Humboldt-Universität zu Berlin
Mathematisch-Naturwissenschaftliche Fakultät
Institut für Informatik

Patching with Matching in Clone And Own

Bachelorarbeit

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

eingereicht von: Lugh-Ole Koepke
geboren am: 26.06.2000
geboren in: Ribnitz-Damgarten

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

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

Abstract

In der Entwicklung von Programmen werden häufig bestehende Projekte kopiert und an neue Bedürfnisse angepasst (Clone-and-Own), obwohl es theoretisch besser ist, ein größeres Programm zu entwickeln, aus dem wiederum andere Programme generiert werden können. Clone-and-Own wird als langfristig schlecht bezeichnet, weil es mit der der Zahl der existierenden Varianten aufwändiger wird, die Varianten zu warten. In dieser Arbeit untersuchen wir, inwiefern Codematchings verwendet werden können, um diesen Aspekt der Wartung zu automatisieren, indem relevante Änderungen von einer Variante auf eine andere Variante übertragen werden. Mit einer solchen Automatisierung kann die Arbeit von Programmierern, die Clone-and-Own betreiben, erleichtert werden. Die Effizienz und Korrektheit dieser matchingbasierten Patchverfahren wurden an einer Simulation von einer Clone-and-Own Umgebung mit 10 Varianten über 5000 Commits hinweg erarbeitet.

Inhaltsverzeichnis

1 Einleitung

Clone-and-Own beschreibt eine in der Praxis verwendete Methode, unterschiedliche Varianten eines Programms zu erzeugen, indem bereits vorhandene Programme kopiert und an neue Bedürfnisse angepasst werden. Clone-and-Own wird beispielsweise verwendet, wenn ein Kunde möglichst schnell und somit auch mit wenig Aufwand ein neues Programm benötigt und große Teile der gewünschten Features bereits in einem bestehenden Programm eines anderen Kunden vorhanden sind. Dieses Programm kann kopiert und im Detail an die neuen Ansprüche des ersten Kunden angepasst werden.

Diese sogenannte Variabilität kann neben Clone-and-Own auch durch *Softwareproduktlinien* sichergestellt werden. Eine Softwareproduktlinie ist ein Programm, aus dem durch das Ein- und Ausschalten von Features unterschiedliche neue Programme abgeleitet werden können. Features sind bestimmte Eigenschaften und Funktionen, die ein Programm aufweisen kann. Dies erfordert Aufwand zum Aufsetzten der Produktlinie, bevor die eigentlichen Features implementiert werden können. So muss zunächst ein Buildsystem gefunden oder erstellt werden, mit dem die Varianten aus der Produktlinie generiert werden können und die Features sowie mögliche Abhängigkeiten müssen festgelegt werden

Beim Clone-and-Own werden bereits existierende Programme kopiert und an die neuen Bedürfnisse angepasst. Hierfür wird kein Voraufwand benötigt, also kann sofort mit der Implementierung der Features begonnen werden. Allerdings entsteht mit der Zeit und wachsender Zahl der Varianten mehr Aufwand. Zum Beispiel muss ein Bug, der in einer Variante auftaucht, in ähnlichen Varianten gefunden und abermals behoben werden. Haupsächlich liegt es am Mehraufwand, der mit der Zeit entsteht, dass die Clone-and-Own-Strategie in der Theorie schlechter [4, 3] als die Verwendung einer Produktlinie ist. In der Praxis ist das Prinzip des Clone-and-Own dennoch weit verbreitet [5].

Um dem Mehraufwand von Clone-and-Own entgegenzuwirken, geht die Forschung in diesem Bereich zwei Wege zur Verringerung der Arbeit für Programmierer, die Clone-and-Own oder Produktlinien verwenden. Eine Möglichkeit ist es, die durch Clone-and-Own entstandenen Varianten zu einer Produktlinie zusammenzufügen [7]. Dadurch werden der anfängliche Aufwand der Produktlinien und der spätere Aufwand durch Clone-and-Own minimiert. Bei der anderen Richtung werden Projekte, in denen Clone-And-Own genutzt wird, durch Tools unterstützt [6, 10].

Zum Beispiel unterstützen Tools, die relevante Änderungen von einer Variante auf eine andere Variante übertragen können, Clone-and-Own, da dadurch Fehlerbehebungen und Featureänderungen einfacher auf alle anderen betroffenen Varianten übertragen werden können. Eines dieser Tools, das unter anderem dafür benutzt werden kann, ist *Unix-Patch*. Schultheiß et al. [10] haben das Unix-Patch-Tool in einer Clone-and-Own-Umgebung untersucht. Dabei haben sie beobachtet, inwiefern Patches automatisiert mit den Unix-Patch-Tool übertragen werden können und wie *Domänenwissen*, bestehend aus den Features einer Variante sowie die von einer Änderung betroffenen Features, zu den Varianten dabei helfen kann. Dabei wurde festgestellt, dass das bestehende Unix-Patch-Verfahren mit Domänenwissen eine Genauigkeit von 93% erzielt, ohne Domänenwissen sind es noch 85%. Wir stellen uns der Frage, ob andere zukünftige Verfahren, die Codematchings

zwischen zwei Varianten ausnutzen, eine ähnliche oder sogar bessere Genauigkeit erzielen können.

Um der Frage nach der Genauigkeit von matchingbasierten Patchverfahren nachzukommen, nutzen wir das von Schultheiß et al. [10] vorgestellte Grundgerüst. Das Grundgerüst zeigt wie Patchverfahren, die zur Automatisierung dienen, getestet werden können. Diesem Grundgerüst folgt diese Arbeit, um die Korrektheit von Patchverfahren, welche Codematchings benutzen, zu erarbeiten. Für diese Korrektheit werden in dieser Arbeit zwei Patchverfahren vorgestellt. Das erste Patchverfahren nutzt alle verfügbaren Informationen aus, um einen perfekten Matcher zu simulieren. Durch die perfekte Funktionsweise stehen die durch das Verfahren entstehenden Ergebnisse als obere Schranke für matchingbasierte Patchverfahren. Das zweite Patchverfahren agiert so einfach wie möglich, indem Änderungen an der gleichen Zeilennummer, ohne das auf den Kontext geachtet wird, durchführt, damit dessen Ergebnisse als untere Schranke für allgemeine Patchingverfahren fungieren können. Letztlich werden die Ergebnisse des perfekten Matchers mit den Ergebnissen von Schultheiß et al. [10] verglichen, um die unterschiedlichen Patchverfahren gegenüberstellen zu können.

Zusammengefasst leistet diese Arbeit folgende Beiträge:

1. Das Aufstellen einer unteren Schranke für automatische Patchverfahren in Clone-And-Own-Projekten, durch ein Patchverfahren, das ganz simpel versucht alle Änderungen an genau der gleichen Stelle unabhängig vom Kontext anzuwenden.

2. Das Aufstellen einer oberen Schranke für matchingbasierte Patchverfahren, durch das Simulieren eines perfekten Matchers.

3. Der Vergleich des perfekten Matchers mit Unix-Patch [10], durch Verwenden des gleichen Auswertungsverfahren.

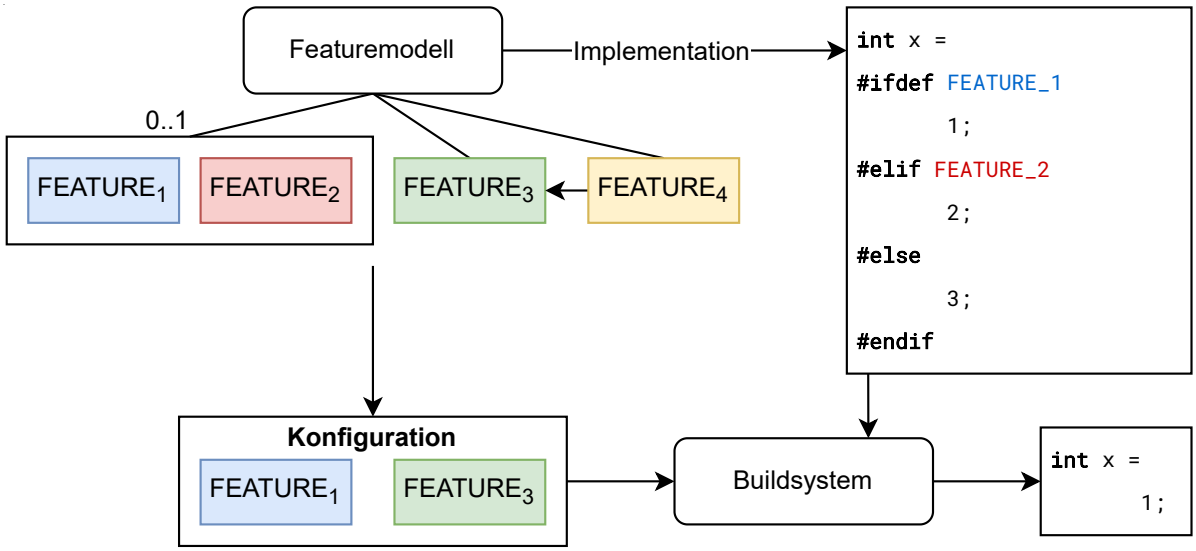

Abbildung 1: Übersicht über den Aufbau einer Produktlinie

2 Hintergrund

In diesem Kapitel werden Begriffe, Strategien und Tools erläutert, die zum Verstehen dieser Arbeit benötigt werden.

2.1 Softwareproduktlinien

Eine Softwareproduktlinie ist ein Programm, aus dem andere Programme mit unterschiedlichen Features abgeleitet werden können. Abbildung 1 verdeutlicht den Aufbau einer Produktlinie. So besitzt eine Produktlinie ein *Featuremodell*, das die vorhandenen Features 1-4, sowie die zwei Einschränkungen beschreibt. Die zwei Einschränkungen sorgen dafür, dass nur ein Feature von 1 und 2 vorhanden sein kann und dass für Feature 4 zusätzlich Feature 3 benötigt wird. Aus der Produktlinie können neue Programme, auch *Varianten* genannt, abgeleitet werden, indem aus dem Featuremodell eine den Einschränkungen entsprechende Menge von Features gewählt wird. Diese Features, die eine Variante ausmachen, werden als *Konfiguration* dieser Variante bezeichnet. Ein gewähltes Buildsystem leitet nun aus dem Sourcecode gemäß der Konfiguration das gewünschte Programm ab. Im Beispiel wird mithilfe von Präprozessoranweisungen der Wert einer Variable x in Abhängigkeit der vorhandenen Features gesetzt.

