



Variability-Aware Differencing with DiffDetective

Paul Maximilian Bittner

paul.bittner@uni-ulm.de

University of Ulm

Germany

Alexander Schultheiß

AlexanderSchultheiss@pm.me

Paderborn University

Germany

Benjamin Moosherr

benjamin.moosherr@uni-ulm.de

University of Ulm

Germany

Timo Kehrer

timo.kehrer@inf.unibe.ch

University of Bern

Switzerland

Thomas Thüm

thomas.thuem@uni-ulm.de

University of Ulm

Germany

ABSTRACT

Diff tools are essential in developers' daily workflows and software engineering research. Motivated by limitations of traditional line-based differencing, countless specialized diff tools have been proposed, aware of the underlying artifacts' type, such as a program's syntax or semantics. However, no diff tool is aware of systematic variability embodied in highly configurable systems such as the Linux kernel. Our software library called DiffDetective can turn any generic diff tool into a variability-aware differencer such that a changes' impact on the actual source code and its superimposed variability can be distinguished and analyzed. Next to graphical diff inspectors, DiffDetective provides an analysis framework for large-scale empirical analyses of version histories on a substantial body of variability-intensive software including the Linux kernel. DiffDetective has been successfully employed to, for example, explain edits, generate benchmark data, or evaluate differencing algorithms and patch mutations.

CCS CONCEPTS

• **Software and its engineering** → **Software configuration management and version control systems**; *Software evolution*.

ACM Reference Format:

Paul Maximilian Bittner, Alexander Schultheiß, Benjamin Moosherr, Timo Kehrer, and Thomas Thüm. 2018. Variability-Aware Differencing with DiffDetective. In *Proceedings of the 32nd ACM Symposium on the Foundations of Software Engineering (FSE '24)*, November 15–19, 2024, Porto de Galinhas, Brazil. ACM, New York, NY, USA, 5 pages. <https://doi.org/XXXXXXX.XXXXXXX>

1 INTRODUCTION

Diff tools are among the most widely used tools in software development, providing a description of how two development artifacts differ from each other. Developers inspect such differences in terms of their daily development workflows, notably to review, understand, and verify changes tracked in version control systems. Moreover, differences are an indispensable basis for many (semi-)automated

tasks, such as reasoning on implications of software changes [39]. Beyond those practical applications, diffing is central in software engineering research, particularly for mining software repositories to investigate software evolution, understand the nature of code changes, and contribute to empirical studies on software maintenance and development [24].

To that end, countless diff tools have been proposed and developed in the past [7, 8, 12, 16–18, 21, 45]. *General* tools, such as commonly used line-based differencing [8, 12, 45], well-known from UNIX and `git diff` utilities, uniformly treat all development artifacts as lines of text, regardless of their actual type (e.g., Java code or XML documents). While being applicable to a wide range of artifacts, general differencing suffers from sensitivity to formatting changes and cannot explain differences on a syntactic or even semantic level of the underlying artifact type. To compensate, researchers have proposed *specialized* diff tools that are aware of the underlying artifacts' syntax or semantics [7, 14, 16–18, 21, 23, 28, 54]. Such structured differencing remains generic in the sense that it can be adapted to a large class of artifact types by providing respective parsers [17, 18, 21] or comparison heuristics [34].

However, there is no diff tool yet that has specialized on software variability as common in software product lines [6, 13]. For instance, the Linux kernel offers over 10,000 configuration options [22, 52], giving rise to uncountably many kernels that can be derived from a common code base [37, 52]. Here, source code is annotated with variability-annotations that denote to exclude or include annotated source code, depending on whether an option is selected or not. Static variability like this is not only present in the Linux kernel but widely used in practice [29, 40]. Given the fact that preprocessor annotations alone are already known to cause bugs [5] and impair code understanding [38, 42], these problems are further aggravated when annotations are subject to continuous change. The lack of diff tools aware of variability hence hampers the evolution of variability-intensive systems and maintaining their quality.

