Rücküberführung von Variation Trees und Diffs

Eugen Shulimov, 11.07.2024

Betreuer: M.Sc. Paul Maximilian Bittner

Gutachter: Prof. Dr. Thomas Thüm

1 Einleitung

Bei der Entwicklung von konfigurierbaren Softwaresystemen, wie zum Beispiel Clone-and-Own, oder Softwareproduktlinien, gibt es im Laufe des Lebenszyklus immer mehr Features. Es ist von Vorteil, eine Möglichkeit zu haben, die Features im Code zu unterscheiden und automatisch zu finden. Einige Möglichkeiten dazu wären Präprozessor-Annotationen, oder Build-Systeme [Ape+13]. Wie bei der Clone-and-Own-Entwicklung, wo für jede Variante der Software eine neue Kopie der gesamten Software angelegt wird und parallel entwickelt wird [Bit+22]. Dort müssen Features gefunden werden, um diese zu aktualisieren [Kui+18; Zho+18; KB20a; KB20b; Keh+21; Bit+22].

Die Entwickler sind bei der Entwicklung von konfigurierbarer Software daran interessiert, zu verstehen, wie sich ihre Änderungen auf die Variabilität auswirken und wie die Variabilität von konfigurierbarer Software aussieht [Bit+22]. Sonst wenn man das Verständnis über die Auswirkungen der Änderung nicht hat, kann das zu Fehlern und Problemen bei der Entwicklung führen [K̈+12; Nev+15; SBT19; Sun+21; Bit+22; Nie+22]. Dies stellt einen Aspekt einer größeren Aufgabe dar, der Aufrechterhaltung und Weiterentwicklung von Informationen über Variabilität bei Quellcodeänderungen [Bit+22]. Für die Entwickler stellt diese Aufgabe eine große Herausforderung dar [SHA12; Pas+13; Pas+16; Bit+22].

Der C-Präprozessor ist eine Möglichkeit, Variabilität zu erzeugen [Ape+13]. Der C-Präprozessor ist ein Tool, das den Quellcode vor dem Kompilieren manipuliert [Ape+13]. Dieses Tool bietet Möglichkeiten zur Dateieinbindung, zu lexikalische Makros, und zur bedingte Kompilierung [Ape+13]. Wie ein mit dem C-Präprozessor annotierter Code aussehen kann, ist in der Abbildung 1 Stelle Ⓥ zu sehen (Abb. 1 St.Ⓥ). Um die Variabilität mithilfe des C-Präprozessors zu erzeugen, brauchen wir dessen Möglichkeit zur bedingten Kompilierung [Ape+13]. Dabei können beliebige Aussageformeln über Features im Quellcode mit den C-Präprozessor-Anweisungen #if, #ifdef und, #ifndef abgebildet werden [Bit+22] (Abb. 1 St.Ⓥ).

Zur Unterstützung der Variabilitätsanalyse kann man Tools verwenden [ZB14; Spr+20], wie zum Beispiel DiffDetective. DiffDetective ist eine Java-Bibliothek [Bit+24]. Der Zweck von DiffDetective ist es, Änderungen im Quellcode und Änderungen der Variabilität darstellbar und den Zusammenhang zwischen ihnen analysierbar zu machen. DiffDetective stellt einen variabilitätsbezogenen Differencer [Bit+22; Bit+24] zur Verfügung, der sich nur auf Aspekte im Code/Text bezieht, welche die Variabilität berücksichtigen. Diese Bibliothek ermöglicht auch die Analyse der Versionshistorie von Softwareproduktlinien [Bit+22] und bietet daher einen flexiblen Rahmen für großangelegte empirische Analysen von Git-Versionsverläufen statisch konfigurierbarer Software [Bit+23; Bit+24].

