Constructing Variation Diffs Using Tree Diffing Algorithms

Benjamin Moosherr, 23.05.2023

Advisor: M.Sc. Paul Maximilian Bittner

Reviewer: Prof. Dr.-Ing. Thomas Thüm

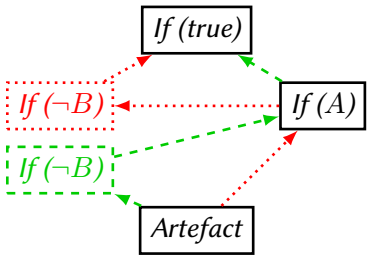
1 Introduction

A software product line implements a collection of related software variants, while sharing implementation artifacts, for instance source code lines, between these variants [Ape+13]. To identify a variant, it is often associated with a set of features implemented in that variant [GBS01; Ape+13]. For example, the Linux kernel implements support for different hardware as a set of features. Some subsets of these features can be chosen to produce a Linux variant. Formally, a software product line can be modelled by a variation tree which associates each artifact with a presence condition stating which variants include a given software artifact [Bit+22]. However, features may have complicated relations to each other and the set of all variants may be very large. Thus the evolution of software product lines can be challenging [Tar+11].

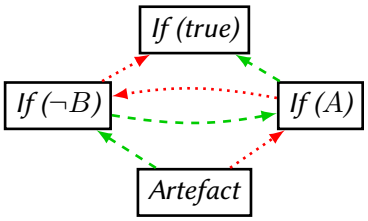
For developers, it is important to understand the implications of changes made to a software product line to reduce the amount of errors during evolution. A useful tool for analyzing changes to a variation tree is the variation diff [Bit+22], an example of which can be as seen in Figure 1a. It encodes two variation trees, one before the change and one after the change, by specifying, for each node and edge, whether it was inserted (green), deleted (red) or unchanged (black). Unfortunately, the construction of such variation diffs is ambiguous, demonstrated by Figure 1b which encodes the same change as Figure 1a. Indeed, the quality[1] of the resulting variation diff depends on the construction algorithm, possibly influencing their usefulness for developers who want to analyze changes using variation diffs.

The current approach for constructing variation diffs uses a line-based diffing algorithm [Vie21] (e.g., Myers algorithm [Mye86]) to detect inserted, deleted and unmodified lines. However, there are alternative diffing algorithms which can, for example, detect moved lines [Hec78; CCDP07] or operate on (syntax) trees instead of lines [Cha+96; PA11; Fal+14]. Notably, variation trees may be diffed using such tree diffing algorithms.

[1] For example, the number of nodes or the number of edges marked as unchanged.



(a) Example variation diff, changing the order of the mapping nodes *If (A)* and *If (¬B)*.



(b) Equivalent variation diff to Figure 1a, but the movement of the *If (¬B)* node was detected.

Figure 1: Comparison of two equivalent variation diffs.

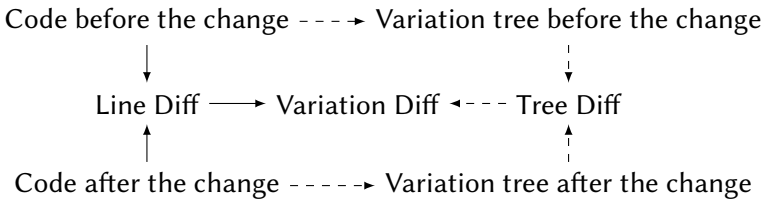


Figure 2: Dataflow during construction of variation diffs. Solid edges mark the algorithm by Viegener [Vie21], dashed edges the proposed algorithm.

2 Problem Statement

The construction of variation diffs given by Viegener [Vie21] does not incorporate the detection of moved artifacts, introducing mental overhead for developers analyzing common changes to source code. Moreover, Bittner et al. [Bit+22] laid the foundation of handling more fine grained changes in variation trees without implementing the possibility for such granularity when constructing their variation diffs. Ideally, the variation diff construction algorithm could detect moved artifacts while being able to process variation trees with different levels of granularity.

