



**UNIVERSITÄT PADERBORN**

*Die Universität der Informationsgesellschaft*

Fakultät für Elektrotechnik, Informatik und Mathematik

Institut für Informatik

Arbeitsgruppe Softwaretechnik

## Bachelorarbeit

Gerichtet an die Arbeitsgruppe Softwaretechnik

zur Erreichung des Grades

Bachelor of Science

# Unparsing von Datenstrukturen zur Analyse von C-Präprozessor-Variabilität

von  
EUGEN SHULIMOV

Betreut durch:  
Prof. Dr. Thomas Thüm

Paderborn, 9. August 2024



# Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen worden ist. Alle Ausführungen, die wörtlich oder sinngemäß übernommen worden sind, sind als solche gekennzeichnet.

---

Ort, Datum

---

Unterschrift



**Zusammenfassung.** We present a full documentation of the Paderborn University Computer Science thesis template (UPB-CS-TT) and how to use it. This document also serves as a demonstrator to show what documents UPB-CS-TT produces. Have fun!



# Inhaltsverzeichnis

<b>1</b>	<b>Einleitung</b>	<b>1</b>
<b>2</b>	<b>Hintergrundwissen</b>	<b>3</b>
2.1	C-Präprozessor . . . . .	3
2.2	Variabilität Umsetzung mit C-Präprozessor . . . . .	4
<b>3</b>	<b>Unparse Algorithmus</b>	<b>7</b>
3.1	Verlorengehende Informationen . . . . .	7
3.2	Unser Algorithmus . . . . .	13
3.3	Laufzeitanalyse . . . . .	13
3.3.1	Laufzeitanalyse für Unparsen von Variation-Trees . . . . .	13
3.3.2	Laufzeitanalyse für Unparsen von Variation-Diffs . . . . .	13
3.4	Metrik . . . . .	13
<b>4</b>	<b>Metrik</b>	<b>15</b>
<b>5</b>	<b>Implementierung</b>	<b>19</b>
5.1	Code . . . . .	19
5.2	Test . . . . .	19
	<b>Bibliography</b>	<b>20</b>





# 1

## Einleitung



## Hintergrundwissen

In diesem Kapitel stellen wir Hintergrundwissen zur Verfügung. Dieses Wissen ist unserer Meinung nach nicht selbstverständlich aber ist von Bedeutung für das Verständnis dieser Arbeit. Es handelt sich um C-Präprozessor und einer seiner Einsatzmöglichkeiten. In dem Abschnitt 2.1 dieses Kapitels wird der C-Präprozessor vorgestellt. Seine Möglichkeiten und Anweisungen zusammen mit einem Beispiel. Wie der C-Präprozessor zur Umsetzung der Variabilität im Code genutzt werden kann und welche Bestandteile von dem C-Präprozessor dazu nötig sind, wird im Abschnitt 2.2 des Kapitels erläutert.