Zentral für DiffDetective sind zwei formal verifizierte Datenstrukturen für Variabilität und Änderungen an dieser [Bit+22]. Das sind Variation-Trees (Abb. 1 St.ⓧ) für variabilitätbezogenen Code (Abb.1 St.ⓥ) und Variation-Diffs (Abb. 1 St.ⓨ) für variabilitätsbezogene Diffs (Abb. 1 St.ⓦ). Diese Datenstrukturen sind generisch. Das bedeutet, dass die Datenstrukturen möglichst von der Umsetzung der Variabilität im Code abstrahieren. Also kann eine Umsetzungsmöglichkeit leicht durch eine andere ersetzt werden, zum Beispiel können C-Präprozessor-Annotationen durch Java-Präprozessor-Annotationen ohne oder geringer Änderungen an den Datenstrukturen selbst, ersetzt werden. Ein Variation-Tree ist ein Baum, welcher die Verzweigungen/Variationen eines annotierten Codes darstellt [Bit+22; Bit+23; Bit+24]. Ein Variation-Diff ist ein Graph, welcher die Unterschiede zwischen zwei Variation-Trees zeigt [Bit+22; Bit+23; Bit+24]. In beiden Fällen werden die Bedingungsknoten, welche Informationen zu Variabilität erhalten, und die Code-Knoten unterschieden. Beim Variation-Diff sind zudem die eingefügten Knoten, die gelöschten Knoten und, die unveränderten Knoten zu unterscheiden.

Das Parsen führt die Eingabe von der konkreten Syntax in die abstrakte Syntax um. In unserem Fall parst DiffDetective C-Präprozessor-Annotationen, dieses kann aber auch auf andere Präprozessor-Annotationen erweitert werden. Beim Parsen wird nur der C-Präprozessor-Annotierter Code in seine abstrakte Syntax überführt, der C- bzw. C++-Code wird als Text behandelt und wird nicht geparst. Das Parsen in DiffDetective funktioniert für Variation-Trees und für Variation-Diffs über einen einzigen gemeinsamen Algorithmus. Der Algorithmus arbeitet wie folgt: Er geht über alle Zeilen des Codes/Textes und schaut sich für jede Zeile an, wie diese Zeile manipuliert wurde, ob die Zeile unverändert geblieben ist, gelöscht wurde, oder neu ist [Vie21]. Dazu wird festgelegt von welchem Typ die Zeile ist, also ob diese Zeile C/C++ Code enthält oder eine C-Präprozessor-Kontrollstruktur. Als

Nächstes wird geprüft, ob die Zeile eine #endif-Annotation enthält. Wenn ja, dann weist das darauf hin, dass ein Bedingungsblock zu Ende ist. Wenn die Zeile kein #endif enthält, dann wird ein neuer Knoten mit Informationen über Elternknoten, den Typ der Zeile und, wie die Zeile manipuliert wurde, erstellt. Wenn dieser Knoten kein Code-Knoten ist, wird er gemerkt und für die Angabe der Elternknoten verwendet. Der Algorithmus ist an sich für das Parsen von textbasierten Diffs in Variation-Diffs ausgelegt (Abb. 1 St.⑤). An den Stellen ① und ⑨ wird anders vorgegangen, da wir als Eingabe ein C-Präprozessor Code (Abb. 1 St.ⓥ) haben und als Ausgabe ein Variation-Tree (Abb. 1 St.ⓧ). Der gegebene Algorithmus ist für das direkte Parsen von C-Präprozessor Code nicht ausgelegt. Deshalb wurde dort Umwege verwendet, um diesen Algorithmus anwendbar zu machen und die benötigte Ausgabe zu erzielen. Ein Text kann in ein nicht verändertes, textbasiertes Diff umgewandelt werden, durch die Bildung eines Diffs mit sich selbst. Dadurch ist es möglich aus C-Präprozessor Code (Abb. 1 St.ⓥ) ein textbasierten Diff (Abb. 1 St.ⓦ) zu erzeugen, also wurden die Stelle ⑪ oder ⑫ verwendet. Da jetzt ein textbasiertes Diff vorhanden ist, kann der Algorithmus darauf angewandt werden (Abb. 1 St.⑤). Um aus dem erhaltenen Variation-Diff (Abb. 1 St.ⓨ) ein Variation-Tree zu bekommen, muss man die Stelle ④ oder ⑧ aus der Abbildung 1 anwenden. So sieht man das die Stelle ① durch die Stellen ⑪,⑤,④ [① → ⑪,⑤,④] und die Stelle ⑨ durch die Stellen ⑫,⑤,⑧ [⑨ → ⑫,⑤,⑧] ersetzt werden kann.