3 Contribution

Conceptual: We apply tree based diffing to variation diff construction. In contrast to using line-based diffing, as seen with solid arrows in Figure 2, we construct variation trees before and after a change and apply a tree diffing algorithm, marked by dashed arrows in Figure 2. Thus, the granularity of the diffed variation trees can be chosen freely and it is possible to detect moved nodes.

Artifact: We implement the proposed variation diff construction algorithm by choosing a suitable

tree diffing algorithm as basis.

Evaluation: We conduct an experiment to evaluate the difference between the variation diff construction algorithm presented by us on the one hand and the algorithm given by Viegener [Vie21] on the other hand.

3.1 Work Packages

Literature research: We must research related work on the evolution of software product lines and select suitable tree diffing algorithms for the implementation.

Implementation: We must implement the proposed construction algorithm as reusable part of DiffDetective[2], a library for analyzing variation diffs. We should reuse, adapt or implement a suitable diffing algorithm for this implementation.

Metric: We should research or conceptualize one or more metric for the quality of the constructed variation diffs.

Qualitative evaluation: We should compare our variation diff construction algorithm with the construction algorithm given by Viegener [Vie21] according to our quality metrics.

Quantitative evaluation We may compare the edit class occurrences quantitatively in comparison to the results in Bittner et al. [Bit+22], to evaluate the benefit of move detection.

Benchmark: We should evaluate the performance difference between our construction algorithm and the algorithm presented in Viegener [Vie21].