### 2.1 C-Präprozessor

C-Präprozessor ist ein Tool, das den Quellcode vor dem Kompilieren manipuliert [1]. Dieses Tool bietet Möglichkeiten zur bedingte Kompilierung, zur Dateieinbindung und zur Erstellung lexikalische Makros [1]. Eine C-Präprozessor-Direktive beginnt mit `#` und geht bis zum ersten Whitespace-Zeichen weiter, optional kann nach der Direktive Argument im Rest der Zeile stehen. Der C-Präprozessor hat solche Anweisungen wie, `#include` zum Einbinden von Dateien, um zum Beispiel Header-Dateien wiederzuverwenden. Wie das Aussehen kann, ist in der Abbildung 2.1 Zeile 1 zu sehen (Abb.2.1 Z1). Mit den Anweisungen `#if` (Abb.2.1 Z6), `#else` (Abb.2.1 Z10), `#elif` (Abb.2.1 Z8), `#ifdef` (Abb.2.1 Z18), `#ifndef` (Abb.2.1 Z3), und `#endif` (Abb.2.1 Z5) wird die bedingte Kompilierung erzeugt. Dabei funktionieren `#if`, `#else`, `#elif`, und `#endif` vergleichbar mit dem, was man aus Programmiersprachen und Pseudocode gewohnt ist. `#ifdef` ist ähnlich zu `#if`, wird aber nur dann wahr, wenn der drauf folgender Makros definiert ist. `#ifndef` ist die Negation von `#ifdef`. Die Makros werden durch die Anweisung `#define` (Abb.2.1 Z4) erstellt. Der Präprozessor ersetzt dann während seiner Arbeit, den Makronamen durch seine Definition. Während dieser Arbeit kann ein Makros definiert, undefiniert und undefiniert, mit `#undef` (Abb.2.1 Z2), werden. Der C-Präprozessor hat noch weitere Anweisungen, auf die wir nicht weiter eingehen. Der C-Präprozessor kann in anderen Programmiersprachen verwendet werden, wenn diese Sprachen syntaktisch ähnlich zu C sind. Beispiel für solchen Sprachen sind C++, Assemblersprachen, Fortran und Java. Der Grund dafür ist, dass der C-Präprozessor ist unabhängig von der zugrundeliegenden Programmiersprache ist. Eine so ähnliche Vorverarbeitungsmöglichkeit ist in vielen anderen Programmiersprachumgebungen.

```

1 | #include <stdio.h>
2 | #undef N
3 | #ifndef N
4 | #define N 10
5 | #endif
6 | #if N > 10
7 | #define A "^-^"
8 | #elif N == 10
9 | #define A ";)"
10 | #else
11 | #define A ":("
12 | #endif
13 |
14 |     int main()
15 |     {
16 |
17 |         int i;
18 |         puts("Hello world!");
19 | #ifdef N
20 |         for (i = 0; i < N; i++)
21 |         {
22 |             puts(A);
23 |         }
24 | #endif
25 |
26 |         return 0;

```

Abbildung 2.1: Beispiel für C Code mit Präprozessor Anweisungen

## 2.2 Variabilität Umsetzung mit C-Präprozessor

Der C-Präprozessor ist eine Möglichkeit, Variabilität zu erzeugen [1]. Um die Variabilität mithilfe des C-Präprozessors zu erzeugen, brauchen wir dessen Möglichkeit zur bedingten Kompilierung [1]. Dies wird mit den C-Präprozessor-Anweisungen `#if` (Abb. 2.1 Z6), `#else` (Abb. 2.1 Z10), `#elif` (Abb. 2.1 Z8), `#ifdef` (Abb. 2.1 Z18), `#ifndef` (Abb. 2.1 Z2), und `#endif` (Abb. 2.1 Z4) bewerkstelligt. Dabei werden Codefragmente von diesen Anweisungen eingeschlossen. Danach, abhängig davon welche Makros definiert sind, werden bestimmte Codefragmente entweder behalten oder entfernt. Es ist möglich, mit diesen Anweisungen beliebige Aussageformeln über Features im Quellcode abzubilden [2]. Die Abbildung 2.2 zeigt, ein von uns erstelltes Beispiel, wie ein mit C-Präprozessor-Annotierter Code aussehen kann. Das Beispiel zeigt, dass die Anweisungen von den C-Präprozessor verschachtelt werden können. Dabei ist auch die Abhängigkeit einiger Features von anderen zu erkennen, wie in Zeile 5, wo die Auswahl des Features D nur dann Sinn ergibt, wenn auch die Features A und B ausgewählt sind. Nicht nur die Definition von Features, sondern auch die nicht Definition kann, einen Einfluss auf das Ergebnis haben, wie in der Zeile 16 zu sehen ist. Was dem Beispiel nicht zu entnehmen ist, ist der häufige Fall des lang kommentierten Codeabschnitts. In dem Beispiel wird, wie bei der Implementierung von funktionsorientierten Softwareproduktlinien, ein Name pro Feature reserviert. Wenn das Feature dann ausgewählt wird, wird dann ein Makro mit Feature-Namen definiert, mit der Anweisung `#define FEATURE_NAME`. Die Abbildungen 2.3 und 2.4 stellen 2 mögliche Ergebnisse der C-Präprozessor Ausführung dar. Dabei wird für die Abbildung 2.3 die Features A und B definiert und für die Abbildung 2.4 nur das Feature C. Dabei ist zu sehen, dass der generierte Code nur in dem Bezeichner `j` gleich ist und sonst nicht. Das veranschaulicht, wie unterschiedlich das Ergebnis von den C-Präprozessor sein kann.

```

1 | #ifdef FEATURE_A && FEATURE_B
2 |     foo();
3 |     bar();
4 |     int i = 18
5 | #ifdef FEATURE_D
6 | #define SIZE 200
7 |     foom();
8 | #else
9 | #define SIZE 175
10 |     i = 17;
11 | #endif
12 |     too(i);
13 | #endif
14 | #ifdef FEATURE_C
15 |     baz();
16 | #ifndef FEATURE_B
17 | #define SIZE 100
18 | #endif
19 |     bazzz();
20 | #else
21 |     boom();
22 |     broo();
23 | #endif
24 | #if SIZE > 180
25 |     long j;
26 | #elif SIZE < 111
27 |     short j;
28 | #else
29 |     int j;
30 | #endif

