Managing Systems Evolving in Space and Time: Four Challenges for Maintenance, Evolution and Composition of Variants

Gabriela Karoline Michelon[1,2], David Obermann[1], Wesley K. G. 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]Pontifical Catholic University of Rio de Janeiro & PPGComp - Western Paraná State University, Brazil
[4]Institute of Software Engineering and Automotive Informatics, Technische Universität Braunschweig, Germany

ABSTRACT

Software companies have to provide a large set of features, i.e., functional and non-functional requirements to cope with *variability in space* and satisfy a large number of customers. Feature location techniques support many activities, such as software maintenance and evolution tasks. So far, there is only one feature location technique that considers system variants *evolution in time*, which is required for feature enhancements and bug fixing. Changing features over time results in feature revisions and requires feature location techniques for keeping track of changes and constantly updating existing variants. Existing tools for managing a set of systems over time do not offer proper management to keep track of feature revisions, updating existing variants, or creating new configurations with features from different points in time, i.e., features revisions. Thus, this paper presents four challenges for improving or introducing approaches for feature/feature revision location as well as for composing new configurations with features or feature revisions. We also provide a benchmark with a ground-truth and metrics computation to motivate the researchers to provide approaches that can be reproducible and comparable to provide better tools for the systems evolving in space and time. Thus, we do not limit the evaluation of techniques with only this benchmark as well we provided instructions on how to use a benchmark extractor for generating ground-truth for other systems. We expect that the feature/feature revision location techniques maximize information retrieval by measuring precision, recall, and F-score, and with less time and memory consumption as possible.

CCS CONCEPTS

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

KEYWORDS

feature location, feature revision, software product line, repository mining, benchmark extractor

SPLC'21, 06–11 September, 2021, Leicester, UK

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

ACM Reference Format:
Gabriela Karoline Michelon[1,2], David Obermann[1], Wesley K. G. Assunção[3], Lukas Linsbauer[4], Paul Grünbacher[1], Alexander Egyed[1]. 2021. Managing Systems Evolving in Space and Time: Four Challenges for Maintenance, Evolution and Composition of Variants. In *25th ACM International Systems and Software Product Line Conference (SPLC '21), September 06–11, 2021, Leicester, United Kingdom.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Software companies have to tailor and maintain variants of systems co-existing simultaneously to serve different customers and new requirements. A variant of a system has a different set of features, i.e., variable assets compared to existing system variants [3]. The system variants with different configurations are part of the *variability in space* as referred by some authors [28, 29]. The *variability in time* results from the need of modifying a variant due to enhancements, for example, to suit the new customers' requirements and environments, such as alternative hardware or the optimization of non-functional properties [29]. This can require the introduction of new features in the existing variants over the system life cycle, a.k.a *evolution in space*. Still, the features of a system can be subject to failures or unwanted behaviors and bug fixes have to be done, introducing new versions of features, which is referred to as *evolution in time* [28]. Furthermore, the evolution in time results in variant revisions, which are sequential versions of a variant, containing different artifacts for the same configuration, i.e., set of features [4, 16].

The scenarios aforementioned lead to many system variants and, consequently, the need of managing and evolving every system's variants in parallel. This highly increases the workload of developers. Furthermore, keeping the system variability consistent across different types of artifacts manually is an error-prone task [16]. Software product line (SPL) is an approach that has been adopted by engineers for systematic variability management and reuse of the core assets and features' artifacts, which enables faster production of variants of a system on a large scale, reduction of the effort and costs to maintain and create products of a system [12]. In the transition of variants, using opportunistic reuse to create an SPL, feature location is the first and most essential task of the re-engineering process to migrate a family of existing system variants into an SPL [2].

However, feature location and SPLs cover the *space* dimension but not the *time* dimension. SPLs standalone do not provide proper

management of evolution in time and engineers have to combine additional mechanisms/tools for managing systems evolving in space and time. Despite SPLs have been implemented in version control systems (VCSs), which track changes of a system over time [5], the VCS has support for managing the versions of variants but not for managing versions of features. Some pieces of work point out the need for an SPL reusing potential versions of a feature that co-exist, as well as having a portfolio that reflects the versions of a feature, i.e., the feature revisions of a sytem [4, 16, 24].

