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, Brazil & PPGComp - Western Paraná State University, Brazil
[4]Institute of Software Engineering and Automotive Informatics, Technische Universität Braunschweig, Germany

ABSTRACT

Software companies need to provide a large set of features satisfying functional and non-functional requirements of diverse customers, thereby leading to *variability in space*. Feature location techniques have been proposed to support software maintenance and evolution in space. However, so far only one feature location technique also analyses the *evolution in time* of system variants, which is required for feature enhancements and bug fixing. Specifically, existing tools for managing a set of systems over time do not offer proper support for keeping track of feature revisions, updating existing variants, and creating new product configurations based on feature revisions. This paper presents four challenges concerning such capabilities for feature (revision) location and composition of new product configurations based on feature/s (revisions). We also provide a benchmark containing a ground truth and support for computing metrics. We hope that this will motivate researchers to provide and evaluate tool-supported approaches aiming at managing systems evolving in space and time. Further, we do not limit the evaluation of techniques to only this benchmark: we introduce and provide instructions on how to use a benchmark extractor for generating ground truth data for other systems. We expect that the feature (revision) location techniques maximize information retrieval in terms of precision, recall, and F-score, while keeping execution time and memory consumption low.

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

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

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

1 INTRODUCTION

Software companies have to tailor and maintain variants of software systems co-existing simultaneously to serve different customers and new requirements. Variants of a system are composed of variable assets related to different features that realize the variability of a system [3]. The system variants reflect different configurations and have been described as *variability in space* [28, 29]. *Variability in time*, on the other hand, results from the need of modifying variants due to enhancements, for example, to address new customer requirements or changes to the environment, such as alternative hardware or the optimization of non-functional properties [29]. Thus, over the system life cycle, the introduction of new features in existing variants, a.k.a *evolution in space* can be required. Further, features can be subject to failures or unwanted behaviors and bug fixes have to be done, introducing new revisions 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., a set of features [4, 16].

The aforementioned scenarios lead to many system variants that need to be managed and evolved in parallel. This highly increases the workload of developers. Furthermore, keeping system variability consistent across different types of artifacts manually is an error-prone task [16]. Software product line (SPL) approaches have been adopted by engineers for systematic variability management and reuse of the core assets and features' artifacts, thereby accelerating the production of variants and reducing the effort and costs for maintaining and creating products [12]. Regarding the transition of existing systems to an SPL, feature location is the first and one of the most essential tasks of the re-engineering process to migrate a family of existing system variants into an SPL [2].

Despite feature location and SPLs cover the *space* dimension, they do not address the *time* dimension [4]. SPLs by themselves do not provide proper management of evolution in time and engineers have to adopt additional mechanisms and tools. SPLs are frequently managed in version control systems (VCSs), which track changes of a system over time [5]. However, current VCSs have 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 to

have a portfolio to reflect potential versions of a feature, i.e., the feature revisions that co-exist, which can be reused for creating different variants [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. Although there are some feature location techniques considering the *space* dimension, they 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 sub-optimal results, and 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 (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 as at multiple points in time (Section 2). Yet, the proposed feature (revision) location techniques support engineers in creating new configurations based on the traced features and their revisions. By proposing these challenges, we aim to motivate researchers and tool developers to optimize and address the limitations of existing techniques and to develop more efficient mechanisms for managing systems evolving in space and time.

Evaluating solutions for the challenges with a common benchmark can enable future work comparisons [21]. For this purpose, we contribute with both a benchmark and 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 utilities[1] to evaluate the efficiency of feature (revision) location techniques that compute 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 on the benchmark. We discuss the scenarios and metrics for evaluating solutions and briefly explain the ground truth extractor. Section 4 concludes the paper.

2 THE CHALLENGES

To describe our challenges, we now present the background of feature (revision) location techniques. Then we explain the importance of these techniques, their current limitation, and observed complexity, or lack of studies, which makes the challenges interesting.

2.1 Feature Location

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

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

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. During maintenance and evolution, developers need the complete locations of features, which can be outside the annotations. This requires manual work that could be automated by feature location techniques. Furthermore, feature location techniques are helpful not only for the software maintenance and evolution tasks as well as for the re-engineering process of cloned software systems into SPLs [2, 26].

