Mining Feature Revisions in Highly-Configurable Software Systems

Gabriela Karoline Michelon[1,2], David Obermann[1], Wesley Klewerton Guez Assunção[3], Lukas Linsbauer[4], Paul Grünbacher[1], Alexander Egyed[1]
[1]Institute for Software Systems Engineering, Johannes Kepler University Linz, Austria
[2]LIT Secure and Correct Systems Lab, Johannes Kepler University Linz, Austria
[3]COTSI, Federal University of Technology - Paraná, PPGComp, Western Paraná State University, Brazil
[4]Institute of Software Engineering and Automotive Informatics, Technische Universität Braunschweig, Germany

ABSTRACT

Highly-Configurable Software Systems (HCSSs) support the systematic evolution of systems in space, i.e., the inclusion of new features, which then allow users to configure software products according to their needs. However, HCSSs also change over time, e.g., when adapting existing features to new hardware or platforms. In practice, HCSSs are thus developed using both version control systems (VCSs) and preprocessor directives (`#ifdefs`). However, the use of a preprocessor as variability mechanism has been criticized regarding the separation of concerns and code obfuscation, which complicates the analysis of HCSS evolution in VCSs. For instance, a single commit may contain changes of totally unrelated features, which may be scattered over many variation points (`#ifdefs`), thus making the evolution history hard to understand. This complexity often leads to error-prone changes and high costs for maintenance and evolution. In this paper, we propose an automated approach to mine HCSS features taking into account evolution in space and time. Our approach uses constraint satisfaction problem solving to mine newly introduced, removed and changed features. It finds a configuration containing the feature revisions which are needed to activate a specific program location. Furthermore, it increments the revision number of each changed feature. Thus, our approach enables to analyze when and which features often change over time, as well as their interactions, for every single commit of a HCSS. Our approach can contribute to future research on understanding the characteristics of HCSS and supporting developers during maintenance and evolution tasks.

CCS CONCEPTS

• **Software and its engineering → Preprocessors**; **Software product lines**; *Traceability*; *Reusability*.

KEYWORDS

system evolution, software product lines, preprocessors, feature evolution, version control systems, repository mining

SPLC '20 Companion, October 19–23, 2020, MONTREAL, QC, Canada

ACM ISBN 978-1-4503-7570-2/20/10…$15.00
https://doi.org/10.1145/3382026.3425776

ACM Reference Format:
Gabriela Karoline Michelon[1,2], David Obermann[1], Wesley Klewerton Guez Assunção[3], Lukas Linsbauer[4], Paul Grünbacher[1], Alexander Egyed[1]. 2020. Mining Feature Revisions in Highly-Configurable Software Systems. In *24th ACM International Systems and Software Product Line Conference Companion (SPLC '20 Companion), October 19–23, 2020, MONTREAL, QC, Canada.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3382026.3425776

1 INTRODUCTION

A software system usually must be delivered with different configurations of features, with each feature representing a functionality of the system accessible to developers and users. To remain competitive, companies have to satisfy different customer needs of the market segment they serve. Software Product Line (SPL) engineering is a systematic approach to deal with the development of customized system products. An SPL is a set of software-intensive systems that share a common set of artifacts developed in a prescribed way to facilitate their systematic reuse [7]. The customized software products, a.k.a. variants, result from the derivation of SPL artifacts, i.e., the selection of a different set of features that are of interest to a customer. To allow customization, the features of an SPL is implemented using variability mechanisms [1].

A widely used variability mechanism in SPLs is based on annotations [26]. Annotations rely on preprocessor directives such as `#ifdef` and `#endif` which enclose blocks of variable code and enable to tailor system variants to different hardware platforms, operating systems, and application scenarios [23]. Annotation-based SPLs are often implemented as Highly-Configurable Software Systems (HCSSs) [16]. HCSSs use techniques such as feature flags, feature toggles, or feature switches, to turn on configuration options/features needed to be included in a product [8, 18, 27]. However, features also need to evolve over time. For instance, when a specific feature is adapted to a new hardware platform, then a new version of a variant is created. This evolution in time [30] is aided by some tools such as version control systems (VCSs) [28].

However, despite the benefits of managing HCSSs in VCSs, they are hardly integrated to support both evolution in space and time [21, 22]. For example, when evolving HCSSs in VCSs, developers often commit unrelated or loosely related implementations of features [13]. Then, evolving a particular feature requires to find the implementation artifacts over many `#ifdefs`, compromising code comprehension and complicating maintenance and evolution tasks [9].

