressiveness of VFs, it is always possible to normalize the CFM into a CFM having VF and attributes in its leaf features only.

[5]For the time being, CaSPL-gen supports features with maximum one attribute. Hence, in case more VFs are added to the same feature, the same attribute is re-used.

the model is void. A SAT-reasoner is a tool that employs a systematic backtracking search procedure to explore the space of variable assignments looking for satisfying a propositional formula and it is often used to check if FM without context have valid configurations [4]. If CaSPL-gen detects that the FM does not admit any single valid configuration, BeTTy is executed again until a valid FM is produced. A limit on the number of retries can be given in input as a parameter by the user.

Note that BeTTy supports the generation of attributes, but unfortunately it does not support the analysis of FM with attributes. Since in the advance modality we use the reasoner, differently from what is done in the simple execution modality, BeTTy is used to generate FMs without attributes that are later added when generating the VFs. In the default execution modality, context and VF are added as in the simple execution modality. As the contexts, attributes are added in a straightforward way by randomly selecting a leaf feature and a random range.

When the CFM is generated, techniques are used to discard CFMs that are too simple that selecting only a few number of features is enough to have a valid configuration. Indeed, real SPLs usually require the selection of more than one feature and therefore a CFM where the root has only optional relations usually does not represent real life problems. To avoid the generation of this kind of CFM, CaSPL-gen counts the paths from the root feature of the generated CFM to features at a given level in the CFM tree. The mandatory features are always followed. Instead, in case of Alternative-groups or optional features are encountered, the feature with the lowest number of paths is followed. CaSPL-gen discards a model if the number of paths is below a given threshold or, for complex models with considerable size, if the check runs out of time. The threshold is dependent on the size of the FM and can be set by the user.

CaSPL-gen also addresses a drawback of the BeTTy framework approach that when it generates CTCs it adds them by choosing first two sub-trees in the FM at random and then selecting a random feature in each of these sub-trees. In some cases, this mechanism allows the selection of two features that are not in separate sub-trees, thus possibly producing dead features and, in the worst case, it may make an entire sub-tree not selectable. Moreover, when a CTC is added, it may be the case that the constraint has no logical implications and is already captured by the structure of the FM. For example, consider the case where features F_a and F_b are mandatory children of F_p, and a CTC $F_a \rightarrow F_b$ is added. Since the structure of the FM already implies the constraints $F_a \rightarrow F_p$ and $F_p \rightarrow F_b$, the CTC is already implied. On the other hand, if instead the added CTC is $F_a \rightarrow \neg F_b$, we would have a contradiction to the constraint implied by the structure of the CFM. In this case F_p becomes a dead feature: no valid configuration of the CFM will be able to select it since if a configuration selects F_p it must also select F_a and F_b because of the mandatory relations, which contradicts the CTC. Since F_p is a root of a sub-tree, all the features in the sub-tree are dead features, making it also a dead sub-tree.

A dead sub-tree is not necessarily a significant problem if it only contains the three features above. However, if F_p is a child of the root, the situation may result in half of the features in the CFM being dead, which clearly is something we would like to avoid. This problem will most likely not occur if F_p is either the root or a feature in a path of mandatory features from the root. Indeed, in this case, the addition of the CTC would lead to the whole model being void, which is detected early by running the BeTTy reasoner. The SAT reasoner included in BeTTy however does not check for dead sub-trees. For this reason, to reduce the risk of having big dead sub-trees, CaSPL-gen checks for two common scenarios where a randomly generated CTC is not strictly a constraint between

features in two sub-trees. The first scenario occurs when the feature on the left-hand side F_l can logically be interpreted to be a predecessor of the feature on the right-hand side F_r (obviously the root feature of a sub-tree is a predecessor of all features in that sub-tree). The second scenario occurs instead when the two features F_l and F_r have both predecessors in the same Alternative-group. If CaSPL-gen finds that a CTC generated by BeTTy falls into one of these two categories, the CTC is not added into the final CFM.

To check for the occurrence of the first scenario, CaSPL-gen locates the feature closest to the root which has a mandatory path leading to F_l. Let us denote this mandatory predecessor as F_{ml} (note that if F_l is not mandatory then $F_l = F_{ml}$). If F_{ml} is a predecessor of F_r, the generated CTC is not included in the final CFM. The reason behind this action is that F_l must be selected if F_{ml} is selected. Assuming that F_{ml} is selected, if the CTC is of the form $F_l \rightarrow \neg F_r$, then F_r will never be selected, and the sub-tree of which it is a root is a dead sub-tree. A CTC on the form $F_l \rightarrow F_r$ creates instead a mandatory path from F_{ml} to F_r, no matter what relations already exist in the path $F_{ml}, \ldots, F_r$. This turns all Optional-relations and Or-relations into the path $F_{ml}, \ldots, F_r$ into mandatory relations, denaturing their intended meaning. In the case of Alternative-relations, the CTC does create dead features because in any Alternative-groups passed by the path $F_{ml}, \ldots, F_r$ there are features that can no longer be selected, meaning that they are dead features. Thus, the CTC is not added to the CFM.