In this paper, we present our tool DiffDetective that can turn any generic differencing technique into a variability-aware diff tool. DiffDetective is modeled around formally verified core data structures for variability and variability-aware diffs [10]. While being open for extensions, DiffDetective has been integrated with two concrete differencing tools, accessible through a reusable software library and graphical diff inspectors. Thereupon, DiffDetective offers a framework for large-scale empirical analyses of Git version histories of variational software, along with a curated dataset of

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

FSE'24, July 15–19, 2024, Porto de Galinhas, Brazil

© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-XXXX-X/18/06

<https://doi.org/XXXXXXX.XXXXXXX>

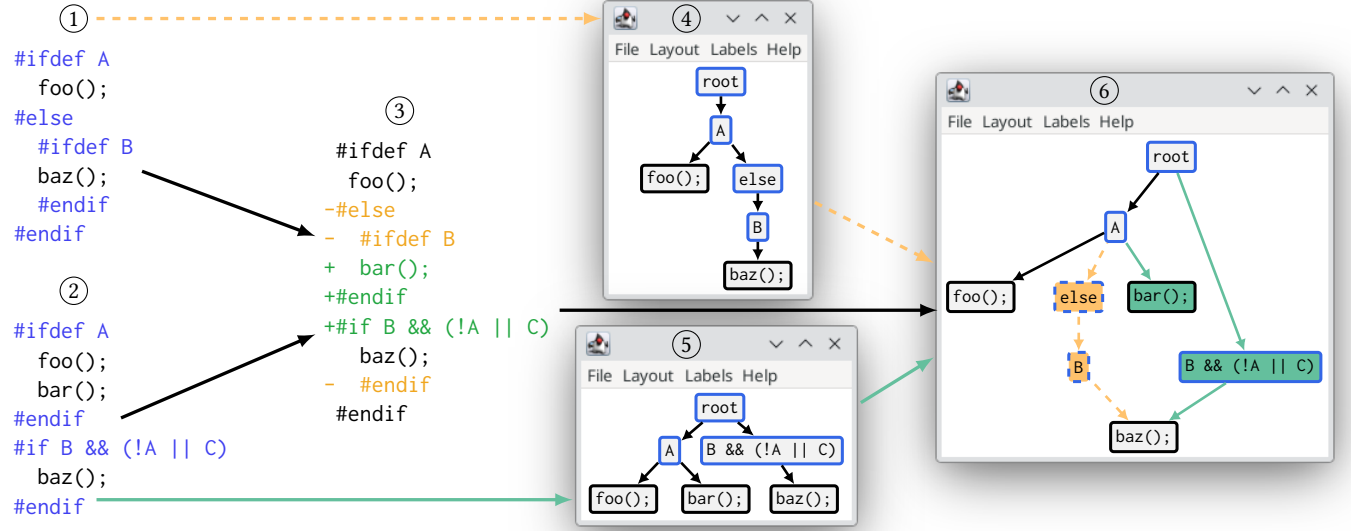


Figure 1: Overview on variability-aware differencing based on a simplified example of commit [afde13b](#) in Vim.

44 real-world software product lines, including the Linux Kernel, GCC, Vim, Emacs, and the Godot game engine.

DiffDetective targets both practitioners and researchers, covering use cases such as explaining edits [10, 27], generating clone-and-own benchmarks [47], commit untangling or view generation [9], or benchmarking diff algorithms [43]. While DiffDetective was developed in parallel to prior research efforts [9–11, 27, 43, 53], it was never their focus and instead mostly a means to conduct those studies. This paper is the first comprehensive and independent overview of DiffDetective to pave the way for a more widespread reuse and adoption.

DiffDetective as well as the replication packages of past studies are open-source and available online [3]. This paper also comes with a demo repository [1] and a corresponding screencast [4].

2 TOOL OVERVIEW

In this section, we give an overview of DiffDetective. We start with a motivating example, and then explain its core components, parsing capabilities, and framework for large-scale empirical analysis.

2.1 Variability-Aware Differencing

The left of Figure 1 shows two versions of a simplified C/C++ code snippet, where the snippet ① was changed to snippet ②. While the snippets are simplified for presentation here, they equal real snippets from the code base of the Vim text editor in structure and complexity, edited in commit [afde13b](#). Both versions exhibit variability implemented by C preprocessor annotations (blue lines starting with #). These annotations conditionally in- or exclude enclosed lines of code depending on the selections of the configuration options A, B, and C. For example, configuring version ① will only contain the call to `foo()` when A is selected. Such annotations form a meta-language that must be evaluated by a preprocessor *before* the underlying C/C++ source code is compiled. Once all selections are made, the preprocessor resolves and removes all annotations

from the code base, handing the remaining, non-variational C/C++ program to the compiler.