In this paper, we present an automated approach[1] for mining HCSSs managed in VCSs to obtain information of the evolution of features in both space and time. For every repository commit, we mine the features that were introduced, changed, and removed. Thus, our approach enables to automatically retrieve the features that evolved in each point in time for every change in the code. The approach takes into account all subsequent lines of code and solves a Constraint Satisfaction Problem (CSP) [5, 29] to assign feature revisions to a specific changed block of code. In addition, our approach finds a configuration for every changed block of code in a Git commit that activates that specific program location, thereby easing the analysis of feature interactions [2].

2 MOTIVATING EXAMPLES

The complexity of HCSSs implementation often makes maintenance, evolution, and testing activities time consuming and error-prone tasks. This happens mainly because source code cluttered with preprocessor directives is difficult to understand [14]. Complex systems have many features which are annotated across many files, and which often depend on or interact with other features. This makes it hard, for instance, to determine without an automated mechanism which specific feature has a bug or causes other faults.

Imagine an HCSS managed in a VCS, which has been evolved for a while. If a bug is reported by the users, the developers need to find where and when the bug was introduced and which features it affects. Developers fixing the bug may need to look through the entire VCS version history to find the commit introducing the defective code. However, manually retrieving the changes related to the desired feature is a complex task, especially when multiple features are changed or added in a single commit [4, 17, 31].

Concrete examples can be found in the commits of the LibSSH[2] system. Analyzing the version history we can see that many changes were made in a single commit (77603db), containing changes of refactoring, cleanup debugging messages, inclusion and enhancing of features, and bug fixing. In this same commit, 15 files were changed, representing a total of 415 additions and 338 deletions. To associate the different changes to specific features, firstly, we have to analyze which features really changed. Performing a manual analysis over all the files requires also to analyze each `#ifdef` block as well as `#defines` and `#includes` directives in the code to correctly assign a change to a feature. Doing this manually can become infeasible and a cumbersome when features have high degrees of scattering, tangling, and nesting [26].

It has been shown that the commit messages often do not reflect the actual changes performed [4]. For instance, the commit 6f47401 of the LibSSH system contains the code implementing the feature `HAVE_SSH1` but also changed the feature name in the `#ifdef` annotation to `WITH_SSH1`. This kind of changes can easily lead to a misunderstanding of features. For example, if a customer using a system version delivered before this name change reports a bug, and the developers try to find the bug by looking for the feature `HAVE_SSH1` in a version of the system with the aforementioned commit, they will be misled, since, at that time, the problem is actually located in the feature `WITH_SSH1`.

[1]https://github.com/GabrielaMichelon/git-ecco
[2]https://gitlab.com/libssh/libssh-mirror/

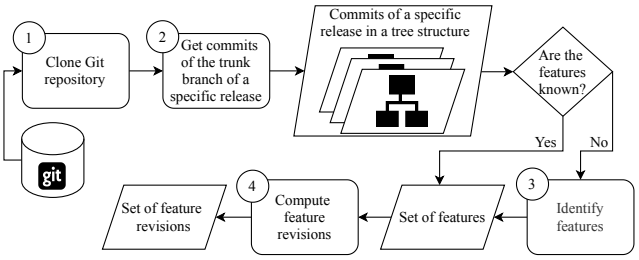

Figure 1: Approach overview.

The examples show that recovering feature implementations is a complex and costly task in HCSSs evolution, which suggests an automated mechanism to retrieve the added and removed features as well as their changes, i.e., revisions.

3 APPROACH

We present an approach for mining feature revisions of HCSSs, which are managed in VCSs. We describe its main steps, input, and output, as well as the internal representation of artifacts. From now on, we refer to feature revision as the change in a feature at a specific point in time, i.e., in a Git commit. Figure 1 presents an overview of our automated approach. Firstly, the Git repository of the HCSS is cloned (step 1) and all commits of the main branch of a specific release, i.e., a Git tag, are retrieved and represented in a tree-like structure (step 2). Since the goal of our approach is to mine feature revisions, we must know the HCSS features. If they are known in advance, they are provided as input for step 4, if not, step 3 is responsible for automatically identifying the features implemented in the HCSS by exploring the tree structure containing the files and source code of each commit of the release. In step 4, our approach performs the process of assigning feature revisions to the changes.