Existing feature location techniques can locate features of a system [6, 21, 25] or a set of systems [1, 22] but only at one point in time. Even there are some feature location techniques for the *space* dimension, they also have limitations as presented in our previous work [21]. Still, regarding the *time* dimension, there is only one feature revision location technique able to retrieve traces of feature revisions [24], which is in the early stages of development, with suboptimal results and scalability limitations in terms of the number of feature revisions that can be located.

We thus stress the need of introducing new or improving existing, feature/feature revision location techniques by describing four challenges to be solved by the research community and tool developers. These challenges are concerned with locating features at one point in time as well at multiple points in time (Section 2). Yet, the feature/feature revision location techniques proposed must be developed in such an approach that enables engineers to create new configurations with the set of features and feature revisions traced. By proposing our challenges, we aim to motivate the tool developers not only to optimize and cover limitations of existing feature/feature revision location techniques, as well to instigate that more efficient mechanisms/tools are developed for easier management of systems evolving in space and time.

Solutions evaluating their techniques with a common benchmark enable future work comparisons [21]. For this purpose, we contribute by making available not only a benchmark as well a ground-truth extractor[1] for evaluating these techniques. Thus, the benchmark contains: (i) a ground-truth dataset[2] with variants from three C open-source systems evolving in space and time with their respective configurations at one point and multiple points in time; (ii) tool utils, available in our Git repository, to evaluate the efficiency of a feature/feature revision location technique that computes automatically three metrics: precision, recall, and f-score.

The remainder of this paper is structured as follows. Section 2 presents the motivation and challenges of this paper. Section 3 provides detailed information of the benchmark, such as the scenarios and metrics for solutions evaluation, and a brief explanation of the ground-truth extractor. Section 4 concludes the paper.

2 THE CHALLENGES

We present now, to describe existing challenges, a background of feature/feature revision location. Then we explain the importance of these tasks, their current limitation, and existing limitations, observed complexity, or lack of studies, which makes the challenges interesting.

[1]https://github.com/GabrielaMichelon/git-ecco/tree/challenge
[2]http://doi.org/10.5281/zenodo.4586774

2.1 Feature Location

A feature can be a functional or non-functional requirement that represents a software system functionality [6]. In order to better explain, let's use the Marlin system as an example. Marlin is an open-source firmware for 3D printers. It has many features, including features for linear acceleration, control of the temperature to melt the filament, a buzzer sounds to warning signals [13]. Some of the features are optional features, i.e., not all products of a system have to include them. These optimal features are thus units of variability because they are responsible for changing the system functionalities and behaviors.

Marlin is an annotated SPL, however, according to a study from Krüger et al. [13], not all optional features are used in variation points, e.g., `#ifdef` preprocessor directives. The maintenance and evolution tasks need their whole actual location that can be out of annotations to be recovered, and it requires manual work that could be easier and automated by feature location techniques. Therefore, there is no question about the importance of feature location techniques, which are helpful for the software maintenance and evolution tasks as well for the re-engineering process of migrating software systems into SPLs [2, 26].

It is true that there already exists a large number of feature location techniques available using in the vast majority static, textual or dynamic analysis, or combining more than one of these types of analysis, called as hybrid aiming to supply the disadvantages each one has [6, 21]. However, there are so many feature location techniques with different evaluations, using different metrics and ground-truth that it is very difficult for practitioners to decide which one is more appropriate to use [26]. Yet, some of the work proposing feature location techniques cannot be reproduced because of not available material, which makes it difficult to compare existing feature location techniques for further improvements in their existing limitations [21, 26].

Therefore, more common benchmarking frameworks in feature location field for evaluating feature location techniques have been required [18] . Currently, there is a benchmark proposed by Martinez et al. [18] with the ArgoUML system, which is implemented with Java programming language. However, differences in source code entities between different languages have a strong impact on feature location [27]. Yet, there are few benchmarks available that can be used for feature/feature revision location applied to C preprocessed systems. A benchmark for C software systems would make easier the proposal of feature revision location techniques because the C programming language is widely used for realizing SPLs with preprocessor directives [19]. We thus now present our first challenge:

> **Challenge 1: Feature location at one point in time.** We aim with this challenge to motivate tool developers to evaluate existing or new feature location techniques for systems developed under C/C++ programming language using a common benchmark to enable studies reproducibility and comparison.