Um eine neue Produktlinie einzurichten wird Anfangs zusätzlicher Aufwand benötigt, um das Featuremodell zu definieren und die Features anhand dessen zu implementieren. Dieser Aufwand macht sich auf lange Sicht bezahlt macht, da es das Beheben von Bugs und Ändern von Features für viele Varianten deutlich vereinfacht.

Eine Möglichkeit, um aus einer gültigen Konfiguration ein Programm zu generieren, ist es den einzelnen Features Codeblöcke zuzuordnen. Wenn das entsprechende Feature

```
1 int x =
2 #ifdef FEATURE_1
3          1;
4 #elif FEATURE_2
5          2;
6 #else
7          3;
8 #endif
```

Codebeispiel 1: Präprozessoranweisungen Beispiel

```
1 int x =
2          2;
```

Codebeispiel 2: Auflösung der Präprozessoranweisung mit FEATURE_2

vorhanden ist, wird der Codeblock in die zu generierende Variante übernommen. Wenn das Feature nicht vorhanden ist, ist auch der Codeblock nicht vorhanden.

Präprozessoranweisungen, welche in bestimmten Programmiersprachen(C, C++, ...) vorhanden sind, bieten sich für die Kombination von Features und Codeblöcken an. Solche Präprozessoranweisungen sind Anweisungen, welche im Sourcecode angegeben aber beim Kompilieren aufgelöst werden. Bei Produktlinien können sie, ähnlich wie in Codebeispiel 1, genutzt werden, um je nach Feature unterschiedliche Codeblöcke zu kompilieren. Im Codebeispiel 1 wird einer Variable x ein Wert basierend auf den vorhandenen Features ausgewählt. Ist *FEATURE_1* vorhanden, entspricht x gleich 1, bei *FEATURE_2* ist x gleich 2, und 3 in allen anderen Fällen. Wenn aus der entstandenen ausführbaren Datei wieder der Sourcecode generiert werden würde, könnte man dort keine Präprozessoranweisungen finden. Beim Kompilieren vom Codebeispiel 1, ist im Sourcecode *int x = 2;* mit entsprechenden Zeilenumbrüchen zu finden, wie es im Codebeispiel 2 zu sehen ist.

2.2 Clone-and-Own

Clone-and-Own wird oft als der Gegensatz von Produktlinien gesehen und bezeichnet das Kopieren und Anpassen einzelner Varianten, die sich unabhängig voneinander weiterentwickeln können. Während bei Produktlinien Variabilität durch Featurekonfigurationen und dem Generieren von unterschiedlichen Programmen gegeben ist, wird bei Clone-and-Own ein bereits bestehendes Projekt kopiert und an die gewünschten Features angepasst. Projekte können ganz einfach mit *Git-Fork* oder *Git-Branch* kopiert werden. Dann kann sofort mit der Implementation begonnen werden ohne vorher ein Buildsystem zu finden und die Features zu definieren.

Der Verwaltungsaufwand steigt hier mit der Anzahl der Projektklone. Da die entstehenden Varianten sich beim Clone-and-Own gleichzeitig weiterentwickeln können, werden Features doppelt implementiert oder wieder von Variante zu Variante kopiert. Dieses Kopieren von Code erzeugt wiederum weiteren Aufwand, wenn darin Bugs enthalten sind. So muss der entsprechende Bug in einer Variante identifiziert und behoben werden. Anschließend müssen alle Varianten mit ähnlichen Features überprüft werden, ob der Bug

```
float distance(Point other) {
  float out = (x + other.x)^2;
  out += (y - other.y)^2;

  return sqrt(out);
}
```

Codebeispiel 3: Euklidische Abstandsmethode in 2D-Raum

```
float distance(Point other) {
  float out = (x + other.x)^2;
  out += (y - other.y)^2;
  out += (z - other.)^2;

  return sqrt(out);
}
```

Codebeispiel 4: Euklidische Abstandsmethode in 3D-Raum

```
float distance(Point other) {
  float out = abs(x - other.x);
  out += abs(y - other.y);
  out += abs(z - other.z);

  return out;
}
```

Codebeispiel 5: Manhatten-Abstandsmethode in 3D-Raum

dort auch existiert, bereits behoben wurde oder gar nicht vorkommt. Wenn in einer der anderen Varianten der Bug beobachtet werden konnte, muss er dort abermals behoben werden. Die Bugbehebung ist auch deswegen so aufwändig, weil fast nie dokumentiert wird, in welchen Varianten, Dateien und auf welche Art Features implementiert wurden. Damit muss jede Variante auf den Bug überprüft werden. Sollten diese Informationen doch aufgezeichnet werden, kann es dennoch passieren, dass sie mit der Zeit verloren gehen oder nicht mehr stimmen. Diese fehlerhafte Aufzeichnungen kommen daher, dass sich die Features einer Variante und die Dateien mit der Zeit ändern können und diese Änderungen jedes Mal dokumentiert werden müssen.

Codebeispiel 3, Codebeispiel 4 und Codebeispiel 5 sind Funktionen aus jeweils einer durch Clone-And-Own entstandener Variante. Zuerst wurde eine Klasse entwickelt, die Punkte in einem 2D-Raum beschreiben. Die gezeigte Funktion Codebeispiel 3 berechnet den euklidischen Abstand zweier Punkte. Jedoch ist diese Funktion fehlerhaft, da in der zweiten Zeile die x-Koordinaten addiert und nicht subtrahiert werden.

In einem anderen Projekt wird eine Klasse benötigt, die Punkte in einem 3D-Raum beschreiben. Dafür wird die Klasse aus dem ersten Projekt kopiert und durch die dritte Dimension erweitert. Die Abstandsmethode, die in Codebeispiel 4 (Seite 5) dargestellt ist, wird durch die z-Koordinate erweitert, der Fehler bleibt hierbei erhalten.

Ein drittes Projekt braucht eine Klasse, die für Punkte in einem 3D-Raum den Manhattanabstand anstelle des euklidischen Abstands berechnet. Diesmal wird die Klasse aus dem zweiten Projekt kopiert und die Abstandsfunktion ausgetauscht. Das Ergebnis ist in Codebeispiel 5 (Seite 5) zu sehen. Dem Entwickler der dritten Klasse fällt auf, dass die Abstandsmethode aus der zweiten Klasse fehlerhaft ist und behebt den Fehler. Nun kann der Programmierer alle anderen vorhanden Varianten, in diesem Beispiel das erste Projekt, nach dem selben Fehler absuchen, und den Bug dort wieder beheben, sofern er vorhanden ist. Andernfalls bleibt der Fehler offen und muss zu einem späteren Zeitpunkt wieder entdeckt und anschließend behoben werden.

2.3 Git Diff und Patch

Versionsverwaltungssysteme wie Git bieten eine Möglichkeit, mit Diff- und Patch-Tools die Clone-And-Own-Entwicklung zu unterstützen. Git-Diff vergleicht Dateien und findet Unterschiede zwischen diesen. Diese Unterschiede können anschließend mit Git-Patch auf andere angewandt werden.

Die von Git-Diff gefundenen Unterschiedene werden in sogenannte *Hunks* unterteilt. Ein Hunk beschreibt einen Bereich, in denen sich zwei Dateien unterscheiden. Hunks beginnen mit einer Headerzeile, welche jeweils die Startzeile und die Anzahl von Zeilen eines Bereichs von Änderungen zwischen den zwei zu vergleichenden Dateien angibt. Nach der Headerzeilen werden die Zeilen aus beiden Dateien aufgelistet, wobei Zeilen, die in der ersten Datei auftreten aber nicht in der Zweiten, mit einem führenden '-' angegeben werden. Ähnlich werden Zeilen, die in der zweiten Datei und nicht in der ersten Datei vorkommen, von einem '+' angeführt. Gleichbleibende Zeilen bleiben unverändert.

Codebeispiel 6 (Seite 7) zeigt ein Beispiel-Diff. Dieses Diff besteht aus Änderungen an einem Hunk, welcher mit *@@ -1,12 +1,5 @@* beginnt. Dieser Hunk startet in diesem Fall mit einer Kontextzeile. Danach werden die gelöschten Zeilen, die für Variabilität in unterschiedlichen Varianten, aufgelistet. Die hinzugefügten Zeilen, die den Features der Variante entsprechen, werden darunter aufgelistet. Hiernach folgen dem Standard entsprechend wieder drei Kontextzeilen, wonach ein neuer abgeschnittener Hunk zu sehen ist. Das Hunkformat ist für diese Arbeit umständlich zu nutzen, weshalb wir alle Hunks durch das Reduzieren auf das Hinzufügen und Löschen einzelner Zeilen auf Zeilenebene bringen.

Eine Änderung in Zeilenebene sagt aus, welche Zeile wo eingefügt oder gelöscht wird. Für das Codebeispiel 6 (Seite 7) sind - *2@#ifdef FEATURE_1* und + *2@ 2;* zwei Beispiele für valide Änderungen auf Zeilenebene. Die erste Änderung - *2@#ifdef FEATURE_1* gibt an, dass die zweite Zeile mit dem Inhalt '*#ifdef FEATURE_1*' gelöscht werden soll. Die zweite Änderung *+ 2@ 2;* beinhaltet das an Zeile 2 ' *2;*' eingefügt werden soll.

```
diff --git a/toDiff.txt b/toDiff.txt
index e39612c..203595a 100644
--- a/toDiff.txt
+++ b/toDiff.txt
@@ -1,12 +1,5 @@
 int x =
-#ifdef FEATURE_1
-        1;
-#endif
-#ifdef FEATURE_2
-        2;
-#else
-        3;
-#endif
+        2;



@@ -14,11 +7,4 @@
...
```

Codebeispiel 6: Diff Beispiel

Diff-Dateien können mittels Git-Patch oder Unix-Patch eine Datei gemäß den Änderungen im Diff verändern. Dafür wird ausgehend von der angegeben Zeilennummer nach dem entsprechenden Kontext gesucht. Wird der gesuchte Kontext in der Zieldatei nicht gefunden, werden die Kontextzeilen nach und nach reduziert bis entweder eine passende Zeile gefunden oder ein Maximum an Reduktionen erreicht wurde.