There already exists a large number of feature location techniques, which use, e.g, textual, static, or dynamic analyses, or combinations thereof [6, 21]. Despite many feature location techniques available, results can be compromised by the number of existing software systems when using a comparison-based static analysis for re-engineering existing software systems into SPLs [22], for example. Yet, the quality results of textual analysis are highly-dependent on index terms and queries, while the results from the dynamic analysis are very sensitive to how scenarios and features are executed [21]. Nonetheless, feature location techniques have different evaluations, metrics, and ground truth data sets, making it difficult for practitioners to decide which one is most appropriate for them [26]. Further, 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 with improvements addressing their limitations [21, 26].

Therefore, more common benchmarking frameworks for evaluating feature location techniques have been suggested [18]. Currently, there is a benchmark proposed by Martinez et al. [18] based on the ArgoUML system, which is implemented in Java. 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 to apply feature (revision) location to C-preprocessor-based systems. A benchmark for C software systems would ease the proposal and evaluation of feature (revision) location techniques because C 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 to motivate researchers and tool developers to evaluate existing or new feature location techniques based on systems developed in C or C++, using a common benchmark enabling the 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 product configurations with a set of features.** We evaluate if the proposed feature location approaches for C/C++ systems can be used to compose new configurations with the traces retrieved to simplify and accelerate the composition 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 manage system variants over time at the level of feature revisions. Even if a software system already manages its features as an SPL, maintenance and evolution will introduce changes, affecting the implementation of the system's features, which may become inconsistent across the existing variants. This makes changes increasingly hard to understand and propagate to variants at any time a feature has to be revised [8]. In the literature and practice, there is no unified mechanism to deal with the evolution of systems in space and time [4]. While 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 need yet to be addressed. For instance, a higher number of feature revisions could be traced with less memory consumption and higher effectiveness in retrieving information. We thus present the third challenge to motivate researchers and tool developers to improve our feature revision location technique or propose new ones overcoming current limitations.

Challenge 3: Feature revision location at multiple points in time. We expect solutions with feature revision location techniques 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 and unified mechanisms and tools for system evolution in space and time, we present our fourth challenge. It 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 by systematically reusing feature revisions. By raising initial solutions for systematic reuse in software systems evolving in space and time to the level of features, developers and engineers can benefit not only when propagating bug fixes and refactoring but also when creating new configurations with different behaviors of the same feature.

Challenge 4: Composition of new product 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 (revision) location techniques for the C programming language with an established set of metrics[1] and dataset[2] from available open-source systems.

3.1 Subject Systems

Our benchmark is composed of 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 managed in Git repositories. We thus believe they are representative target systems to be used to evaluate feature (revision) location techniques. 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 now has around 5000 commits in the master branch. The Marlin[4] system is a variant-rich open-source embedded firmware for 3D printers created in 2011 and with currently around 15000 commits. The Irssi[5] system is an internet relay chat client for Linux with around 6000 commits since 1999.

3.2 Evaluation Scenarios

3.2.1 Variants with Features. The scenarios for evaluating solutions for *Challenge 1* are from variants containing a set of features from one release, i.e., the state of the system after the last commit of a release in the repository. We designed 13 scenarios (Table 1) of each system with 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 *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

3.2.2 Variants with Feature Revisions. The scenarios for evaluating solutions for *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 nine scenarios (Table 2) for each system according to a specific number of Git commits, by varying the number of variants for each system. For both all systems, we present scenarios that consist of locating feature revisions from 1 point in time to up 400 points in time. Furthermore, we make available an additional scenario for the LibSSH system with a set of 6730 variants, resulting in 6596 feature revisions from 103 features, thereby covering the entire evolution of the master branch.

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

[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.

3.3 Format of the Proposed Solutions

The ground truth is composed of a set of variants and no traces, which allows for multiple valid traces. Then, the solutions for all challenges have to show as result the variants composed with the mappings of each feature (revision) to its artifacts that are part of a configuration, i.e., the artifacts that form the input and new product 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 (revision) location technique regarding its quality (correctness) and performance (scalability). The correctness must be computed based on the comparison between the ground truth variants with corresponding retrieved ones obtained after performing the feature (revision) location.

3.4.1 Correctness. To evaluate the effectiveness of the feature revision location technique, i.e., the quality of its search results, we adopt efficient and frequently used information retrieval metrics precision (P) and recall (R) [17, 26] (cf. Equations 1 and 2). Furthermore, we also adopt the F-score (F), i.e., the harmonic average between precision and recall, as a single value equally balancing precision and recall (cf. Equation 3) and assessing the techniques' effectiveness [7].