We also want to motivate the development of approaches for automation of the reuse of features for composing new configurations. We thus present our second challenge:

Challenge 2: Composition of new configurations with a set of features. With this challenge we will evaluate if proposed feature location approaches for C/C++ systems can be used to compose new configurations with the traces retrieved to make easier and fast the large production of not yet existing variants of a system.

2.2 Feature Revision Location

A feature revision represents the change of the implementation artifacts associated with a feature at a specific point in time [11, 23]. Previous studies [4, 11, 23, 24] stress the need to managing system variants over time at the level of feature revisions. Even a software system has already its features defined into a platform such as an SPL, maintenance and evolution tasks introduce changes in the software system over time. These changes can affect the implementation of the system's features, which are inconsistent across the existing variants. Then, understanding and propagating the changes in variants become harder any time a feature has to be revised [8]. In the literature and practice, there is no unified mechanism to deal with the system evolution in space and time [4]. Despite we presented a feature revision location technique for software systems evolving in space and time in our previous work [24], there are some limitations that can be improved. For instance, a higher number of feature revisions could be traced with less memory consumption and higher effectiveness in retrieving information. We thus present our third challenge to motivate researchers and tool developers to improve our feature revision location or introducing new ones that fulfill its current limitations.

Challenge 3: Feature revision location at multiple points in time. We expect a feature revision location technique to automate the process of mapping implementation artifacts to feature revisions for every existing different implementation of a feature at multiple points in time.

Aiming to motivate better mechanisms/tools for system evolution in space and time, we present our fourth challenge. The fourth challenge is intended to use the feature revision location technique solution from the third challenge as an extractive approach [2] for re-engineering existing variants' versions with feature revisions to be systematically reused. Then, raising initial solutions for systematic reuse in software systems evolving in space and time at the level of features can benefit developers and engineers not only for propagating bug fixes and refactoring but also for creating new configurations with different behaviors of the same feature.

Challenge 4: Composition of new configurations with a set of feature revisions. We expect solutions that can automate the reuse of existing feature revisions of a system in order to compose different configurations with the different implementations of features at different points in time.

3 BENCHMARK

Our benchmark can be used for evaluating and comparing feature/feature revision location techniques for the C programming language with an established set of metrics[1] and dataset[2] from open source systems available. We thus present our benchmark systems.

3.1 Subject Systems

We choose for the benchmark preprocessed SPLs implemented in combination with version control systems, which keep a history of the changes over time and enable us to generate ground-truth variants with features from multiple points in time. The systems are LibSSH, Irssi, and Marlin. These systems have been used in previous studies [9, 10, 14, 15, 20, 23, 24], and are being maintained in Git repositories. We thus believe they are representative target systems to be used to evaluate a feature/feature revision location technique. The LibSSH[3] system is a multi-platform C library implementing the SSHv2 protocol on the client and server-side. This project was initiated in 2005 and has now around 5000 Git commits at the master branch. The Marlin system is a variant-rich open-source embedded firmware for 3D printers[4] created in 2011 and containing currently around 15000 Git commits. The Irssi system is an internet relay chat client program for Linux[5] with around 6000 Git commits being maintained since 1999.

3.2 Evaluation Scenarios

3.2.1 Variants with Features. The scenarios for evaluating solutions for the *Challenge 1* are from variants containing a set of features from a release, i.e., the state of the system after the last commit of a release in the repository. We designed 13 scenarios (see Table 1) of each system according to a specific number of variants, where scenarios 1-10 have 1-10 input configurations and scenario 11 has 100, scenario 12 has 200 and scenario 13 has 300 input configurations.

For the *Challenge 2*, we make available 50 new configurations that do not exist in any one of the scenarios to evaluate the solutions.

Table 1: Scenarios to feature revision location.

		Number of Features		
Scenario	Number of Variants	LibSSH	Marlin	Irssi
1	1	65	41	26
2	2	91	56	32
3	3	99	61	37
4	4	102	65	40
5	5	104	65	40
6	6	104	67	40
7	7	104	67	41
8	8	104	67	41
9	9	104	67	41
10	10	104	67	41
11	100	104	67	41
12	200	104	67	41
13	300	104	67	41

[3] https://gitlab.com/libssh/libssh-mirror
[4] https://github.com/MarlinFirmware/Marlin
[5] https://github.com/irssi/irssi

Table 2: Scenarios for feature revision location.

