HUMBOLDT-UNIVERSITÄT ZU BERLIN
MATHEMATISCH-NATURWISSENSCHAFTLICHE FAKULTÄT
INSTITUT FÜR INFORMATIK

Inkrementelle Extraktion von Feature-Mappings

Bachelorarbeit

zur Erlangung des akademischen Grades
Bachelor of Science (B. Sc.)

eingereicht von: Angelina Jellinek
geboren am: 01.03.1997
geboren in: Berlin

Gutachter/innen: Prof. Dr. Lars Grunske
Prof. Dr. Timo Kehrer

eingereicht am: .... verteidigt am: ....

Inhaltsverzeichnis

1 Einleitung

Firmen entwickeln oft mehrere verschiedene Varianten einer Software die an unterschiedliche Kundinnen und Kunden angepasst sind. Eine gute Möglichkeit für die Entwicklung und das Management vieler Varianten ist die Verwendung von Softwareproduktlinien. Eine Softwareproduktlinie ist eine Reihe von Softwarevarianten mit gemeinsamen Features (Merkmalen), die sich auf bestimmte Weise kombinieren lassen. Hierzu wird möglichst vorher festgelegt, welche Varianten es später geben wird, welche Features diese haben können und welche Abhängigkeiten zwischen den Features existieren. Es gibt aber mehrere Probleme, weswegen die Verwendung von Softwareproduktlinien in der Praxis oft nicht umgesetzt wird [1]. Häufig weiß man beispielsweise am Anfang noch nicht, welche Varianten später mal existieren sollen. Deshalb erfordert der Entwicklungsprozess viel Arbeit und Zeit, bevor man die erste Variante fertigstellen kann. Aus diesem Grund ist es oft so, dass statt der Verwendung von Softwareproduktlinien einfach eine bestehende Variante der Software kopiert und angepasst wird [2]. Diese Herangehensweise ist als Clone-and-Own bekannt. Clone-and-Own erhöht allerdings den Wartungsaufwand der Varianten exponentiell mit jeder neuen Variante, denn leider speichert man dabei nicht, welches Code-Stück zu welchem Feature gehört. Das wiederum bedeutet, dass man Änderungen eines Features manuell in jeder Variante vornehmen muss. Dadurch kann die Wartung von Software extrem aufwändig und kostenintensiv werden. Zudem stellt der redundante Code auch ein hohes Risiko für Fehler dar [3].

Um die Probleme wachsender Clone-and-Own-Projekte zu lösen, arbeiten verschiedene Forschungsgruppen daran, Feature-Mappings automatisch zu ermitteln. Basierend auf den Feature-Mappings ist es beispielsweise möglich, Varianten in eine Softwareproduktlinie zu überführen und somit die Wartbarkeit deutlich zu erhöhen. Ein Werkzeug um Feature-Mappings automatisch zu ermitteln ist ECCO [4]. Hierfür benötigt man lediglich bestehende Varianten und zugehörige Informationen über die darin enthaltenen Features. Michelon et al. [5] zeigten, dass ECCO auf der ArgoUML-Softwareproduktlinie für zehn oder mehr Varianten sehr gute Ergebnisse erzielen konnte.

In der Praxis kann man jedoch nicht davon ausgehen, dass genügend Varianten existieren, um daraus nutzbare Feature-Mappings zu ermitteln. Aus diesem Grund entstanden Ansätze, wie man Feature-Mappings bereits während des Bearbeitungsprozesses der Varianten ableiten kann. Das VariantSync-Projekt [1] bietet die Möglichkeit mehrere Varianten einer Software mit Hilfe von Feature-Mappings auf Codefragmente zu verwalten, was einen Kompromiss zwischen Softwareproduktlinien und Clone-and-Own darstellt. Außerdem lassen sich Feature-Mappings durch Feature-Trace-Recording [2] aus Annotationen während des Entwicklungsprozesses ableiten und über Varianten hinweg weitergeben. Dadurch kann der manuelle Aufwand des Annotierens reduziert werden [2].

In dieser Arbeit möchten wir einen neuen Ansatz vorstellen, um die Überführung eines

---

[1] https://github.com/tthuem/VariantSync

[2] https://github.com/VariantSync/FeatureTraceRecording

Clone-and-Own-Projekts in eine Softwareproduktlinie zu erleichtern. Wir werden den Algorithmus von ECCO so erweitern, dass man bereits vorhandene Feature-Mappings mit eingeben kann, die man beispielsweise im Bearbeitungsprozess mit VariantSync erstellt hat. Wir werden untersuchen, welchen Einfluss diese initialen Feature-Mappings auf die Experimente von Michelon et al. [5] mit der ArgoUML-Softwareproduktlinie haben.

Wir werden sehen, dass der erweiterte Algorithmus von ECCO in der Lage ist, mit steigender Anzahl an initialen Feature-Mappings die berechneten Ergebnisse zu verbessern. Insbesondere konnten wir durch initiale Feature-Mappings in Szenarien mit geringer Anzahl an Varianten einen deutlichen Anstieg von Precision, Recall und F1-Score verzeichnen.

Zusammenfassend werden wir folgende Beiträge leisten:

- Wir präsentieren den um die Möglichkeit der Eingabe von initialen Feature-Mappings erweiterten Algorithmus von ECCO.
- Wir untersuchen das Potential der Eingabe initialer Feature-Mappings auf die Ergebnisse von ECCO in Abhängigkeit von der Anzahl der eingegebenen Varianten.

2 Theoretischer Hintergrund

In diesem Abschnitt möchten wir das nötige Hintergrundwissen erläutern, welches für diese Arbeit relevant ist. Außerdem werden wir einen Einblick in die verwendeten Tools und das Entwicklungsprojekt geben.

2.1 Verfahren zur Entwicklung mehrerer Varianten einer Software

2.1.1 Softwareproduktlinien

Eine Softwareproduktlinie beschreibt eine Reihe von Softwarevarianten, welche gemeinsame Merkmale (Features) haben, die den bestimmten Anforderungen einer Marktrichtung oder Aufgabe entsprechen. Die Varianten setzen sich auf vorgeschriebene Weise aus den gemeinsamen Funktionen zusammen und werden aus einer gemeinsamen Code-Basis generiert [6].

Unter einem Feature verstehen wir eine für den Nutzer oder die Nutzerin sichtbare Charakteristik einer Software [7]. Die logische Verknüpfung von mehreren Features wird als Feature-Interaktion bezeichnet [4]. Ein Feature-Trace identifiziert die Artefakte (beispielsweise Codestellen) einer Software, die ein bestimmtes Feature oder eine Feature-Interaktion implementieren [8]. Das Mapping oder Feature-Mapping eines Artefakts bezeichnet den Term, der angibt, welches Features oder welche Feature-Interaktion durch dieses Artefakt implementiert wird. [7]. Bei der *Presence Condition* eines Artefakts handelt es sich um einen booleschen Ausdruck der Features, welcher den Wert `true` annimmt, falls das Artefakt für die entsprechende Auswahl an Features in der Softwarevariante vorhanden ist [9]. Die Presence Condition eines Softwareartefakts wird im Folgenden als Feature-Mapping angegeben. Falls A und B Features sind, so könnte ein Feature-Mapping beispielsweise so aussehen: A & B.

Zum besseren Verständnis möchten wir das Prinzip von Softwareproduktlinien anhand des simplen Beispiels von Legofiguren erklären. Legofiguren sind die menschlichen Miniatur-Figuren aus Kunststoff des Spielzeugherstellers *The LEGO Group*. Sie können sehr unterschiedlich aussehen und haben doch gemeinsame Eigenschaften anhand derer wir sie als zusammengehörig erkennen. Wir betrachten zuerst das Feature-Modell in Abbildung 1. Aus solch einem Feature-Modell lässt sich ablesen, welche Features es gibt und welche Feature-Interaktionen zulässig sind. Die hier im Beispiel dargestellten Features sind:

- Körper
- Haare
- Helm
- Visier
- Werkzeug

Wir sehen, dass ein Körper für eine Legofigur erforderlich ist, während ein Werkzeug ein optionales Feature darstellt. Außerdem bekommen wir Informationen über die erlaubten Feature-Interaktionen, damit wir als Ergebnis eine Legofigur erhalten. In diesem

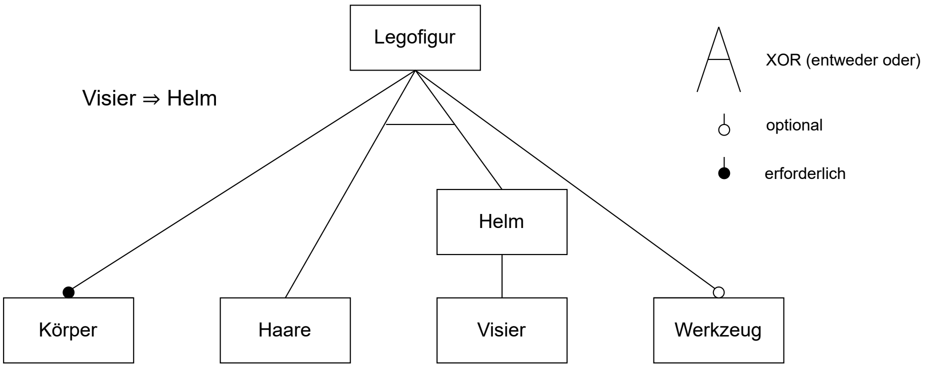

Abbildung 1: Feature-Modell für Legofiguren

Fall erkennen wir beispielsweise, dass nur entweder Haare oder ein Helm vorhanden sein dürfen.

Die Verwendung von Softwareproduktlinien ist besonders geeignet für die Entwicklung einer großen Zahl von Varianten, weil die Entwicklung in diesem Fall deutlich günstiger werden kann, als wenn jedes Produkt einzeln entworfen wird [10]. Dies verbessert außerdem die Qualität der Codestücke, da diese in verschiedenen Konstellationen getestet werden. Allerdings sind die Möglichkeiten der Variation eingeschränkt und die Verwendung von Softwareproduktlinien verursacht einen größeren verwaltungstechnischen Aufwand, woraus sich erst spät ein positiver Langzeiteffekt ergibt [3].

Dies führt dazu, dass Softwareproduktlinien nicht immer einsetzbar sind und stattdessen Clone-and-Own genutzt wird.

2.1.2 Clone-and-Own

Als Clone-and-Own bezeichnet man in der Softwareentwicklung das Kopieren eines vorhandenen Stücks Code aus einem bestehenden Programm, welches anschließend in ein anderes Programm eingefügt wird [3]. Neue Varianten einer Software entstehen also durch Wiederverwendung einzelner Teile aus bereits existierenden Varianten bis hin zu ganzen Varianten [11].

Vorteile hiervon sind die Einfachheit und der geringe zeitliche Aufwand bei der Erstellung. Außerdem kann man problemlos Änderungen vornehmen, ohne die bestehende Variante zu verändern. Allerdings ist die Verwendung von Clone-and-Own auch sehr kostenintensiv, was die nachträgliche Bearbeitung der Softwarevarianten angeht, weil es keine Dokumentation darüber gibt, welches Stück Code wo wiederzufinden ist. Dies kann die Wartung sehr teuer und komplex werden lassen, da Änderungen dann in allen betroffenen Varianten umzusetzen sind [2, 3].

Besonders für eine große Menge an Varianten einer Software ist die Verwendung von Clone-and-Own also schwierig [4].

Wir betrachten nun ein kleines Beispiel-Clone-and-Own-Projekt. Nachfolgend sehen wir die vereinfachte Darstellung von sieben Varianten einer Software mit einer unterschiedlichen Auswahl an Features von `A` bis `C` bzw. Konjunktionen dieser Features. Die

I_1 // A
I_5 // A

Variante 1: A

I_2 // B
I_4 // B | C
I_6 // B

Variante 2: B

I_4 // B | C
I_7 // C
I_9 // C

Variante 3: C

I_1 // A
I_2 // B
I_3 // A & B
I_4 // B | C
I_5 // A
I_6 // B

Variante 4: A & B

I_1 // A
I_4 // B | C
I_5 // A
I_7 // C
I_8 // A & C
I_9 // C

Variante 5: A & C

I_2 // B
I_4 // B | C
I_6 // B
I_7 // C
I_9 // C
I_{10} // B & C

Variante 6: B & C

I_1 // A
I_2 // B
I_3 // A & B
I_4 // B | C
I_5 // A
I_6 // B
I_7 // C
I_8 // A & C
I_9 // C
I_{10} // B & C

Variante 7: A & B & C

Abbildung 2: Beispiel-Clone-and-Own-Projekt

Softwareartefakte (Codeteile, die jeweils ein bestimmtes Feature-Mapping implementieren) sind beispielhaft mit I_1 bis I_{10} dargestellt und mit Kommentaren annotiert. Dieses Projekt könnte so entstanden sein, dass Variante 1 mit Feature A entworfen wurde. Nun wurde von unterschiedlichen Stellen zusätzlich ein Feature B und C angefordert. Es wurden also Variante 4 und 5 generiert. Anschließend gab es keinen Bedarf an Feature A mehr und es entstanden Variante 2 und 3. Daraufhin war die Kombination beider Features gefragt und Variante 6 wurde entwickelt. Schließlich sollte Feature A doch wieder integriert werden und Variante 7 entstand. Es ist erkennbar, dass viele Teile Code für die Entwicklung der neuen Varianten wiederverwendet werden konnten. Dies geschah, indem eine vorherige Variante kopiert wurde und die Grundlage für die Entwicklung einer neuen Variante bildete.

2.1.3 Umsetzung in der Praxis

Für die Entwicklung vieler Varianten eignen sich Softwareproduktlinien in der Theorie also besser als Clone-and-Own-Projekte. In der Praxis findet die Entwicklung der Varianten aber oft nicht so gut dokumentiert statt, da dieser Prozess viel Arbeit und Zeit kosten würde [2]. Aus diesem Grund werden sehr viele Projekte mit dem Clone-and-Own-Verfahren entwickelt. Dies ist dann problematisch, wenn man die Varianten beispielsweise nachträglich erweitern möchte [4]. Manchmal kann es vorkommen, dass die Varianten eines Projekts um ein neues Feature ergänzt werden sollen. Hierfür werden dann Informationen über die bisherigen Features der Varianten benötigt und wie diese kombiniert werden können.
Im nächsten Abschnitt werden wir uns ECCO anschauen, welches versucht die Informationen über die Features einzelner Codestücke automatisch herzuleiten und das Problem fehlender Dokumentation dieser Informationen zu lösen.