The center of Figure 1 shows a text-based diff ③ of both code snippets. Such a generic diff inevitably mixes source code, variability annotations, and diff syntax, and makes no distinction between annotations and annotated code. Crucially, further variability annotations enclosing the changed lines might be excluded from the diff because they are too far away. This makes it hard to tell how the scope of annotations changed, and how variability is impacted. While our example shows a simplified diff from commit [afde13b](#) in Vim, more complex cases arise in practice [10, 29, 36, 51].

To make a generic diff variability-aware, DiffDetective is designed around two data structures, also depicted in Figure 1. First, *variation trees* ④ ⑤ represent files or chunks of variational source code. Second, *variation diffs* ⑥ are diffs of variation trees and hence show the differences between two versions of variational code. Together, they enable lifting a generic diff ③ into a variability-aware diff ⑥. Variation trees and diffs abstract from the syntactical details of the concrete technology used for variability annotations, such as the # symbols or #endif directives that close a scope. Instead, variation trees and diffs reflect the abstract syntax of annotations by only distinguishing between annotations (blue outlines) and affected source code (black outlines). Similarly, variation trees and diffs are not tied to any particular programming language either. While this example is centered around lines of C/C++ code, variation trees and diffs may reference any language (e.g., other programming languages, documentation, or build files) at any granularity (e.g., lines of code, tokens, or abstract syntax tree nodes) [10].

With DiffDetective, a generic differencing algorithm can be made variability-aware in two ways, loosely drawn as a commuting diagram in Figure 1: either diff both versions generically and then distinguish variability and artifacts (black path ① ② → ③ → ⑥) or vice versa (orange path ① → ④ → ⑥ and green path ② → ⑤ → ⑥ combined). Both ways are semantically equivalent

(i.e., either diff will yield the same result when applied as a patch) but might be syntactically different (e.g., different lines or nodes might be flagged for insertion or deletion), as different algorithms may have different heuristics, for example, to detect moved lines. As of now, DiffDetective interfaces with `git diff` (via the JGit library) and its supported algorithms for the first path, and the tree-differencer GumTree [18] for the second path.

Next to diffing facilities, the DiffDetective API provides a lightweight graphical user interface as illustrated by the frames in Figure 1. Opening the user interface will show a single variation tree or diff, and offer drag'n'drop interaction with graph nodes, screenshots, and a Tikz export. The graphical user interface can be opened via DiffDetective's API by invoking dedicated Java methods at any point during a program's execution, which has DiffDetective integrated as a library. Then, DiffDetective will open the user interface and, if desired, block program execution until the user interface is closed. This enables developers to debug their implementations of analyses or transformations for variation trees and diffs.

2.2 Implementing Variability Awareness

To make diffs aware of variability, DiffDetective must support concrete implementations of variability mechanisms. This allows DiffDetective, to distinguish variability annotations from the annotated implementation artifacts.

DiffDetective offers a heuristic parser for C preprocessor annotations, one of the most widely used variability mechanisms [29, 40]. Our parser recognizes conditional statements that induce variability and separates them from non-variational preprocessor statements, such as `#include` directives. Non-variational annotations are instead treated as plain lines of code because they are also subject to the variability induced by conditional annotations (e.g., an `#include` statement can be conditionally in- or excluded just like any other code).

While variation trees and diffs may represent any tree-like data in any granularity, the parser for C preprocessor annotations within DiffDetective treats all source code as plain lines of text for multiple reasons. It is accurate regarding the C preprocessor, and it is fast, general, less prone to errors, and robust to language changes. To be accurate with respect to the employed variability mechanism, DiffDetective must treat implementation artifacts in the same way as the mechanism does. The C preprocessor neglects all code structure by treating code as lines of text, and so does the respective parser in DiffDetective. In fact, an exact parsing of C preprocessor statements and the underlying code to their abstract syntax is known as variability-aware parsing and was implemented in TypeChef and SuperC [25, 32]. However, variability-aware parsing may take more than ten hours for only a single commit, rendering the analysis of entire commit histories infeasible. The reason for excessive run times comes from the need to parse all variants of the code simultaneously, while typically a compiler only sees exactly one such variant (cf. Section 2.1). Hence, we found our parsing to be a good trade-off of accuracy and speed (cf. Section 3).