		LibSSH			Marlin			Irssi		
S	C	V	F	R	V	F	R	V	F	R
1	1	14	14	0	1	1	0	7	7	0
2	5	22	14	8	19	13	6	11	7	4
3	10	32	14	18	24	13	11	16	7	9
4	15	40	15	25	37	16	20	21	7	14
5	50	111	34	77	81	16	64	65	15	50
6	100	182	34	148	333	130	179	84	16	101
7	200	322	39	283	463	135	303	170	28	209
8	300	458	40	418	579	139	413	341	28	314
9	400	575	40	536	683	142	514	441	28	414

S = Scenario; C = Number of Git commits; V = Number of variants; F = Number of features; R = Number of feature revisions.

3.2.2 Variants with Feature Revisions. The scenarios for evaluating solutions for the *Challenge 3* are from variants containing a set of feature revisions from 400 points in time, i.e., from the first 400 Git commits of the master branch. We designed 9 scenarios (see Table 2) for each system according to a specific number of Git commits, which varies the number of variants for each system. However, for both systems the first scenario consists of locating feature revisions from 1 point in time, the second scenario from 5 points in time, the third scenario 10 points in time up to 400 points in time. Furthermore, we make available an additional scenario for the LibSSH system with a set of 6730 variants totalizing 6596 feature revisions from 103 features, covering its entire evolution from the master branch.

For the *Challenge 4*, we make available one new configuration for each point in time where we could combine different feature revisions for all the scenarios presented in this work.

3.3 Format of the Proposed Solutions

The solutions for all challenges have to show as result the variants composed with the mappings of each feature/feature revision to its artifacts that are part of a configuration, i.e., the artifacts that form the input and new configurations of the ground-truth. Additionally, we expect the result files from the metrics computed (see Section 3.4).

3.4 Metrics

In this section, we present the metrics suggested to evaluate the feature/feature revision location technique regarding its quality (correctness) and performance (scalability).

3.4.1 Correctness. To evaluate the effectiveness of the feature revision location technique, i.e., the quality of its search results, we will use the most effective and frequent metrics used for information retrieval: Precision (P) and Recall (R) [17, 26] (Equations 1 and 2). Furthermore, we will use F-Score (F) as a single value that balances precision and recall equally because it is the harmonic average between precision and recall, as shown in Equation 3, and facilitates comparing the technique effectiveness [7].

We used two levels of granularity due to the granularity of the ground-truth extraction, which can be obtained from text files or Git commits with lines that have been added/removed/changed in C source code or binary and text files. Therefore, the metrics will be computed at two levels of granularity: file- and line-level. The (i) file-level comparison consists of comparing if two complete files (ground-truth and retrieved) match their content; and (ii) line-level consists of comparing every line of source code of two files (ground-truth and retrieved).

The precision (Equation 1) is the relation of the true positives (TP), i.e., the correctly retrieved files, which entire content matches, or lines of source code, and false positives (FP), i.e., the files that their entire content does not match, or lines of source code retrieved by the technique that do not exist in the ground-truth variants.

$$Precision = \frac{TP}{TP + FP} \tag{1}$$

The recall (Equation 2) is the relation of the false negatives (FN), i.e., the files or lines of source code that exist in the ground-truth variant but were not retrieved by the technique.

$$Recall = \frac{TP}{TP + FN} \tag{2}$$

The F-Score (Equation 3) is the weighted average of Precision and Recall.

$$F\text{-}Score = 2 * \frac{Precision * Recall}{Precision + Recall} \tag{3}$$

Instructions on how to use our tool utils for computing automatically these metrics are available in our Git repository[1].

3.4.2 Scalability. To evaluate the scalability of the feature/feature revision location technique, we suggest computing runtime and memory consumption as well report the system description used to run the tests. These metrics can help to compare and improve techniques regarding the time complexity and space complexity, i.e., how much time a technique takes to locate features/feature revisions for each variant and how much memory is necessary to locate features/feature revisions of a specific number of variants.

3.5 Ground-Truth Extractor

The ground-truth extractor and instructions on how to use it are available in our Git repository[1]. We now explain how we mine features/feature revisions and how we generate a ground-truth with a small running example shown in Listing 1, where we use the first Git commit of the Marlin system.