2.2 ECCO (Extraction and Composition for Clone-and-Own)

ECCO [3] ist ein Tool, welches automatisch Feature-Mappings aus bestehenden Softwarevarianten und zugehörigen Informationen, in welcher Softwarevariante welche Features vorkommen, ableiten kann. Es findet Feature-Traces und berechnet ihre Zusammenhänge [4]. Michelon et al. [5] konnten zeigen, dass ECCO für zehn oder mehr Softwarevarianten in der Lage ist, fast alle Feature-Mappings richtig zu bestimmen. Die Ergebnisse zeigen jedoch auch, dass die Korrektheit und Vollständigkeit der Feature-Mappings bei Eingabe von weniger als zehn Softwarevarianten deutlich schlechter wird. Wir wollen nun die grundlegenden Begriffe und die Funktionsweise von ECCO erklären, um anschließend den Algorithmus um initiale Feature-Mappings zu erweitern und die Auswirkungen auf weniger als zehn Softwarevarianten zu untersuchen.

2.2.1 Grundlagen

ECCO wird mit einer Menge an existierenden Softwarevarianten und den zugehörigen Informationen, welche Features darin enthalten sind, aufgerufen. Die grundlegenden Begriffe übernehmen wir von Linsbauer et al. [4]. Diese bezeichnen die Menge aller in den Softwarevarianten enthaltenen Features als $\mathbb{F}$. Unter einer Softwarevariante P verstehen sie die darin enthaltenen Features, bezeichnet mit $P.Features \subseteq \mathbb{F}$, und den zugehörigen AT (*artifact tree*), bezeichnet mit $P.AT$. Der AT enthält die Softwareartefakte (*implementation artifacts*) der Softwarevariante als Baumstruktur. Die Softwareartefakte I für eine Softwarevariante P entsprechen den Artefakten, die die Features implementieren, welche in P enthalten sind. Die Softwareartefakte müssen nicht zwingend nur aus Quellcode bestehen, sondern können auch beispielsweise Modelle beinhalten. Im weiteren Verlauf betrachten wir allerdings aus Verständnisgründen den Fall, dass wir nur Quellcode vorliegen haben, da wir für unser Beispielprojekt auch nur

---

[3] https://github.com/jku-isse/ecco

diesen benötigen. Andere Softwareartefakte können jedoch analog verwendet werden. Außerdem führen Linsbauer et al. [4] den Begriff Modul (*module*) ein, um zwischen einfachen Features und Feature-Interaktionen unterscheiden zu können. Ein Modul ist eine Menge von Features, die entweder positiv (nicht-negiert) oder negativ (negiert) auftreten können. Als Basismodule (*base modules*) bezeichnen sie Module, die genau ein positives und kein negatives Feature beinhalten. Das Basismodul $f = \{F\}$ beschreibt das Softwareartefakt, welches das Feature $F \in \mathbb{F}$ implementiert. Es erhält also die kleingeschriebene Bezeichnung des Features. Ein Softwareartefakt, welches mit dem Basismodul a bezeichnet wird, muss also in allen Softwarevarianten vorkommen, welche in irgendeiner Weise das Feature A beinhalten. Hierbei ist es egal, ob noch weitere Features oder Feature-Interaktionen vorkommen. Als Ableitungsmodul (*derivative modules*) werden Module bezeichnet, die mindestens ein positives und beliebig viele negative Features haben. Ein Ableitungsmodul $\delta^n(F, f1, f2, ..., fn) = \{F, f1, f2, ..., fn\}$ bezeichnet ein Softwareartefakt, welches die Interaktion von $n + 1$ Features beinhaltet. $F \in \mathbb{F}$ ist hierbei ein positives Feature und fi das positive Feature Fi oder das negative Feature $\neg Fi \; \forall i \in \{1, 2, ..., n\}$. Dieses n wird als Grad des Ableitungsmoduls bezeichnet. Linsbauer et al. [4] betrachten ein Ableitungsmodul als Menge mit Kardinalität $n + 1$, welche alle (positiven und negativen) Features beinhaltet, die an der Feature-Interaktion beteiligt sind. Ein Ableitungsmodul mit Grad $n = 0$ ist ein Basismodul.

Außerdem wollen wir noch ein paar Funktionen zum Arbeiten mit Features und Modulen von Linsbauer et al. [4] einführen. Die Namen der Funktionen wurden von uns aus ihrem Papier übernommen.

Negate Features nF: Generiert eine Menge $\overline{F}$, welche alle Features F einer Menge in negiertem Zustand enthält.

$$nF(F) = \{\neg f \mid f \in F\} \qquad (= \overline{F}) \tag{1}$$

Als Ergebnis erhalten wir eine Menge von Features.

Compute Modules from Features f2m: Berechnet die Menge der Module aus den positiven Features F und den negativen Features $\overline{F}$.

$$f2m(F, \overline{F}) = \{p \cup n \mid p \in 2^F \setminus \emptyset \land n \in 2^{\overline{F}}\} \tag{2}$$

Die Module bilden sich aus der paarweisen Vereinigung von jeweils einem Element aus der Produktmenge der positiven Features ohne leere Menge ($p \in 2^F \setminus \emptyset$) und einem Element aus der Produktmenge der negativen Features ($n \in 2^{\overline{F}}$). Man erhält also eine Menge von Modulen.

Update Modules with Features uM: Aktualisiert eine Menge von Modulen M mit einer Menge an bisher unbekannten Features $\overline{F}$.

$$uM(M, \overline{F}) = \{m \cup n \mid m \in M \wedge n \in 2^{\overline{F}}\} \tag{3}$$

Die Module werden paarweise mit den Elementen aus der Produktmenge der Features vereinigt. Das Ergebnis ist wieder eine Menge von Modulen.

2.2.2 Trace-Extraction-Algorithmus

Der Trace-Extraction-Algorithmus von ECCO basiert auf fünf Regeln [4]. Seien A und B zwei Softwarevarianten. Dann befolgt der Algorithmus folgende Regeln:

1. Gemeinsame Softwareartefakte bilden *mindestens* Feature-Traces zu gemeinsamen Modulen.
2. Softwareartefakte die in A und nicht B sind, bilden *mindestens* Feature-Traces zu Modulen, die in A und nicht B sind. Und umgekehrt.
3. Softwareartefakte die in A und nicht B sind, bilden *auf keinen Fall* Feature-Traces zu Modulen, die in B und nicht A sind. Und umgekehrt.
4. Softwareartefakte die in A und nicht B sind, bilden *höchstens* Feature-Traces zu Modulen, die in A sind. Und umgekehrt.

Höchstens bedeutet in diesem Fall, dass keine Feature-Traces zu Modulen in nicht A gebildet werden.

5. Softwareartefakte die in A und B sind, bilden *höchstens* Feature-Traces zu Modulen, die in A oder B sind. Und umgekehrt.

Um die Informationen zu verwalten, arbeitet ECCO mit Assoziationen [4]. Eine Assoziation $a \in A$ ist eine Relation (M, AT), wobei M (auch $a.M$) ein vier-Tupel aus Modul-Mengen $M = (Min, All, Max, Not)$ ist und AT (auch $a.AT$) der Artefaktbaum (artifact tree) für die zugehörigen Softwareartefakte. Die Modul-Mengen enthalten die folgenden Informationen.

$a.M.Min$	ist die Menge an Modulen, zu denen die Softwareartefakte (im AT) *mindestens* Feature-Traces bilden.
$a.M.All$	ist die Menge an Modulen mit denen die Softwareartefakte jemals assoziiert wurden.
$a.M.Max$	ist die Menge an Modulen, zu denen die Softwareartefakte *höchstens* Feature-Traces bilden.
$a.M.Not$	ist die Menge an Modulen, zu denen die Softwareartefakte *auf keinen Fall* Feature-Traces bilden.

Um anschließend die Feature-Mappings zu bestimmen, betrachten wir meistens nur die Min-Mengen [4]. Sollte die Min-Menge leer sein, so wird die Max-Menge verwendet. Um die Max-Menge zu bestimmen, benötigen wir die *All*- und die Not-Menge. Aus diesem Grund speichern wir uns zu jedem Artefaktbaum alle vier Modul-Mengen als Assoziationen. In den Trace-Extraction-Algorithmus wird jeweils eine Softwarevariante p und eine Menge von aus vorherigen Iterationen bekannten Assoziationen A eingegeben. Beim Aufruf mit der ersten Softwarevariante ist die Menge der Assoziationen noch leer. Jede Softwarevariante enthält die Information, welche Features in dieser implementiert sind.

2.2.3 Beispiel

Wir wollen nun zum besseren Verständnis der Regeln und Struktur der Assoziationen das Beispiel-Clone-and-Own-Projekt aus Abschnitt 2.1.2 betrachten. Wir geben nacheinander die Varianten (ohne initiale Feature-Mappings) in den Trace-Extraction-Algorithmus ein und betrachten die berechneten Assoziationen. Am Ende werden wir sehen, welche Feature-Mappings abgeleitet werden können.
Wir wollen die Informationen der Assoziationen der Übersichtlichkeit halber in einer Tabelle veranschaulichen.

Eingabe in 1. Aufruf (P_1)

$\mathtt{I}_1$
$\mathtt{I}_5$

Abbildung 3: Variante P_1 mit Feature A

ID	Artefakte in AT	M.Min	M.All	M.Max	M.Not

Tabelle 1: (Noch keine) Assoziationen zu Beginn

In Abbildung 3 sehen wir die Softwarevariante P_1, welche zusammen mit einer leeren Menge an Assoziationen (siehe Tabelle 1) in den Trace-Extraction-Algorithmus von ECCO eingegeben wird. Wir haben noch keine initialen Informationen über die Feature-Mappings. Da wir wissen, dass die Softwarevariante das Feature A implementiert, legen wir eine neue Assoziation

$$a_{new} = ((\{A\}, \{A\}, \{A\}, \emptyset), P_1.AT)$$

an. Zur Min-, *All*- und Max-Menge fügen wir die Module hinzu, die mit Hilfe der $f2m$-Funktion aus dem enthaltenen Feature A gebildet werden können (also weiterhin nur {A}). Da wir bisher keine Module ausschließen können, bleibt die Not-Menge vorerst leer. Diese Assoziation wird nun mit ID a_1 zur Gesamtmenge der Assoziationen hinzugefügt (siehe Tabelle 2). Beim Aufruf mit der ersten Softwarevariante können wir noch nicht mehr als diese eine Assoziation über die Feature-Mappings erfahren. Würden wir nun hieraus die Feature-Mappings der Softwareartefakte ableiten, so könnten wir sehen, dass alle Artefakte das Feature-Mapping A bekommen würden (siehe Min-Menge). In Abbildung 4 sehen wir eine Illustration von a_1. Dargestellt ist der Artefaktbaum von P_1 mit den aus der Min-Menge von a_1 ableitbaren Feature-Mappings für die Softwareartefakte I_1 und I_5.

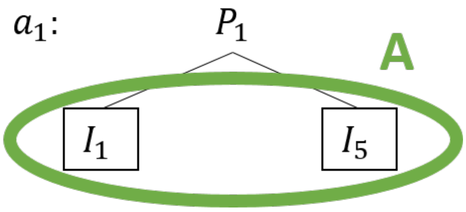

Abbildung 4: Bedeutung von a_1; I_1 und I_5 erhalten das Feature-Mapping A (hier in grün dargestellt)

Die berechnete Gesamtmenge der Assoziationen (hier vorerst nur a_1) wird nun gemeinsam mit der nächsten Variante erneut in den Trace-Extraction-Algorithmus eingegeben.

Eingabe in 2. Aufruf (P_2)

$\mathtt{I}_2$
$\mathtt{I}_4$
$\mathtt{I}_6$

Abbildung 5: Variante P_2 mit Feature B

ID	Artefakte in AT	M.Min	M.All	M.Max	M.Not
a_1	I_1, I_5	{A}	{A}	{A}	$\emptyset$

Tabelle 2: Assoziationen nach Durchlauf mit P_1

In Abbildung 5 und Tabelle 2 sehen wir nun die zweite Eingabe in den Trace-Extraction-Algorithmus von ECCO. Wir geben die Softwarevariante P_2 und die aus dem vorherigen Durchlauf berechneten Assoziationen ein. Wir verfahren anfangs ähnlich wie bei P_1 und legen eine neue Assoziation

$$
\begin{array}{rl}
a_{new} = ((& \{\{B, \neg A\}, \{B\}\}, \\
& \{\{B, \neg A\}, \{B\}\}, \\
& \{\{B, \neg A\}, \{B\}\}, \\
& \emptyset \qquad\qquad\qquad), P_2.AT)
\end{array}
$$

für P_2 an. Wir können den Modulen hier bereits die Potenzmenge des Features $\neg A$, also $\{\neg A, \emptyset\}$ elementweise hinzufügen, da wir wissen, dass das Feature A in P_2 nicht implementiert ist. Da die Varianten P_1 und P_2 keinen gemeinsamen Code vorliegen haben, können wir die Module in Assoziation a_1 mit der uM-Funktion um das elementweise hinzufügen der Potenzmenge des Features ¬B aktualisieren. Es ergibt sich

$$\begin{aligned} a_1 = ((&\{\{A, \neg B\}, \{A\}\}, \\ &\{\{A, \neg B\}, \{A\}\}, \\ &\{\{A, \neg B\}, \{A\}\}, \\ &\emptyset \qquad\qquad\qquad\qquad), P_1.AT). \end{aligned}$$

Wir berechnen außerdem eine Assoziation

$$\begin{aligned} a_{int} = ((\ &a_1.M.Min \cap a_{new}.M.Min, \\ &a_1.M.All \cup a_{new}.M.All, \\ &\emptyset, \\ &a_1.M.Not \cap a_{new}.M.Not \qquad\qquad),\ a_1.AT \cap a_{new}.AT\ \ ) \\ \\ &a_{int}.M.Max = a_{int}.M.All \backslash a_{int}.M.Not \end{aligned}$$

für die Schnittmenge der aktualisierten Assoziation a_1 und a_{new}. Hier sehen wir die Anwendung der Regel 1 – **Gemeinsame Softwareartefakte bilden *mindestens* Feature-Traces zu gemeinsamen Modulen**. Betrachten wir die Min-Menge und die Berechnung des Artefaktbaums, so sehen wir, dass wir nur die gemeinsamen Module beziehungsweise Softwareartefakte in a_{int} behalten. Im Fall der entstandenen Assoziation

$$\begin{aligned} a_{int} = ((&\emptyset, \\ &\{\{B, \neg A\}, \{B\}, \{A, \neg B\}, \{A\}\}, \\ &\{\{B, \neg A\}, \{B\}, \{A, \neg B\}, \{A\}\}, \\ &\emptyset \qquad\qquad\qquad\qquad\qquad\qquad), P_1.AT \cap P_2.AT) \end{aligned}$$

gibt es allerdings keine gemeinsamen Softwareartefakte. In der All- und Not-Menge vereinigen wir alle Informationen aus a_1 und a_{new}. Daraus berechnen wir dann die Max-Menge. Nach der Berechnung der Schnittmenge ziehen wir a_{int} nacheinander von a_{new} und a_1 ab.