Solche Patchverfahren sind für Clone-And-Own-Umgebungen nicht optimal. Die gewollten Abweichungen zwischen zwei Varianten sorgen auch innerhalb der Dateien für Unterschiede. Dadurch passiert es oft, dass der Kontext, der für die Kontextsuche unabdingbar ist, nicht genau der selbe ist und nicht mehr gefunden werden kann. Angenommen aus der Produktlinie aus Codebeispiel 1 (Seite 4) werden folgende Varianten abgeleitet:

- eine erste Variante mit *FEATURE_1* mit dem Code `int x = 1;`
- eine zweite Variante mit *FEATURE_2* mit dem Code `int x = 2;`

Zum Beispiel wird in der ersten Variante nun am Ende eine Zeile hinzugefügt. Von dieser Änderung wird ein Diff erstellt, der auf die zweite Variante angewandt werden soll. Da der Kontext der Änderung `int x = 1;` ist und diese Zeile in der zweiten Variante nicht auftritt, kann die Änderung nicht übertragen werden, auch wenn das die richtige Entscheidung wäre.

2.4 Verwandte Arbeiten

Saha et al. [9] stellen das Tool *Hercules* vor, das innerhalb eines Programms einen Bug finden und beheben kann. Zunächst findet Hercules fehlerhafte Stellen im Code mittels *spectrum-based-fault-localization*-Techniken. Für diese fehlerhaften Stellen werden ähnliche Stellen mit ähnlichem Kontext gefunden, da diese Stellen eine hohe Wahrscheinlichkeit haben ebenfalls fehlerhaft zu sein. Anschließend wird ein Patchmuster für den entsprechenden Fehler bestimmt und angewandt. Schließlich werden Tests durchgeführt um zu bestimmen ob der Bug behoben wurde.

Der Aufbau von Hercules und dem in dieser Arbeit vorgestellten Verfahren ähneln sich in den Grundzügen. Zuerst wird ein Bug bestimmt: bei Hercules automatisch, hier durch Änderungen in einer Variante. Danach werden andere Stellen gesucht, in denen der Bug behoben werden soll: bei Hercules sind es ähnliche Stellen im gleichen Programm, hier in einer anderen Variante desselben Programms. In beiden Fällen werden Codematchings benutzt. Und zum Schluss werden die gefundenen Bugs behoben: bei Hercules durch bekannte Patchmuster und Tests, hier durch den Bugfix in einer Variante. Durch diese Ähnlichkeit könnte ein Patchverfahren entwickelt werden, das sich die Fähigkeiten von Hercules zu nutzen macht, indem im ersten und letzten Schritt ein durch einen Programmierer gegebener Patch eingespeist wird. Dieses Verfahren könnte auch ähnliche Bugs fixen, die durch die Variabilität der Varianten entstanden sind.

Da die Verwendung von Hercules im Clone-and-Own-Bereich bisher nur theoretisch möglich ist, wollten wir zunächst die bestmöglichen erreichbaren Ergebnisse durch die Simulation eines perfekten Matchers abschätzen.

ReDeBug [2] ist ein System um Code-Klone in Distributionen von Betriebssystemen zu finden, das im Vergleich zu anderen Systemen weniger Klone insgesamt findet, dafür aber besser skaliert, schneller läuft und weniger falsche Klone findet. ReDeBug verwandelt in einem Preprocessing-Schritt Codedateien in eine Folge von Tokens, die benutzt werden um Klone zwischen zwei Dateien zu finden. Ein Nutzer des Systems kann nach dem Preprocessing eine Diff-Datei als Patch eingeben und das System findet Stellen, deren Tokens ähnlich bis identisch zu den Tokens der Ausgangslage des Patches sind. Somit werden mögliche fehlerhafte Stellen gefunden, die mit der gleichen Patchart behoben werden könnten.

PaReco [8] ist eine für Clone-And-Own weiterentwickelte Variante von ReDeBug, die Pull-Requests von einer Source-Variante nach Bugfixes durchsucht. Anschließend wird mit einem erweiterten ReDeBug-System eine Target-Variante analysiert, um Stellen zu finden, wo der Bug noch existiert, behoben wurde, oder teilweise behoben wurde.

Sowohl ReDeBug als auch PaReco finden fehlerhafte Code-Klone ohne diese zu beheben. Dies mag Entwickler im Finden von fehlerhaften Stellen unterstützen, aber wir wollen untersuchen, inwiefern solche Patches auch automatisch angewandt werden können, um Entwickler zusätzlich diese Arbeit abzunehmen.

Schultheiß et al. [10] haben sich bereits mit dem automatisierten Patching von Varianten beschäftigt. Hierfür haben sie das Unix-Patch-Tool als Patchverfahren verwendet

und ihre Korrektheit analysiert. Dabei haben sie diese Art des automatisierten Patching als valide Option eingestuft. Allerdings gehen wir der Korrektheit einer anderen Art von Patchingverfahren nach und vergleichen diese anschließend mit der von Unix-Patch.

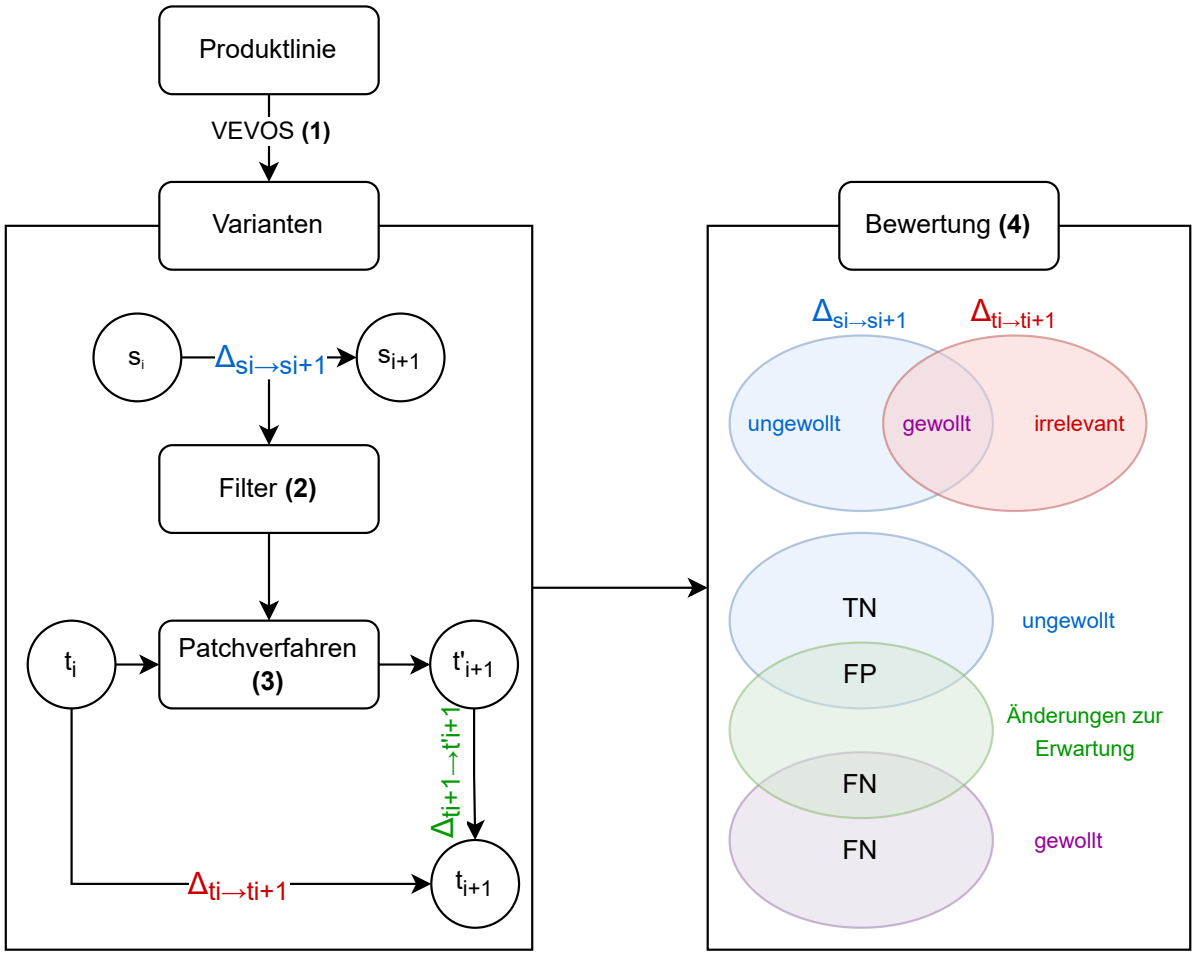

Abbildung 2: Skizzierung des Ablaufs der Studie

3 Methodik

Unsere Studie folgt dem Aufbau der Arbeit von Schultheiß et al. [10]. Der Aufbau, der in Abbildung 2 skizziert wird, lässt sich mit den folgenden drei Schritten, welche in diesem Kapitel genauer betrachtet werden, grob skizzieren:

(1) **Generation von Varianten** Zunächst werden mithilfe eines Benchmark-Generation-Tool aus einer Produktlinie Varianten generiert.

(2) **Filtern relevanter Änderungen** Durch einen Filter können erste ungewollte Änderungen von den zu übertragenden Änderungen schon vor dem Patchingschritt aussortiert werden.

(3) **Patching** Hierbei werden relevante Änderungen aus einer Variante auf eine andere Variante übertragen.

(4) **Bewertung** Zum Schluss werden die übertragenen Änderungen in richtig und falsch erkannte Änderungen klassifiziert.

Mit diesem Aufbau haben wir uns folgende Forschungsziele (RQ) gesetzt:
RQ 1 Wie sieht eine untere Schranke für allgemeine automatische Patchverfahren in Clone-And-Own-Projekten aus?
Für die Untersuchung der unteren Schranke wurde ein simples Patchverfahren eingebaut, das Änderungen weiterleitet. Aufgrund dieser einfachen Funktionsweise, die nicht auf Codematchings beruht, sind seine Ergebnisse als untere Schranke geeignet. Der simple Matcher wird in Abschnitt 3.3.2 (Seite 15) genauer beschrieben.