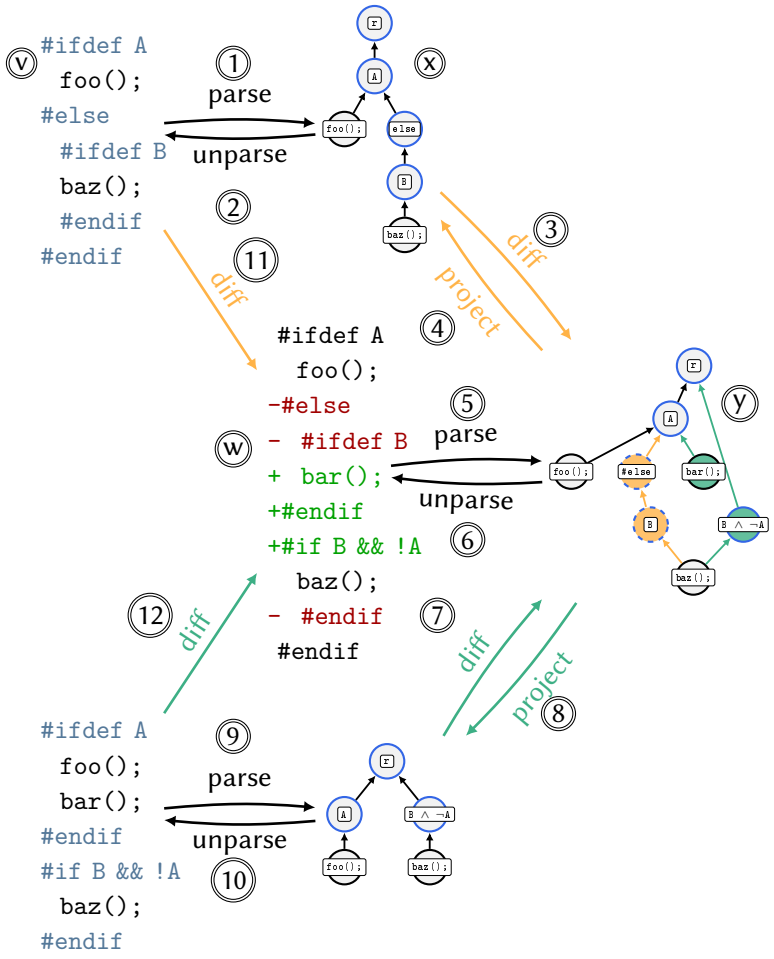

Abbildung 1: Überblick über Variabilität bezogene Konvertierungen

2 Problemstellung

Obwohl DiffDetective Funktionen zum Parsen (Abb. 1 St.①,⑤,⑨) hat, besitzt dieses Tool keine Funktion zum Unparsen (Abb. 1 St.②,⑥,⑩) von Variation-Trees und Variation-Diffs. Das Unparsen ist die Überführung aus der abstrakten Syntax in die konkrete Syntax, also ist das Unparsen die Invertierung des Parsens. Unser Ziel ist es, das zu ändern. Dazu müssen wir einen Unparser entwickeln, welcher auf direktem oder indirektem Wege, Variation-Trees (Abb. 1 St.ⓧ) in C-Präprozessor-Annotierten Code (Abb. 1 St.ⓥ) und Variation-Diffs (Abb. 1 St.ⓨ) in textbasierte Diffs (Abb. 1 St.ⓦ) überführt.

Eine Einsatzmöglichkeit des Unparsers wäre, das Unparsen von Variation-Trees, bei denen die Variabilität mutiert wurde. Eine Möglichkeit zu Analyse von Softwareproduktlinien ist Mutation-Tests. Bei Mutation-Tests werden Mutation-Operatoren verwendet, welche aber nur auf der abstrakten

Ebene, also auf Variation-Trees, angewandt werden können [AH+16]. Um weiter in der Analyse vorzugehen, muss man von der abstrakten Ebene zu der konkreten Ebene übergehen und hier wird der Unparser angewandt. Eine andere Einsatzmöglichkeit wäre die Verwendung von Variation-Diffs als Patches. Wenn ein Patch modifiziert werden muss, um ihn für andere Versionen zu verwenden oder um Änderungen zu sehen, die nur ein bestimmtes Feature betreffen [Bit+23].

Für das Unparsen stellt das Fehlen einiger Informationen, die im annotierten Code vorhanden sein müssen, aber in Variation-Trees bzw. Variation-Diffs nicht vorhanden sind, das größte Problem dar. Diese Informationen sind entweder durch das Parsen verloren gegangen oder waren von Anfang an nicht vorhanden, wenn Variation-Trees bzw. Variation-Diffs ohne Parsen gebildet wurden. Diese Informationen sind die exakte Formel, die ein Mapping-Knoten $\tau(v) = \texttt{mapping}$ besitzt [Bit+22], die Position von #endif und deren Einrückung. Aus diesem Grund müssen wir entweder Annahmen treffen, oder DiffDetective so erweitern, dass er diese Information explizit speichert. Eine Annahme könnte sein, dass das #endif genauso eingerückt ist, wie die Bedingung, zu der es gehört.