$$\begin{aligned} a_1 = ((\ & a_1.M.Min \backslash a_{int}.M.Min, \\ & a_1.M.All, \\ & \emptyset, \\ & a_1.M.Not \cup a_{new}.M.All \qquad),\ a_1.AT \backslash a_{int}.AT\ \ ) \end{aligned}$$

$$a_1.M.Max = a_1.M.All \backslash a_1.M.Not$$

$$\begin{aligned} \Rightarrow a_1 = \ & ((\{\{A, \neg B\}, \{A\}\}, \\ & \{\{A, \neg B\}, \{A\}\}, \\ & \{\{A, \neg B\}, \{A\}\}, \\ & \{\{B, \neg A\}, \{B\}\} \qquad), P_1.AT) \end{aligned}$$

$$\begin{aligned} a_{new} = ((\ & a_{new}.M.Min \backslash a_{int}.M.Min, \\ & a_{new}.M.All, \\ & \emptyset, \\ & a_{new}.M.Not \cup a_1.M.All \qquad),\ a_{new}.AT \backslash a_{int}.AT\ \ ) \end{aligned}$$

$$a_{new}.M.Max = a_{new}.M.All \backslash a_{new}.M.Not$$

$$\begin{aligned} \Rightarrow a_{new} = \ & ((\{\{B, \neg A\}, \{B\}\}, \\ & \{\{B, \neg A\}, \{B\}\}, \\ & \{\{B, \neg A\}, \{B\}\}, \\ & \{\{A, \neg B\}, \{A\}\} \qquad), P_2.AT) \end{aligned}$$

Hier wird also Regel 2 – **Softwareartefakte die in A und nicht B sind, bilden *mindestens* Feature-Traces zu Modulen, die in A und nicht B sind. Und umgekehrt.** – und Regel 3 – **Softwareartefakte die in A und nicht B sind, bilden *auf keinen Fall* Feature-Traces zu Modulen, die in B und nicht A sind. Und umgekehrt.** – und Regel 4 – **Softwareartefakte die in A und nicht B sind, bilden *höchstens* Feature-Traces zu Modulen, die in A sind. Und umgekehrt.** – verwendet. Die Regel 2 finden wir wieder in der Berechnung der Min-Menge und dem Artefaktbaum. Die Anwendung von Regel 3 sehen wir in der Berechnung der Not-Mengen. Hier werden die All-Mengen der jeweiligen anderen Assoziation zur Not-Menge hinzugefügt, weil diese keine gemeinsamen Softwareartefakte (mehr) haben und diese somit ausgeschlossen werden können. Für Regel 4 schauen wir uns die Max-Mengen an. Wir können sehen, dass in der Max-Menge von a_1 nur Module vorhanden sind, die A enthalten (kein nicht A), und in der Max-Menge von a_{new} nur Module, die

B enthalten.
Die Max-Mengen wurden wieder aus den All- und Not-Mengen berechnet. Nun fügen wir a_1 und a_{int} (mit ID a_2) zur Gesamtmenge der Assoziationen hinzu. Da wir keine weiteren Assoziationen in der Eingabe bekommen haben, fügen wir auch a_{new} mit ID a_3 hinzu. Hätten wir noch weitere Assoziationen eingegeben bekommen, dann würden wir diese nun aktualisieren, dann die Schnittmenge mit a_{new} berechnen und dieses a_{int} erneut von den Assoziationen abziehen.
In Abbildung 6 sehen wir die Veranschaulichung der Feature-Mappings von P_1 und P_2. Die Informationen können wir aus den Min-Mengen von a_1 und a_3 entnehmen. Falls man mehrere Module in der Min-Menge zur Auswahl haben sollte, entscheidet man sich für die kleinsten und ver$oder$t diese. In unserem Fall wählen wir also die Basismodule. Bei Ableitungsmodulen bestimmt man die entsprechenden Feature-Mappings durch Verundung der enthaltenen Features. Analog verfährt man mit der Max-Menge, falls die Min-Menge einmal leer sein sollte.

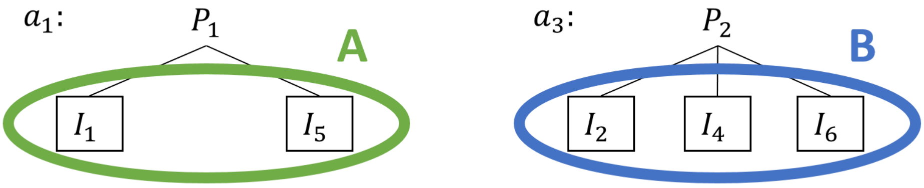

Abbildung 6: Bedeutung von a_1 und a_3; I_1 und I_5 erhalten das Feature-Mapping A und nicht B, I_2, I_4 und I_6 erhalten das Feature-Mapping B und nicht A

Wir konnten nun Regel 1 bis 4 nachvollziehen. Um die Anwendung von Regel 5 zu sehen, geben wir als nächste Variante P_4 in den Algorithmus ein. Wir rufen den Trace-Extraction-Algorithmus von ECCO wieder gemeinsam mit den eben berechneten Assoziationen auf. In Abbildung 7 sehen wir die Variante P_4 und in Tabelle 3 sind die bisher berechneten Assoziationen dargestellt.

Eingabe in 3. Aufruf (P_4)

$\mathtt{I}_1$
$\mathtt{I}_2$
$\mathtt{I}_3$
$\mathtt{I}_4$
$\mathtt{I}_5$
$\mathtt{I}_6$

Abbildung 7: Variante P_4 mit Feature A & B

ID	Artefakte in AT	M.Min	M.All	M.Max	M.Not
a_1	I_1, I_5	$\{A, \neg B\}, \{A\}$	$\{A, \neg B\}, \{A\}$	$\{A, \neg B\}, \{A\}$	$\{B, \neg A\}, \{B\}$
a_2	$\emptyset$	$\emptyset$	$\{A, \neg B\}, \{A\}, \{B, \neg A\}, \{B\}$	$\{A, \neg B\}, \{A\}, \{B, \neg A\}, \{B\}$	$\emptyset$
a_3	I_2, I_4, I_6	$\{B, \neg A\}, \{B\}$	$\{B, \neg A\}, \{B\}$	$\{B, \neg A\}, \{B\}$	$\{A, \neg B\}, \{A\}$

Tabelle 3: Assoziationen nach Durchlauf mit P_1 und P_2

Genau wie bei Softwarevariante P_1 und P_2 bestimmen wir als erstes eine neue Assoziation

$$
\begin{aligned}
a_{new} = ((&\{\{A, B\}, \{A\}, \{B\}\},\\
&\{\{A, B\}, \{A\}, \{B\}\},\\
&\{\{A, B\}, \{A\}, \{B\}\},\\
&\emptyset \qquad\qquad\qquad\qquad), P_4.AT)
\end{aligned}
$$

für die neue Variante P_4. Da wir dieses Mal mehrere Assoziationen eingegeben bekommen haben, betrachten wir diese nun nacheinander. Die Betrachtung von a_2 können wir vernachlässigen, da der AT dieser Assoziation keine Softwareartefakte enthält und wir somit keine Informationen über die Feature-Mappings daraus gewinnen können. Wir beginnen mit Assoziation a_1. Da keine neuen Features in der Softwarevariante P_4 auftauchen, müssen wir die eingegebenen Assoziationen nicht updaten. Wir berechnen also die Überschneidung von a_1 mit a_{new}. Es ergibt sich die neue Assoziation

$$
\begin{aligned}
a_{int} = ((&\{A\},\\
&\{\{A, \neg B\}, \{A, B\}, \{A\}, \{B\}\},\\
&\{\{A, \neg B\}, \{A, B\}, \{A\}\},\\
&\{\{B\}, \{B, \neg A\}\} \qquad\qquad\qquad), P_1 \cap P_4.AT).
\end{aligned}
$$

Bei dieser Berechnung sehen wir Regel 5 – **Softwareartefakte die in A und B sind, bilden *höchstens* Feature-Traces zu Modulen, die in A oder B sind. Und umgekehrt**. Die Softwareartefakte in unserem Beispiel sind nun A und A & B. Betrachten wir die Min-Menge, so sehen wir, dass wir das Feature-Mapping A ableiten würden.
Um mit dem Trace-Extraction-Algorithmus fortzufahren, würden wir a_{int} nun wieder von a_1 und a_{new} abziehen und anschließend a_3 betrachten. Die entstehenden Assoziationen werden wieder zur Gesamtmenge der berechneten Assoziationen hinzugefügt und können anschließend mit weiteren Softwarevarianten aufgerufen werden.

In Abbildung 8 sehen wir die berechneten Feature-Mappings nach vollständiger Verarbeitung der drei eingegebenen Varianten P_1, P_2 und P_4. Die Zuordnung ist für die meisten Softwareartefakte korrekt. ECCO kann allerdings nicht bestimmen, dass das Software-

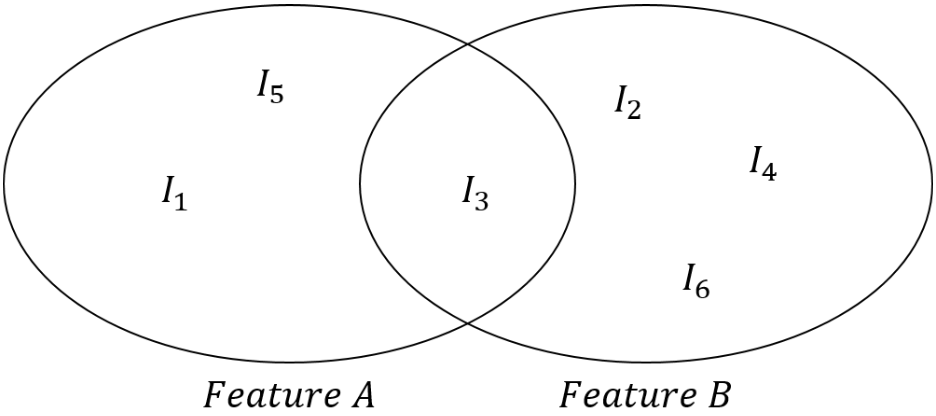

Abbildung 8: Zuordnung der Features A und B zu den implementierenden Softwareartefakten der Softwarevarianten P_1, P_2 und P_4

artefakt I_4 auch mit Feature C zusammenhängt (siehe korrekte Feature-Annotationen in Abbildung 2), da Feature C nicht in den betrachteten Varianten auftauchte.
Ein weiteres Problem des Algorithmus von ECCO ergibt sich durch die Wahl der kleinsten Module für die Bestimmung der Feature-Mappings. Auf diese Weise können Informationen verloren gehen. In diesem Beispiel konnten wir diesen Fehler nicht beobachten, jedoch könnte er in komplexeren Szenarien vorkommen.

Durch initiale Feature-Mappings könnten wir auch solche Informationen über Features erhalten, die nicht in den betrachteten Softwarevarianten implementiert sind bzw. wo der bisherige Algorithmus von ECCO noch fehleranfällig ist. Partielle initiale Feature-Mappings könnten die Informationen teilweise liefern und wir möchten untersuchen, wie die Eingabe initialer Feature-Mappings die Ergebnisse in verschiedenen Szenarien beeinflussen kann.

3 Methodik

In Abschnitt 2.1 haben wir gesehen, welche Probleme es beim Generieren und Verwalten von mehreren Varianten einer Software gibt, wenn der Entwicklungsprozess nicht ausreichend dokumentiert wird. Anschließend haben wir ECCO vorgestellt, welches uns erlaubt, die Informationen über die Features einzelner Codestücke automatisch herzuleiten. Michelon et al. [5] zeigten, dass ECCO auf der ArgoUML-Softwareproduktlinie für zehn oder mehr Varianten sehr gute Ergebnisse erzielen konnte.
In der Praxis kann man jedoch nicht davon ausgehen, dass genügend Varianten existieren um nutzbare Feature-Mappings mit ECCO zu ermitteln. Aus diesem Grund möchten wir in dieser Arbeit den Algorithmus von ECCO so erweitern, dass man bereits vorhandene Feature-Mappings mit eingeben kann, die man beispielsweise im Bearbeitungsprozess mit VariantSync erstellt hat oder einer unvollständigen Dokumentation der Softwarevarianten entnehmen kann.

3.1 Ziele und Forschungsfragen

In dieser Arbeit werden wir die Veränderungen im ECCO-Algorithmus erläutern. Die konkrete Implementierung des erweiterten Algorithmus [4] soll nicht Teil dieser Arbeit sein, wird aber in Abschnitt 3.2 beschrieben. Wir werden anschließend eine empirische Studie zum Vergleich des vorherigen Algorithmus und dem erweiterten Algorithmus mit der ArgoUML-SPL durchführen. Wir vermuten, dass es hierdurch möglich ist, auch bei Eingabe von wenigen Varianten gute Ergebnisse zu erzielen. Diese Vermutung möchten wir mit Hilfe folgender Forschungsfragen (RQ – Research Questions) untersuchen:

RQ1: Können wir die Ergebnisse von ECCO replizieren?
Für die Beantwortung dieser Frage werden wir dieselben Experimente, die Michelon et al. [5] mit dem ECCO-Algorithmus durchgeführt haben, mit dem erweiterten Algorithmus von ECCO und einer Eingabe von 0 % initialen Feature-Mappings wiederholen. Da wir den Algorithmus komplett neu implementieren werden, erwarten wir etwas abweichende Ergebnisse zu denen von Michelon et al. [5], welche auf Implementierungs- und Auswertungsunterschiede zurückzuführen sind.