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

Precision (Equation 1) is the relation of the true positives (TP), i.e., the correctly retrieved files, which entire content matches, and false positives (FP), i.e., the files that their entire content does not match. At the line-level, TP are the lines of source code that match related to the lines of source code retrieved by the technique that does not exist in the ground truth variants.

$$Precision = \frac{TP}{TP + FP} \quad (1)$$

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.

$$Recall = \frac{TP}{TP + FN} \quad (2)$$

The F-score (Equation 3) is the harmonic average of precision and recall.

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

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 (revision) location technique, we expect proponents to compute runtime and memory consumption, reporting the specification of the infrastructure used to run the proposed solutions. 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 (revisions) for each variant and how much memory is necessary to locate features and their revisions for 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 our extractor mines features and feature revisions to generate a ground truth. Our explanation relies on a small running example shown in Listing 1, where we use the first Git commit of the Marlin system.

We first need the set of features of a system defined to be able to preprocess variants. Then, we make available the possibility of setting up manually the set of existing features of a system in our extractor or computing features 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 are the ones that have never been used in *#define* directives in the range of Git commits analyzed.

In the case of our running example (Listing 1), the possible feature candidates are the macros `CONFIGURATION_H`, `ADVANCE`, `MOTHERBOARD`, and `__AVR_ATmega644P__`. In the next analysis, we look for *define* directives and eliminate the macros `CONFIGURATION_H` and `MOTHERBOARD`. Thus, the set of features we consider is composed of features `ADVANCE`, `__AVR_ATmega644P__` and `BASE`. The feature called `BASE` is the feature containing the core of the system. The `BASE` can be represented by the files that are not source code files, and all code of conditional blocks, 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

```
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.

set of features to preprocess the variants or to start the process of mining feature revisions.

Mining feature revisions. Mining feature revisions consists of finding features, which were affected by changes to their implementation in some Git commits, with lines added, changed, or removed at specific points in time. For this analysis, we consider all conditional blocks, all *#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 [23, 24]). For example, Line 3 in Listing 1 will be executed if the macro `CONFIGURATION_H` is not defined. In this example, we then know 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 on Line 3 in Listing 1 and there is another file containing a conditional block with a feature that 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 is the macro `MOTHERBOARD` defined in two locations: on 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 on Line 3, which is part of the feature `BASE` before preprocessing the block of Lines 9-13 in Listing 1. Finally, 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 added, removed, or changed between two Git commits, which features are part of it. The feature(s) is/are then selected to preprocess a variant, which contains the artifacts of at least the feature `BASE` and possible other feature(s). 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`. The preprocessed result comprises Lines 2, 3, and 5 from Listing 1. A second variant contains the feature revisions `BASE.1` and `ADVANCE.1`. The result from preprocessing comprises Lines 2 and 3 from Listing 1. The third variant containing the feature revisions `BASE.1` and `__AVR_ATmega644P__` has then 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 work [23, 24].

We used the ChocoSolver[6] to implement our benchmark extractor and to automate the analysis of building correctly the set of constraints and getting correct solutions, i.e, the features to be selected or excluded 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 would not be possible with an SAT solver, for example.

4 CONCLUSION

We presented four challenges relevant for feature (revision) location techniques to motivate the proposal of solutions for better mechanisms and tools to support the evolution of systems in space and time. We made available a benchmark for comparing future work for supporting reproducibility. It contains a dataset and tool utilities for computing metrics. The dataset comprises 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. This allows to evaluate if the resulting traces can be used to compose variants with a new set of features and feature revisions.

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 previously the feature revisions of a range of Git commits from the SPLs in Git version control systems. Thus, although the benchmark comprises three systems, our ground truth extractor can be used to generate ground truth data sets 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; FAPPR, grant no. 51435; and FAPERJ PDR-10 program, grant no. 202073/2020. The support by the Austrian Federal Ministry for Digital and Economic Affairs and the National Foundation for Research, Technology and Development is gratefully acknowledged.

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

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.* O'Reilly Media, Stanford, California, USA. 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, New York, USA, 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 *Proceeedings 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, New York, USA, 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. https://doi.org/10.1109/TSE.2017.2688333

[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, New York, USA, 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, New York, USA, 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, New York, USA, 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