Eine Möglichkeit könnte sein, das Unparsen von Variation-Trees direkt umzusetzen. Dazu muss ein entsprechender Algorithmus entwickelt werden. Für das Unparsen von Variation-Diffs ziehen wir in Betracht indirekt vorzugehen, ähnlich wie bei dem Parsen von C-Präprozessor-Annotierten Code zu Variation-Trees.

3 Beitrag

Der Beitrag setzt sich aus Konzept, Implementierung und Auswertung zusammen. Beim Konzept wird ein Vorgehen zum Unparsen von Variation-Trees und Variation-Diffs in das ursprüngliche Textformat ausgearbeitet. In der Implementierung wird dieses Vorgehen in das DiffDetective-Tool eingebaut. In der Bachelorarbeit wird eine Metrik spezifiziert, anhand derer die Korrektheit bewertet wird. Zurzeit wird in Betracht gezogen, die Korrektheit, der Implementierung anhand folgender Kriterien festzustellen: syntaktische Gleichheit, syntaktische Gleichheit ohne Whitespace und semantische Gleichheit. Ein ähnliches Kriterium für die Gleichheit bezogen aber auf Variation-Trees bzw. Variation-Diffs ist im Konferenzbeitrag von Bittner et al. zu finden [Bit+23]. Die syntaktische Gleichheit bedeutet, dass das Verglichene in jedem Zeichen übereinstimmt, so wie das erste Beispiel in Abbildung 2 . Das zweite Beispiel der Abbildung 2 zeigt die syntaktische Gleichheit ohne Whitespace, bei der das Verglichene gleich sein muss, wenn man alle Zeichen, die Whitespace sind, entfernen

würde. Bei der semantischen Gleichheit muss der Sinn gleich sein, was uns das letzte Beispiel der Abbildung 2 zeigt. Das ist wie folgt zu verstehen, zwei Diffs sind semantisch gleich, wenn ihre Projektionen syntaktisch gleich bzw. syntaktisch gleich ohne Whitespace sind. Am Ende der Auswertung wird anhand der vorher spezifizierten Metrik festgelegt, wie korrekt die Implementierung und somit das Vorgehen ist.