The first need the set of features of a system defined to be able to preprocess variants. Thus, we make available the possibility of set features manually if there is the knowledge of the features of the system or they can be computed automatically based on our approach to identifying features.

Identifying features. From a specific range of Git commits we analyze all macros used in preprocessor directives. The macros used in the *#ifdefs* directives are candidates to be part of the features of the system. We also analyze the macros used in *#define* directives, which we discard from being features of the system. Therefore, the macros considered features of the system are the ones that have

```
1  #ifndef CONFIGURATION_H
2      #define CONFIGURATION_H
3      #define MOTHERBOARD 5
4      #ifdef ADVANCE
5          #define EXTRUDER_ADVANCE_K 0.02
6      #endif
7  #endif
8
9  #if MOTHERBOARD == 1
10     #ifndef __AVR_ATmega644P__
11         #error
12     #endif
13 #endif
```

Listing 1: Code snippet adapted from file Configurations.h from the first Git commit 750f6c3 of the Marlin system.

never been used in *#define* directives in the range of Git commits analyzed.

Looking to our running example (Listing 1), the possible candidates to be the features of the system are the macros CONFIGURATION_H, ADVANCE, MOTHERBOARD and __AVR_ATmega644P__. In the next analysis, we look for *define* directives and we eliminate the macros CONFIGURATION_H and MOTHERBOARD. Thus, the set of features we consider has the features ADVANCE, __AVR_ATmega644P__ and BASE. The feature called BASE is the feature containing the core of the system, the files that are not source code files, and all code of conditional block, i.e., *#ifdefs* with macros that are not part of the set of features, for example, the conditional block from Lines 1-9 in Listing 1. Now we have the set of features to preprocess the variants or to start the process of mining feature revisions.

Mining feature revisions. Mining feature revisions consist of finding which features have been affected in every Git commit, i.e., which features' implementation was added/changed/removed at a specific point in time. For this analysis, we consider every conditional blocks, every *#define* directives, and also the blocks and directives from the top of the file including recursively all the ones in header files, i.e., files used in the *#include* directives. We create then a set of constraints to represent the conditions that must be satisfied to execute a specific line of source code (see previous work [23, 24]). For example, Line 3 in Listing 1 will be executed if the macro CONFIGURATION_H is not defined. In this example, we know then that CONFIGURATION_H is not a feature and the conditional block of the macro CONFIGURATION_H belongs to the BASE.

However, in more complex cases, let us suppose the macro MOTHERBOARD is not defined in Line 3 in Listing 1 and there is another file containing a conditional block with a feature that inside it is defining a value 1 for the macro MOTHERBOARD. Thus, the conditional block from Line 9-13 would belong to that specific feature defining MOTHERBOARD, instead of belonging to the BASE feature. Yet, another example if the macro MOTHERBOARD is defined in two locations, in Line 3 in Listing 1 and in another file as we mentioned, we thus consider the conditional block of the macro MOTHERBOARD as part of the closest feature, which in this example, is BASE. We use the closest feature because the MOTHERBOARD value would have already been replaced by the value in Line 3, which is part of the feature BASE before preprocessing the block of Lines 9-13 in Listing 1. Thus, when preprocessing the source code, the lines of the conditional block of the macro MOTHERBOARD would be executed from the code of the feature BASE.

A feature revision then is a feature that is introduced or changed when comparing one point in time to another. We thus get a range of Git commits and compare the first commit with the second, the second commit with the third, and so on. The comparison consists of analyzing for each line of source code that is added/removed/changed between two Git commits which features are part of it. The feature(s) is then selected to preprocess a variant, which contains the artifacts of at least the feature BASE and possible other feature(s) when the source code added/removed/changed requires that more features are selected to be executed. Thus, from the features used to preprocess a variant, only the closest feature will have an increment in its revision, i.e., in the number that proceeds the name of the feature, which represents kind of a new version of a feature revision.