Due to the verified generality of variation trees and diffs, there are no barriers to extend DiffDetective with further parsers for other technologies in the future.









2.3 Empirical Analysis Pipeline

DiffDetective offers a framework to iterate entire version histories of variational software, for example, to diff and analyze commits and patches, or to evaluate research methods and tools that operate on such changes. In a nutshell, the framework asynchronously iterates over all commits within a given set of Git repositories, giving users access to each commit, each patch within a commit (i.e., changes to a single file), and the variation diff representing that patch. To implement their own study, users can compose existing and new implementations of a dedicated Java interface. This interface contains callbacks that are invoked by the framework, when a commit, patch, or variation diff is processed, respectively.


DiffDetective can process any Git repository in which software is annotated with supported variability mechanisms. Given the wide adoption of the C preprocessor to implement static variability, DiffDetective has a significant body of large-scale, real-world datasets available. In past studies, we analyzed 44 open-source and real-world datasets, including famous projects such as the Linux kernel, the Vim and Emacs text editors, GCC, and Godot [9, 10]. Together, the 44 projects cover a wide spectrum of about 30 domains, including but not limited to web servers, operating systems, database systems, antivirus, media players, editors, and game engines. For easy access to all datasets and to foster exact replications of experiments, we provide forks of all 44 repositories, frozen at their state around February 2022 [2]. The dataset of these repositories can be easily customized and loaded into DiffDetective by using supplied dataset markdown files. When starting the evaluation framework, DiffDetective will then automatically clone each specified repository of which there is no local copy yet.

3 VALIDATION AND USE CASES

At a larger scale, DiffDetective was employed for the empirical analysis within multiple published research papers [9, 10, 27] as well as student theses [11, 26, 43, 53]. In each of these works, DiffDetective was employed to perform a particular analysis or data acquisition. Thereby, DiffDetective was iteratively refined, extended, and validated. Here, we report on our results and experiences along these empirical studies.

Explaining Edits [10, ][27, ][53]. Our initial motivation for developing DiffDetective was explaining an edit's effect on the variability of a code base [10]. To this end, we introduced variation trees and diffs and an algorithm for classifying nodes in variation diffs depending on how their variability has changed. Based on formal models for variation trees and diffs, we discuss the completeness and soundness of variation trees (i.e., variation trees are general enough to model any kind of static variability, and all variation trees describe a valid state of variability) and prove completeness and soundness of variation diffs (i.e., variation diffs can describe any edit to variation trees, and all variation diffs describe legal edits) [10]. The proofs are made available within appendices in DiffDetective's repository, including a small Haskell project with formal specifications of variation trees and diffs. To conduct an empirical evaluation, we extended DiffDetective by a SAT solving interface, the edit classification, as well as a scheme for implementing any custom classification. DiffDetective processed all 44 repositories (cf. Section 2.3) with about 1.7 million commits and

4.9 million variation diffs in about 70 min on a compute server [10]. Hence, creating the variation diffs of a commit and classifying the nodes took around 21.5 ms on average.

Commit Untangling and Operation Lifting [9, ][11]. Sometimes, commits address multiple concerns at once, for example when fixing a bug and simultaneously refactoring or adding a feature. Motivated by untangling such composite commits, we developed a method and formal foundations for lifting any conventional operation on configurable software to an operation on variations diffs, for example to reduce a diff to contain only changes to a certain configuration option. We designed and implemented two algorithms for this lifting and proved their correctness (both algorithms do not produce illegal diffs) and semantic equivalence (both algorithms produce diffs that describe the same change but might do so differently, cf. Section 2.1). We extended DiffDetective with a graphical user interface for variation trees and diffs, and implementations for both lifting algorithms. On a compute server, our empirical evaluation on all 44 datasets took about three hours for around 10 million lift operations, which corresponds to about 900 operations per second [9].

Benchmark Generation. Next to using variability annotations, new system variants can also be implemented by copying and adapting existing software, also known as clone-and-own. To evaluate research and tools for clone-and-own development, respective datasets are required with certain governance metadata (e.g., when cloning happened or where source code of clones differ) [47], which are not readily available [50]. Hence, we developed VEVOS, a generator for clone-and-own benchmarks from variational software [47, 48]. Recently, we migrated VEVOS to use DiffDetective as its core analysis library, which considerably improved scalability, allows the analysis of diffs as opposed to independent versions, and gives VEVOS access to all 44 datasets (cf. Section 2.3).