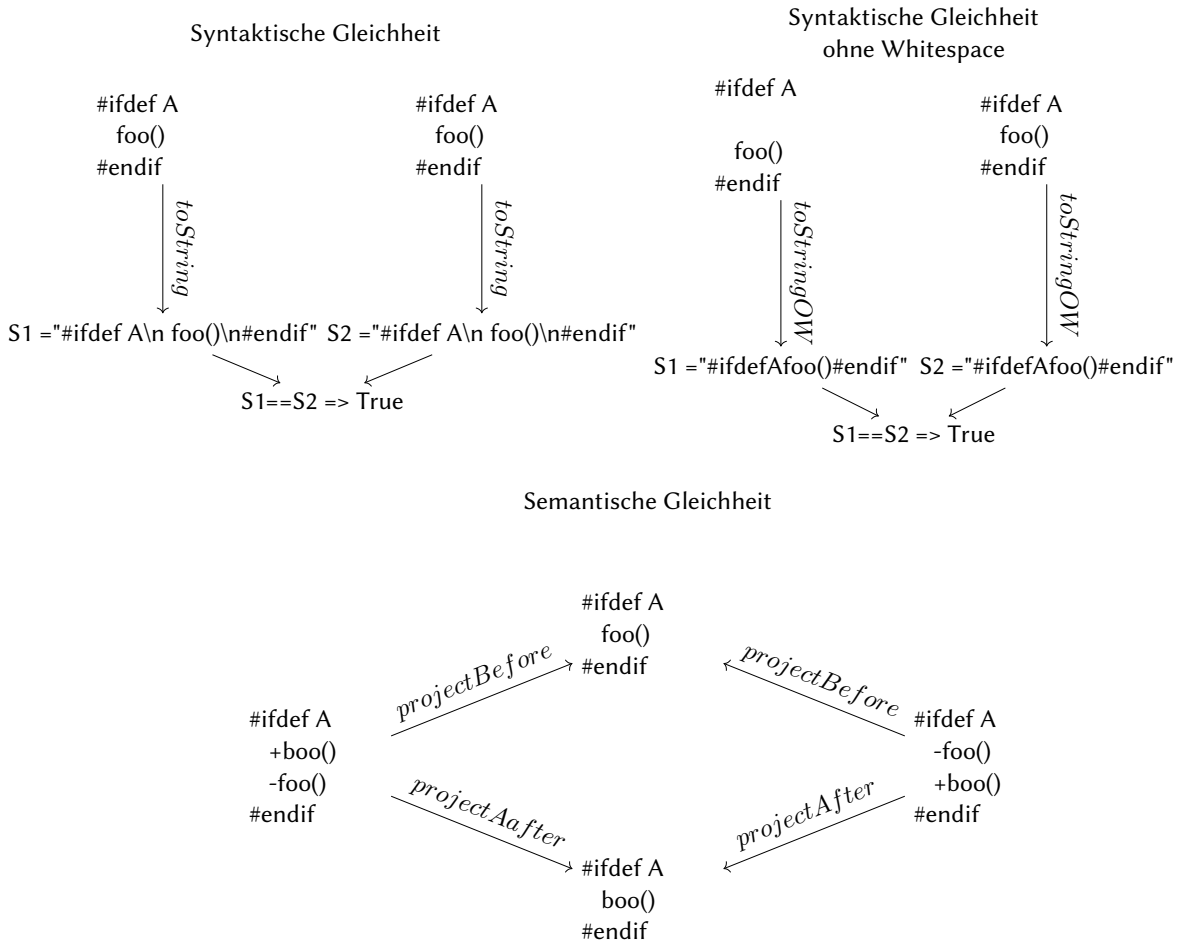

Abbildung 2: Beispiel für Metrik

3.1 Arbeitspakete

Literaturrecherche: Suche der passenden Literatur.

Konzept: Ausarbeitung eines Algorithmus zum Unparsen von Vatiation-Trees/Variation-Diffs.

Metrik: Metrik oder Metriken für die Korrektheit werden festgelegt.

Implementierung: Implementierung des Vorgehens in das DiffDetective-Tool.

Auswertung: Anhand der Metrik/Metriken entscheiden, wie korrekt die Implementierung ist.

Schreiben: Die Bachelorarbeit wird geschrieben.