Taking into account that all the lines from Listing 1 were added, we have three variants representing the changes of this point in time. One is a variant containing the feature revision BASE.1 with the number 1 as it is the first revision of the feature BASE and the preprocessed result is the Lines 2, 3, and 5 from Listing 1. A second variant containing the feature revisions BASE.1 and ADVANCE.1, which result from preprocessing are the Lines 2 and 3 from Listing 1, and a third variant containing the feature revisions BASE.1 and __AVR_ATmega644P__ has then the Lines 2, 3 and 11 from Listing 1. This same process repeats for every change over all artifact files of a system for every Git commits of a selected range. More details of the approach we used to create variants with feature revisions are shown in our previous pieces of work [23, 24].

Regarding implementation aspects of our benchmark extractor, we used ChocoSolver[6] to automate the analysis of building correctly the set of constraints and getting correct solutions, i.e, which features must be selected or not to execute a specific line of source code. We chose this solver because it enables us to get basic arithmetic operations and comparisons of numeric values in the range of integer or double despite basic logic operations and Boolean values, which is not possible with an SAT solver, for example.

4 CONCLUSION

We presented four challenges observed for feature/feature revision location techniques to motivate the proposal of solutions for better mechanisms/tools for systems evolving in space and time. We made available a benchmark for future work comparison and reproducibility containing an established dataset and tool utils for metrics computation. The dataset contains a set of variants and their configurations to be used as input for the techniques and a set of variants and their configurations to be used as new configurations aiming to evaluate if the traces resulted can be used to compose variants with a new set of features/feature revisions and cope with the system evolution in space and time.

The ground-truth of features at one point in time was generated by preprocessing SPLs with our benchmark extractor, as well as the ground-truth of feature revisions from multiple points in time. The ground-truth extractor for generating variants with feature revisions was also used in our previous studies [23, 24], which mines

[6] https://choco-solver.org/

previously the feature revisions of a range of Git commits from the SPLs in Git version control systems. Thus, despite we defined a benchmark with three systems, our ground-truth extractor is also available and can be used to generate more ground-truths and variants for any point in time.

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] Ra'Fat AL-Msie'deen, Abdelhak Seriai, Marianne Huchard, Christelle Urtado, Sylvain Vauttier, and Hamzeh Eyal Salman. 2013. Feature Location in a Collection of Software Product Variants Using Formal Concept Analysis. In *Safe and Secure Software Reuse*, John Favaro and Maurizio Morisio (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 302–307.

[2] Wesley K. G. Assunção, Roberto E. Lopez-Herrejon, Lukas Linsbauer, Silvia R. Vergilio, and Alexander Egyed. 2017. Reengineering legacy applications into software product lines: a systematic mapping. *Empir. Softw. Eng.* 22, 6 (2017), 2972–3016. https://doi.org/10.1007/s10664-017-9499-z

[3] Wesley K.G. Assunção, Silvia R. Vergilio, and Roberto E. Lopez-Herrejon. 2020. Automatic extraction of product line architecture and feature models from UML class diagram variants. *Information and Software Technology* 117 (2020), 106198. https://doi.org/10.1016/j.infsof.2019.106198

[4] Thorsten Berger, Marsha Chechik, Timo Kehrer, and Manuel Wimmer. 2019. Software Evolution in Time and Space: Unifying Version and Variability Management (Dagstuhl Seminar 19191). *Dagstuhl Reports* 9, 5 (2019), 1–30. https://doi.org/10.4230/DagRep.9.5.1

[5] Ben Collins-Sussman, Brian W. Fitzpatrick, and C. Michael Pilato. 2002. *Version Control with Subversion.* http://svnbook.red-bean.com/

[6] Bogdan Dit, Meghan Revelle, Malcom Gethers, and Denys Poshyvanyk. 2013. Feature location in source code: a taxonomy and survey. *Journal of Software: Evolution and Process* 25, 1 (2013), 53–95. https://doi.org/10.1002/smr.567

[7] Marc Eaddy, Alfred V. Aho, Giuliano Antoniol, and Yann-Gaël Guéhéneuc. 2008. CERBERUS: Tracing Requirements to Source Code Using Information Retrieval, Dynamic Analysis, and Program Analysis. In *The 16th IEEE International Conference on Program Comprehension, ICPC 2008, Amsterdam, The Netherlands, June 10-13, 2008*, René L. Krikhaar, Ralf Lämmel, and Chris Verhoef (Eds.). IEEE Computer Society, 53–62. https://doi.org/10.1109/ICPC.2008.39

[8] T. Eisenbarth, R. Koschke, and D. Simon. 2003. Locating features in source code. *IEEE Transactions on Software Engineering* 29, 3 (2003), 210–224. https://doi.org/10.1109/TSE.2003.1183929

[9] Angelo Gargantini, Justyna Petke, Marco Radavelli, and Paolo Vavassori. 2016. Validation of Constraints Among Configuration Parameters Using Search-Based Combinatorial Interaction Testing. In *Search Based Software Engineering*, Federica Sarro and Kalyanmoy Deb (Eds.). Springer International Publishing, New York, NY, USA, 49–63.