Below, we describe in detail how the tree-like representation of the artifacts is created and we explain the steps of identifying features (step 3) and computing feature revisions (step 4). To exemplify these activities, we use the running example presented in Listing 1.

Artifact representation. Existing tools such as TypeChef [20], SuperC [10], and KernelHaven [19] allow transforming systems that are implemented in C and annotated with preprocessor directives into an Abstract Syntax Tree (AST). In addition, TypeChef and SuperC represent the variability in the AST in the form of choice nodes. However, for our purpose, we only need nodes at the level of preprocessor directives and it would be computationally too expensive and time-consuming to analyze all commits of a system at the AST level. Thus, we decided to create our own tree structure suitable for our approach, which only needs to distinguish the preprocessor directives to easily build constraints for CSP problems and to identify the features of our subject systems, which do not have a variability model and tristate type such as the Linux Kernel.

Therefore, in our approach, the artifacts, i.e., the source code and any other files, are represented based on a tree-like structure. For this, we assume that conditional blocks wrap code that may belong to one feature, multiple features, or no feature. An example of such code blocks is presented in Listing 1. The Lines 1-4 are part

```
1  #ifdef WITH_SERVER
2      #define _LIBSSH_H
3      #define MD5_DIGEST_LEN 16
4  #endif
5
6  #ifdef __cplusplus
7      #if _LIBSSH_H && MD5_DIGEST_LEN > 5
8          <code>
9      #endif
10 #endif
```

Listing 1: Conditional blocks of feature implementations.

of a conditional block of the feature WITH_SERVER, while the block from Lines 6-10 is part of the feature __cplusplus. We can also see an internal block within the conditional block of code of the feature __cplusplus in Lines 7-9. Using this information we can build a tree structure to represent the artifacts of the repository at a certain point in time, i.e., for a specific Git commit. The tree structure contains the content of each source code, text or binary file. For the source code files, e.g., .c/.cpp, we add child nodes. The child nodes can be conditional nodes, i.e., #ifdef, #if, #ifndef, #elif, #else; definition nodes, i.e., #define and #undef directives; or import nodes, containing #include directives. The lines of code inside a source file that are not wrapped by a conditional block will be part of an #if conditional node belonging to the feature BASE. The #include nodes contain child nodes corresponding to the source code of each included file. For the other files, we do not add define and include nodes, we just consider all their lines belonging to a conditional node containing the feature BASE, i.e., a code that belongs to the base of the project and not to a particular feature of the system.

As we can see in Figure 1, the tree is built from the output of step 2. From this step, we get a partially preprocessed version of the code, which consists of resolving macros in the statements of the conditional blocks of code [18]. For example, if we have: #define FEAT_A(X,Y) X+Y, then, the partial preprocessing of: #if FEAT_A(FEAT_B,FEAT_C) > 10, would result in: #if (FEAT_B+FEAT_C) > 10. This process is necessary to correctly get all the features from the conditions. All #define and #include directives remain in the partially preprocessed files. Finally, we transform these files into the tree structure.

Identify Features. After cloning the Git repository (step 1) and obtaining the commits of a specific release into the tree structure (step 2), our approach proceeds with step 3, which extracts all macros from the conditional and define nodes. Identifying possible features is based on classifying the macros inside conditional nodes within three classes: external, internal, and transient, where only the external will be the features of the system. *External macros* can only be defined externally and represent the features that are selected or not when creating a variant of the SPL. Therefore, the conditional blocks of code with external macros are the variation points. For example, in Listing 1, WITH_SERVER and __cplusplus are the external macros. *Internal macros* are defined at some point in the code via a #define directive. In Listing 1 _LIBSSH_H and MD5_DIGEST_LEN are internal macros, defined in Lines 2 and 3, respectively. *Transient macros* are used in some Git commits as external and internal at the same release, i.e., in some Git commits a specific macro is used in a condition and never defined in the system code, but in some other Git commits, it is defined in the source files. We thus compute as features the macros classified as external in all commits of a specific release. The set of features obtained as output is not limited to be exactly those features of the external macros. Therefore, the output can be manually adjusted to the array of features in the program, if necessary, before being used as input for the process of computing feature revisions (step 4).