RQ 2 Was ist eine obere Schranke für automatische Patchverfahren, die Codematchings zwischen zwei Variante ausnutzen?
Für diese obere Schranke wurde ein Matchingverfahren implementiert, das alle verfügbaren Codematchings von der verwendeten Produktlinie zu den einzelnen Varianten nutzt, um ein perfektes Matchingvefahren zu simulieren. Diese Matchings werden aus der Ground-Truth, die das Benchmark-Generation-Tool bereitstellt, ausgelesen. Dadurch soll das Verfahren auch die bestmögliche Funktionsweise und somit auch Ergebnisse haben. Dieser perfekte Matcher wird in Abschnitt 3.3.1 (Seite 13) genauer beschrieben.

RQ 3 Können matchingbasierte Patchverfahren eine Alternative oder sogar eine Verbesserung gegenüber einem auf Unix-Patch basierten Verfahren sein?
Hierfür wird das in dieser Arbeit vorgestellte Verfahren mit einem auf Unix-Patch basierenden Verfahren [10] verglichen. Dieser Vergleich wird durch die Verwendung des gleichen Auswertungsverfahren ermöglicht und dient zum Abwägen des besseren Patchverfahrens.

Um Ergebnisse für diese Forschungsziele zu finden, haben wir ein Java-Programm geschrieben, dass automatisch die drei beschriebenen Schritte durchführt. Das Programm erlaubt verschiedene Konfigurationen von Matchverfahren, Filtern und Bewertern und erlaubt durch eigene zusätzliche Implementationen erweitert zu werden. Die für diese Arbeit implementierten Elemente werden im Laufe des Kapitels weiter erläutert.

3.1 Generation von Varianten

Für die Durchführung der Studie benötigen wir einen Datensatz der Varianten mit der zugehörigen Commithistorie. Um den perfekten Matcher zu simulieren wird zusätzlich eine Ground-Truth benötigt, die die Codematchings beinhaltet. Da unseres Wissens nach kein Datensatz existiert, der diese Daten beinhaltet, nutzen wir wie Schultheiß et al. das Benchmark-Generation-Tool VEVOS [11][1]. Hauptsächlich wird das Tool benutzt, um aus den Softwareproduktlinen BusyBox[2] und Linux[3] Varianten zu generieren. VEVOS besteht aus zwei Komponenten, die zusammen die Generierung ermöglichen.

Im ersten Schritt wird die Ground-Truth extrahiert, indem die Präprozessoranweisungen ausgelesen und in Presence-Conditions (Anwesenheitsbedingung) für die einzelnen Zeilen

[1] https://github.com/VariantSync/VEVOS_Simulation
[2] https://busybox.net
[3] https://github.com/torvalds/linux

umgewandelt werden. Um das Featuremodell auszulesen, wird die entsprechende Produktlinie mit dem Analysetool *KernelHaven* untersucht. Die entstandene Ground-Truth, bestehend aus Featuremodell und Presence-Conditions, wird in einer Datei gespeichert. Somit kann sie im zweiten Schritt verwendet werden, ohne dass sie erneut aus der Produktlinie extrahiert werden muss.

Die zweite Komponente ist die Simulationskomponente, welche Funktionen zum Laden der gespeicherten Ground-Truth, zum Ableiten von gültigen Konfigurationen aus dem Featuremodell und zum Generieren von feature-aware (feature-bewussten) Varianten. Feature-aware bedeutet, dass zusätzlich zu den Varianten auch ihre Konfigurationen und die Presence-Conditions von Codezeilen generiert und gespeichert werden; aus letzteren können die gewünschten Matchings gebaut werden.

Ein solches Matching ist eine bijektive Abbildung endlicher Mengen von den Zeilen der Produktlinien zu den Zeilen einer generierten Variante. Codematchings sagen aus, welche Zeile in der Produktlinie zu welcher Zeile in der generierten Variante gehört. Wie am Codebeispiel 1 (Seite 4) und Codebeispiel 2 (Seite 4) gesehen werden kann, unterscheiden sich die Zeilen in der Variante und der Produktlinie dadurch, dass die Präprozessoranweisungen und bestimmte Codeblöcke wegfallen. Wenn eine Zeile nicht übernommen wird, weil die Featurekonfiguration dies verhindern, dann wird diese ungültige Zeile durch den Wert -1 angegeben. Diese Codematchings werden im Patchingverfahren genutzt, um Zeilennummern von einer Variante in eine andere zu übertragen.

Mit dem Tool VEVOS können nun die eigentlich Varianten generiert werden. Für jeden Commit $p_i, 2 \leq i, \leq n$ in der Historie der Softwareproduktlinie, startend mit dem zweiten, werden 10 Versionspaare erstellt. Ein Versionspaar zum Zeitpunkt $2 \leq t \leq n$ besteht aus einer Version V_{t-1} und einer Version V_t, die beide die selbe Konfiguration besitzen. Jedes Versionspaar entspricht dem Entwicklungsschritt in einer Variante. Dieser Entwicklungsschritt kann mit den verschiedenen Patch-Verfahren, die wir betrachten, auf die anderen Varianten übertragen und ausgewertet werden.

3.2 Match-Filter

Für das Patchverfahren werden alle Kombinationen aus zwei Versionspaaren in Betracht gezogen, wobei die Änderungen, die im ersten Paar auftreten, auf das zweite Paar übertragen werden sollen. Das erste Paar von Versionen ist hierbei das *Source-Paar*, deren Änderungen auf eine andere Variante übertragen werden sollen. Die Variante, auf die die Änderungen übertragen werden sollen, wird zusammen mit dem erwarteten Patchergebnis als *Target*-Paar bezeichnet.

Zunächst werden mittels Git-Diff die Änderungen gesammelt, die in dem Source-Paar vorhanden sind, und auf Zeilenebene gebracht. Als optionaler Schritt können diese Änderungen danach gefiltert werden, sodass ein Teil der Änderungen, die nicht auf die Target-Variante übertragen werden sollen, schon vorher entfernt werden können. Diese ungewollten Änderungen entstehen durch die gewollten Unterschiede zwischen der Source- und Target-Variante, weshalb die Änderungen unwichtig für die Target-Variante sind.

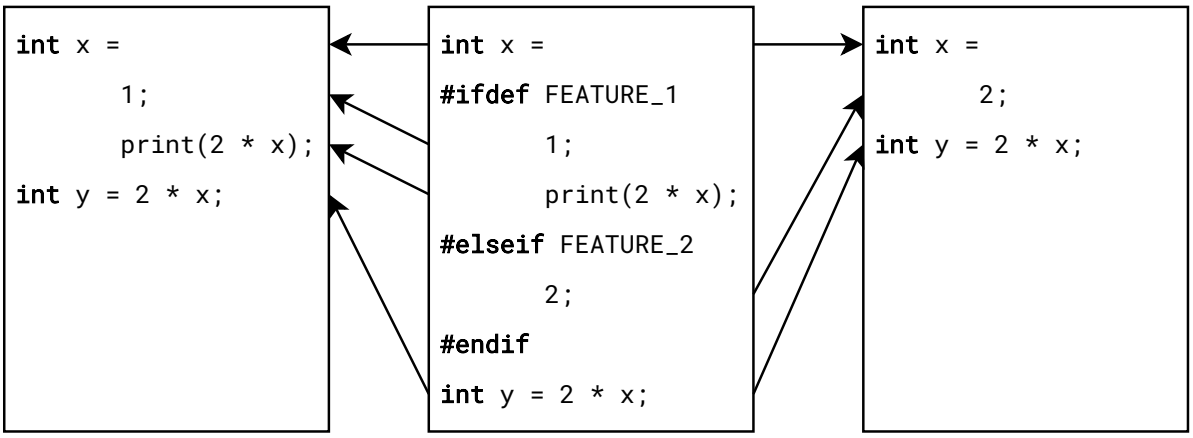

Abbildung 3: Codematching von einer Produktlinie zu zwei Varianten

Dieser Schritt des Filterns ist optional, damit die Auswirkungen des Filters beobachtet werden.

Das Filtern wird durch die gegebenen Matchings umgesetzt. Dafür wird im Falle einer Löschung in der ersten Target-Variante, und im Falle einer Einfügung in der zweiten Target-Variante, überprüft, ob die zu ändernde Zeile fehlt oder vorhanden ist. Damit jetzt eine Änderung übernommen werden kann, muss die richtige Stelle in der Target-Variante gefunden werden. Hier gibt es viele Möglichkeiten, wie das bewerkstelligt werden kann. Wir haben zwei dieser Möglichkeiten untersucht, welche im folgenden Abschnitt genauer erklärt werden.

3.3 Patching

Im Rahmen dieser Arbeit wurden zwei mögliche Matchingverfahren implementiert, die auf unterschiedliche Arten eine mögliche Stelle für den Patch finden. Somit können sie den ersten zwei Forschungszielen *RQ1* und *RQ2* dienen. Die erste Art ist der perfekte Matcher, welcher alle gegebenen Informationen ausnutzt, auch wenn diese in realen Szenarien nicht immer vorhanden sind. Damit kann der perfekte Matcher das bestmögliche matchingbasierte Verfahren simulieren. Dadurch können die Ergebnisse des perfekten Matchers als obere Grenze von Verfahren, die ebenfalls Codematchings benutzen, gesehen werden.

Der zweite implementierte Matcher ist der simple Matcher. Dieser nutzt keine der gegebenen Informationen. Somit kann an seinen Ergebnissen erkannt werden, wie gut Patchingmethoden ohne großen Aufwand funktionieren können.

3.3.1 Perfekter Matcher

Dieser Matcher soll ein perfektes Matchverfahren simulieren, damit für *RQ2* eine obere Schranke für matchingbasierte Patchverfahren gesetzt werden kann. Hierfür werden alle durch Vevos [11] gegebenen Codematchings verwendet.

Die genaue Funktionsweise wird an den im Abbildung 3 (Seite 13) gezeigten Codebeispielen erklärt. Das Beispiel in der Mitte zeigt eine Produktlinie aus der zwei Varianten(links und rechts) mitsamt Codematchings generiert wurden. Die linke Variante besitzt *FEATURE_1* und die rechte *FEATURE_2*, was für Variabilität in den Varianten sorgt. Das Hinzufügen und Löschen einer Zeile sind die einzigen Änderungen, die betrachtet werden müssen. Das Ändern einer einzelnen Zeile kann nämlich auf das Löschen der ursprünglichen Zeile und das Hinzufügen der neuen Zeile reduziert werden.