[10] Huong Ha and Hongyu Zhang. 2019. Performance-Influence Model for Highly Configurable Software with Fourier Learning and Lasso Regression. In *35th International Conference on Software Maintenance and Evolution* (Cleveland, OH, USA) *(ICSME 2019)*. IEEE, New York, USA, 470–480. https://doi.org/10.1109/ICSME.2019.00080

[11] Daniel Hinterreiter, Michael Nieke, Lukas Linsbauer, Christoph Seidl, Herbert Prähofer, and Paul Grünbacher. 2019. Harmonized Temporal Feature Modeling to Uniformly Perform, Track, Analyze, and Replay Software Product Line Evolution. In *18th International Conference on Generative Programming: Concepts & Experiences* (Athens, Greece) *(GPCE 2019)*. ACM, New York, USA, 115–128. https://doi.org/10.1145/3357765.3359515

[12] Charles W. Krueger. 1992. Software Reuse. *ACM Comput. Surv.* 24, 2 (June 1992), 131–183. https://doi.org/10.1145/130844.130856

[13] Jacob Krüger, Wanzi Gu, Hui Shen, Mukelabai Mukelabai, Regina Hebig, and Thorsten Berger. 2018. Towards a Better Understanding of Software Features and Their Characteristics: A Case Study of Marlin. In *Proceedings of the 12th International Workshop on Variability Modelling of Software-Intensive Systems* (Madrid, Spain) *(VAMOS 2018)*. Association for Computing Machinery, New York, NY, USA, 105–112. https://doi.org/10.1145/3168365.3168371

[14] Jacob Krüger, Mukelabai Mukelabai, Wanzi Gu, Hui Shen, Regina Hebig, and Thorsten Berger. 2019. Where is my feature and what is it about? A case study on recovering feature facets. *Journal of Systems and Software* 152 (2019), 239–253. https://doi.org/10.1016/j.jss.2019.01.057

[15] Jörg Liebig, Sven Apel, Christian Lengauer, Christian Kästner, and Michael Schulze. 2010. An Analysis of the Variability in Forty Preprocessor-based Software Product Lines. In *32nd ACM/IEEE International Conference on Software Engineering - Volume 1* (Cape Town, South Africa) *(ICSE 2010)*. ACM, New York, USA, 105–114. https://doi.org/10.1145/1806799.1806819

[16] Lukas Linsbauer, Felix Schwägerl, Thorsten Berger, and Paul Grünbacher. 2021. Concepts of variation control systems. *J. Syst. Softw.* 171 (2021), 110796. https://doi.org/10.1016/j.jss.2020.110796

[17] Christopher D Manning, Hinrich Schütze, and Prabhakar Raghavan. 2008. *Introduction to information retrieval.* Cambridge university press, Cambridge, England.

[18] Jabier Martinez, Nicolas Ordoñez, Xhevahire Tërnava, Tewfik Ziadi, Jairo Aponte, Eduardo Figueiredo, and Marco Tulio Valente. 2018. Feature location benchmark with argoUML SPL. In *Proceedings of the 22nd International Systems and Software Product Line Conference - Volume 1, SPLC 2018, Gothenburg, Sweden, September 10-14, 2018*, Thorsten Berger, Paulo Borba, Goetz Botterweck, Tomi Männistö, David Benavides, Sarah Nadi, Timo Kehrer, Rick Rabiser, Christoph Elsner, and Mukelabai Mukelabai (Eds.). ACM, 257–263. https://doi.org/10.1145/3233027.3236402

[19] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Sarah Nadi, and Rohit Gheyi. 2015. The Love/Hate Relationship with the C Preprocessor: An Interview Study (Artifact). *Dagstuhl Artifacts Ser.* 1, 1 (2015), 07:1–07:32. https://doi.org/10.4230/DARTS.1.1.7

[20] 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.