```

Abbildung 2.2: Beispiel für Umsetzung der Variabilität mit C-Präprozessor

```

1 |     foo();
2 |     bar();
3 |     int i = 18
4 |     i = 17
5 |     too(i);
6 |     boom;
7 |     broo();
8 |     int j;

```

Abbildung 2.3: Ausgabe des C-Präprozessors wenn Feature A=1 B=1 C=0 D=0

```

1 |     baz();
2 |     bazzz();
3 |     short j;

```

Abbildung 2.4: Ausgabe des C-Präprozessors wenn Feature A=0 B=0 C=1 D=0

Die Abbildung 2.5 zeigt Beispiele für vier Pattern, welche häufig bei der Umsetzung der Variabilität mit C-Präprozessor verwendet werden. Oben rechts in der Abbildung 2.5 ist alternative Includes zu sehen. Abhängig von der Definition des Features werden unterschiedliche Header-Dateien eingefügt. Das Beispiel zeigt, dass wenn das Feature `WINDOWS` definiert wird, der Windows-Header eingefügt, sonst der von Unix. Bei alternative Funktionsdefinitionen oben links zu sehen, gibt das Feature an, ob oder wie eine oder mehrere Funktionen definiert sind. In der Abbildung ist zu sehen, dass entweder eine Funktion `foo()` definiert wird oder alle Stellen, wo diese auftaucht durch 0 ersetzt werden. Das dritte Beispiel zeigt, dass wir die Makros während der C-Präprozessor Ausführung definieren und undefinieren können. Dazu ist es auch Möglich, Makros umzudefinieren. In dem Beispiel ist das Feature `FEAT_WINDOWS` automatisch definiert. Wenn aber des Feature `FEAT_SELINUS` definiert wird, wird das Feature `FEAT_LINUS` efiniert und das Feature `FEAT_WINDOWS` undefiniert. Damit wird das automatisch definierte Feature außer Kraft gesetzt. In dem letzten Beispiel rechts unten ist alternative Makrodefinition abgebildet. Abhängig davon, ob das Feature `A` definiert ist, wird des Makro `SIZE` mit unterschiedlichen Werten definiert, was ich auf den allokierten Speicher auswirkt.

## 2.2 VARIABILITÄT UMSETZUNG MIT C-PRÄPROZESSOR

```
1| #ifdef WINDOWS
2| #include <windows.h>
3| #else
4| #include <unix.h>
5| #endif
6| ...

1| #ifdef FEAT_SELINUX
2| #define FEAT_LINUX 1
3| #undef FEAT_WINDOWS
4| #endif
5|
6| #ifdef FEAT_WINDOWS
7| ...

1| #ifdef FOO
2| int foo(){...}
3| #else
4| #define foo(...) 0
5| #endif
6|
7| int i = 429 + foo()
8| ...

1| #ifdef A
2| #define SIZE 128
3| #else
4| #define SIZE 64
5| #endif
6|
7| ...allocate(SIZE)...
```