RQ2: Welchen Effekt hat die Anzahl der Varianten bei fester Prozentzahl von eingegebenen Feature-Mappings?
Für dieses Experiment wählen wir jeweils 30 Mal eine Eingabe von 25 %, 50 % und 75 % initialen Feature-Mappings für die Variantenanzahlen 3, 6, 9, 12 und 15. Die Feature-Mappings werden jedes Mal neu zufällig ausgewählt. Das bedeutet, dass für gleiche Prozentangabe an Feature-Mappings und Varianten trotzdem 30 Mal andere Teile des Codes mit Feature-Mappings versehen werden. Wir wollen beobachten, ob und wie sich die Ergebnisse mit steigender Variantenanzahl verändern.

---

[4] https://gitlab.informatik.hu-berlin.de/mse/VariantSync/automatedfeaturelocation/ecco-light

RQ3: Welchen Effekt hat die Prozentzahl von eingegebenen Feature-Mappings bei fester Anzahl an Varianten?
Zu diesem Zweck wählen wir ein beliebiges Szenario mit 3 Varianten und erhöhen die Prozentzahl der Eingabe der initialen Feature-Mappings von 0 % bis auf 100 % in 10 %-Schritten. Für jede Prozentzahl werden 30 Mal neu die eingegebenen Feature-Mappings zufällig ausgewählt. Das bedeutet, dass wieder jedes Mal andere Teile des Codes mit Feature-Mappings versehen werden. Damit wollten wir untersuchen, wie sich die Prozentzahl der eingegebenen Feature-Mappings auf die Ergebnisse auswirkt.

RQ4: Welchen Einfluss hat die Auswahl der Varianten?
Hierfür werden wir dieselben Experimente wie bei RQ2 und RQ3 nochmal ausführen, allerdings dieses Mal nicht die Auswahl der entsprechenden Prozentzahl mit Feature-Mappings versehener Knoten, sondern die Auswahl der Varianten variieren. Dadurch lässt sich feststellen, ob die Auswahl spezieller Varianten einen Einfluss auf die Ergebnisse haben kann.

RQ5: Wie verhält sich die Laufzeit abhängig von der Prozentzahl eingegebener Feature-Mappings?
Um diese Frage zu beantworten, wählen wir ein beliebiges Szenario mit 3 Varianten und erhöhen die Prozentzahl der Eingabe der initialen Feature-Mappings von 0 % bis auf 100 % in 10 %-Schritten. Für jede Prozentzahl messen wir die Laufzeiten der Experimente und evaluieren, wie sich die Laufzeit abhängig von der Prozentzahl eingegebener Feature-Mappings verändert. Dies betrachten wir sowohl für 3 festgehaltene Varianten mit variierter Auswahl initialer Feature-Mappings, als auch für eine variierte Auswahl von 3 Varianten.

3.2 Erweiterter Algorithmus

Als Nächstes wollen wir uns anschauen, wie der erweiterte Algorithmus von ECCO implementiert ist und funktioniert. Die Überlegungen zur Implementierung und die Java-Implementierung selbst sind eine Gemeinschaftsarbeit von Alexander Schultheiß, Luisa Gerlach und mir und sollen nicht Teil dieser Arbeit sein.

In Abschnitt 2.2.2 haben wir uns die fünf Regeln angeschaut, auf denen der Trace-Extraction-Algorithmus von ECCO basiert. Dieselben Regeln sollten auch für den erweiterten Algorithmus gelten. Linsbauer et al. [4] haben in ihrem Papier den Pseudocode für ihren Trace-Extraction-Algorithmus angegeben. Auf der folgenden Seite sehen wir den erweiterten Algorithmus mit gelben Markierungen an den Stellen, die für unsere Zwecke verändert wurden (siehe Algorithmus 1).
Es ist zu beachten, dass bei der Berechnung der Not-Menge der Überschneidung (a_{int}; hier Zeile 16) ein Tippfehler bei dem Pseudocode von Linsbauer et al. [4] zu finden ist. Dieser wurde hier korrigiert. Dabei handelt es sich also nicht um eine Veränderung des Algorithmus.

Algorithm 1 Erweiterter Trace-Extraction-Algorithmus

```
Input: MainTree mainTree, Product[] products

A = $\emptyset$; //Set of Associations
$F_{all}$ = $\emptyset$;
for p in products do
  mainTree.unite(p); //merge p into MainTree
  M = f2m(p.Features, nF($F_{all}$\p.Features));
  $F_{neg}$ = nF(p.Features\$F_{all}$);
  $a_{new}$ = ((M,M,M, $\emptyset$), p.AT);
  $A_{new}$ = $\emptyset$;
  $F_{all}$ = $F_{all}$ $\cup$ p.Features;
  for a in A do
    a = ((uM(a.M.Min, $F_{neg}$),  uM(a.M.All, $F_{neg}$),  uM(a.M.Max, $F_{neg}$),
          uM(a.M.Not, $F_{neg}$)),  a.AT);

    $a_{int}$ = ((a.M.Min $\cap$ $a_{new}$.M.Min, a.M.All $\cup$ $a_{new}$.M.All , $\emptyset$,
            a.M.Not $\cap$ $a_{new}$.M.Not), a.AT $\cap$ $a_{new}$.AT );
    $a_{int}$ .M.Max = $a_{int}$ .M.All \$a_{int}$ .M.Not;
    a = ((a.M.Min \$a_{int}$ .M.Min, a.M.All , $\emptyset$,
          a.M.Not $\cup$ $a_{new}$.M.All), a.AT \$a_{int}$ .AT );
    a.M.Max = a.M.All \a.M.Not ;
    $a_{new}$ = (($a_{new}$.M.Min \$a_{int}$ .M.Min, $a_{new}$.M.All, $\emptyset$,
            $a_{new}$.M.Not $\cup$ a.M.All), $a_{new}$.AT \$a_{int}$.AT );
    $a_{new}$.M.Max = $a_{new}$.M.All \$a_{new}$.M.Not ;
    $A_{new}$ = $A_{new}$ $\cup$ {$a_{int}$, a};
  end for
  A = $A_{new}$ $\cup$ {$a_{new}$};
end for

return A
```

Der bisherige Algorithmus von ECCO funktioniert so, dass eine Softwarevariante `p` und eine Menge von aus vorherigen Iterationen bekannten Assoziationen `A` eingegeben wird [4]. Im erweiterten Trace-Extraction-Algorithmus geben wir einen leeren `MainTree` und alle Softwarevarianten ein. Ein MainTree ist eine Baumstruktur, die die Knoten aus allen eingegebenen Produkten enthält, ähnlich einem 150 %-Modell [12]. Zusätzlich existieren auch Knoten für die Verzeichnisstruktur der Produkte. In den Knoten finden sich unter anderem Informationen über die Position innerhalb der Produkte (bei Knoten, die in mehreren Produkten vorkommen, auch mehrere Positionen).

Wenn man den erweiterten Algorithmus genauer betrachtet, dann zeigt sich, dass sich der bisherige Algorithmus von ECCO fast identisch innerhalb der `for`-Schleife

in Zeile 4 des erweiterten Algorithmus wiederfindet. Die Menge der Assoziationen `A` erstellen wir vor dieser Schleife und füllen sie sukzessiv. Da wir durch die Produkte iterieren, betrachten wir auch jedes Mal nur eine Softwarevariante. Unser Input des ursprünglichen Algorithmus wird also auch in die Schleife in Zeile 4 des erweiterten Algorithmus eingegeben.
Außerdem wird die Menge $\texttt{F}_{\texttt{all}}$ anders berechnet, enthält aber für die Berechnung von `M` und $\texttt{F}_{\texttt{neg}}$ dieselben Informationen, nämlich jeweils alle Features, die in den zuvor bearbeiteten Softwarevarianten schon gesehen wurden.
Zusätzlich gibt es im erweiterten Algorithmus einen MainTree. Anfangs ist dieser leer. Innerhalb der Schleife über die Produkte fügen wir diese nacheinander in den MainTree ein. Dadurch können wir die Position eines Knotens direkt im MainTree bestimmen und brauchen keine *doAlignmentAndSequencing()*-Funktion mehr, welche im ursprünglichen Algorithmus von ECCO Verwendung fand, um die Sequenznummern der Knoten (Artifact nodes) so neu zu ordnen, dass die neuen Knoten ebenfalls Sequenznummern erhielten und gleichzeitig möglichst viele alte Knoten ihre Sequenznummer behalten konnten.
Innerhalb der `for`-Schleife über die Assoziationen `a` in `A` in Zeile 11 des erweiterten Algorithmus aktualisieren wir die Module mit neuen Features und die Assoziationen nach den Regeln 1 bis 5, die in Abschnitt 2.2.2 vorgestellt wurden. Anschließend werden $\texttt{a}_{\texttt{int}}$ und `a` zu der neuen Menge der Assoziationen hinzugefügt.
Nachdem durch alle Assoziationen iteriert wurde, wird auch $\texttt{a}_{\texttt{new}}$ und die Menge aller neuen Assoziationen $\texttt{A}_{\texttt{new}}$ zur Gesamtmenge aller Assoziationen hinzugefügt und eine weitere Softwarevariante könnte betrachtet werden.

Die Eingabe der initialen Feature-Mappings kann problemlos über schon initial mit Feature-Mappings versehene Knoten der Softwarevarianten erfolgen.

3.3 ArgoUML-SPL

Wir wollen uns nun die ArgoUML-SPL [5] anschauen, mit welcher wir unsere Experimente durchführen wollen. Die ArgoUML-SPL ist die Softwareproduktlinie für das Open-Source-UML-Modellierungstool ArgoUML [13]. Sie hat acht optionale Features und insgesamt 256 verschiedene Varianten. Sie stellt außerdem eine Ground-Truth (korrekte Annotation von Features) zur Verfügung [5].
Ein paar charakteristische Eigenschaften der ArgoUML-SPL sind [5]:

- ArgoUML ist ein großes, reales Softwaresystem.
- Es gibt Feature-Interaktionen und -Negierungen.
- Die Feinheit der Softwareartefakte variiert von ganzen Klassen bis zu einzelnen Statements in Methoden.

Michelon et al. [5] haben das Feature-Lokalisierungstool ECCO mit der ArgoUML-SPL Challenge [14] getestet. Die Challenge enthält 15 Szenarien mit jeweils unterschiedlicher

---

[5] https://github.com/marcusvnac/argouml-spl

Szenario	**Größe**	**Beschreibung**
Original	1	Originale ArgoUML-Variante mit allen Features
Traditional	10	Varianten mit keinem, allen, und der Kombination aus je 7 optionalen Features
PairWise	9	Varianten mit allen paarweisen Featurekombinationen
2-10 Random	2-10	Zufällig ausgewählte Teilmenge der Varianten
50 Random	50	Zufällig ausgewählte Teilmenge der Varianten
100 Random	100	Zufällig ausgewählte Teilmenge der Varianten
All	256	Alle möglichen Varianten der ArgoUML-SPL

Tabelle 4: ArgoUML Challenge Szenarien [5]

Anzahl an Varianten aus der Softwareproduktlinie. In Tabelle 4 ist zu sehen, welche und wie viele Varianten die 15 Szenarien jeweils enthalten. Das *Original*-Szenario enthält das initiale ArgoUML System als eine Variante, welche alle Features inkludiert hat [5]. Das *Traditional*-Szenario hat 10 Varianten, eine mit allen optionalen Features, eine mit keinem optionalen Feature und für jedes optionale Feature eine Variante bei der nur dieses Feature nicht vorhanden ist. Das *PairWise*-Szenario enthält Varianten mit Kombinationen aus jeweils zwei Features. Die *Random*-Szenarien der verschiedenen Größe enthalten die jeweilige Anzahl an zufällig ausgewählten Varianten und das *All*-Szenario enthält alle 256 Varianten.

Die Ground-Truth besteht aus 24 Traces [5]. Als Trace wird die Zuordnung einer logischen Formel von Features und einer Menge an Softwareartefakten (hier Java-Codefragmente) bezeichnet. Die Ground-Truth besteht aus einem Trace für jedes einzelne Feature, zwei Traces mit jeweils einem negierten einzelnen Feature, 13 Traces mit Konjunktionen aus jeweils zwei Features und ein Trace mit einer Konjunktion aus drei Features.

3.4 Auswertungsmethodik

In diesem Abschnitt wollen wir uns die Auswertungsumgebung anschauen. Hier kann eingestellt werden, wie der erweiterte Algorithmus von ECCO aufgerufen wird. Wir werden sehen, wie die Ground-Truth generiert wird und schließlich die Berechnung der Metriken funktioniert. Hierbei werden wir die Unterschiede zu der Arbeit von Michelon et al. [5] betrachten.

3.4.1 Ground-Truth

Eigentlich stellt die ArgoUML-SPL bereits eine Ground-Truth zur Verfügung, welche auch von Michelon et al. [5] verwendet wurde. Allerdings ist diese Ground-Truth in Text-Dateien angelegt. Innerhalb dieser Dateien sind die Pfade zu den jeweiligen Softwareartefakten angegeben, die zu den Traces der entsprechenden Dateinamen gehören. Hieraus lässt sich allerdings nicht mehr die knotenweise Ground-Truth für unseren MainTree ableiten, weil darin keine Zeilennummern angegeben sind. Aus diesem Grund verwenden wir die #ifdef-Annotationen in den Quellcode-Dateien der ArgoUML-SPL, um eine eigene Ground-Truth zu erhalten. Wir nutzen das Ground-Truth-Extraction-Tool von VEVOS [15], um entsprechend der 24 Traces der ArgoUML-SPL die Varianten zu generieren und anschließend die Annotationen auslesen und speichern zu können. Um das Ground-Truth-Extraction-Tool nutzen zu können, verwenden wir zuerst den ArgoUMLExtractor [6], um die nötigen Informationen zu extrahieren. Durch die erneute Generierung der Varianten mit Hilfe des Tools können wir so außerdem sicherstellen, dass die Varianten zur neuen Ground-Truth passen. Die gespeicherten Annotationen können dann mit Hilfe von VEVOS für einzelne Knoten unseres MainTrees anhand ihrer `ProductPosition` (Information in welcher Softwarevariante und in welchen Zeilennummern das Softwareartefakt zu finden ist) als Ground-Truth geladen und verwendet werden.

3.4.2 Metriken

Zur Validierung berechneten Michelon et al. [5] für jeden Trace der Challenge Benchmark [14] automatisch drei Metriken, die die Ground-Truth-Traces T_{gt} mit den von ECCO berechneten Traces T_{ecco} vergleichen.