Betrachten wir zunächst den Fall des Löschens. Die Zeile 3 `print(2 * x);` wird in der linken Variante gelöscht und diese Löschung soll auf die rechte Variante übertragen werden, sofern diese Änderung auch die rechte Variante betrifft. Dafür wird zunächst die Zeilennummer 3 der linken Variante in Zeilennummer 4 in der Produktlinie umgewandelt. Diese Umwandlung wird durch die Surjektivität und endliche Größe der Codematchings ermöglicht.

Da die Menge der Zeilennummern der Produktlinie eine endliche Größe besitzt, kann diese Menge nach einem Element i durchsucht werden, für welches das Codematching m_l zur linken Variante Zeile 3 ergibt $m_l(i) = 3$. Das ist für Zeile 4 in der Produktlinie der Fall. Im nächsten Schritt wird die Zeilennummer aus der Produktlinie zur rechten Variante umgewandelt.

Dafür kann das Codematching m_r zur rechten Variante als normale Abbildung betrachtet werden. Da die gesuchte Zeile in dieser Variante nicht existiert, liefert das Matching $m_r(4) = -1$ als Fehlerwert, also wird keine Zeile gelöscht. Würde eine existierende Zeile gelöscht werden, dann würde das Matching die entsprechende Zeilennummer in der rechten Variante ausgeben.

Betrachten wir nun den Fall des Hinzufügens. In der linken Variante wird vor Zeile 4 der Kommentar `//calculate y` eingefügt. Da bei den Diff-Änderungen die Nummern von hinzugefügten Zeilen in Relation zu der Datei nach allen Änderungen angegeben werden, müssen hier die Abbildungen bezüglich der Variante zum späteren Zeitpunkt durchgeführt werden. Beim Löschen werden die Abbildungen in Bezug zu der früheren Variante durchgeführt.

Also wird die Zeilennummer der Änderung zunächst auf die Produktlinie abgebildet, das ist dort Zeile 8. Anschließend wird die Zeile 8 `int y = 2 * x;` auf der neuen Variante abgebildet, wobei Zeile 3 in der rechten Variante gefunden wird. Also wird auch in der rechten Variante vor Zeile 3 der Kommentar `//calculate y` eingefügt. Wenn die Transformation von der Produktlinie in die zweite Variante fehlschlägt, dann wird keine Änderung vorgenommen.

Da das perfekte Matchingverfahren den umliegenden Kontext benötigt, schlägt das Verfahren fehl, wenn eine neue Datei angelegt wird. Das liegt daran, dass die Datei in der ersten Target-Variante nicht existiert und somit kein Kontext vorhanden ist. Dieser Fehler wird umgangen, indem das Hinzufügen einer Datei erkannt und als Randfall betrachtet wird. Die Zeilen werden hierbei in der Reihenfolge übernommen, in der sie im Source-Paar auftreten.

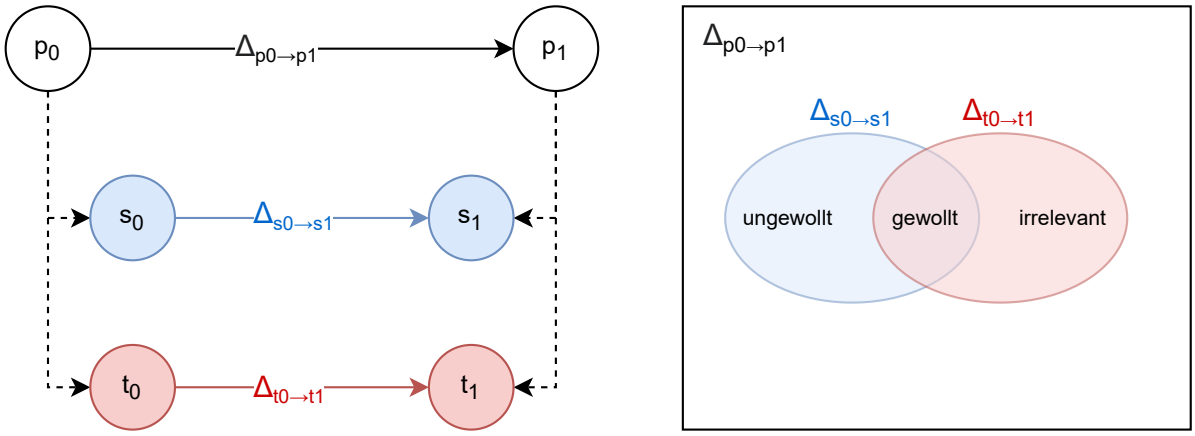

Abbildung 4: Veranschaulichung der Klassifikation von Änderungen

3.3.2 Simpler Matcher

Der simple Matcher nutzt, wie bereits beschrieben, keine gegebenen Informationen aus, damit seine simple Funktionsweise gemäß *RQ1* eine untere Schranke liefern kann. Da hierbei keine Codematchings verwendet werden, gilt diese untere Schranke für Patchingverfahren allgemein. Seine simple Funktionsweise ist dadurch gegeben, dass er Änderungen weiterleitet. Wenn in der Source-Variante die zweite Zeile gelöscht wird, dann wird auch die zweite Zeile in der Target-Variante gelöscht. Hierbei findet keine Überprüfung statt, ob es der Inhalt der Zeile der selbe ist. Das Gleiche geschieht auch beim Hinzufügen von Zeilen. Wird in der Source-Variante nach Zeile 5 eine Neue eingefügt, wird diese neue Zeile ebenfalls nach Zeile 5 in der Target-Variante eingefügt.

3.4 Bewertung

Für die Bewertung werden die Änderungen in ungewollte und gewollte Änderungen unterteilt, welche mit den gemachten Änderungen verglichen werden und somit weiter in True Positives, False Positives, True Negatives und False Negatives klassifiziert werden können. Für dieses Bewertungsverfahren sind die Source-Versionen s_0, s_1 und Target-Versionen t_0, t_1, ihre Änderungen $\Delta s_0 \to s_1, \Delta t_0 \to t_1$, und die gepatchte Variante t_1' gegeben. Die Änderungen werden zunächst in gewollte und ungewollte Änderungen unterteilt.

Die gewollten Änderungen sind jene, die in beiden Varianten vorhanden sind, also $required = \Delta s_0 \to s_1 \cap \Delta t_0 \to t_1$. Die ungewollten Änderungen sind Änderungen, die nur in der Source-Variante sind, also $undesired = \Delta s_0 \to s_1 \backslash required$. Die ungewollten Änderungen sind deshalb ungewollt, weil sie aufgrund der Variabilität nur in der Source-Variante vorkommen und beim Übertragen die Dateien der Target-Variante fehlerhaft machen könnten. Dementsprechend kommen sie nicht in der Target-Variante vor und sind dort ungewollt.

Die gewollten Änderungen kommen, wie beschrieben, in beiden Varianten vor. Sie sind gewollt, da das die Änderungen sind, die von der Source-Variante auf die Target-Variante übertragen werden sollen.

Die letzte Gruppe an Änderungen sind die, die nur in der Target-Variante auftauchen: die irrelevanten Änderungen. Da sie nicht in der Source-Variante erscheinen, können sie nicht übertragen werden. Die letzte Gruppe der irrelevanten Änderungen ergibt sich aus den Target-Änderungen, nachdem die Source-Änderungen entfernt wurden $irrelevant = \Delta t_0 \to t_1 \cap \Delta s_0 \to s_1$. Diese irrelevanten Änderungen können aber nicht von der Source-Variante auf die Target-Variante übertragen werden, da sie in der Source-Variante überhaupt nicht existieren. Also sind das irrelevante Änderungen, die gleichzeitige Änderungen in der Target-Variante simulieren.

Die gefundenen gewollten und ungewollten Änderungen können nun genauer unterteilt werden, indem geprüft wird, ob sie richtig erkannt und umgesetzt wurden. Für diese genauere Einteilung betrachten wir den Unterschied der gepatchten Datei zur erwarteten Datei $toExpectation = \Delta_{t_1' \to t_1}$. Unter Verwendung der Mengenoperationen mit den gewollten und ungewollten Änderungen mit den Änderungen zur erwarteten Datei entstehen die gesuchten True Positives, False Positives, True Negatives und False Negatives. Die erwartete Datei wird aus der Target-Version t_1 entnommen, da diese Version bereits alle gewollten Änderungen enthält.

Aus dem Schnitt der gewollten Änderungen und den Änderungen, die benötigt werden, damit aus der gepatchten Datei die erwartete wird, entsteht die Menge der Änderungen, die gewollt sind aber nicht umgesetzt werden konnten, die False Negatives $FN = required \cap toExpectation$. Befindet sich eine Änderung in den gewollten Änderungen, ist sie gewollt, und wenn sie sich in den Änderungen zum Erreichen der Erwartung befinden, ist es eine nicht - oder an der falschen Stelle - durchgeführte Änderung. Wäre die Änderung richtig gewesen, würde sie schließlich nicht in dieser Menge vorhanden sein.

Die Menge der ungewollten aber durchgeführten Änderungen, die False Positives $FP = undesired \cap toExpectation$, kann ähnlich aus dem Schnitt der ungewollten Änderungen und den Änderungen zur erwarteten Datei gebildet werden. Wenn die Änderung ungewollt durchgeführt wurde, befindet sich in den Änderungen zur erwarteten Datei eine Löschung dieser eingefügten Zeile. Oder im Fall einer ungewollten Löschung wird die Zeile dann wieder hinzugefügt.

Danach können die True Positives $TP = required \backslash FN$, also gewollte Änderungen, die auch umgesetzt wurden, gebildet werden. Das sind dann die gewollten Änderungen, nachdem man die bereits definierten False Negatives entfernt hat.

Genauso werden auch die True Negatives $TN = undesired \backslash FP$, ungewollte Änderungen, die nicht angewandt wurden, gebildet. Nur hier sind es die ungewollten Änderungen ohne die False Positives.