To check for the occurrence of the second scenario, CaSPL-gen looks through all Alternative-groups and checks if they contain one predecessor of F_l and a different predecessor of F_r. In this case the CTC is removed. The logic behind this action is straightforward: let the predecessor of F_l be called F_{pl} and the predecessor of F_r be F_{pr}. If the CTC is of the form $F_l \rightarrow \neg F_r$ it is superfluous: F_l can only be selected if F_{pl} is selected, and F_r can only be selected if F_{pr} is selected. Since by definition of Alternative-groups F_{pl} and F_{pr} can not be selected at the same time, F_l and F_r can not be selected at the same time either. The contrapositive meaning of an Alternative-group is that for any descendant F_l' of F_{pl} and for any descendant F_r' of F_{pr} there is an implied constraint that one excludes the other. With that in mind, if the CTC is of the form $F_l \rightarrow F_r$ we have that F_l is a dead feature. Hence, also in this case, to avoid the presence of dead features and dead sub-trees it is better to remove the CTC.

Note that applying all these strategies helps avoiding some of the dead features that may be created, but it does not guarantee that the final model comes without dead features or that it is not void. Indeed, there may still exist less obvious relationships which ultimately creates dead features (e.g., CTCs contradicting each other) or void CFMs.

Usage

CaSPL-gen is open source and developed in Java ($\sim$ 2k lines of code). It is freely available from `https://github.com/magnurh/CaSPL`. In order to facilitate its use, we adopted the Gradle technology [10] to facilitate its deployment and a JSON file to set its parameters. Detailed instruction for the deployment and its usage are available in the repository Readme file.

As an example of the running time of CaSPL-gen, Table 2 reports some preliminary experiments showing how long it takes to generate benchmarks varying two of the key parameters: the number of features (*numberOfFeatures*) and the maximal number of context (*contextMaxSize*) of the generated CFMs, keeping the other parameters set to their default value.

contextMaxSize	numberOfFeatures			
	500	1000	2000	4000
10	367	1274	3458	21472
20	363	1270	5948	25590
40	513	1459	6065	28316
80	455	1769	6755	30614

Table 2: CaSPL-gen running time performance (in ms) varying two parameters.

4 Related work & Conclusions

The definition of random instances for evaluating the performance of algorithms has attracted a lot of attention in the last decades. For instance, in the SAT and Constraint community, different algorithms have been proposed to generate hard and realistic enough instances (see, e.g., [21]). In the SPL community, as reported in the comprehensive survey [5], numerous techniques and tools have proliferated to study and analyze FM. The rapid progress of this discipline is naturally leading to an increasing concern about the quality of the tools, especially when extensions of the standard FM formalism are taken into account. In this context, current testing methods are mainly guided by intuition rather than by well-studied testing techniques. This makes testing conclusions rarely rigorous and verifiable, weakening the value and scope of research contributions. The presence of a good selection of benchmarks alleviates this problem, allowing a rigorous comparison of the tools and in the discovery of bugs. However, creating good quality benchmark is a difficult and challenging task, due to the fact that a good benchmark should i) cover all the different types of instances that can be submitted to the tools, ii) not be bias toward a specific class of instances, iii) be small enough to allow a fast evaluation without wasting a lot of computational resources. Ideally, the best approach for the creation of a benchmark is the use of real feature models but, unfortunately, it is well known that companies are reluctant to show their feature models to avoid revealing to their competitors strategic business information. There are tentatives to derive variability models from open source software [8,23] but these approaches are usually tailored to the mined open source project, thus disallowing the production of benchmark with a big number of varying instances.

For this reason, usually random FMs are generated. One advantage of generating the FM randomly is that the results are not limited by the lack of generalization that one might get from hand-crafted or real models: randomly generated models might indeed take into account currently unusual, yet possible problems that production tool of tomorrow may face. Conversely, a disadvantage of random generated datasets is that there are no guarantees that generated CFMs represent real-life situations, possibly diverting the tool development towards instances that may never be encountered in practice, to the detriment of the tool performance on real instances.

Up to today, random generated benchmarks are the most used way for testing and comparing SPL reasoners and tools [16, 17, 22, 24]. However, all the available random generated benchmarks consider standard FM only, few times taking into account also the generation of attributes. To the best of our knowledge, CaSPL-gen is instead the first tool to be able to generate benchmarks for CFM, taking into account attributes and context.

In this work we have presented CaSPL-gen and how it can be used in a simpler or default modality, detailing the operation performed to reduce the likelihood to have void CFMs or a huge number of dead features. A preliminary version of CaSPL-gen has been

already used to evaluate the performance of some analysis conducted by the HyVarRec tool [15]. The CaSPL-gen has proven its usefulness by allowing to generate benchmarks of CFM with thousand of features and up to 10 contexts in few minutes.

As a future work, we are interested in extending CaSPL-gen by investigating how to further reduce the number of void CFMs that are generated and in speeding up the generation process. Moreover, we would like to study how the finding on the hardness of SAT problems [21] can be used in combination with techniques such as [13] to generate harder and more industrial like CFMs.