Benchmarking Differencing Algorithms [43]. In a bachelor's thesis, we integrated tree-differencing and used DiffDetective's evaluation framework to evaluate the performance of differencing algorithms [43]. For 161,000 patches, we compared the performance and diff quality of both differencing strategies described in Section 2.1, based on the tree-differ GumTree [18], the default git diff, and a hybrid version of both [41]. We found the hybrid approach to yield best diff quality, with git diff second, and GumTree last. The runtime for creating variation diffs in DiffDetective is dominated by the tree-differs whereas git diff is orders of magnitudes faster. Currently, we are extending DiffDetective with support for the recent *truediff* algorithm [17].

4 RELATED WORK

In this section, we discuss tools related to DiffDetective. For research related to DiffDetective but without explicit tool support, we refer to the respective research papers [9, 10, 27] and student theses [11, 26, 43, 53] using DiffDetective for various studies.

Diff tools presented in the literature are motivated by mitigating the limitations of traditional line-based differencing in one way or another [14, 17, 21, 28, 54]. While we do not provide a complete classification for the sake of brevity here, a distinguishing characteristic of DiffDetective is its variability-awareness. In fact, one may argue that DiffDetective is not a diff tool itself, but rather a

wrapper around existing generic diff tools. While, to date, we provide integrations with git diff and GumTree [18] due to their widespread usage and maturity, in principle, other generic diff tools such as ChangeDistiller [21], Diff/TS [28], srcDiff [14], truediff [17], to only mention a few, may be wrapped as well.

Software product-line tools such as TypeChef [32], CIDE [31], SuperC [25], FeatureIDE [33], the e4CompareFramework [46], KernelHaven [35], and others, analyze or support development of configurable software, for example to perform variability-aware parsing [25, 31, 32], syntax or type checking [30, 31]. These tools deal with single revisions of variational software and do not explicitly model changes as done by DiffDetective. While some tools, such as variation control systems [20, 49, 51], analyze changes for specific purposes, they do not provide a general method for variability-aware differencing.

Evolution in software product lines is also targeted by DarwinSPL [44], FEVER [15], and a tool by Kröher et al. [36]. DarwinSPL is an IDE, centered around modeling and analyzing the evolution of *constraints* among features over time (e.g., one feature excludes another for all versions above 2.0). In contrast, DiffDetective's focus lies on studying changes to the *implementation* of features in the code base. FEVER [15] extracts information about changes to variability from a version history to model and populate a project history database [19], and to track the evolution of individual features. While this facilitates certain kinds of empirical studies, such a large-scale data warehouse approach does not provide a lightweight and readily reusable diff engine like DiffDetective. Kröher et al. [36] present a framework for detailed analyses of product line commits but do not provide a differencing approach.

5 CONCLUSION

In this paper, we gave an overview of the purpose, implementation, validation, and use cases of DiffDetective – a Java library for variability-aware differencing and large-scale empirical analyses of changes to statically configurable software, such as the Linux Kernel, GCC or Vim. DiffDetective has been applied and validated in multiple research endeavors on a significant body of datasets, and covering various use cases. Designed as a reusable library, DiffDetective may be used as a basis to implement or evaluate algorithms and tools based on changes to variational software. Paired with three years of active development, DiffDetective has become a mature library for researchers and practitioners.

ACKNOWLEDGMENTS

We thank everyone else involved in developing DiffDetective, including Christof Tinnes, Sören Viegner, and Kevin Jedelhauser. This work has been supported by the German Research Foundation within the project *VariantSync* (TH 2387/1-1 and KE 2267/1-1).