Compute Feature Revisions. Now, we get each changed code block for each commit of the HCSS release. The approach compares the files of the commit n with the commit $n-1$ to get the differences from one point in time to another. For this, it gets the tree structure created for each commit of the release and obtains the changes mapped to the nodes of the tree structure, which returns the changed nodes. The obtained set of changed nodes allows creating the constraints to represent our problem. The basic idea here is to create a set of constraints that will be handed to a solver. Solving these constraints delivers the assignment for the macros, which we will need for computing the feature revisions.

To understand how the set of constraints is created we can look at Listing 1 and assume that the code in Line 8 changed. We first get the local condition, i.e., the condition that is closest to the changed code, which will be in this case the condition in Line 7 (_LIBSSH_H && MD5_DIGEST_LEN > 5). Next, we get the second part of the constraint, which is the condition of the closest block with a conjunction of all conditions of its parents' blocks. We obtain the conditions of the parents' blocks by walking up the tree, starting from the changed node. In the example, this would be __cplusplus && (_LIBSSH_H && MD5_DIGEST_LEN > 5). The constraint needs a mapping of all internal macros to external macros that influence the activation of the code block of the changed node because the external macros are our set of features. In Listing 1 we see that WITH_SERVER defines _LIBSSH_H and MD5_DIGEST_LEN=16. This means we can map the internal macros of the changed node to the code block of the feature WITH_SERVER. This mapping will be a queue of implications for the internal macros _LIBSSH_H and MD5_DIGEST_LEN. The queue of implications is built by traversing the tree of the corresponding source file. When a #define is found, the approach takes the condition of the conditional code block containing the #define directive. Note that when a #define is not wrapped by an #ifdef, the tree structure will contain the BASE conditional block as a parent node. With this, we form an implication, similar to Nadi et al. [24], in the form of $(Condition \rightarrow (Macro = Value)) \wedge (\neg Condition \rightarrow \neg Macro)$ for the base case, or of the form $(Condition \rightarrow (Macro = Value)) \wedge (\neg Condition \rightarrow ElsePart)$ for the non-base cases. The first part of the implication is the base case, which means that if there are no further implications the macros have $value = false$. In case there are further implications, they are concatenated in the same way and placed as the $ElsePart$. This representation makes it possible to overwrite assignments of macros via #defines. By using this approach we can simulate the behavior of the preprocessor.

Having all the constraints created, the approach has to provide an adequate assignment. For this, a solver is needed. TypeChef uses an SAT solver to analyze ifdef variability in C code. However, the statements in the conditional code blocks may not only contain basic logic operations and Boolean values but can also involve

basic arithmetic operations and comparisons, as well as numeric values in the range of integer or double. Thus our approach uses the ChocoSolver [3]. We provide the first possible solution for a given constraint in step 4, since getting the optimal solution increases significantly the computation effort. After adding all these parts to the solver, we retrieve a solution in case a solution exists for the given constraints. If the solver finds no solution it means that the part of code we wanted to activate is dead, i.e., there is no configuration that can activate it.

To compute the revisions, we chose a particular heuristic that considers as changed features only the features closest to the changed node that are assigned the value 1 by the solution given by the solver. Initially, we do not consider for a new feature revision all parent nodes that wrap the changed node. When more than one parent node exists, only the closest is considered as changed. If the macros of the changed node are external, the approach considers the features of the condition of the changed node as new revisions. When this is not the case, as in our example shown in Listing 1 in Line 7, the approach looks at the condition of the closest parent node. We repeat the same process with parent conditions until we find a solution. When no positive features are wrapping the changed code, i.e., solution $\neq 1$, then we assign the change as a new revision of the feature `BASE`. In our example in Listing 1, a change in Line 8 will lead to a solution from the solver: $WITH\_SERVER = 1$ and $\_\_cplusplus = 1$. Thus, the approach assigns this change to the closest parent node, i.e., the feature `__cplusplus`, thus leading to a new revision caused by the change in Line 8.

Output results. The output of our mining approach will be a configuration to activate the changed code (Line 8 in Listing 1): `BASE.1`, `WITH_SERVER.1`, `__cplusplus.2`. In case of a conditional expression of a feature interaction, for example, `WITH_SERVER && __cplusplus`, the configuration will contain an incremented revision of both `WITH_SERVER` and `__cplusplus`. A dataset of the LibSSH system is available at https://github.com/jku-isse/LibSSH-dataset. It contains for every release: the classification of the macros; the considered set of features; the Git commits where a feature was present; the number of Git commits that a feature changed, i.e., had new revisions; the number of deleted features per Git commit; the configuration for every changed block of code in a Git commit; the number of Git commits that a feature was removed; and the number of new and changed features per Git commit.