Abbildung 2.5: Beispiele für Variabilität Umsetzung mit C-Präprozessor Pattern

## Unparse Algorithmus

### 3.1 Verlorengelungende Informationen

Um zu verstehen was für Informationen verloren werden und der Unparser wiederherstellen muss, betrachten wir im folgenden den Parser Algorithmus von Viegner [3]. Der Algorithmus überführt einen textbasierten Diff in einen Variation-Diff um. Ein Variation-Diff ist ein gerichteter azyklischer Graph. Dieser Graph stellt dabei zeilenbasiert den textbasierten Diff dar. Die Knoten des Graphen werden durch einen Diff-Typ und einen Code-Typ eingeordnet. Diese Informationen werden der Zeile entnommen, die der Knoten repräsentiert und gelten damit auf für Zeilen eines textbasierten Diffs. Der Diff-Typ kann die Werte add, remove oder none einnehmen. Add bedeutet das diese Zeile dem textbasierten Diff hinzugefügt wurde, remove das diese Zeile entfernt wurde und none das die Zeile unverändert geblieben ist. Der Code-Typ kann die Werte if, elif, else, code, oder endif haben. Dabei gibt der Code-Type an, das bei dem Wert if die Zeile eine Anweisung des if-Blocks, das können die Präprozessor Anweisungen #if, #ifdef, oder #ifndef sein, enthält und der Knoten darstellt. Bei den Wert elif es ist die Anweisung #elif, bei else die Anweisung #else. Bei dem Wert code des Code-Typs enthält die Zeile Code und der Knoten stellt dies dar. Der Wert endif gibt and das die Zeile die Anweisung #endif enthält, in dem Variation-Diff ist dieser Code-Typ nicht enthalten. Die Knoten des Variation-Diffs haben höchstens zwei Elternknoten. Es gibt einen befor Elternknoten, das ist der Elternknoten, welchen der Knoten vor der Änderung hatte. Dieser Elternknoten gibt den umgebenden Präprozessor-Block vor der Änderung an. Es gibt noch einen after Elternknoten das ist der Elternknoten, welchen der Knoten nach der Änderung hat. Dieser Elternknoten gibt den umgebenden Präprozessor-Block nach der Änderung an. Nur Knoten mit Diff-Typ none haben zwei Elternknoten. Die Knoten mit dem Diff-Typ remove haben nur den befor Elternknoten und die Knoten mit dem Diff-Typ add haben nur after Elternknoten. Dabei kann ein befor Elternknoten nicht den Diff-Typ add haben und ein after Elternknoten nicht den Diff-Typ remove. Der Variation-Diff hat noch einen Knoten welcher keine widerspiegelung in dem textbasierten Diff enthält, das ist der Wurzelknoten. Der Wurzelknoten repräsentiert den ganzen textbasierten Diff. Er hat als einziger Knoten in dem Variation-Diff kein Elternknoten. Der Wurzelknoten hat immer den Diff-Typ none und den Code-Typ if, dabei ist das Feature-Mapping wahr.

**Algorithmus 1:** Erstellung eines Variation-Diffs aus einem Patch

---

**Data:** ein textbasierter Diff  
**Result:** ein Variation-Diff

```

1 initialisiere ein Stack/Keller before mit dem Wurzelknoten
2 initialisiere ein Stack/Keller after mit dem Wurzelknoten
3
4 foreach Zeile in dem Patch/Diff do
5    $\delta \leftarrow$  identifiziere Diff-Typ
6    $\gamma \leftarrow$  identifiziere Code-Typ
7    $\sigma \leftarrow$  gib relevante Stacks mithilfe von  $\delta$  an
8
9   if  $\gamma = \text{endif}$  then
10    | entpacke  $\sigma$  bis ein Knoten mit  $\gamma = \text{if}$  entpackt wurde
11  else
12    | erstelle einen neuen Knoten mit  $\delta$ ,  $\gamma$  und Eltern aus  $\sigma$ 
13    | if  $\gamma \neq \text{code}$  then
14    |   | füge den neuen Knoten  $\sigma$  hinzu
15    | end
16  end
17 end

```

---

Der Algorithmus arbeitet wie folgt, ganz am Anfang werden zwei Stacks erstellt und jeweils mit dem Wurzelknoten initialisiert, was in Zeilen 1 und 2 des Algorithmus 1 zu sehen ist. In Zeile 4 ist eine Schleife zu sehen, welche über alle Zeilen des textbasierten Diffs geht. Dabei wird für jede Zeile zuerst der Diff-Typ  $\delta$  in Zeile 5 und dann der Code-Typ  $\gamma$  in Zeile 6 festgelegt. In Zeile 7 werden die relevanten Stacks  $\sigma$  anhand von Diff-Typ  $\delta$  bestimmt und zwar wie folgt:

$$\sigma = \begin{cases} \text{Stack } \textit{after} & , \quad \delta = \text{add} \\ \text{Stack } \textit{before} & , \quad \delta = \text{remove} \\ \text{Stacks } \textit{before} \text{ und } \textit{after} & , \quad \delta = \text{none} \end{cases}$$