REFERENCES

- [1] 2024. DiffDetective Demo. <https://github.com/VariantSync/DiffDetective-Demo>.
- [2] 2024. DiffDetective Forked Data Sets. <https://github.com/DiffDetective>.
- [3] 2024. DiffDetective Website. <https://variantsync.github.io/DiffDetective>.
- [4] 2024. Screencast. <https://www.youtube.com/watch?v=q6ight5EDQY>.
- [5] Iago Abal, Jean Melo, Stefan Stănculescu, Claus Brabrand, Márcio Ribeiro, and Andrzej Wasowski. 2018. Variability Bugs in Highly Configurable Systems: A Qualitative Analysis. *TOSEM* 26, 3, Article 10 (Jan. 2018), 10:1–10:34 pages. <https://doi.org/10.1145/3149119>

- [6] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines*. Springer. <https://doi.org/10.1007/978-3-642-37521-7>
- [7] Taweessup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold. 2004. A Differencing Algorithm for Object-Oriented Programs. In *ASE*. IEEE, 2–13. <https://doi.org/10.1109/ASE.2004.10015>
- [8] Muhammad Asaduzzaman, Chanchal K. Roy, Kevin A. Schneider, and Massimiliano Di Penta. 2013. LHDiff: A Language-Independent Hybrid Approach for Tracking Source Code Lines. In *ICSM*. IEEE, 230–239. <https://doi.org/10.1109/ICSM.2013.34>
- [9] Paul Maximilian Bittner, Alexander Schultheiß, Sandra Greiner, Benjamin Moosher, Sebastian Krieter, Christof Tinnes, Timo Kehler, and Thomas Thüm. 2023. Views on Edits to Variational Software. In *SPLC*. ACM, 141–152. <https://doi.org/10.1145/3579027.3608985>
- [10] Paul Maximilian Bittner, Christof Tinnes, Alexander Schultheiß, Sören Viegner, Timo Kehler, and Thomas Thüm. 2022. Classifying Edits to Variability in Source Code. In *ESEC/FSE*. ACM, 196–208. <https://doi.org/10.1145/3540250.3549108>
- [11] Lukas Bormann. 2022. *Reverse Engineering Feature-Aware Commits From Software Product-Line Repositories*. Bachelor's Thesis. University of Ulm. <https://doi.org/10.18725/OPARU-47892>
- [12] Gerardo Canfora, Luigi Cerulo, and Massimiliano Di Penta. 2009. Ldiff: An Enhanced Line Differencing Tool. In *ICSE*. IEEE, 595–598. <https://doi.org/10.1109/ICSE.2009.5070564>
- [13] Krzysztof Czarnecki and Ulrich Eisenberger. 2000. *Generative Programming: Methods, Tools, and Applications*. ACM/Addison-Wesley.
- [14] Michael John Decker, Michael L. Collard, L. Gwenn Volkert, and Jonathan I. Maletic. 2020. srcDiff: A Syntactic Differencing Approach to Improve the Understandability of Deltas. *JSEP* 32, 4 (2020). <https://doi.org/10.1002/smr.2226>
- [15] 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. *EMSE* 23, 2 (April 2018), 905–952. <https://doi.org/10.1007/s10664-017-9557-6>
- [16] Georg Dotzler and Michael Philippsen. 2016. Move-Optimized Source Code Tree Differencing. In *ASE*. ACM, 660–671.
- [17] Sebastian Erdweg, Tamás Szabó, and André Pacak. 2021. Concise, Type-Safe, and Efficient Structural Diffing. In *PLDI* (Virtual, Canada). ACM, New York, NY, USA, 406–419. <https://doi.org/10.1145/3453483.3454052>
- [18] Jean-Rémy Falleri, Flóreal Morandart, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-Grained and Accurate Source Code Differencing. In *ASE*. 313–324. <https://doi.org/10.1145/2642937.2642982>
- [19] Michael Fischer, Martin Pinzger, and Harald C. Gall. 2003. Populating a Release History Database from Version Control and Bug Tracking Systems. In *ICSM*. IEEE, 23. <https://doi.org/10.1109/ICSM.2003.1235403>
- [20] Stefan Fischer, Lukas Linsbauer, Roberto E. Lopez-Herrejon, and Alexander Egyed. 2015. The ECCO Tool: Extraction and Composition for Clone-and-Own. In *ICSE*. IEEE, 665–668. <https://doi.org/10.1109/ICSE.2015.218>