Die **Precision** ist der Prozentwert der korrekt ermittelten Feature-Mappings in Relation zu der Gesamtzahl aller ermittelten Feature-Mappings:

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

TP (true positive) ist die Anzahl der korrekt und FP (false positive) die Anzahl der nicht korrekt ermittelten Feature-Mappings [5].
Der **Recall** ist der Prozentwert der korrekt ermittelten Feature-Mappings in Relation zu der Gesamtzahl aller Feature-Mappings in der Ground-Truth:

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

FN (false negative) ist die Anzahl der nicht ermittelten Feature-Mappings, die in der Ground-Truth vorhanden sind [5].
Der **F1-Score** ist das Verhältnis zwischen Precision und Recall: [5]

$$F1 = 2 * \frac{precision * recall}{precision + recall}$$

---

[6] https://github.com/AlexanderSchultheiss/ArgoUMLExtractor

Da wir als Ausgabe des erweiterten Algorithmus von ECCO einen MainTree erhalten, konnten wir diese Art der Auswertung nicht so einfach übernehmen. In Abschnitt 3.4.1 haben wir beschrieben, dass wir die Ground-Truth knotenweise für alle Knoten dieses MainTrees laden können. Wir schauen uns also knotenweise an, wie viele Einträge in den Wahrheitstabellen der beiden Mappings als logische Formeln übereinstimmen. Daraus berechnen wir dann Precision, Recall und F1-Score. Hierfür definieren wir:

- TP := Anzahl der Zeilen, bei denen die entsprechende Belegung in der Zeile für das berechnete Feature-Mapping `true` und für die Ground Truth `true` ergibt,
- TN := Anzahl der Zeilen, bei denen die entsprechende Belegung in der Zeile für das berechnete Feature-Mapping `false` und für die Ground Truth `false` ergibt,
- FP := Anzahl der Zeilen, bei denen die entsprechende Belegung in der Zeile für das berechnete Feature-Mapping `true` und für die Ground Truth `false` ergibt,
- FN := Anzahl der Zeilen, bei denen die entsprechende Belegung in der Zeile für das berechnete Feature-Mapping `false` und für die Ground Truth `true` ergibt.

Zum besseren Verständnis wollen wir hierfür ein kleines Beispiel betrachten. Angenommen wir hätten eine Softwareproduktlinie mit den Features A, B und C. Gehen wir davon aus, dass wir für einen Knoten das Feature-Mapping (FM) ”A und B” berechnet hätten, die Ground-Truth (GT) für diesen Knoten allerdings ”A und C” entspricht. Dann würde die Wahrheitstabelle folgendermaßen aussehen:

A	B	C	FM ($A \wedge B$)	GT ($A \wedge C$)	
`false`	`false`	`false`	`false`	`false`	TN
`false`	`false`	`true`	`false`	`false`	TN
`false`	`true`	`false`	`false`	`false`	TN
`false`	`true`	`true`	`false`	`false`	TN
`true`	`false`	`false`	`false`	`false`	TN
`true`	`false`	`true`	`false`	`true`	FN
`true`	`true`	`false`	`true`	`false`	FP
`true`	`true`	`true`	`true`	`true`	TP

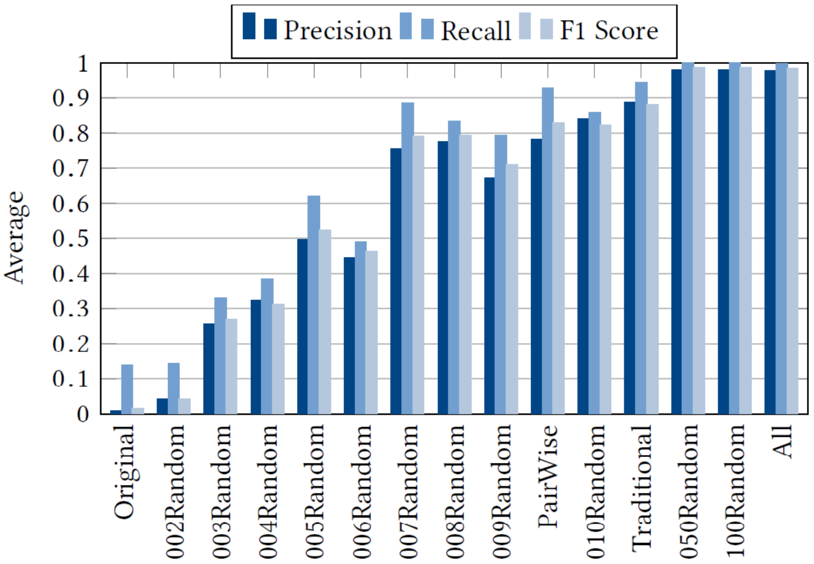

Abbildung 9: Precision, Recall und F1-Score der Szenarios von Michelon et al. [5]

4 Ergebnisse und Diskussion

In diesem Abschnitt wollen wir die Ergebnisse darstellen und diskutieren und schließlich die Forschungsfragen beantworten. Bei den Grafiken ist zu beachten, dass sich auf Grund der Sichtbarkeit die Skalierung der Achsen von Grafik zu Grafik unterscheiden kann.

4.1 RQ1: Können wir die Ergebnisse von ECCO replizieren?

Wir haben in Abschnitt 3.4 gesehen, dass wir eine andere Auswertung verwenden als Michelon et al. [5]. Wir generieren die Varianten und die zugehörige Ground-Truth mit Hilfe des Ground-Truth-Extraction-Tools von VEVOS [15]. Außerdem definieren wir unterschiedliche TP, TN, FP und FN, da wir unsere Ground-Truth knotenweise auswerten anstatt traceweise. Hierdurch kann es zu Unterschieden in den Ergebnissen kommen. Dennoch möchten wir untersuchen, ob wir mit unserer Implementierung und Auswertung zu denselben Erkenntnissen gelangen.

In Abbildung 9 sind die Ergebnisse von Michelon et al. [5] dargestellt. Wir wollen nun ihre Folgerungen daraus mit unseren Ergebnissen abgleichen. Hierfür betrachten wir die Ergebnisse des erweiterten Algorithmus von ECCO mit einer Eingabe von 0 % Feature-Mappings für dieselben Szenarien. Zur Erinnerung, die Szenarien enthalten festgelegte Varianten. Jedes Szenario wird 30 Mal untersucht und die Ergebnisse in Abbildung 10 und 11 dargestellt. In Abbildung 10 ist eine ähnliche Ansicht wie in Abbildung 9 gewählt. In Abbildung 11 sind die zu Abbildung 10 gehörigen Box-Plots zu sehen.

Für den bisherigen Algorithmus wurden die besten Ergebnisse für die größten Szenarien mit einem Recall von 100 % und einer Precision und F1-Score von 99 % erzielt [5].

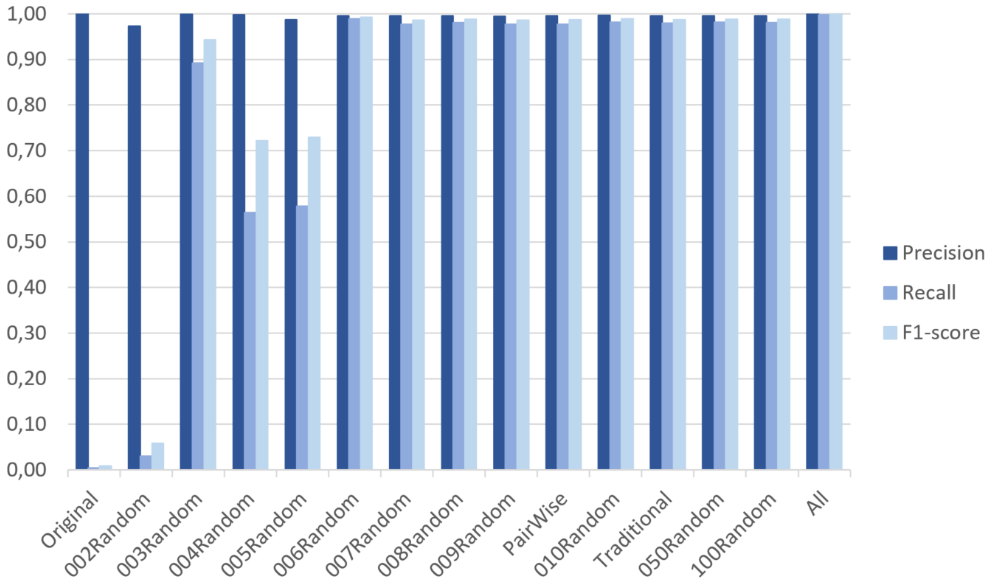

Abbildung 10: Ergebnisse des erweiterten Algorithmus von ECCO für die ArgoUML-SPL Challenge Szenarios mit Eingabe von 0 % Feature-Mappings (arithmetische Mittelwerte)

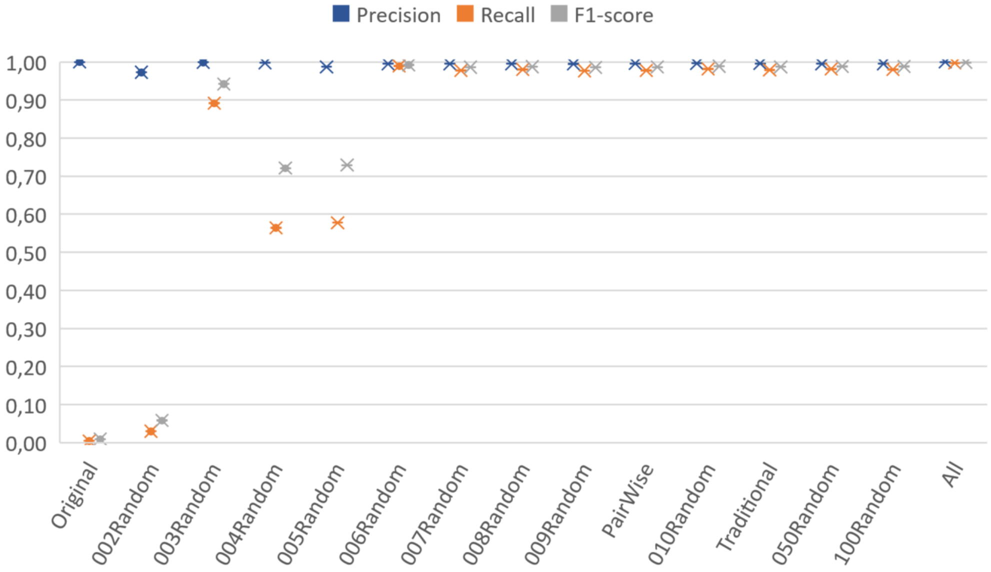

Abbildung 11: Ergebnisse des erweiterten Algorithmus von ECCO für die ArgoUML-SPL Challenge Szenarios mit Eingabe von 0 % Feature-Mappings (Box-Plot)

Beim erweiterten Algorithmus stellen wir ebenfalls fest, dass größere Szenarien zu einer besseren Precision und einem besseren Recall führen. Allerdings verzeichnen wir einen schnelleren Anstieg und erreichen mit dem Recall nur etwas geringere Maximalwerte von durchschnittlich 99,74 % beim All-Szenario [5]. Der niedrigste Recall-Wert wurde jedoch auch nach der Erweiterung des Algorithmus bei dem kleinsten Szenario (Original) verzeichnet [5]. Bei der Precision konnten wir durchgängig Werte zwischen 97,3 % (*002Random*) und 100 % (*Original- und All-Szenario*) verzeichnen. Die Abweichungen in der Precision sind jedoch so gering, dass wir über diese keine Aussage treffen. Wir konnten jedoch bei Recall und F1-Score die Tendenz reproduzieren, dass die Ergebnisse besser wurden je mehr Varianten eingegeben wurden [5].

Genau wie bei dem bisherigen Algorithmus war dies bei dem erweiterten Algorithmus auch nicht generell der Fall, wie man beim Betrachten des *003Random* und *004Random* Szenarios erkennen kann [5].

Wir konnten beim erweiterten Algorithmus nicht beobachten, dass die 10 Traditional-Varianten ein besseres Ergebnis als die 010Random-Varianten erzeugt hätten [5]. Allerdings weisen die Ergebnisse dieser Szenarien beim erweiterten Algorithmus generell nur eine kleine durchschnittliche Differenz auf (siehe Abbildung 10 und 11).

Andererseits können wir beobachten, dass die Precision beim erweiterten Algorithmus durchgängig deutlich höher liegt als beim bisherigen Algorithmus von ECCO [5]. Dies können wir auf die unterschiedliche Art der Auswertung zurückführen. Die Ergebnisse scheinen jedoch plausibel, da wir erwarten, dass durch die Verundung der Features im (erweiterten) ECCO-Algorithmus nur sehr wenige FPs bei unserer Auswertung entstehen. Durch die Verundung müssen alle Features des Feature-Mappings aktiv sein, damit das entsprechende Softwareartefakt in der Variante vorkommt. Es wird also angenommen, dass alle in Frage kommenden Features an dem Softwareartefakt beteiligt sind. Da FPs entstehen, wenn nicht erkannt wird, dass ein Feature an einem Softwareartefakt beteiligt ist, führt dies zu wenigen FPs (und außerdem vielen FNs) und somit zu einer hohen Precision. Hätten wir statt der Verundung eine Veroderung gewählt, so würden wir stattdessen einen hohen Recall erwarten. Durch eine Veroderung müssen nicht alle Features des Feature-Mappings aktiv und damit an dem Softwareartefakt beteiligt sein. Es wären also wenige FNs (und viele FPs) zu erwarten.

In Abbildung 10 und 11 beobachten wir zusätzlich noch besonders gute Ergebnisse bei dem 003Random-Szenario. Schaut man sich die Varianten im 003Random-Szenario an, so fällt auf, dass diese Varianten sehr unterschiedlich sind. Das heißt es gibt wenige Features, die in mehreren Varianten vorkommen. Diese Traces sind also leicht zu erkennen. Außerdem enthalten die Varianten insgesamt auch recht wenige Features (zwei, drei und fünf), sodass große Teile des restlichen Codes sich auch leicht den übrigen Features zuordnen lassen. Aus diesen Gründen können wir beim 003Random-Szenario eine große Anzahl der Feature-Mappings korrekt ermitteln und erhalten diesen auffällig hohen Wert für den Recall.

Zum Schluss schauen wir noch die Box-Plots in Abbildung 11 an. Es fällt auf, dass die einzelnen Messwerte nur sehr gering vom arithmetischen Mittelwert abweichen. Daraus kann man schließen, dass der erweiterte Algorithmus bei mehrfacher Ausführung desselben Experiments jedes Mal nahezu die gleichen Ergebnisse liefert.