4 RELATED WORK

Recent work from Gazzillo et al. [11] poses a challenge to propose an approach to apply an automatic analysis to find concrete configurations that include a specific program location. Our approach is able to find configurations of a specific block of code and may help the software testing and maintenance. Because knowing which feature(s) and revision(s) are needed to be selected or not to execute a specific block of code, can easier to identify bugs or defects.

An approach to automatically identify and summarize features in forks of a project is proposed by Zhou et al. [31]. It is based on source code analysis and on a cluster of changes by information-retrieval to identify keywords of features developed in forks. Their approach is not focused on mining feature revisions on tangled changes within preprocessor directives from HCSSs. Our approach is focused to mine feature revisions of changes in single commits, and thus, can also be applied to identify which changes in forks correspond to which feature revisions.

Passos et al [25] studied the scattering of features in some releases of the Linux kernel. Also, Chaikalis et al. [6] analyzed the feature scattering but based on the number of classes and methods involved in the implementation of a certain feature over versions (not Git commits) of Java projects. Godfrey and Qiang [12] studied the Linux evolution in terms of lines of code. Israeli and Feitelson [15] studied the Linux evolution in relation to the complexity of functions. The FEVER approach [8] is also applied to extract information of the Linux kernel evolution, extracting information about which features changed in the variability models (KConfig files), assets (preprocessor based C code), and mappings (Makefiles). They considered as features the ones defined in the KConfig file, and the source files are the ones mapped to the features in the Makefiles. However, some files in the Linux kernel cannot be mapped directly to features, such as `#include` files.

Our approach differs from the existing ones as we get features from the external macros. We then analyze every constraint from all subsequent lines of the source code up to the `#ifdefs` or line of code that changed to compute a possible solution of which feature changed in a specific Git commit. Furthermore, we provide an automated mechanism for our approach, which can retrieve the feature revisions over all commits of HCSSs that do not rely on KConfig and Makefiles. Our approach captures the features added or removed based on the ifdefs containing or not containing the external macros (our default set of features) instead of evaluating the variability model as done by FEVER. This does not limit our approach to systems containing a variability model. Furthermore, our approach mines automatically which features have been changed, i.e., the feature revisions over all commits of all releases of a system.

5 CONCLUSIONS AND FUTURE WORK

This paper presented an approach and an automated mechanism to mine feature revisions over all commits of HCSSs developed in VCSs. Our goal is to provide information on how the variable systems continuously evolve in space and time, i.e., about the features that have been introduced and changed, aiming to ease the maintenance and evolution tasks and to benefit both the research and the practice of HCSSs. Specifically, we believe that our approach and automated mechanism can help developers and the research community to further explore at what level features have been changed, to what degree their changes have been affecting the implementation of other features, and hence, the system behavior.

ACKNOWLEDGMENTS

This research was funded by the LIT Secure and Correct Systems Lab; the Austrian Science Fund (FWF), grant no. P31989; Pro2Future, a COMET K1-Centre of the Austrian Research Promotion Agency (FFG), grant no. 854184; CNPq, grant no. 408356/2018-9 and FAPPR, grant no. 51435. The support by the Austrian Federal Ministry for Digital and Economic Affairs and the National Foundation for Research, Technology and Development is gratefully acknowledged.

REFERENCES

[1] Sven Apel, Don Batory, Christian Kstner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines: Concepts and Implementation.* Springer Publishing Company, Incorporated, New York, NY, USA.

[2] Sven Apel, Sergiy Kolesnikov, Norbert Siegmund, Christian Kästner, and Brady Garvin. 2013. Exploring Feature Interactions in the Wild: The New Feature-Interaction Challenge. In *Proceedings of the 5th International Workshop on Feature-Oriented Software Development* (Indianapolis, Indiana, USA) *(FOSD '13)*. Association for Computing Machinery, New York, NY, USA, 1–8. https://doi.org/10.1145/2528265.2528267

[3] IMT Atlantique. 2020. *Choco-Solver*. IMT Atlantique. Retrieved September 16, 2020 from https://choco-solver.org/