Danach in Zeile 9 kommen wir zu einer if-Abfrage. Wenn der Code-Typ der bearbeiteten Zeile *endif* entspricht, dann wird aus den relevanten Stacks in  $\sigma$  solange Knoten entnommen bis man ein Knoten mit dem Code-Type  $\gamma$  *if* entnommen hat. In Fall das beide Stacks relevant sind, muss der *if* Knoten in beiden Stacks gefunden werden. Wenn der Code-Typ nicht *endif* entspricht kommen wir in dem *else*-Teil ab Zeile 11 des Algorithmus 1. Dort wird zuerst ein neuer Knoten erstellt, welcher unter anderem auch Diff-Typ  $\delta$ , Code-Typ  $\gamma$  und Elternknoten aus den Stacks von  $\sigma$  enthält. Als nächstes wird in Zeile 13 Überprüft ob der erstellter Knoten nicht von Code-Typ *code* ist, also den Code-Type *if*, *elif*, oder *else* hat. Den Code-Typ *endif* kann dieser Knoten nicht haben, wegen der if-Abfrage in Zeile 9, welche nicht zulässt das ein Knoten mit diesem Typ zu dieser Stelle gelangen kann. Wenn der Knoten nicht von Code-Typ *code* ist, dann wird dieser Knoten den relevanten Stacks aus  $\sigma$  hinzugefügt, sonst wenn der Knoten den Code-Type *code* hat wird nichts gemacht.

Der vorgestellter Algorithmus ist für das Parsen von textbasierten Diffs zu Variation-Diffs ausgelegt aber es ist auch möglich den zum Parsen von C-Präprozessor-Annotierten Code zu Variation-Tree zu verwenden. Um das anstellen zu können müssen wir zwei Sachen anstellen. Zuerst wäre da die Anpassung der Eingabe, da wir C-Präprozessor-Annotierten Code haben aber der Algorithmus einen textbasierten Diff erwartet. Die zweite Sache wäre die Anpassung der Ausgabe, die Ausgabe des Algorithmus ist ein Variation-Diff, wir brauchen aber ein Variation-Tree.



Um die Eingabe gerecht für den Algorithmus zu machen, müssen wir unseren C-Präprozessor-Annotierten Code in ein textbasiertes Diff verwandeln. Dazu bilden wir ein Diff mit unseren C-Präprozessor-Annotierten Code als davor und danach Zustand und bekommen ein textbasiertes Diff wo alle Zeilen gleich dem C-Präprozessor-Annotierten Code sind. Dabei hat jede Zeile dieses Diffs den Diff-Typ none. Da jetzt ein Diff gegeben ist, können wir auf den Diff den Algorithmus anwenden. Die Ausgabe ist dann ein Variation-Diff und der muss in eine Variation-Tree umgewandelt werden. Um dies anzustellen, bilden wir eine Projektion des Variation-Diffs auf den davor bzw. danach Zustand und bekommen ein Variation-Tree, was auch gewollt ist. Mit den gezeigten zwischen Schritten lässt sich dieser Algorithmus auch für das Parsen von C-Präprozessor-Annotierten Code zu Variation-Trees verwenden.

Wir wollen die Arbeitsweise des Algorithmus veranschaulichen. Dazu wenden wir den Algorithmus auf den untenstehende künstlich generierte C-Präprozessor-Annotation anwenden. Da hier eine C-Präprozessor-Annotation gegeben ist aber wir ein textbasiertes Diff brauchen, wird wie in dem Abschnitt davor vorgegangen und diese C-Präprozessor-Annotation bildet ein Diff mit sich selbst, somit ist die nötige Eingabe gegeben. Am Anfang des Algorithmus werden die

```

1  Anweisung1 (a1)
2  #if Bedingung1 (b1)
3      #if Bedingung2 (b2)
4          Anweisung2 (a2)
5      #else
6          Anweisung3 (a3)
7  #endif
8  Anweisung4 (a4)
9  #endif