Abschließend können wir sagen, dass wir zwar nicht die exakten Ergebnisse von ECCO replizieren konnten, allerdings konnten wir die meisten Erkenntnisse von Michelon et al. [5] ebenfalls beobachten.

4.2 RQ2: Welchen Effekt hat die Anzahl der Varianten bei fester Prozentzahl von eingegebenen Feature-Mappings?

Um diesen Effekt zu untersuchen, wählen wir einige Werte für die Eingabe von Feature-Mappings aus und untersuchen eine unterschiedliche Variantenanzahl. Wir haben uns für eine Eingabe von 25 %, 50 % und 75 % Feature-Mappings entschieden, da die Grenzfälle bei dieser Forschungsfrage uninteressant sind. Für 0 % haben wir die Frage schon in RQ1 beantwortet und für 100 % erhalten wir unabhängig von der Anzahl der eingegebenen Varianten auch 100 % korrekte Ergebnisse. Wir wollen eine Anzahl von 3, 6, 9, 12 und 15 Varianten betrachten, weil sich die Ergebnisse vor allem in den geringen Szenariogrößen stark verbessern, wie auch in Abbildung 10 und 11 zu sehen ist. Außerdem ist der Vergleich mit dem ursprünglichen Algorithmus von ECCO vor allem für geringe Szenariogrößen am interessantesten. In Abbildung 12, 13 und 14 sehen wir die Ergebnisse des erweiterten ECCO-Algorithmus für die gewählten Experimente. Es wurden in allen Experimenten jeweils dieselben 3, 6, 9, 12, beziehungsweise 15 Varianten verwendet, die zuvor einmal zufällig ausgewählt wurden.

Wir können in allen drei Grafiken sehen, dass der Recall und F1-Score für mehr Varianten immer besser werden. Wir erkennen also eine Tendenz, dass mehr Varianten zu besseren Ergebnissen führen. Die Werte werden allerdings auch von der Auswahl der Varianten beeinflusst (siehe RQ4), sodass diese Tendenz nicht der Allgemeinheit entspricht.

Wir erreichen wieder durchgehend eine sehr hohe Precision zwischen 99,8 % und 100 %. Über die Precision lassen sich also keine Aussagen treffen, die nicht durch kleine Schwankungen der Ergebnisse durch die jeweilige Auswahl der entsprechenden Prozentzahl an Feature-Mappings verzerrt werden könnten.

Wir konnten beobachten, dass eine größere Anzahl an eingegebenen Varianten tendenziell dazu führt, dass der erweiterte Algorithmus von ECCO mehr Feature-Mappings korrekt bestimmen kann.

4.3 RQ3: Welchen Effekt hat die Prozentzahl von eingegebenen Feature-Mappings bei fester Anzahl an Varianten?

In Abbildung 15 sehen wir die Ergebnisse des erweiterten Algorithmus von ECCO für drei Varianten bei einer Eingabe von Feature-Mappings von 0 % bis 100 % in 10 %-Schritten. Es wurden bei allen Durchführungen dieselben Varianten verwendet. Wir können beobachten, dass der Recall und F1-Score mit zunehmender Prozentzahl der eingegebenen Feature-Mappings ebenfalls steigt. Bei der Precision kann man in den Rohdaten denselben Effekt beobachten. Dieser ist jedoch in der Abbildung nicht gut zu erkennen, da die Werte sehr dicht beieinanderliegen.

Wir konnten zeigen, dass eine höhere Prozentzahl von eingegebenen Feature-Mappings bei fester Variantenzahl zu einer Verbesserung der Ergebnisse führt.

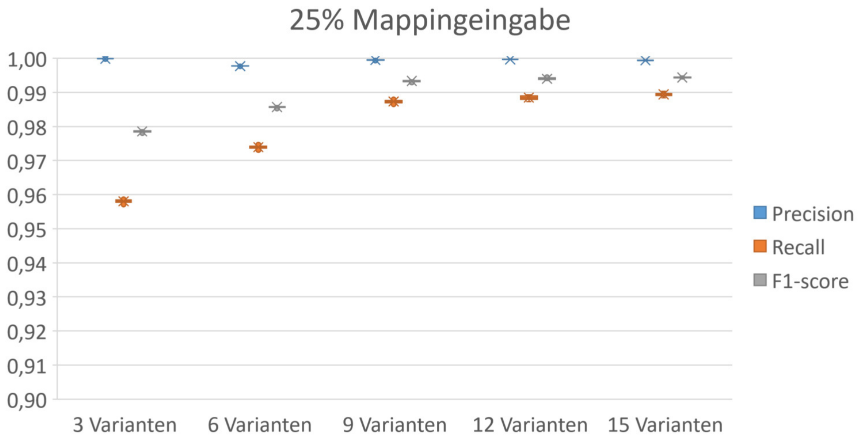

Abbildung 12: Ergebnisse des erweiterten Algorithmus von ECCO für eine Eingabe von 25 % Feature-Mappings mit 3, 6, 9, 12 und 15 Varianten (jedes Mal dieselben) (Box-Plot)

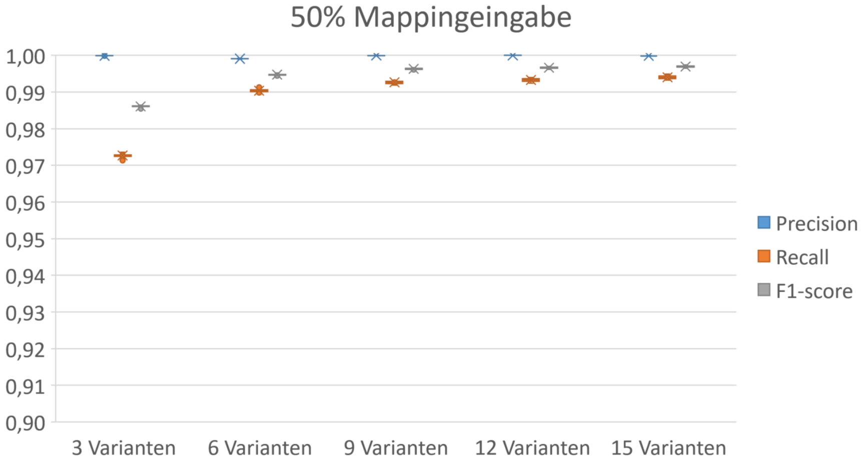

Abbildung 13: Ergebnisse des erweiterten Algorithmus von ECCO für eine Eingabe von 50 % Feature-Mappings mit 3, 6, 9, 12 und 15 Varianten (jedes Mal dieselben) (Box-Plot)

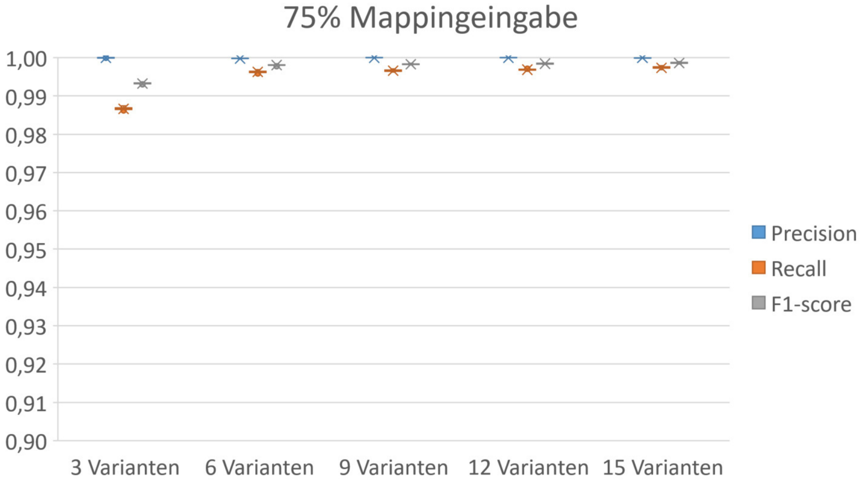

Abbildung 14: Ergebnisse des erweiterten Algorithmus von ECCO für eine Eingabe von 75 % Feature-Mappings mit 3, 6, 9, 12 und 15 Varianten (jedes Mal dieselben) (Box-Plot)

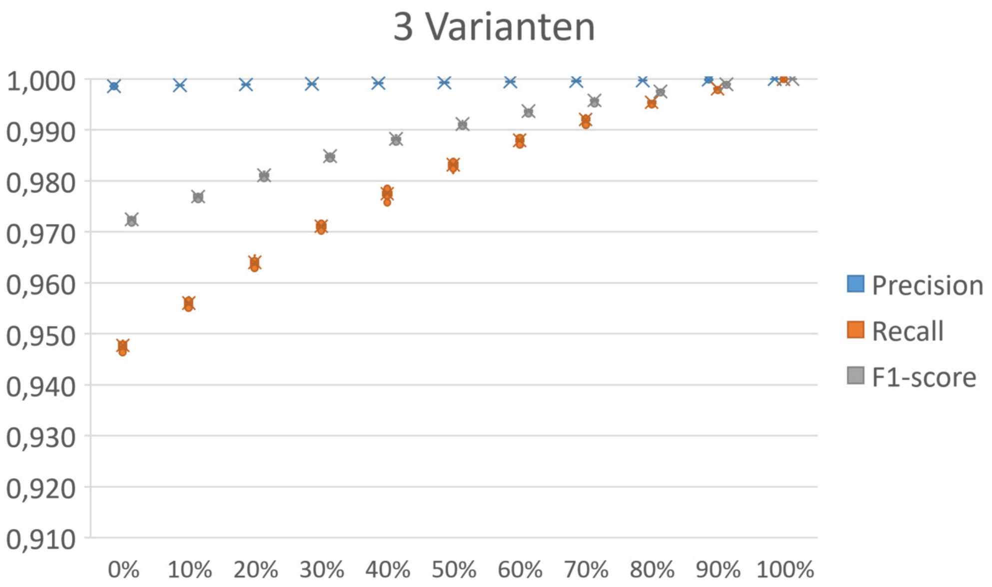

Abbildung 15: Ergebnisse des erweiterten Algorithmus von ECCO für 3 Varianten (jedes Mal dieselben) mit Eingabe von 0 %, 10 %, ..., 100 % Feature-Mappings (Box-Plot)

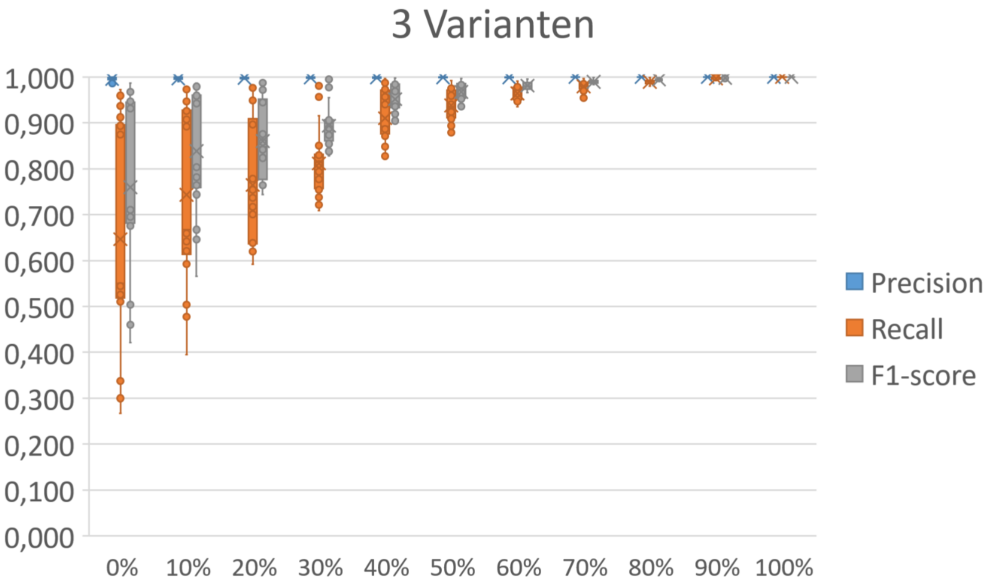

Abbildung 16: Ergebnisse des erweiterten Algorithmus von ECCO für 3 Varianten (jedes Mal andere) mit Eingabe von 0 %, 10 %, ..., 100 % Feature-Mappings (Box-Plot)

4.4 RQ4: Welchen Einfluss hat die Auswahl der Varianten?

Für die Beantwortung dieser Frage führen wir die Experimente aus RQ2 und RQ3 erneut aus, wobei wir dieses Mal die Auswahl der Varianten variieren. In Abbildung 16 sehen wir die Ergebnisse des erweiterten Algorithmus von ECCO für das Experiment

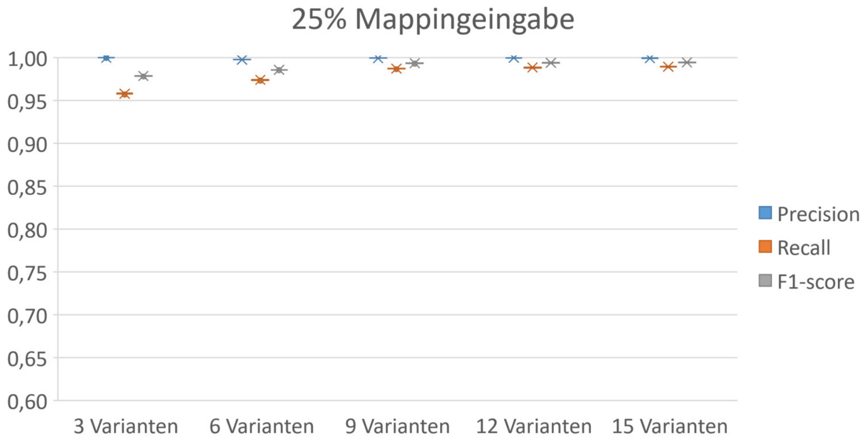

Abbildung 17: Ergebnisse des erweiterten Algorithmus von ECCO für eine Eingabe von 25 % Feature-Mappings mit 3, 6, 9, 12 und 15 Varianten (jedes Mal dieselben) (Box-Plot)

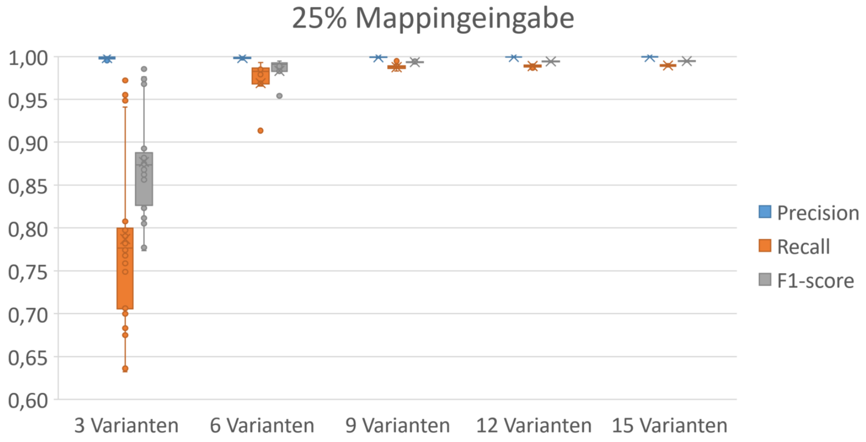

Abbildung 18: Ergebnisse des erweiterten Algorithmus von ECCO für eine Eingabe von 25 % Feature-Mappings mit 3, 6, 9, 12 und 15 Varianten (jedes Mal andere) (Box-Plot)

aus RQ3 mit variierter Auswahl der drei Varianten. Beim Betrachten der Abbildung 16 wird deutlich, dass die Auswahl der Varianten einen großen Einfluss auf die Ergebnisse nimmt. Wir können Spannweiten der Prozentzahlen von 1,43 bei der Precision, von 70,053 beim Recall und von 56,47 beim F1-Score sehen. Vergleicht man Abbildung 16 mit Abbildung 15, so kann man erkennen, dass die Auswahl der Varianten eine deutlich größere Varianz erzeugt, als die variierte Auswahl der jeweiligen Prozentzahl an initialen Feature-Mappings. Man beachte die unterschiedlichen Achsen, um alle Ergebnisse darstellen zu können.