[4] Mike Barnett, Christian Bird, João Brunet, and Shuvendu K. Lahiri. 2015. Helping Developers Help Themselves: Automatic Decomposition of Code Review Changesets. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1* (Florence, Italy) *(ICSE '15)*. IEEE Press, San Francisco, CA, USA, 134–144. https://doi.org/10.1109/ICSE.2015.35

[5] David Benavides, Sergio Segura, Pablo Trinidad, and Antonio Ruiz-Cortés. 2006. *Using Java CSP Solvers in the Automated Analyses of Feature Models*. Springer Berlin Heidelberg, Berlin, Heidelberg, 399–408. https://doi.org/10.1007/11877028_16

[6] Theodore Chaikalis, Alexander Chatzigeorgiou, and Georgina Examiliotou. 2013. Investigating the effect of evolution and refactorings on feature scattering. *Software Quality Journal* 23, 1 (May 2013), 79–105. https://doi.org/10.1007/s11219-013-9204-4

[7] Paul C. Clements and Linda Northrop. 2001. *Software Product Lines: Practices and Patterns*. Addison-Wesley, Boston, MA, USA.

[8] 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* 23, 2 (2018), 905–952. https://doi.org/10.1007/s10664-017-9557-6

[9] Gabriel Ferreira, Momin Malik, Christian Kästner, Jürgen Pfeffer, and Sven Apel. 2016. Do #ifdefs Influence the Occurrence of Vulnerabilities? An Empirical Study of the Linux Kernel. In *Proceedings of the 20th International Systems and Software Product Line Conference* (Beijing, China) *(SPLC '16)*. Association for Computing Machinery, New York, NY, USA, 65–73. https://doi.org/10.1145/2934466.2934467

[10] Paul Gazzillo and Robert Grimm. 2012. SuperC: Parsing All of C by Taming the Preprocessor. *SIGPLAN Not.* 47, 6 (June 2012), 323–334. https://doi.org/10.1145/2345156.2254103

[11] Paul Gazzillo, Ugur Koc, ThanhVu Nguyen, and Shiyi Wei. 2018. Localizing Configurations in Highly-Configurable Systems. In *Proceedings of the 22nd International Systems and Software Product Line Conference - Volume 1* (Gothenburg, Sweden) *(SPLC '18)*. Association for Computing Machinery, New York, NY, USA, 269–273. https://doi.org/10.1145/3233027.3236404

[12] Godfrey and Qiang Tu. 2000. Evolution in open source software: a case study. In *Proceedings International Conference on Software Maintenance ICSM-94* (San Jose, CA, USA). IEEE Comput. Soc. Press, San Francisco, CA, USA, 131–142. https://doi.org/10.1109/icsm.2000.883030

[13] Kim Herzig and Andreas Zeller. 2013. The Impact of Tangled Code Changes. In *10th Working Conference on Mining Software Repositories (MSR '13)*. IEEE Press, San Francisco, CA, USA, 121–130. https://doi.org/10.1109/MSR.2013.6624018

[14] Wanja Hofer, Christoph Elsner, Frank Blendinger, Wolfgang Schröder-Preikschat, and Daniel Lohmann. 2010. Leviathan: SPL Support on Filesystem Level. In *Software Product Lines: Going Beyond*, Jan Bosch and Jaejoon Lee (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 491–491. https://doi.org/10.1007/978-3-642-15579-6_43

[15] Ayelet Israeli and Dror G. Feitelson. 2010. The Linux kernel as a case study in software evolution. *Journal of Systems and Software* 83, 3 (March 2010), 485–501. https://doi.org/10.1016/j.jss.2009.09.042

[16] Dongpu Jin, Myra B. Cohen, Xiao Qu, and Brian Robinson. 2014. PrefFinder: Getting the Right Preference in Configurable Software Systems. In *29th ACM/IEEE International Conference on Automated Software Engineering* (Vasteras, Sweden) *(ASE '14)*. ACM, New York, NY, USA, 151–162. https://doi.org/10.1145/2642937.2643009

[17] Christian Kastner, Alexander Dreiling, and Klaus Ostermann. 2014. Variability Mining: Consistent Semi-Automatic Detection of Product-Line Features. *IEEE Trans. Softw. Eng.* 40, 1 (Jan. 2014), 67–82. https://doi.org/10.1109/TSE.2013.45

[18] Christian Kästner, Paolo G. Giarrusso, Tillmann Rendel, Sebastian Erdweg, Klaus Ostermann, and Thorsten Berger. 2011. Variability-Aware Parsing in the Presence of Lexical Macros and Conditional Compilation. *SIGPLAN Not.* 46, 10 (Oct. 2011), 805–824. https://doi.org/10.1145/2076021.2048128