3.2 Zeitplan

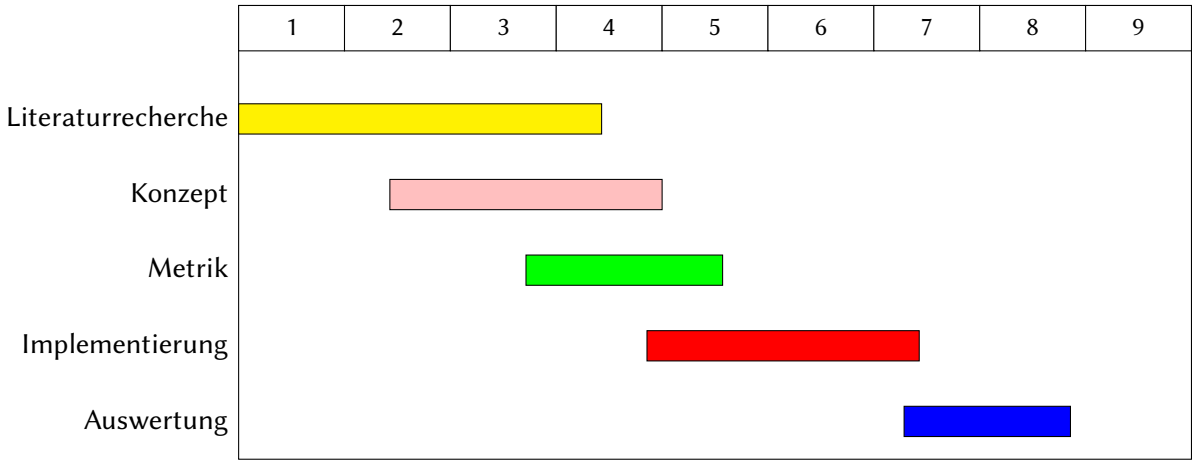

Abbildung 3: Zeitplan für die Ausarbeitung der Bachelorarbeit

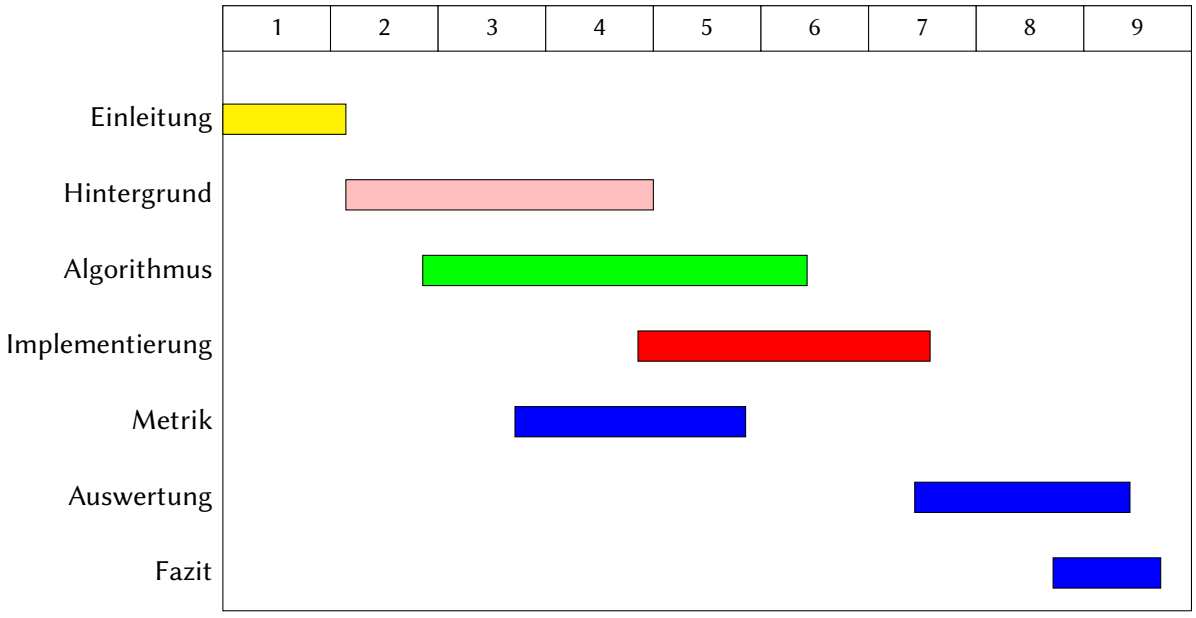

Abbildung 4: Zeitplan für das Schreiben der Bachelorarbeit

References

[K+12] Christian Kästner et al. “Type Checking Annotation-Based Product Lines”. In: *Trans. on Software Engineering and Methodology (TOSEM)* 21.3 (July 2012), 14:1–14:39. ISSN: 1049-331X. DOI: 10.1145/2211616.2211617 (see Page 1).

[SHA12] Christoph Seidl, Florian Heidenreich, and Uwe Aßmann. “Co-Evolution of Models and Feature Mapping in Software Product Lines”. In: *Proc. Int’l Systems and Software Product Line Conf. (SPLC)*. Salvador, Brazil: ACM, 2012, pp. 76–85. DOI: 10.1145/2362536.2362550 (see Page 1).

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

[Pas+13] Leonardo Passos et al. “Feature-Oriented Software Evolution”. In: *Proc. Int’l Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*. Pisa, Italy: ACM, 2013, pp. 1–8. DOI: 10.1145/2430502.2430526 (see Page 1).

[ZB14] Bo Zhang and Martin Becker. “Variability code analysis using the VITAL tool”. In: *Proceedings of the 6th International Workshop on Feature-Oriented Software Development*. FOSD ’14. Västerås, Sweden: Association for Computing Machinery, 2014, 17–22. ISBN: 9781450329804. DOI: 10.1145/2660190.2662113. URL: https://doi.org/10.1145/2660190.2662113 (see Page 2).

[Nev+15] Laís Neves et al. “Safe Evolution Templates for Software Product Lines”. In: *J. Systems and Software (JSS)* 106 (2015), pp. 42–58. DOI: 10.1016/j.jss.2015.04.024. URL: https://www.sciencedirect.com/science/article/pii/S0164121215000801 (see Page 1).