Wir betrachten nun Abbildung 18, 19 und 20. Hier sind die gleichen Experimente wie für RQ2 aber mit variierter Auswahl der Varianten dargestellt. Zu Vergleichszwecken sehen wir in Abbildung 17 dasselbe Experiment wie in Abbildung 12, allerdings mit derselben Achse wie in Abbildung 18. Beim Betrachten dieser beiden Abbildungen und

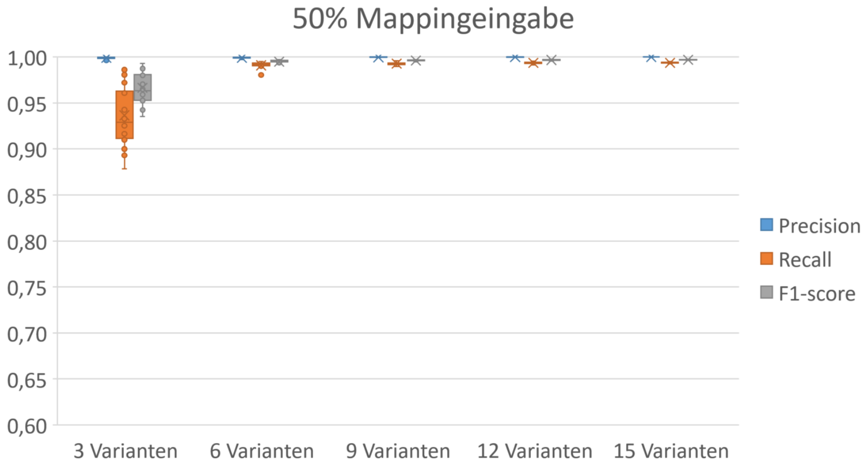

Abbildung 19: Ergebnisse des erweiterten Algorithmus von ECCO für eine Eingabe von 50 % Feature-Mappings mit 3, 6, 9, 12 und 15 Varianten (jedes Mal andere) (Box-Plot)

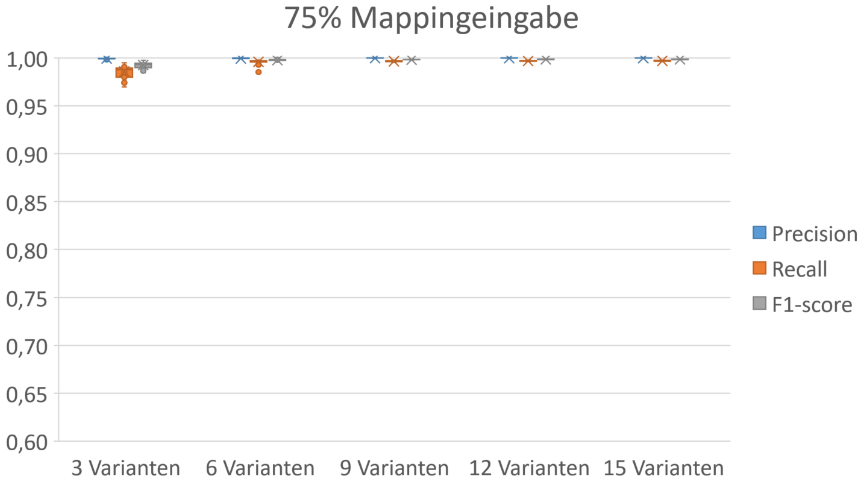

Abbildung 20: Ergebnisse des erweiterten Algorithmus von ECCO für eine Eingabe von 75 % Feature-Mappings mit 3, 6, 9, 12 und 15 Varianten (jedes Mal andere) (Box-Plot)

dem Vergleich von 19 mit 13 und 20 mit 14 können wir erneut die Effekte beobachten, die wir im vorherigen Absatz beschrieben haben.
Wir wollen nun Abbildung 18, 19 und 20 untereinander vergleichen. In diesen ist zu erkennen, dass wir größere Box-Plots für kleinere Variantenanzahlen verzeichnen. Das bedeutet, dass die Varianz der Ergebnisse mit größerer Variantenanzahl abnimmt. Am deutlichsten ist dieser Effekt zwischen 3 und 6 Varianten zu sehen. Dies bestätigt unsere Aussage zu RQ1, dass die besonders guten Ergebnisse beim 003Random-Szenario auf eine spezielle Featureauswahl in den Varianten zurückzuführen sind.
Außerdem beobachten wir größere Box-Plots für eine geringere Prozentzahl an ausgewählten Feature-Mappings. Dies erklärt sich damit, dass die Überschneidung der

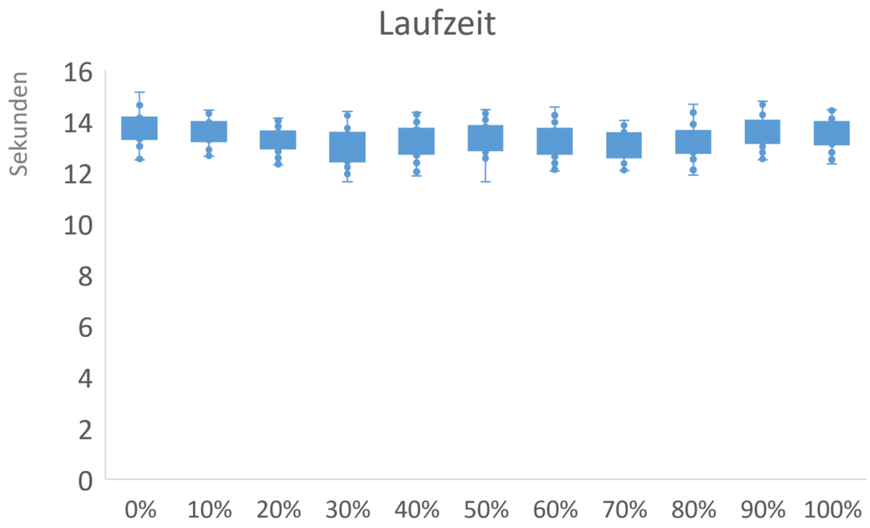

Abbildung 21: Laufzeit des erweiterten Algorithmus von ECCO für 3 Varianten (jedes Mal dieselben) mit Eingabe von 0 %, 10 %, ..., 100 % Feature-Mappings (Box-Plot)

ausgewählten Feature-Mappings mit größerer Prozentzahl immer größer wird. Dadurch werden häufiger dieselben Feature-Mappings eingegeben und die Ergebnisse variieren weniger stark.

Wir können sagen, dass eine größere Prozentzahl an eingegebenen Feature-Mappings und eine höhere Anzahl an ausgewählten Varianten zu stabileren und besseren Ergebnisse führen.

4.5 RQ5: Wie verhält sich die Laufzeit abhängig von der Prozentzahl eingegebener Feature-Mappings?

Um diese Frage zu beantworten, wollen wir die Laufzeit des erweiterten Algorithmus von ECCO für 3 Varianten mit 0 %, 10 %, ..., 100 % aus RQ3 und RQ4 betrachten. In Abbildung 21 ist die Laufzeit des erweiterten ECCO-Algorithmus für das Experiment aus RQ3 zu sehen. In Abbildung 22 sehen wir die Laufzeit für das Experiment aus RQ4. Es ist erkennbar, dass die Laufzeit in beiden Experimenten nicht durch die Prozentzahl der eingegebenen Feature-Mappings beeinflusst wurde. Beim Vergleich von Abbildung 21 und 22 stellen wir außerdem fest, dass die Laufzeit auch nicht von der Auswahl der Varianten abhängt.

Wir konnten zeigen, dass die Laufzeit des erweiterten Algorithmus von ECCO unabhängig von der Prozentzahl der eingegebenen Feature-Mappings und der Anzahl der Varianten ist.

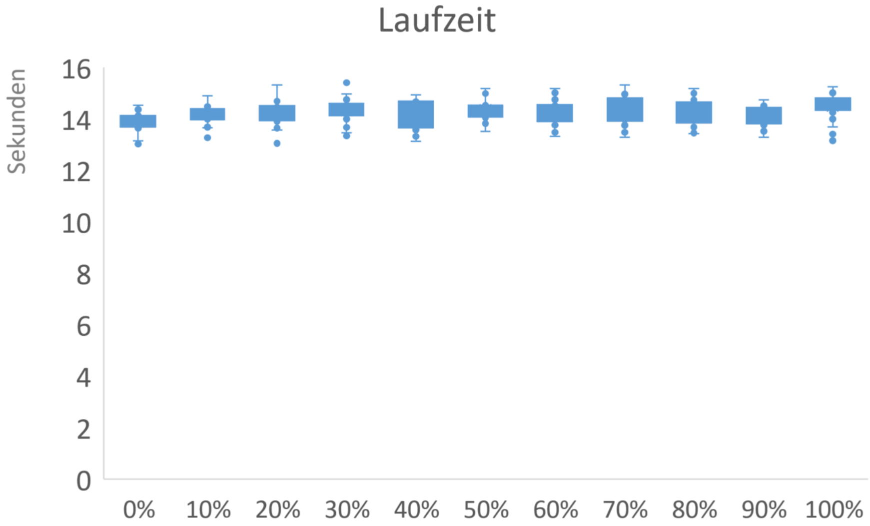

Abbildung 22: Laufzeit des erweiterten Algorithmus von ECCO für 3 Varianten (jedes Mal andere) mit Eingabe von 0 %, 10 %, ..., 100 % Feature-Mappings (Box-Plot)

5 Kritische Punkte

In diesem Abschnitt wollen wir kritische Punkte der Arbeit diskutieren und darlegen, wie wir damit umgegangen sind.

5.1 Konstruktvalidität

Auswertungsmetrik

Bei der Auswertungsmetrik stellte sich die Frage, ob diese auch validiert werden kann. Um diese Frage nach der Validität zu beantworten, wollten wir zeigen, dass eine beliebige Prozentzahl eingegebener Feature-Mappings zu derselben Prozentzahl richtig berechneter Feature-Mappings führt.
In Abbildung 23 sind die Abweichungen von der korrekten Prozentzahl an Feature-Mappings abgebildet. Eingegeben wurde jeweils nur eine Variante, da mehrere Varianten durch das Verwalten der Knoten in einem MainTree die Prozentzahl bereits verbessern würden.
Die Abweichungen sind dadurch zu erklären, dass nur komplette Feature-Mappings eingegeben werden und nicht immer die exakte Prozentzahl an Knoten ausgewählt werden kann. Es kann also vorkommen, dass wir die Anzahl initial mit Feature-Mappings versehener Knoten aufrunden müssen.
Wir sehen in Abbildung 23, dass die Abweichungen sehr gering sind und wir somit zeigen konnten, dass unsere Auswertungsmetrik valide ist.

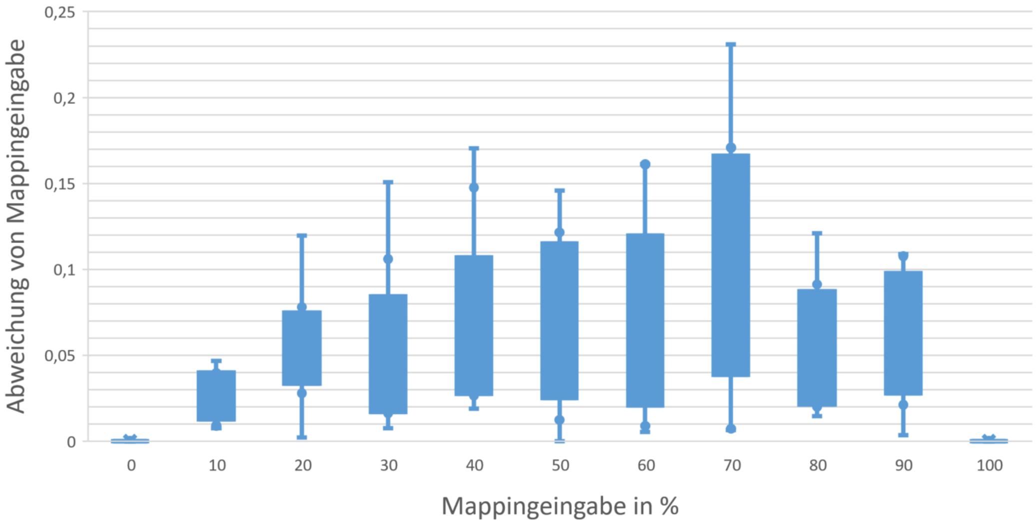

Abbildung 23: Abweichung der Prozentzahl eingegebener Feature-Mappings von der berechneten Prozentzahl korrekter Feature-Mappings

5.2 Interne Validität

VEVOS

In dieser Arbeit nutzen wir das Ground-Truth-Extraction-Tool von VEVOS [15], um entsprechend der 24 Traces der ArgoUML-SPL die Varianten zu generieren und anschließend die Annotationen auslesen und speichern zu können. VEVOS und weitere externe Tools, die es verwendet, könnten Fehler enthalten, die dazu führen, dass die verwendeten Varianten und/oder die Ground-Truth fehlerhaft generiert werden. Außerdem kann die eigene Generierung der Varianten und Ground-Truth zu Unterscheidungen von der ursprünglichen ArgoUML-SPL [13] führen.