[19] Christian Kröher, Sascha El-Sharkawy, and Klaus Schmid. 2018. KernelHaven: An Open Infrastructure for Product Line Analysis. In *Proceedings of the 22nd International Systems and Software Product Line Conference - Volume 2* (Gothenburg, Sweden) *(SPLC '18)*. Association for Computing Machinery, New York, NY, USA, 5–10. https://doi.org/10.1145/3236405.3236410

[20] Christian Kästner. 2013. *TypeChef*. Christian Kästner. Retrieved September 15, 2020 from https://ckaestne.github.io/TypeChef/

[21] Lukas Linsbauer, Thorsten Berger, and Paul Grünbacher. 2017. A Classification of Variation Control Systems. *SIGPLAN Not.* 52, 12 (Oct. 2017), 49–62. https://doi.org/10.1145/3170492.3136054

[22] Lukas Linsbauer, Felix Schwägerl, Thorsten Berger, and Paul Grünbacher. 2021. Concepts of Variation Control Systems. *Journal of Systems and Software* 171 (2021), 110796. https://doi.org/10.1016/j.jss.2020.110796

[23] F. Medeiros, M. Ribeiro, R. Gheyi, S. Apel, C. Kästner, B. Ferreira, L. Carvalho, and B. Fonseca. 2018. Discipline Matters: Refactoring of Preprocessor Directives in the #ifdef Hell. *IEEE Transactions on Software Engineering* 44, 5 (May 2018), 453–469. https://doi.org/10.1109/TSE.2017.2688333

[24] S. Nadi, T. Berger, C. Kästner, and K. Czarnecki. 2015. Where Do Configuration Constraints Stem From? An Extraction Approach and an Empirical Study. *IEEE Transactions on Software Engineering* 41, 8 (2015), 820–841. https://doi.org/10.1109/TSE.2015.2415793

[25] Leonardo Passos, Jesús Padilla, Thorsten Berger, Sven Apel, Krzysztof Czarnecki, and Marco Tulio Valente. 2015. Feature Scattering in the Large: A Longitudinal Study of Linux Kernel Device Drivers. In *Proceedings of the 14th International Conference on Modularity* (Fort Collins, CO, USA) *(MODULARITY 2015)*. Association for Computing Machinery, New York, NY, USA, 81–92. https://doi.org/10.1145/2724525.2724575

[26] Rodrigo Queiroz, Leonardo Passos, Tulio Marco Valente, Claus Hunsen, Sven Apel, and Krzysztof Czarnecki. 2017. The shape of feature code: an analysis of twenty C-preprocessor-based systems. *Software and Systems Modeling (SoSyM)* 16 (2017), 77–96. https://doi.org/10.1007/s10270-015-0483-z

[27] Md Tajmilur Rahman, Louis-Philippe Querel, Peter C. Rigby, and Bram Adams. 2016. Feature Toggles: Practitioner Practices and a Case Study. In *Proceedings of the 13th International Conference on Mining Software Repositories* (Austin, Texas) *(MSR '16)*. Association for Computing Machinery, New York, NY, USA, 201–211. https://doi.org/10.1145/2901739.2901745

[28] Nayan B. Ruparelia. 2010. The History of Version Control. *SIGSOFT Softw. Eng. Notes* 35, 1 (Jan. 2010), 5–9. https://doi.org/10.1145/1668862.1668876

[29] Thomas Schiex and Simon de Givry (Eds.). 2019. *Principles and Practice of Constraint Programming* (Stamford, CT, USA). LNCS '19, Vol. 11802. Springer.

[30] Christoph Seidl, Ina Schaefer, and Uwe Aßmann. 2014. Integrated Management of Variability in Space and Time in Software Families. In *Proceedings of the 18th International Software Product Line Conference - Volume 1* (Florence, Italy) *(SPLC '14)*. Association for Computing Machinery, New York, NY, USA, 22–31. https://doi.org/10.1145/2648511.2648514

[31] Shurui Zhou, Ştefan Stănciulescu, Olaf Leßenich, Yingfei Xiong, Andrzej Wąsowski, and Christian Kästner. 2018. Identifying features in forks. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. ACM, New York, NY, USA, 105–116. https://doi.org/10.1145/3180155.3180205