[AH+16] Mustafa Al-Hajjaji et al. “Mutation Operators for Preprocessor-Based Variability”. In: *Proc. Int’l Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*. Salvador, Brazil: ACM, Jan. 2016, pp. 81–88. ISBN: 978-1-4503-4019-9. DOI: 10.1145/2866614.2866626 (see Page 5).

[Pas+16] Leonardo Passos et al. “Coevolution of Variability Models and Related Software Artifacts”. In: *Empirical Software Engineering (EMSE)* 21.4 (2016). DOI: 10.1007/s10664-015-9364-x (see Page 1).

[Kui+18] Elias Kuiter et al. “Getting Rid of Clone-and-Own: Moving to a Software Product Line for Temperature Monitoring”. In: *Proc. Int’l Systems and Software Product Line Conf. (SPLC)*. Gothenburg, Sweden: ACM, Sept. 2018, pp. 179–189. ISBN: 9781450364645. DOI: 10.1145/3233027.3233050 (see Page 1).

[Zho+18] Shurui Zhou et al. “Identifying Features in Forks”. In: *Proc. Int’l Conf. on Software Engineering (ICSE)*. Gothenburg, Sweden: ACM, May 2018, pp. 105–116. DOI: 10.1145/3180155.3180205. URL: https://dl.acm.org/citation.cfm?id=3180205 (see Page 1).

[SBT19] Gabriela Sampaio, Paulo Borba, and Leopoldo Teixeira. “Partially Safe Evolution of Software Product Lines”. In: *J. Systems and Software (JSS)* 155 (2019), pp. 17–42. ISSN: 0164-1212. DOI: 10.1016/j.jss.2019.04.051 (see Page 1).

[KB20a] Jacob Krüger and Thorsten Berger. “Activities and Costs of Re-Engineering Cloned Variants Into an Integrated Platform”. In: *Proc. Int’l Working Conf. on Variability Modelling of Software-Intensive Systems (VaMoS)*. Magdeburg, Germany: ACM, 2020. ISBN: 9781450375016. DOI: 10.1145/3377024.3377044 (see Page 1).

[KB20b] Jacob Krüger and Thorsten Berger. "An Empirical Analysis of the Costs of Clone- and Platform-Oriented Software Reuse". In: *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*. Virtual Event, USA: ACM, 2020, pp. 432–444. ISBN: 9781450370431. DOI: 10.1145/3368089.3409684 (see Page 1).

[Spr+20] Joshua Sprey et al. "SMT-Based Variability Analyses in FeatureIDE". In: *Proc. Int'l Working Conf. on Variability Modelling of Software-Intensive Systems (VaMoS)*. Magdeburg, Germany: ACM, Feb. 2020. ISBN: 9781450375016. DOI: 10.1145/3377024.3377036 (see Page 2).

[Keh+21] Timo Kehrer et al. "Bridging the Gap Between Clone-and-Own and Software Product Lines". In: *Proc. Int'l Conf. on Software Engineering (ICSE)*. Piscataway, NJ, USA: IEEE, May 2021, pp. 21–25. ISBN: 978-1-6654-0140-1. DOI: 10.1109/ICSE-NIER52604.2021.00013 (see Page 1).

[Sun+21] Chico Sundermann et al. "Applications of #SAT Solvers on Feature Models". In: *Proc. Int'l Working Conf. on Variability Modelling of Software-Intensive Systems (VaMoS)*. Krems, Austria: ACM, Feb. 2021. ISBN: 9781450388245. DOI: 10.1145/3442391.3442404 (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 Page 2).

[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, 2, 5).

[Nie+22] Michael Nieke et al. "Guiding the Evolution of Product-Line Configurations". In: *Software and Systems Modeling (SoSyM)* 21 (1 Feb. 2022), pp. 225–247. DOI: 10.1007/s10270-021-00906-w (see Page 1).

[Bit+23] Paul Maximilian Bittner et al. "Views on Edits to Variational Software". In: *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. Tokyo, Japan: ACM, Aug. 2023, pp. 141–152. ISBN: 9798400709910. DOI: 10.1145/3579027.3608985 (see Pages 2, 5).

[Bit+24] Paul Maximilian Bittner et al. "Variability-Aware Differencing with DiffDetective". In: *Proc. Int'l Conference on the Foundations of Software Engineering (FSE)*. To appear. New York, NY, USA: ACM, July 2024 (see Page 2).