- [21] Beat Fluri, Michael Wuersch, Martin Pinzger, and Harald Gall. 2007. Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction. *TSE* 33, 11 (Nov. 2007), 725–743. <https://doi.org/10.1109/TSE.2007.70731>
- [22] Patrick Franz, Thorsten Berger, Ibrahim Fayaz, Sarah Nadi, and Evgeny Groshev. 2021. ConfigFix: Interactive Configuration Conflict Resolution for the Linux Kernel. In *ICSE-SEIP*. IEEE, 91–100. <https://doi.org/10.1109/ICSE-SEIP52600.2021.00018>
- [23] Veit Frick, Thomas Grassauer, Fabian Beck, and Martin Pinzger. 2018. Generating Accurate and Compact Edit Scripts Using Tree Differencing. In *ICSME*. IEEE, 264–274.
- [24] Harald C. Gall, Beat Fluri, and Martin Pinzger. 2009. Change Analysis With Evolizer and Changedistiller. *IEEE Software* 26, 1 (2009), 26–33.
- [25] Paul Gazzillo and Robert Grimm. 2012. SuperC: Parsing All of C by Taming the Preprocessor. In *PLDI*. ACM, 323–334. <https://doi.org/10.1145/2254064.2254103>
- [26] Lukas Güthing. 2023. *Inspecting the Evolution of Feature Annotations in Configurable Software*. Master's Thesis. University of Ulm. To appear.
- [27] Lukas Güthing, Paul Maximilian Bittner, Ina Schaefer, and Thomas Thüm. 2024. Explaining Edits to Variability Annotations in Evolving Software Product Lines. In *VaMoS*. ACM. To appear.
- [28] Masatomo Hashimoto and Akira Mori. 2008. Diff/TS: A Tool for Fine-Grained Structural Change Analysis. In *WCRE*. 279–288. <https://doi.org/10.1109/WCRE.2008.44>
- [29] Claus Hunsen, Bo Zhang, Janet Siegmund, Christian Kästner, Olaf Leßenich, Martin Becker, and Sven Apel. 2016. Preprocessor-Based Variability in Open-Source and Industrial Software Systems: An Empirical Study. *EMSE* 21, 2 (April 2016), 449–482. <https://doi.org/10.1007/s10664-015-9360-1>
- [30] Christian Kästner and Sven Apel. 2008. Type-Checking Software Product Lines—A Formal Approach. In *ASE*. IEEE, 258–267.
- [31] Christian Kästner, Sven Apel, and Martin Kuhlemann. 2008. Granularity in Software Product Lines. In *ICSE*. ACM, 311–320. <https://doi.org/10.1145/1368088.1368131>
- [32] 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. In *OOPSLA*. ACM, 805–824. <https://doi.org/10.1145/2048066.2048128>
- [33] Christian Kästner, Thomas Thüm, Gunter Saake, Janet Feigenspan, Thomas Leich, Fabian Wielgorz, and Sven Apel. 2009. FeatureIDE: A Tool Framework for Feature-Oriented Software Development. In *ICSE*. IEEE, 611–614. <https://doi.org/10.1109/ICSE.2009.5070568> Formal demonstration paper.
- [34] Timo Kehler, Udo Kelter, Pit Pietsch, and Maik Schmidt. 2012. Adaptability of Model Comparison Tools. In *ASE*. ACM, 306–309. <https://doi.org/10.1145/2351676.2351731>
- [35] Christian Kröher, Sascha El-Sharkawy, and Klaus Schmid. 2018. KernelHaven: An Experimentation Workbench for Analyzing Software Product Lines. In *ICSEC*. ACM, New York, NY, USA, 73–76. <https://doi.org/10.1145/3183440.3183480>
- [36] Christian Kröher, Lea Gerling, and Klaus Schmid. 2023. Comparing the Intensity of Variability Changes in Software Product Line Evolution. *JSS* 203 (Sept. 2023), 111737. <https://doi.org/10.1016/j.jss.2023.111737>
- [37] Elias Kuitert, Sebastian Krieter, Chico Sundermann, Thomas Thüm, and Gunter Saake. 2022. Tseitin or not Tseitin? The Impact of CNF Transformations on Feature-Model Analyses. In *ASE*. ACM, 110:1–110:13. <https://doi.org/10.1145/3551349.3556938>
- [38] Duc Le, Eric Walkingshaw, and Martin Erwig. 2011. #ifdef Confirmed Harmful: Promoting Understandable Software Variation. In *VL/HCC*. IEEE, 143–150. <https://doi.org/10.1109/VLHCC.2011.6070391>