[21] Gabriela Karoline Michelon, Lukas Linsbauer, Wesley K. G. Assunção, Stefan Fischer, and Alexander Egyed. 2021. A Hybrid Feature Location Technique for Re-engineeringSingle Systems into Software Product Lines. In *VaMoS'21: 15th International Working Conference on Variability Modelling of Software-Intensive Systems, Virtual Event / Krems, Austria, February 9-11, 2021*, Paul Grünbacher, Christoph Seidl, Deepak Dhungana, and Helena Lovasz-Bukvova (Eds.). ACM, 11:1–11:9. https://doi.org/10.1145/3442391.3442403

[22] Gabriela Karoline Michelon, Lukas Linsbauer, Wesley K. G. Assunção, and Alexander Egyed. 2019. Comparison-Based Feature Location in ArgoUML Variants. In *23rd International Systems and Software Product Line Conference - Volume A* (Paris, France) *(SPLC '19)*. Association for Computing Machinery, New York, NY, USA, 93–97. https://doi.org/10.1145/3336294.3342360

[23] Gabriela Karoline Michelon, David Obermann, Wesley Klewerton Guez Assunção, Lukas Linsbauer, Paul Grünbacher, and Alexander Egyed. 2020. Mining Feature Revisions in Highly-Configurable Software Systems. In *Proceedings of the 24th ACM International Systems and Software Product Line Conference - Volume B* (Montreal, QC, Canada) *(SPLC '20)*. Association for Computing Machinery, New York, NY, USA, 74–78. https://doi.org/10.1145/3382026.3425776

[24] Gabriela Karoline Michelon, David Obermann, Lukas Linsbauer, Wesley Klewerton Guez Assunção, Paul Grünbacher, and Alexander Egyed. 2020. Locating feature revisions in software systems evolving in space and time. In *SPLC '20: 24th ACM International Systems and Software Product Line Conference, Montreal, Quebec, Canada, October 19-23, 2020, Volume A*, Roberto Erick Lopez-Herrejon (Ed.). ACM, 14:1–14:11. https://doi.org/10.1145/3382025.3414954

[25] Denys Poshyvanyk, Yann-Gael Gueheneuc, Andrian Marcus, Giuliano Antoniol, and Vaclav Rajlich. 2007. Feature Location Using Probabilistic Ranking of Methods Based on Execution Scenarios and Information Retrieval. *IEEE Transactions on Software Engineering* 33, 6 (June 2007), 420–432. https://doi.org/10.1109/TSE.2007.1016

[26] Abdul Razzaq, Asanka Wasala, Chris Exton, and Jim Buckley. 2018. The State of Empirical Evaluation in Static Feature Location. *ACM Trans. Softw. Eng. Methodol.* 28, 1, Article 2 (Dec. 2018), 58 pages. https://doi.org/10.1145/3280988

[27] Ripon K. Saha, Julia Lawall, Sarfraz Khurshid, and Dewayne E. Perry. 2014. On the Effectiveness of Information Retrieval Based Bug Localization for C Programs. In *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014*. IEEE Computer Society, 161–170. https://doi.org/10.1109/ICSME.2014.38

[28] Christoph Seidl, Ina Schaefer, and Uwe Aßmann. 2014. Capturing Variability in Space and Time with Hyper Feature Models. In *Proceedings of the Eighth International Workshop on Variability Modelling of Software-Intensive Systems* (Sophia Antipolis, France) *(VaMoS '14)*. Association for Computing Machinery, New York, NY, USA, Article 6, 8 pages. https://doi.org/10.1145/2556624.2556625

[29] Thomas Thüm, Leopoldo Teixeira, Klaus Schmid, Eric Walkingshaw, Mukelabai Mukelabai, Mahsa Varshosaz, Goetz Botterweck, Ina Schaefer, and Timo Kehrer. 2019. Towards Efficient Analysis of Variation in Time and Space. In *Proceedings of the 23rd International Systems and Software Product Line Conference - Volume B* (Paris, France) *(SPLC '19)*. Association for Computing Machinery, New York, NY, USA, 57–64. https://doi.org/10.1145/3307630.3342414