Nr.	Patchverfahren	Filter
1	Perfekter Matcher	Match-Filter
2	Perfekter Matcher	Kein Filter
3	Simpler Matcher	Match-Filter
4	Simpler Matcher	Kein Filter
5	Unix-Patch	Feature-Filter
6	Unix-Patch	Kein Filter

Tabelle 1: Benutzte Konfigurationen

4 Auswertung

Für die Beantwortung unserer Forschungsfragen generieren wir Varianten aus der Produktlinie BusyBox. BusyBox ist eine Sammlung vieler Unix-Tools für Systeme mit limitierten Ressourcen. Die Toolsammlung wurde aufgrund dieser Limitierung so angepasst, dass die einzelnen Tools möglichst wenig Speicher benötigen, indem sie weniger Einstellungsmöglichkeiten haben. BusyBox hat eine Historie von über 17000 Commits, was es zu einer gefragten und in der Praxis viel verwendeten Toolsammlung macht.

Außerdem benutzten Schultheiß et al. [10] BusyBox für die Generierung der Varianten. Durch das Verwenden der selben Produktlinie, können die Ergebnisse besser verglichen werden, da sie auf ähnlichen Daten beruhen. Auch wenn BusyBox über 17000 Commits hat, kann die von uns verwendete Version von Vevos zu nur ungefähr 5600 davon Varianten generieren. Das angefertigte Programm konnte für den perfekten Matcher alle dieser Commits verarbeiten. Aufgrund einer späteren Fehlerbehebung und der langen Laufzeit der Studie konnte der simple Matcher im Rahmen dieser Bachelorarbeit jedoch auf nur etwa 3100 Commits angewandt werden. Es war wichtiger den Fehler im simplen Matcher zu beheben, als Ergebnisse, die auf einer vollständige Durchführung des fehlerhaften Matchers beruhen, darzustellen.

Wie in Abschnitt 3 (Seite 10) bereits erwähnt, kann das Programm mit unterschiedlichen Kombinationen aus Match-, Filter- und Bewertungsverfahren gestartet werden. Tabelle 1 zeigt die in dieser Arbeit verwendeten Konfigurationen. Die Konfigurationen 1 und 2 mit dem perfekten Matcher dienen zum Setzen der oberen Schranke von matchingbasierten Patchverfahren. Die Konfigurationen 3 und 4 mit dem simplen Verfahren sind dem Setzen einer allgemeinen unteren Schranke dienlich. Beide Matchverfahren werden einmal mit und einmal ohne Filter ausgeführt, damit die korrekte Funktionsweise der Filter überprüft werden kann. Die in Tabelle 1 vorgestellten Konfigurationen 1-4 laufen mit dem in Abschnitt 3 (Seite 10) vorgestellten Bewertungsverfahren. Hierbei werden Änderungen in True Positives, False Positives, True Negatives und False Negatives unterteilt.

Konfigurationen 5 und 6 beschreiben die von Schultheiß et al. [10] benutzten Verfahren. Deren Feature-Filter sortieren Änderungen aus, wenn die von einer Änderung betroffenen Features nicht in der Variante vorhanden sind. Unsere Match-Filter filtern anhand dessen, ob die betroffene Zeile in der Variante vorhanden ist oder nicht. So werden Änderungen rausgefiltert, die eine Zeile löschen sollen, die nicht existiert, oder Änderungen, die eine

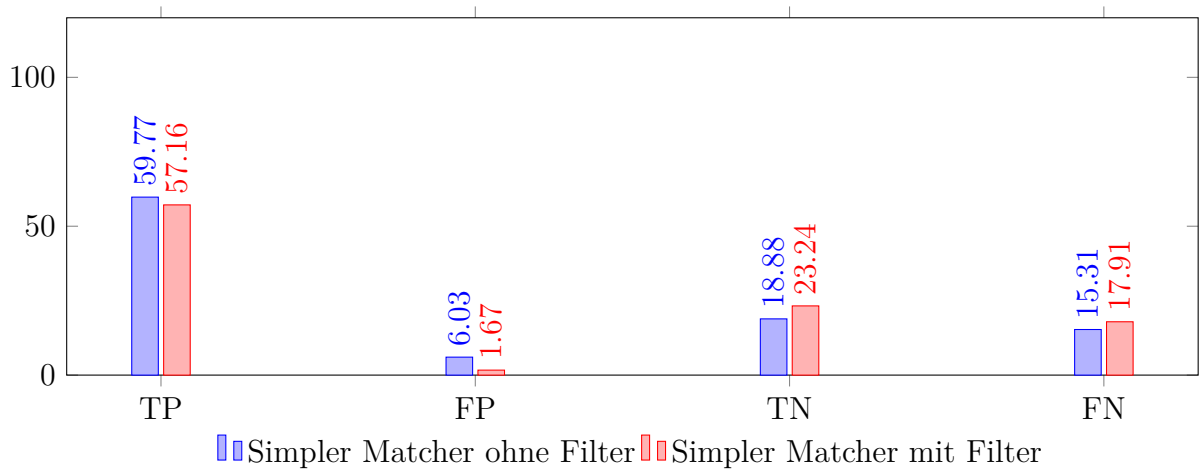

Abbildung 5: Ergebnisse der Konfigurationen mit dem simplen Matcher nach 3310 Commits

Zeile hinzufügen, welche nicht zu den Features der Variante passt.

Für die Unterteilung des verwendeten Bewertungsverfahren ist es wichtig, ob Änderungen gewollt oder ungewollt sind. Genauso wichtig ist es, ob Änderungen weitergeleitet wurden oder nicht. So sind die True Positives gewollte Änderungen, die auch umgesetzt wurden. Während False Positives ungewollte Änderungen sind, die trotzdem weitergeleitet wurden. Die True Negatives wiederum sind ungewollte, nicht weitergeleitete Änderungen. Zum Schluss bleiben die False Negatives, das sind gewollte Änderungen, die nicht richtig durchgeführt wurden, also entweder an der falschen Stelle oder sogar gar nicht angewandt wurden.

4.1 RQ 1: Die untere Schranke

Aus den in Abbildung 5 dargestellten Ergebnissen geht hervor, dass es bei dem simplen Matcher ohne Filter 59.77% True Positives, 6.03% False Positives, 8.88% True Negatives und 15.31% False Negatives gibt. Daraus lässt sich eine *Precision* (Präzision) von 0.90 berechnen, die angibt wie viele der angewandten Änderungen auch gewollte Änderungen sind. Zudem ergibt sich ein *Recall* (Trefferquote) von 0.80, der beschreibt wie viele der gewollten Änderungen richtig angewandt wurden. Die gewichtete Genauigkeit (balanced accuracy) liegt bei 0.73.

Da der simple Matcher ohne Filter alle Änderungen durchführt, ist anzunehmen, dass er einen Großteil der gewollten Änderungen an der falschen Stelle durchführt. Die Positionen der Änderungen bleiben beim simplen Matcher gleich, auch wenn sich durch die Variabilität in den meisten Fällen die Zeilennummer ändert. Das bedeutet, dass die False Negatives sehr hoch sein müssten, was bei rund 15% auch der Fall ist. Dennoch sind die True Positives mit 60.12% deutlich höher. Eine Erklärung dafür ist, dass die

Variabilität in BusyBox oft durch unterschiedliche Dateien umgesetzt wurde oder, dass es viele Dateien ohne Variabilität gibt. Der simple Matcher wendet Änderungen in der Target-Variante genau dort an, wo sie in der Source-Variante auftauchen. Genau diese Funktionsweise sorgt dafür, dass er ganze Dateien stets richtig kopiert, da immer die gewünschte Stelle getroffen wird.

Durch die Funktionsweise werden allerdings auch alle der ungewollten Änderungen durchgeführt, also müssten die False Positives deutlich höher als die True Negatives sein, da True Negatives nur auftreten können, wenn eine Datei nicht existiert und somit der Patch nicht angewandt werden kann. Überraschenderweise ist das Gegenteil der Fall und die True Negatives treten öfter auf als die False Positives. Das deutet ebenso auf viel Variabilität auf Dateiebene hin. Änderungen, die eine in der Target-Variante nicht existierende Datei betreffen können auch nicht angewandt werden. Dies resultiert in ungewollten Änderungen, die nicht angewandt werden - den True Negatives.

Wird der simple Matcher mit einem Filter ausgeführt, sinken die True Positives auf 57.16% und die False Positives auf 1.67%. Gleichzeitig steigen die True Negatives auf auf 23.24% und die False Negatives auf 17.91%. Das entspricht einem Unterschied von 2.61 Prozentpunkten bei den True Positives und False Negatives, sowie einem Unterschied von 4.46 Prozentpunkten bei den False Positives und True Negatives. Die Precision beträgt hier 0.97, der Recall 0.76 und die gewichtete Genauigkeit 0.76.

Die True Positives sinken, weil ein Teil davon zusammen mit einem Teil der False Positives durch den Filter verschwinden. Es kann auch passieren, dass ungewollte Änderungen herausgefiltert werden, die für eine gewollte Änderung an der richtigen Stelle sorgen. Es ist durch den Filter nicht überraschend, dass die True Negatives hoch und die False Positives niedrig sind, da genau das die Aufgabe des Filters ist.

Der simple Matcher besitzt ohne einen Filter eine Precision von 0.9, einen Recall von 0.8 und eine gewichtete Genauigkeit von 0.73. Der simple Matcher ändert in Target-Datei die Stelle, wie in der Source-Datei, wodurch viele Änderungen an den falschen Stellen und auch einige ungewollte Änderungen durchgeführt werden.
Die Ergebnisse sprechen für eine Variabilität in BusyBox, die hauptsächlich auf Dateiebene umgesetzt wird. In Projekten, in denen das nicht der Fall, sollten die Werte des simplen Matchers deutlich geringer sein, und somit auch die untere Schranke.

4.2 RQ 2: Die obere Schranke

Abbildung 6 (Seite 20) zeigt die Ergebnisse des perfekten Matchers. Ohne Filter sind 67.71% True Positives, 2.71% False Positives, 24.22% True Negatives und 5.34% False Negatives. Daraus lassen sich eine Precision von 0.96, ein Recall von 0.93 und eine gewichtete Genauigkeit von 0.89 berechnen.