(Erweitertes) ECCO

Da wir eine Implementierung des erweiterten Algorithmus von ECCO verwenden, könnten sowohl in der Grundidee von ECCO [4] als auch in der Erweiterung und Implementierung des neuen Algorithmus Fehler zu finden sein. Durch Tests haben wir versucht dieses Risiko zu verringern.

Eigene Implementierung

Potentielle Fehler in der Experimentierumgebung stellen ein weiteres Risiko dar. Um dieses Risiko zu verringern, wurden der Code getestet und die Ergebnisse manuell überprüft.

Auswahl der Varianten bzw. der initialen Feature-Mappings

Die Auswahl der Varianten erfolgt zufällig. Eine (un-)günstige Auswahl könnte die Ergebnisse beeinflussen, wenn beispielsweise für ein Experiment sehr ähnliche Varianten gewählt werden oder eine Variante häufiger in den Experimenten vorkommt. Auch die initialen Feature-Mappings werden zufällig ausgewählt. Auch hier könnte es zu nicht repräsentativen Ergebnissen kommen. Um das Risiko einer schlechten Stichprobe zu minimieren, führen wir alle Experimente in dieser Arbeit jeweils 30 Mal durch. Dies schien ein guter Kompromiss zwischen Risikominimierung und benötigter Rechenzeit zu sein.

5.3 Externe Validität

Ausgewählte SPL

Für die Experimente haben wir die ArgoUML-SPL gewählt. Die Betrachtung nur einer einzigen Softwareproduktlinie gefährdet die externe Validität. Allerdings handelt es sich bei ArgoUML um ein großes, reales Softwaresystem, was die Anwendung des erweiterten Algorithmus von ECCO auf ein tatsächliches Clone-and-Own-Projekt gut repräsentiert und die Probleme eines solchen Projektes, welches nicht als SPL konstruiert wurde, verdeutlicht.

6 Related Work

Diese Bachelorarbeit ist in das Thema Feature-Mapping-Extraction einzuordnen. 2008 veröffentlichten Apel et al. [16] das Papier ”An Overview of Feature-Oriented Software Development” um einen Überblick über die Grundkonzepte der Feature-orientierten Softwareentwicklung zu geben. Sie stellen unterschiedliche Definitionen und Erklärungen der grundlegenden Begriffe wie beispielsweise eines Features dar und vergleichen diese miteinander.
Berger et al. [9] zeigten 2010 in ihrem Papier ”Feature-to-Code Mapping in Two Large Product Lines” das Extrahieren von Feature-to-Code-Mappings aus den Build-Systemen von zwei Betriebssystemen.
In ihrem Papier ”Extracting Software Product Lines: A Case Study Using Conditional Compilation” beschrieben Couto et al. [13] 2011, wie sie die ArgoUML-SPL aus dem Softwaresystem ArgoUML extrahierten. Diese SPL bildet die Grundlage der Experimente in dieser Arbeit.
Das Papier ”Recovering traceability between features and code in product variants” von Linsbauer et al. [17] stellte 2013 erstmals eine Technik zum Ableiten von Feature-Traces zwischen Features und Code in Softwarevarianten dar. Hierfür werden Überschneidungen von Code und Features abgeglichen.
Pfofe et al. [1] präsentierten 2016 VariantSync in ihrem Papier ”Synchronizing Software Variants with VariantSync”. Dabei handelt es sich um ein Eclipse Plug-In, welches durch automatische Synchronisierung von Softwarevarianten einen späteren Übergang eines undokumentierten Clone-and-Own-Projekts zu einer Softwareproduktlinie erleichtern soll. Softwareartefakte wie Quellcode können mit den zugehörigen Feature-Mappings annotiert und die Informationen automatisch an andere Varianten weitergegeben werden.
2021 veröffentlichten Bittner et al. [2] das Papier ”Feature Trace Recording”, in welchem sie eine Möglichkeit vorstellten, um teilautomatisch Feature-Traces während des Bearbeitungsprozesses von mehreren Varianten abzuleiten, obwohl möglicherweise noch Wissen über bereits existierende oder neue Varianten fehlen könnte.
In dem Papier ”Quantifying the Potential to Automate the Synchronization of Variants in Clone-and-Own” 2022 von Schultheiß et al. [18] ist beschrieben, wie Feature-Trace-Recording die Ergebnisse von VariantSync durch Domainwissen verbessern konnte. An dieser Stelle können anschließend die teilweise berechneten Feature-Mappings in den erweiterten Algorithmus von ECCO eingegeben und so Feature-Mappings für weitere Codestellen bestimmt werden.

Die wichtigste Grundlage dieser Bachelorarbeit bildet ECCO, welches automatisch Feature-Mappings aus bestehenden Varianten und zugehörigen Informationen, in welcher Variante welche Features vorkommen, ableiten kann. Vorgestellt wurde ECCO 2016 von Linsbauer et al. [4] im Papier ”Variability extraction and modeling for product variants”, welches in vielen Abschnitten dieser Arbeit herangezogen wird. Bisher war es bei dem Algorithmus von ECCO nicht möglich initiale Informationen mit einzugeben. Aus diesem Grund wollten wir mit dieser Arbeit den ECCO-Algorithmus um die

Eingabe initialer partieller Mappings erweitern.
Außerdem haben wir dieselben Experimente wie in dem Papier ”Comparison-Based Feature Location in ArgoUML Variants” von Michelon et al. [5] 2019 mit dem erweiterten Algorithmus von ECCO durchgeführt. Da wir den Algorithmus komplett neu implementiert haben, erhielten wir trotz einer Eingabe von 0 % Feature-Mappings abweichende Ergebnisse zu denen von Michelon et al. [5]. Dennoch konnten wir betrachten, welche Auswirkungen die Eingabe initialer Informationen auf die Berechnung der Feature-Mappings der ArgoUML Softwareproduktlinie hat.

7 Fazit

In dieser Arbeit haben wir die Veränderungen des erweiterten Algorithmus von ECCO im Vergleich zu dem ursprünglichen Algorithmus von Linsbauer et al. [4] erläutert. Anschließend wurden die Ergebnisse beider Algorithmen mit Hilfe der ArgoUML-SPL verglichen und in einer empirischen Studie die Auswirkungen der Anzahl der eingegebenen Varianten und Prozentzahl initialer Feature-Mappings in den erweiterten Algorithmus untersucht.
Insbesondere wurden dabei folgende Forschungsfragen beantwortet:

- RQ1: Können wir die Ergebnisse von ECCO replizieren?
- RQ2: Welchen Effekt hat die Anzahl der Varianten bei fester Prozentzahl von eingegebenen Feature-Mappings?
- RQ3: Welchen Effekt hat die Prozentzahl von eingegebenen Feature-Mappings bei fester Anzahl an Varianten?
- RQ4: Welchen Einfluss hat die Auswahl der Varianten?
- RQ5: Wie verhält sich die Laufzeit abhängig von der Prozentzahl eingegebener Feature-Mappings?

Abschließend können wir sagen, dass wir mit unseren Experimenten und dem erweiterten Algorithmus im Wesentlichen zu denselben Erkenntnissen gekommen sind wie Michelon et al. [5] mit dem ursprünglichen ECCO-Algorithmus. Wir konnten die Tendenz reproduzieren, dass die Werte für Precision, Recall und F1-Score anstiegen je mehr Varianten eingegeben wurden. Die Werte werden allerdings auch von der Auswahl der Varianten beeinflusst, sodass diese Tendenz nicht der Allgemeinheit entspricht.
Ebenfalls konnten wir durch Steigerung der Prozentzahl eingegebener initialer Feature-Mappings einen Anstieg von Precision, Recall und F1-Score beobachten. Der Effekt war für wenige Varianten besser erkennbar.
Mit dem erweiterten Algorithmus von ECCO konnten wir durchgängig sehr hohe Werte für die Precision feststellen, was zeigt, dass dieser nur sehr wenige FPs (false positives) liefert. Die ermittelten Feature-Mappings sind also zu einem sehr großen Teil korrekt. Besonders gute Werte für den Recall und F1-Score konnten wir bei einer Eingabe von sehr unterschiedlichen Varianten mit generell wenigen Features verzeichnen.
Betrachtet man die Box-Plots der Ergebnisse, so fällt auf, dass wir bei Mehrfachdurchführung desselben Experiments mit denselben Varianten jedes Mal sehr ähnliche Ergebnisse erzielen. Verwenden wir stattdessen unterschiedliche Varianten, so weichen die Ergebnisse stärker voneinander ab. Die Varianz der Werte hat für größere Variantenanzahl abgenommen, war also am besten bei wenigen Varianten zu beobachten.
Wir konnten allerdings keine Abhängigkeit der eingegebenen Prozentzahl initialer Feature-Mappings und der Laufzeit des erweiterten Algorithmus von ECCO feststellen.

Zukünftige Forschung könnte insgesamt davon profitieren initiale Feature-Mappings für die Rekonstruktion fehlender Dokumentation von Softwareprojekten zu verwenden. In dieser Arbeit haben wir gesehen, dass mit der Verwendung initialer Feature-Mappings auf der ArgoUML-SPL eine Verbesserung der Ergebnisse im Vergleich zu keiner Eingabe initialer Feature-Mappings erzielt werden konnte. Falls lückenhafte Informationen vorhanden sind, sollten diese also genutzt werden, um die Qualität der resultierenden Ergebnisse zu verbessern. Auch die Weiterverarbeitung in anderen Tools könnte von den verbesserten Ergebnissen profitieren.

Literatur

[1] Tristan Pfofe, Thomas Thüm, Sandro Schulze, Wolfram Fenske, and Ina Schaefer. Synchronizing software variants with variantsync. In *Proceedings of the 20th International Systems and Software Product Line Conference*, pages 329–332, 2016.

[2] Paul Maximilian Bittner, Alexander Schultheiß, Thomas Thüm, Timo Kehrer, Jeffrey M Young, and Lukas Linsbauer. Feature trace recording. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1007–1020, 2021.

[3] Eddy Ghabach. *Supporting Clone-and-Own in software product line*. PhD thesis, COMUE Université Côte d'Azur (2015-2019), 2018.

[4] Lukas Linsbauer, Roberto Erick Lopez-Herrejon, and Alexander Egyed. Variability extraction and modeling for product variants. In *Proceedings of the 22nd International Systems and Software Product Line Conference-Volume 1*, pages 250–250, 2018.

[5] Gabriela Karoline Michelon, Lukas Linsbauer, Wesley KG Assunção, and Alexander Egyed. Comparison-based feature location in argouml variants. In *Proceedings of the 23rd International Systems and Software Product Line Conference-Volume A*, pages 93–97, 2019.

[6] Linda M Northrop. Sei's software product line tenets. *IEEE software*, 19(4):32–40, 2002.

[7] Thorsten Berger, Daniela Lettner, Julia Rubin, Paul Grünbacher, Adeline Silva, Martin Becker, Marsha Chechik, and Krzysztof Czarnecki. What is a feature? a qualitative study of features in industrial software product lines. In *Proceedings of the 19th International Conference on Software Product Line*, pages 16–25, 2015.

[8] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. *Feature-oriented software product lines*. Springer, 2016.

[9] Thorsten Berger, Steven She, Rafael Lotufo, Krzysztof Czarnecki, and Andrzej Wasowski. Feature-to-code mapping in two large product lines. In *SPLC*, pages 498–499. Citeseer, 2010.

[10] Elisabeth Niehaus, Klaus Pohl, and Günter Böckle. Software product line engineering: Foundations, principles and techniques, kapitel product management, 2005.

[11] Yael Dubinsky, Julia Rubin, Thorsten Berger, Slawomir Duszynski, Martin Becker, and Krzysztof Czarnecki. An exploratory study of cloning in industrial software product lines. In *2013 17th European Conference on Software Maintenance and Reengineering*, pages 25–34. IEEE, 2013.

[12] Ina Schaefer, Rick Rabiser, Dave Clarke, Lorenzo Bettini, David Benavides, Goetz Botterweck, Animesh Pathak, Salvador Trujillo, and Karina Villela. Software diversity: state of the art and perspectives. *International Journal on Software Tools for Technology Transfer*, 14(5):477–495, 2012.

[13] Marcus Vinicius Couto, Marco Tulio Valente, and Eduardo Figueiredo. Extracting software product lines: A case study using conditional compilation. In *2011 15th European conference on software maintenance and reengineering*, pages 191–200. IEEE, 2011.

[14] Jabier Martinez, Nicolas Ordoñez, Xhevahire Tërnava, Tewfik Ziadi, Jairo Aponte, Eduardo Figueiredo, and Marco Tulio Valente. Feature location benchmark with argouml spl. In *Proceedings of the 22nd International Systems and Software Product Line Conference-Volume 1*, pages 257–263, 2018.

[15] Alexander Schultheiß, Paul Maximilian Bittner, Sascha El-Sharkawy, Thomas Thüm, and Timo Kehrer. Simulating the evolution of clone-and-own projects with vevos. In *Proceedings of the International Conference on Evaluation and Assessment in Software Engineering 2022*, pages 231–236, 2022.

[16] Sven Apel and Christian Kästner. An overview of feature-oriented software development. *J. Object Technol.*, 8(5):49–84, 2009.

[17] Lukas Linsbauer, E Roberto Lopez-Herrejon, and Alexander Egyed. Recovering traceability between features and code in product variants. In *Proceedings of the 17th International Software Product Line Conference*, pages 131–140, 2013.

[18] Alexander Schultheiß, Paul Maximilian Bittner, Thomas Thüm, and Timo Kehrer. Quantifying the potential to automate the synchronization of variants in clone-and-own. In *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 269–280. IEEE, 2022.

Selbständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und noch nicht für andere Prüfungen eingereicht habe. Sämtliche Quellen einschließlich Internetquellen, die unverändert oder abgewandelt wiedergegeben werden, insbesondere Quellen für Texte, Grafiken, Tabellen und Bilder, sind als solche kenntlich gemacht. Mir ist bekannt, dass bei Verstößen gegen diese Grundsätze ein Verfahren wegen Täuschungsversuchs bzw. Täuschung eingeleitet wird.

Berlin, den 14. Februar 2023 Angelina