[2] `https://github.com/VariantSync/DiffDetective/`

3.2 Schedule

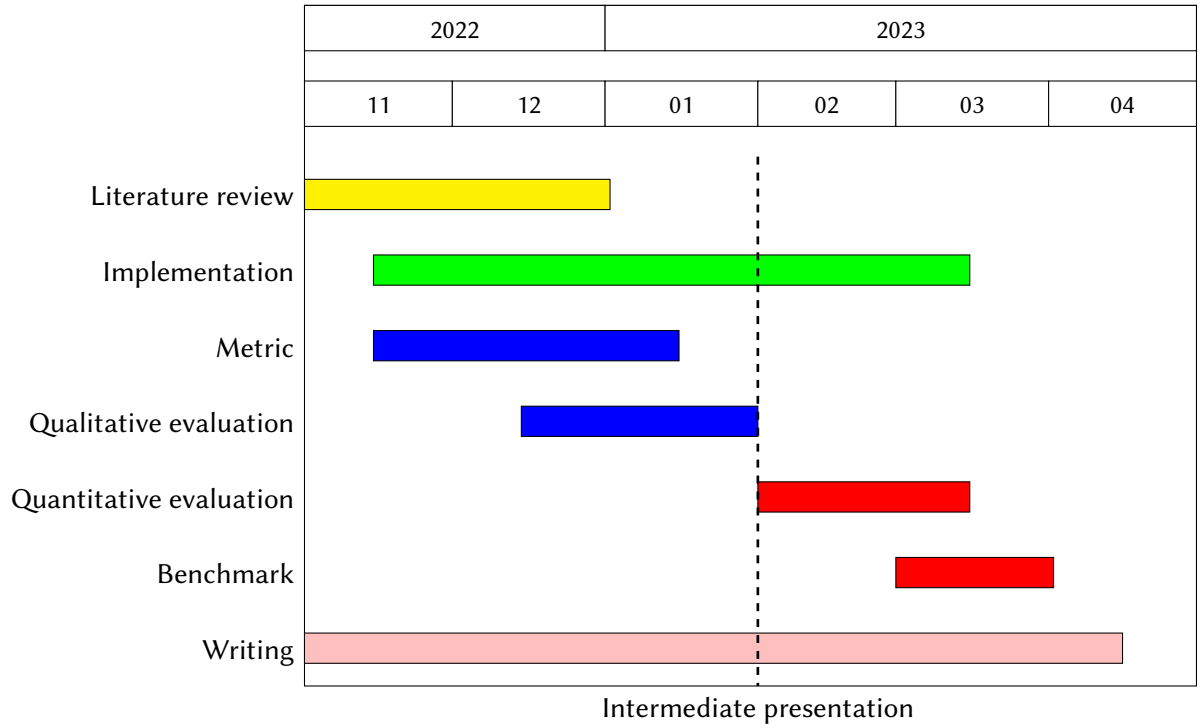


Figure 3: Thesis time schedule in months

References

[Hec78] Paul Heckel. "A Technique for Isolating Differences between Files". In: *Comm. ACM* 21.4 (Apr. 1978), 264–268. ISSN: 0001-0782. DOI: 10.1145/359460.359467. URL: https://doi.org/10.1145/359460.359467 (see Page 1).

[Mye86] Eugene W. Myers. "An O(ND) Difference Algorithm and Its Variations". In: *Algorithmica* 1.1-4 (1986), pp. 251–266. DOI: 10.1007/BF01840446 (see Page 1).

[Cha+96] Sudarshan S. Chawathe et al. "Change Detection in Hierarchically Structured Information". In: *Proc. Int'l Conf. on Management of Data (SIGMOD)*. Montreal, QC, Canada: ACM, 1996, pp. 493–504. ISBN: 0-89791-794-4. DOI: 10.1145/233269.233366. URL: http://doi.acm.org/10.1145/233269.233366 (see Page 1).

[GBS01] Jilles van Gurp, Jan Bosch, and Mikael Svahnberg. "On the Notion of Variability in Software Product Lines". In: *Proc. Working Conf. on Software Architecture (WICSA)*. 2001, pp. 45–54 (see Page 1).

[CCDP07] Gerardo Canfora, Luigi Cerulo, and Massimiliano Di Penta. "Identifying Changed Source Code Lines from Version Repositories". In: *Proc. Working Conf. on Mining Software Repositories (MSR)*. IEEE, May 2007, pp. 14–14. DOI: 10.1109/MSR.2007.14 (see Page 1).

[PA11] Mateusz Pawlik and Nikolaus Augsten. “RTED: A Robust Algorithm for the Tree Edit Distance”. In: *Computing Research Repository (CoRR)* 5.4 (2011), pp. 334–345. ISSN: 2150–8097. URL: http://arxiv.org/abs/1201.0230 (see Page 1).

[Tar+11] Reinhard Tartler et al. “Feature Consistency in Compile-Time-Configurable System Software: Facing the Linux 10,000 Feature Problem”. In: *Proc. Europ. Conf. on Computer Systems (EuroSys)*. Salzburg, Austria: ACM, 2011, pp. 47–60. ISBN: 978-1-4503-0634-8. DOI: http://doi.acm.org/10.1145/1966445.1966451 (see Page 1).

[Ape+13] Sven Apel et al. *Feature-Oriented Software Product Lines*. Berlin, Heidelberg, Germany: Springer, 2013. ISBN: 978-3-642-37520-0. DOI: 10.1007/978-3-642-37521-7. URL: https://doi.org/10.1007/978-3-642-37521-7 (see Page 1).

[Fal+14] Jean-Rémy Falleri et al. “Fine-Grained and Accurate Source Code Differencing”. In: *Proc. Int’l Conf. on Automated Software Engineering (ASE)*. 2014, pp. 313–324. DOI: 10.1145/2642937.2642982. URL: http://doi.acm.org/10.1145/2642937.2642982 (see Page 1).

[Vie21] Sören Viegener. “Empirical Evaluation of Feature Trace Recording on the Edit History of Marlin”. Bachelor’s Thesis. Germany: University of Ulm, Apr. 2021. DOI: 10.18725/OPARU-38603. URL: https://oparu.uni-ulm.de/xmlui/bitstream/handle/123456789/38679/BA_Viegener.pdf (see Pages 1–3).

[Bit+22] Paul Maximilian Bittner et al. “Classifying Edits to Variability in Source Code”. In: *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*. Singapore, Singapore: ACM, Nov. 2022, pp. 196–208. ISBN: 9781450394130. DOI: 10.1145/3540250.3549108 (see Pages 1–3).