Der perfekte Matcher erkennt ohne Matcher bereits 92% der gewollten Änderungen und 90% der ungewollten Änderungen. Es ist anzunehmen, dass es beim Verwenden von

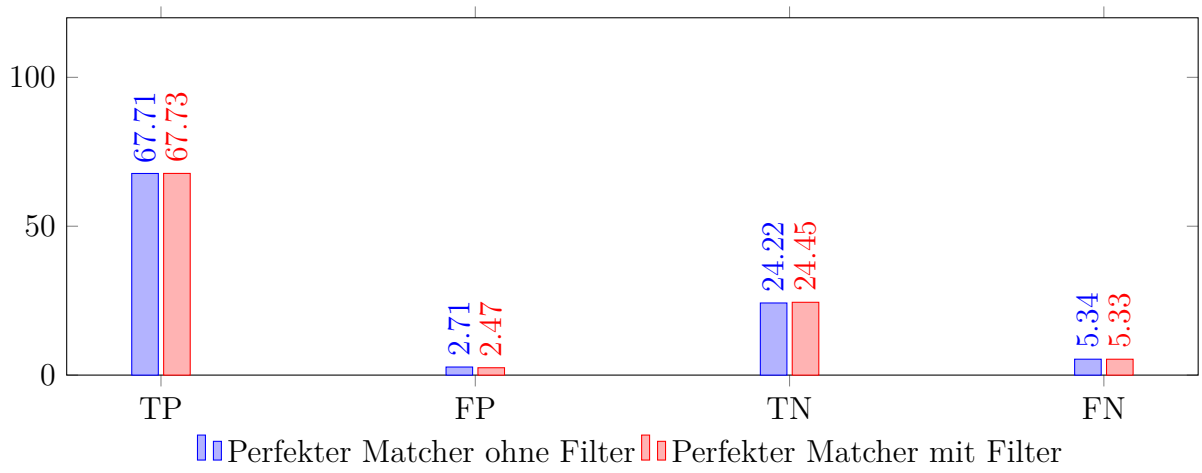

Abbildung 6: Ergebnisse der Konfigurationen mit dem perfekten Matcher

perfekten Matchings keine False Positives gibt, da die Matchings immer richtige Stelle finden, an der eine Änderung angewandt werden soll, wenn es eine solche Stelle gibt. Allerdings sind False Positives vorhanden. Möglicherweise kommt es bei Commits zu Änderungen in den Presence-Conditions innerhalb BusyBox. Dadurch können Änderungen in der Source-Variante entstehen, die in der Target-Variante schon implementiert wurden, da dort die für die nötigen Features bereits vorhanden waren. In dem Fall, befinden sich diese Änderungen aus Sicht unserer Ground-Truth in der Menge der ungewollten Änderungen. Der perfekte Matcher wird diese Änderungen ein zweites Mal anwenden, da die Änderung von der Source-Variante auf die Target-Variante abgebildet werden kann. Somit können mit dem perfekten Matcher Patches doppelt angewandt werden und False Positves entstehen.

Auf die gleiche Art können auch False Negatives entstehen. So ändern sich zum Beispiel die Dateien in dem Target-Paar aufgrund geänderter Presence-Conditions, aber nicht im Source-Paar. Die zu patchenden Target-Dateien bekommen die Änderungen von den Source-Dateien und beinhalten dementsprechend nicht die Änderungen, die durch die geänderten Presence-Conditions entstehen. Die Matchings gehen allerdings davon aus, dass diese angewandt wurden, und bilden somit auf die falschen Zeilen ab. Als Resultat entstehen False Negatives.

Betrachten wir nun die Ergebnisse des perfekten Matchers mit Filter. Hier gibt es 67.73% True Positives, 2.47% False Positives, 24.45% True Negatives und 5.33% False Negatives. Die True Positives und False Negatives sind also bei einer Änderung von 0.01 Prozentpunkten gleich geblieben, während bei den False Positives und True Negatives eine Änderung von etwa 0.23 Prozentpunkten stattfindet. Damit bleibt die Precision bei 0.96, der Recall bei 0.93 und die gewichtete Genauigkeit bei 0.89.

Der zusätzliche Filter sorgt für eine minimale Änderung von 0.01 Prozentpunkten, die sich von den False Negatives zu den True Positives verschieben. Das könnte ein

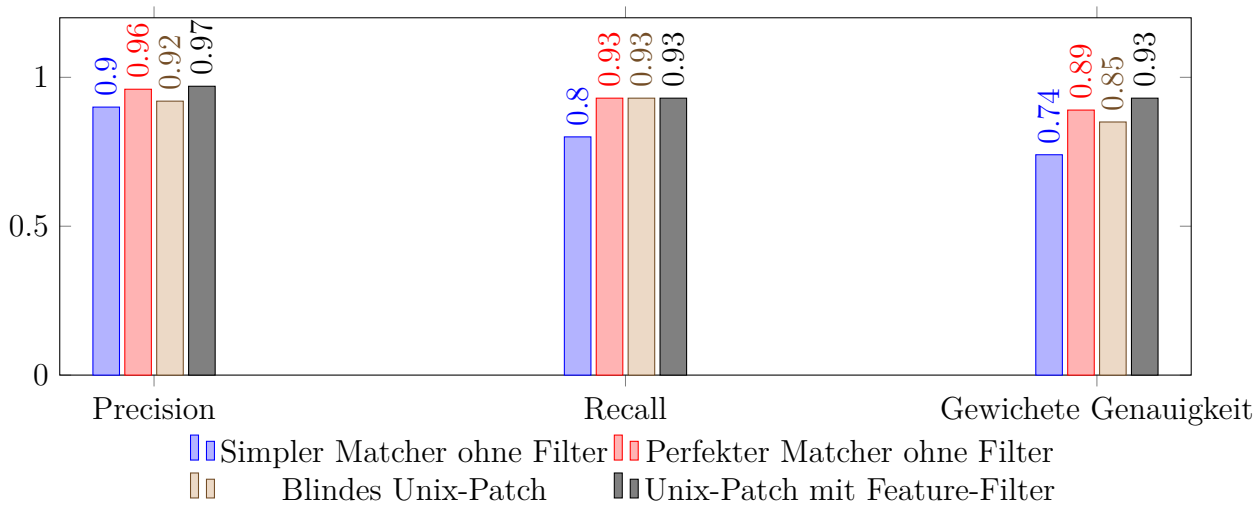

Abbildung 7: Vergleich von matchingbasierten Patchverfahren mit den unterschiedlichen Anwendungen von Unix-Patch [10]

Nebeneffekt von den verringerten False Positives sein, da diese die Dateien weniger beeinflussen und somit auch die Matchings wieder akkurater sind.

Bei den False Positives hingegen konnten 0.23 Prozentpunkte gefiltert werden, die bei den True Negatives geblieben sind. Auch hier ist der Unterschied sehr gering. Das liegt an der ähnlichen Funktionsweise des perfekten Matchers und des Filters. Beide bilden Änderung von der Source-Variante auf die Target-Variante ab und entscheiden anhand dessen, ob eine Änderung eine gültige Zeile betrifft oder nicht.

Der perfekte Matcher erzielt ohne Filter eine Precision von 0.96, einen Recall von 0.93 und eine gewichtete Genauigkeit von 0.89. Er hat hauptsächlich Probleme mit Änderungen, die aufgrund geänderter Presence-Conditions entstehen.
Das zusätzliche Verwenden der Match-Filter bringt aufgrund der gleichen Funktionsweise keine großen Änderungen mit sich. Andere Filter könnten mehr Einfluss auf die Ergebnisse haben.

4.3 RQ 3: Vergleich mit Unix-Patch

Abbildung 7 vergleicht unser simples und unser perfektes Patchverfahren mit dem blinden Unix-Patch-Verfahren und dem Unix-Patch-Verfahren mit Feature-Filter von Schultheiß et al. [10]. Deren Feature-Filter funktioniert anders als unser Match-Filter. Er nutzt das gesammelte Domänenwissen in Form von Wissen über die von Änderungen betroffenen Features, um Änderungen zu filtern, wenn diese keine für die Target-Variante relevante Features betrifft.

Aus der Abbildung 7 geht hervor, dass der simple Matcher eine Precision von 0.9, Recall von 0.8 und eine gewichtete Genauigkeit von 0.74 besitzt. Der perfekte Matcher übertrifft diese Werte mit einer Precision von 0.96, einem Recall von 0.93 und einer gewichteten Genauigkeit von 0.89. Das blinde Anwenden Unix-Patch ordnet sich zwischen

dem simplen und perfekten Matcher ein. Die Precision liegt hier bei 0.92, der Recall wieder bei 0.93 und die gewichtete Genauigkeit bei 0.85. Wird Unix-Patch mit dem Feature-Filter verwendet, ist das Verfahren sichtbar besser als der perfekte Matcher. Die Precision ist mit 0.97 leicht höher, der Recall abermals bei 0.93 und die gewichtete Genauigkeit mit 0.93 wieder höher als die vom perfekten Matcher.

Die hohe Precision des simplen Matchers kann mit denen der anderen Verfahren mithalten, da die Variabilität von BusyBox hauptsächlich auf Dateiebene umgesetzt zu sein scheint. Somit kann der simple Matcher weniger ungewollte Änderungen anwenden. Der Recall des simplen Verfahrens ist niedriger als die Werte der anderen Verfahren, weil er keinen Aufwand betreibt, die richtigen Stellen für Änderungen zu finden und diese meist an den falschen Orten durchführt. Das erklärt auch die geringe gewichtete Genauigkeit. Da die Werte des simplen Matchers niedriger sind als die der anderen Verfahren, hält seine Position als untere Schranke. Diese Schranke ist wahrscheinlich in Projekten niedriger, in denen die Variabilität innerhalb der Dateien größer ist.

Der perfekte Matcher hat dem blinden Unix-Patch gegenüber eine höhere Precision, wendet also im Verhältnis zu den ungewollten Änderungen mehr gewollte Änderungen an. Hinsichtlich des Recalls sind beide Verfahren gleich gut. Bei der gewichteten Genauigkeit ist der perfekte Matcher besser. Das bedeutet auch, dass der perfekter Matcher mehr der ungewollten Änderungen erkennt, wofür die höhere Precision auch ein Zeichen ist. Dementsprechend ist der perfekte Matcher dem blinden Unix-Patch-Verfahren vorzuziehen.