Acknowledgements

This work has been supported by the EU projects H2020-644298 *HyVar: Scalable Hybrid Variability for Distributed Evolving Software Systems* (`http://www.hyvar-project.eu`).

References

[1] M. Acher, P. Collet, F. Fleurey, P. Lahire, S. Moisan, and J.-P. Rigault. Modeling Context and Dynamic Adaptations with Feature Models. In *International Workshop Models@run.time*, page 10, 2009.

[2] L. Atzori, A. Iera, and G. Morabito. The Internet of Things: A Survey. *Comput. Netw.*, 54(15):2787–2805, 2010.

[3] M. Baldauf, S. Dustdar, and F. Rosenberg. A survey on context-aware systems. *IJAHUC*, 2(4):263–277, 2007.

[4] D. Benavides, P. T. Martín-Arroyo, and A. R. Cortés. Automated Reasoning on Feature Models. In *CAiSE*, volume 3520 of *LNCS*, pages 491–503. Springer, 2005.

[5] D. Benavides, S. Segura, and A. R. Cortés. Automated analysis of feature models 20 years later: A literature review. *Inf. Syst.*, 35(6):615–636, 2010.

[6] R. de Lemos et al. Software Engineering for Self-Adaptive Systems: A second Research Roadmap. volume 10431 of *Dagstuhl Seminar Proceedings*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany, 2010.

[7] P. Fernandes, C. Werner, and E. Teixeira. An Approach for Feature Modeling of Context-Aware Software Product Line. *J. UCS*, 17(5):807–829, 2011.

[8] J. A. Galindo, D. Benavides, and S. Segura. Debian Packages Repositories as Software Product Line Models. Towards Automated Analysis. In *International Workshop on Automated Configuration and Tailoring of Applications*, volume 688 of *CEUR Workshop Proceedings*, pages 29–34. CEUR-WS.org, 2010.

[9] H. Gomaa and M. Hussein. Dynamic Software Reconfiguration in Software Product Families. In *PFE*, volume 3014 of *LNCS*, pages 435–444. Springer, 2003.

[10] Gradle Inc. Gradle. `https://gradle.org/`.

[11] H. Hartmann and T. Trew. Using feature diagrams with context variability to model multiple product lines for software supply chains. In *SPLC*. IEEE Computer Society, 2008.

[12] K. Kang. *Feature-oriented Domain Analysis (FODA): Feasibility Study ; Technical Report CMU/SEI-90-TR-21 - ESD-90-TR-222*. Software Engineering Inst., Carnegie Mellon Univ., 1990.

[13] Y. Malitsky, M. Merschformann, B. O'Sullivan, and K. Tierney. Structure-Preserving Instance Generation. In *LION*, volume 10079 of *LNCS*, pages 123–140. Springer, 2016.

[14] J. Mauro, M. Nieke, C. Seidl, and I. C. Yu. Context Aware Reconfiguration in Software Product Lines. *VAMOS*, 2016.

[15] J. Mauro, M. Nieke, C. Seidl, and I. C. Yu. Anomaly Detection and Explanation in Context-Aware Software Product Lines. *SPLC*, 2017.

[16] M. Mendonça, M. Branco, and D. D. Cowan. S.P.L.O.T.: software product lines online tools. In *OOPSLA*, pages 761–762. ACM, 2009.

[17] M. Mendonça, D. D. Cowan, W. Malyk, and T. C. de Oliveira. Collaborative Product Configuration: Formalization and Efficient Algorithms for Dependency Analysis. *JSW*, 3(2):69–82, 2008.

[18] S. Neskovic and R. Matic. Context modeling based on feature models expressed as views on ontologies via mappings. *Comput. Sci. Inf. Syst.*, 12(3):961–977, 2015.

[19] M. Nieke, J. Mauro, C. Seidl, and I. C. Yu. User Profiles for Context-Aware Reconfiguration in Software Product Lines. In *ISoLA*, volume 9953 of *LNCS*, pages 563–578, 2016.

[20] K. Pohl, G. Böckle, and F. J. v. d. Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag New York, Inc., 2005.

[21] F. Rossi, P. v. Beek, and T. Walsh. *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*. Elsevier Science Inc., New York, NY, USA, 2006.

[22] S. Segura, J. A. Galindo, D. Benavides, J. A. Parejo, and A. R. Cortés. BeTTy: benchmarking and testing on the automated analysis of feature models. In *International Workshop on Variability Modelling of Software-Intensive Systems*, pages 63–71. ACM, 2012.

[23] S. She, R. Lotufo, T. Berger, A. Wasowski, and K. Czarnecki. The Variability Model of The Linux Kernel. In *International Workshop on Variability Modelling of Software-Intensive Systems*, volume 37 of *ICB-Research Report*, pages 45–51. Universität Duisburg-Essen, 2010.

[24] J. White, B. Dougherty, and D. C. Schmidt. Selecting highly optimal architectural feature sets with Filtered Cartesian Flattening. *Journal of Systems and Software*, 82(8):1268–1284, 2009.