- [39] Bixin Li, Xiaobing Sun, Hareton Leung, and Sai Zhang. 2013. A Survey of Code-Based Change Impact Analysis Techniques. *STVR* 23, 8 (2013), 613–646.
- [40] 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 *ICSE*. IEEE, 105–114. <https://doi.org/10.1145/1806799.1806819>
- [41] Junnosuke Matsumoto, Yoshiaki Higo, and Shinji Kusumoto. 2019. Beyond GumTree: A Hybrid Approach to Generate Edit Scripts. In *MSR*. IEEE, 550–554. <https://doi.org/10.1109/MSR.2019.00082>
- [42] Flávio Medeiros, Márcio Ribeiro, Rohit Gheyi, Sven Apel, Christian Kästner, Bruno Ferreira, Luiz Carvalho, and Baldoino Fonseca. 2018. Discipline Matters: Refactoring of Preprocessor Directives in the #ifdef Hell. *TSE* 44, 5 (2018), 453–469. <https://doi.org/10.1109/TSE.2017.2688333>
- [43] Benjamin Moosher. 2023. *Constructing Variation Diffs Using Tree Diffing Algorithms*. Bachelor's Thesis. University of Ulm. <https://doi.org/10.18725/OPARU-50108>
- [44] Michael Niekie, Gil Engel, and Christoph Seidl. 2017. DarwinSPL: An Integrated Tool Suite for Modeling Evolving Context-aware Software Product Lines. In *VaMoS*. ACM, 92–99. <https://doi.org/10.1145/3023956.3023962>
- [45] Yusuf Sulisty Nugroho, Hideaki Hata, and Kenichi Matsumoto. 2020. How Different are Different Diff Algorithms in Git? *EMSE* 25, 1 (2020), 790–823. <https://doi.org/10.1007/s10664-019-09772-z>
- [46] Kamil Rosiak and Ina Schaefer. 2023. The e4CompareFramework: Annotation-based Software Product-Line Extraction. In *SPLC*. ACM, 34–38. <https://doi.org/10.1145/3579028.3609012>
- [47] Alexander Schultheiß, Paul Maximilian Bittner, Sascha El-Sharkawy, Thomas Thüm, and Timo Kehler. 2022. Simulating the Evolution of Clone-and-Own Projects With VEVOS. In *EASE*. ACM, 231–236. <https://doi.org/10.1145/3530019.3534084>
- [48] Alexander Schultheiß, Paul Maximilian Bittner, Sandra Greiner, and Timo Kehler. 2023. Benchmark Generation With VEVOS: A Coverage Analysis of Evolution Scenarios in Variant-Rich Systems. In *VaMoS*. ACM, 13–22. <https://doi.org/10.1145/3571788.3571793>
- [49] Felix Schwägerl and Bernhard Westfechtel. 2016. SuperMod: Tool Support for Collaborative Filtered Model-Driven Software Product Line Engineering. In *ASE*. ACM, 822–827. <https://doi.org/10.1145/2970276.2970288>
- [50] Daniel Strüber, Mukelabai Mukelabai, Jacob Krüger, Stefan Fischer, Lukas Linsbauer, Jabier Martinez, and Thorsten Berger. 2019. Facing the Truth: Benchmarking the Techniques for the Evolution of Variant-Rich Systems. In *SPLC*. ACM, New York, NY, USA, 26:1–26:12. <https://doi.org/10.1145/3336294.3336302>
- [51] Stefan Stănculescu, Thorsten Berger, Eric Walkingshaw, and Andrzej Wąsowski. 2016. Concepts, Operations, and Feasibility of a Projection-Based Variation Control System. In *ICSME*. IEEE, 323–333. <https://doi.org/10.1109/ICSME.2016.88>
- [52] Chico Sundermann, Tobias Heß, Michael Niekie, Paul Maximilian Bittner, Jeffrey M. Young, Thomas Thüm, and Ina Schaefer. 2023. Evaluating State-of-the-Art #SAT Solvers on Industrial Configuration Spaces. *EMSE* 28 (Jan. 2023). <https://doi.org/10.1007/s10664-022-10265-9>
- [53] Sören Viegner. 2021. *Empirical Evaluation of Feature Trace Recording on the Edit History of Marlin*. Bachelor's Thesis. University of Ulm. <https://doi.org/10.18725/OPARU-38603>
- [54] Zhenchang Xing and Eleni Stroulia. 2005. UMLDiff: An Algorithm for Object-Oriented Design Differencing. In *ASE*. ACM, 54–65.