Das Unix-Patch-Verfahren, mit dem gesammelten Domänenwissen und dem darauf basierenden Filter, hat eine 0.01 Prozentpunkte höhere Precision, die bei 0.97 liegt, als der perfekte Matcher. Der Recall ist wieder genauso gut, und die gewichtete Genaugkeit ist mit einem Wert von 0.93 0.04 Prozentpunkte höher als die des perfekten Matchers.

Während der perfekte Matcher besser ist als das blinde Anwenden von Unix-Patch, ist er wiederum schlechter als Unix-Patch mit dem Feature-Filter. Das heißt aber nicht, dass das Unix-Patch-Verfahren mit Domänenwissen stets dem perfekten Matcher vorgezogen werden sollte. Das Domänenwissen muss erstmal durch zum Beispiel *Feature-Trace-Recording* [1] gesammelt werden, wobei Entwickler bei Commits die bearbeiteten Features mit angeben. Somit kann das Domänenwissen nicht automatisch zusammengetragen werden. Wenn das Feature-Trace-Recording nicht vom Anfang des Projekts an verwendet wird, ist es umso aufwändiger für alle bereits vorhandenen Commits die bearbeiteten Features zusammenzustellen. Codematchings könnten allerdings automatisch aus der Git-Historie oder durch Tools, die Codeklone finden können, bestimmt werden. Also muss zwischen etwas mehr Aufwand für die Programmierer zum Aufzeichnen bearbeiteter Features und mehr Aufwand für das Programm zum Berechnen der Matchings abgewogen werden.

Eine andere Möglichkeit wäre es, den perfekten Matcher mit dem Feature-Filter zu kombinieren, da dieser anders funktioniert als der perfekte Matcher und somit die Än-

derungen anders filtert. Außerdem ist es fraglich, inwiefern die Ergebnisse mit diesem Studienaufbau noch verbessert werden können, da sowohl der perfekte Matcher als auch das Unix-Patch-Verfahren auf idealisierten Daten beruhen und ihre Ergebnisse nah beieinander liegen.

Der perfekte Matcher liegt zwischen dem blinden Anwenden von Unix-Patch und dem Unix-Patch mit gesammelten Domänenwissen. Der perfekte Matcher bietet also eine gute Alternative, wenn kein Domänenwissen gesammelt werden kann, da die Matchings vollkommen automatisch bestimmt werden können, während die von Änderungen betroffenen Features von den Entwicklern angegeben werden müssen.
Denkbar ist auch eine Kombination des perfekten Matchers mit dem Feature-Filter, was für nochmals bessere Ergebnisse als Unix-Patch mit Domänenwissen sorgen kann.

5 Bedrohung der Validität

Diese Arbeit erbt nicht nur den Aufbau von Schultheiß et al.[10], sondern auch dessen möglichen Bedrohungen. So ist die Existenz von Bugs in den verwendeten Bibliotheken oder unserem Programm möglich. Allerdings wurde unser Programm viele Male akribisch auf Bugs untersucht, indem die einzelnen Varianten, samt Matchings und Konfigurationen, sowie die gepatchten Dateien und alle für die Bewertung relevanten Mengen gespeichert wurden. Mit diesen Daten wurde die richtige Funktionsweise mittels manueller Arbeit stichprobenartig überprüft. Dadurch konnte ein Fehler im simplen Matcher behoben werden. Zu dem Zeitpunkt der Behebung war eine vollständige Durchführung jedoch nicht mehr möglich. Korrekte Daten sind vollständigen Daten vorzuziehen, weswegen wir uns entschlossen haben, die korrekten Daten anstelle der vollständigen, fehlerhaften Daten zu behandeln.

Des Weiteren können die generierten Varianten und die zugehörigen Matchings nicht realistisch und damit idealisiert sein. Allerdings bietet VEVOS[11] die Möglichkeit beliebig viele Varianten aus 5000 Commits von BusyBox zu generieren, welches ein in der Praxis viel verwendetes Tool ist. Unseren Wissens nach existieren keine Benchmarks, die dem Zweck diesen Experiments dienen, besonders nicht in dieser Größenordnung. Das Fehlen von dieser Arbeit dienlichen Benchmarks liegt an der oft fehlenden Git-Historie oder der benötigten Ground-Truth, welche die Codematchings beinhaltet, die für diese Arbeit wesentlich sind.

Außerdem könnten die generierten Varianten eingeschränkt werden, damit nur realistische Varianten erzeugt werden. Es ist ebenso unrealistisch und idealisiert, Codematchings von jeder Variante zu jeder anderen Variante als gegeben zu nehmen. Aber der Zweck dieser Arbeit ist es eine obere Schranke zu setzen, dafür sind diese unrealistischen Matchings genau richtig. Zukünftige Forschung könnte sich damit beschäftigen, wie realistische Matchings erzeugt werden können oder ob ähnliche Stellen genügen, wie es bei der automatischen Bugbehebung der Fall sein kann[9].

Wir haben nur BusyBox betrachtet, welches eine Softwareproduktlinie ist, kein Clone-and-Own-Projekt. Aufgrund der Funktionalität von VEVOS[11] könnte das Experiment zusätzlich auf dem Linux Kernel wiederholt werden. Das Ausführen der Studie auf dem Linux-Kernel war für diese Arbeit nicht geplant. Natürlich ist auch der Linux Kernel kein Clone-and-Own-Projekt, das mehr Unterschiede zwischen zwei Varianten aufweist. Aber das Experiment simuliert eine Clone-and-Own-Umgebung, in der Änderungen von einer Variante auf die andern Varianten übertragen werden, sobald diese auftauchen. Durch dieses sofortige Angleichen der Varianten, wird die Divergenz zwischen Varianten klein gehalten.

6 Zusammenfassung

In der Softwareenwicklung wird häufig die Strategie des Clone-and-Own verfolgt, bei welcher bestehende Projekte kopiert und die dadurch entstehenden Varianten an neue Bedürfnisse angepasst werden. Diese Strategie kommt mit einem Aufwand, der mit der Zahl der Varianten steigt. So müssen zum Beispiel Bugs, die in einer Variante auftreten auch in anderen Varianten mit ähnlichen Features behoben werden. Wir haben untersucht inwiefern matchingbasierte Patchverfahren geeignet sind, um solche Fehlerbehebungen automatisch von einer Variante auf andere Varianten übertragen werden können.

Dafür wurden zwei Patchverfahren implementiert: eines, das als perfekter Matcher fungiert, dadurch können seiner erzielten Ergebnisse als obere Schranke für matchingbasierte Patchverfahren angesehen werden; und eines, das so simpel wie möglich agiert, somit können seine Ergebnisse als untere Schranke für Patchingverfahren allgemein gesehen werden. Des Weiteren haben wir unsere Ergebnisse mit dem Unix-Patch-Tool von Schultheiß et al. [10] verglichen.

Unser perfekter Matcher erkennt 89% aller Änderungen richtig und kann diese auch richtig anwenden. Beim simplen Matcher sind es nur 73%. Also haben matchingbasierte Patchverfahren eine (gewichtete) Genauigkeit von 73%-89%. Das Verfahren, welches auf Unix-Patch [10] basiert und Domänenwissen benutzt, besitzt eine Genauigkeit von 93%. Diese Genauigkeit ist geringfügig besser als die von unserem perfektem Matcher. Dahingehend ist es von Fall zu Fall abzuwägen, ob Entwickler Änderungen an Features für ein etwas genaueres Patchverfahren dokumentieren sollen oder ob automatisch Matchings bestimmt werden sollen.

Für realistische Ergebnisse bleibt zu untersuchen, inwiefern Matchings für Clone-and-Own-Umgebungen erstellt werden können. Ein Möglichkeit ist durch die Commithistorie gegeben, da zum Zeitpunkt des Klonens zwei Varianten vollständig übereinstimmen. Eine Kombination des perfekten Matchers mit den Feature-Filtern ist eine Alternative, welche noch besser als die einzelnen Verfahren sein könnte und untersucht werden kann. Dies sind aber Themen für zukünftige Arbeiten.

Literatur

[1] Paul Maximilian Bittner et al. “Feature Trace Recording”. In: *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*. Athens, Greece: ACM, Aug. 2021, S. 1007–1020. ISBN: 9781450385626. DOI: 10.1145/3468264.3468531. URL: https://doi.org/10.1145/3468264.3468531.

[2] Jiyong Jang, Abeer Agrawal und David Brumley. “ReDeBug: finding unpatched code clones in entire os distributions”. In: *2012 IEEE Symposium on Security and Privacy*. IEEE. 2012, S. 48–62.

[3] Cory J Kapser und Michael W Godfrey. ““Cloning considered harmful” considered harmful: patterns of cloning in software”. In: *Empirical Software Engineering* 13.6 (2008), S. 645–692.

[4] Jacob Krüger und Thorsten Berger. “An empirical analysis of the costs of clone-and platform-oriented software reuse”. In: *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2020, S. 432–444.

[5] Elias Kuiter et al. “Getting rid of clone-and-own: Moving to a software product line for temperature monitoring”. In: *Proceedings of the 22nd International Systems and Software Product Line Conference-Volume 1*. 2018, S. 179–189.

[6] Raúl Lapeña, Manuel Ballarin und Carlos Cetina. “Towards clone-and-own support: locating relevant methods in legacy products”. In: *Proceedings of the 20th International Systems and Software Product Line Conference*. 2016, S. 194–203.

[7] Wardah Mahmood et al. “Seamless variability management with the virtual platform”. In: *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE. 2021, S. 1658–1670.

[8] Poedjadevie Kadjel Ramkisoen et al. “PaReco: patched clones and missed patches among the divergent variants of a software family”. In: *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2022, S. 646–658.

[9] Seemanta Saha et al. “Harnessing evolution for multi-hunk program repair”. In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE. 2019, S. 13–24.

[10] Alexander Schultheiß et al. “Quantifying the Potential to Automate the Synchronization of Variants in Clone-and-Own”. In: 2022.

[11] Alexander Schultheiß et al. “Simulating the Evolution of Clone-and-Own Projects with VEVOS”. In: *The International Conference on Evaluation and Assessment in Software Engineering 2022*. 2022, S. 231–236.

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 21. Dezember 2022