```

Stacks erstellt und mit dem Wurzelknoten initialisiert. Wir betrachten jetzt die Schleife, die über alle Zeilen des obigen Diffs geht. Wir kommen zur Zeile 1 der C-Präprozessor-Annotation, dort befindet sich eine normale Codezeile, welche nicht zur C-Präprozessor-Annotation gehört. Das ergibt dass diese Zeile den Code-Typ code hat und den Diff-Typ none. Alle anderen Zeilen haben auch den Diff-Typ none, aus dem Grund wie dieser Diff gebildet wurde und deshalb lassen wir die Erwähnung des Diff-Typ für jede Zeile sein. Dasselbe gilt auch für die relevanten Stacks in  $\sigma$ , da alle Zeilen den Diff-Typ none haben, gilt für alle Zeilen auch die gleichen relevanten Stacks und das sind beide. Da diese Zeile nicht Code-Typ endif hat wird ein Knoten mit Code-Type, Diff-Typ, Elternknoten aus den Stacks und dem Inhalt der Zeile und den Variation-Diff hinzugefügt, wie das aussieht ist in Abbildung 3.1 in dem Kasten nach Z.1 zu sehen. In Abbildung 3.1 beim Kasten Nach Z.2 ist der Variation-Diff nach der Bearbeitung der Zeile 2 zu sehen. Es wurde ein neuer Knoten erstellt, welcher eine if-Anweisung enthält. Die Schleife wurde fast gleich mit dem vorherigen Fall durchgelaufen, außer an der letzten if-Abfrage. Diese Abfrage was in Fall von Zeile 1 false in diesem Fall, da wir keinen Code-Typ code haben, wird diese Abfrage ausgeführt und der neu erstellte Knoten den Stacks hinzugefügt. Der nächste Kasten rechts zeigt den Variation-Diff nach Zeile 3. Der Algorithmus ist genauso wie in vorherigen Fall vorgegangen. Weiter Voran wird dem Variation-Diff im nächsten Schritt ein Code-Knoten hin-

zugefügt, da für die Erstellung dieses Knotens der Code selbst irrelevant ist, wurde hier genauso vorgegangen wie bei dem erstellen eines Code-Knotens in Zeile 1. In der Zeile 5 ist `#else` als Anweisung gegeben. Diese Zeile hat den Code-Typ `else` und somit auch kein `endif`. Es wird in den `else`-Zweig der ersten Abfrage gegangen. Dort wird ein neuer Knoten mit Inhalt dieser Zeile erstellt. Der Knoten wird den Stacks hinzugefügt, da der Knoten `else` als Code-Typ hat und nicht `code`, was die innere Abfrage erfüllt (Abb. 3.1 Nach Z.5). In der Nächsten Zeile ist wieder eine Codezeile vorhanden und aus der wird ein Code-Knoten erstellt. Wie es danach aussieht ist in Abbildung 3.1 Nach Z.6 zu sehen. Danach in der Zeile 7 treffen wir das erste mal auf die Anweisung `#endif`, welche den Code-Typ `endif` hat. Damit gelangen wir in den `if`-Teil der ersten Abfrage. Den Algorithmus nach muss man aus den Stacks die Knoten solange entnehmen bis ein `if` Knoten kommt. Dabei werden aus den Stacks die Knoten mit `else` und `if b2` entnommen und übrig bleiben der `if b1` Knoten und der Wurzelknoten. Dieser Schritt verändert den Variation-Diff nicht. Im nächsten Schritt treffen wir wieder auf eine Codezeile und erstellen ein Knoten mit der und fügen den dem Variation-Diff hinzu, welcher in Abbildung 3.1 Nach Z.8 zu sehen ist. In der Zeile 9 ist wieder ein `#endif` und wir müssen wieder Knoten aus den Stacks entnehmen. Dieses mal wird der Knoten `if b1` entnommen und es bleibt nur der Wurzelknoten übrig. Damit wäre die Arbeit des Algorithmus zu Ende und wir haben als Rückgabewert einen Variation-Diff erhalten. Da aber wir einen Variation-Tree brauchen müssen wir noch eine Projektion auf den Zustand davor oder danach bilden. Danach erhalten wir ein Variation-Tree, welches genauso aufgebaut ist, wie der Variation-Diff aus der Abbildung 3.1 Kasten Nach Z.8.

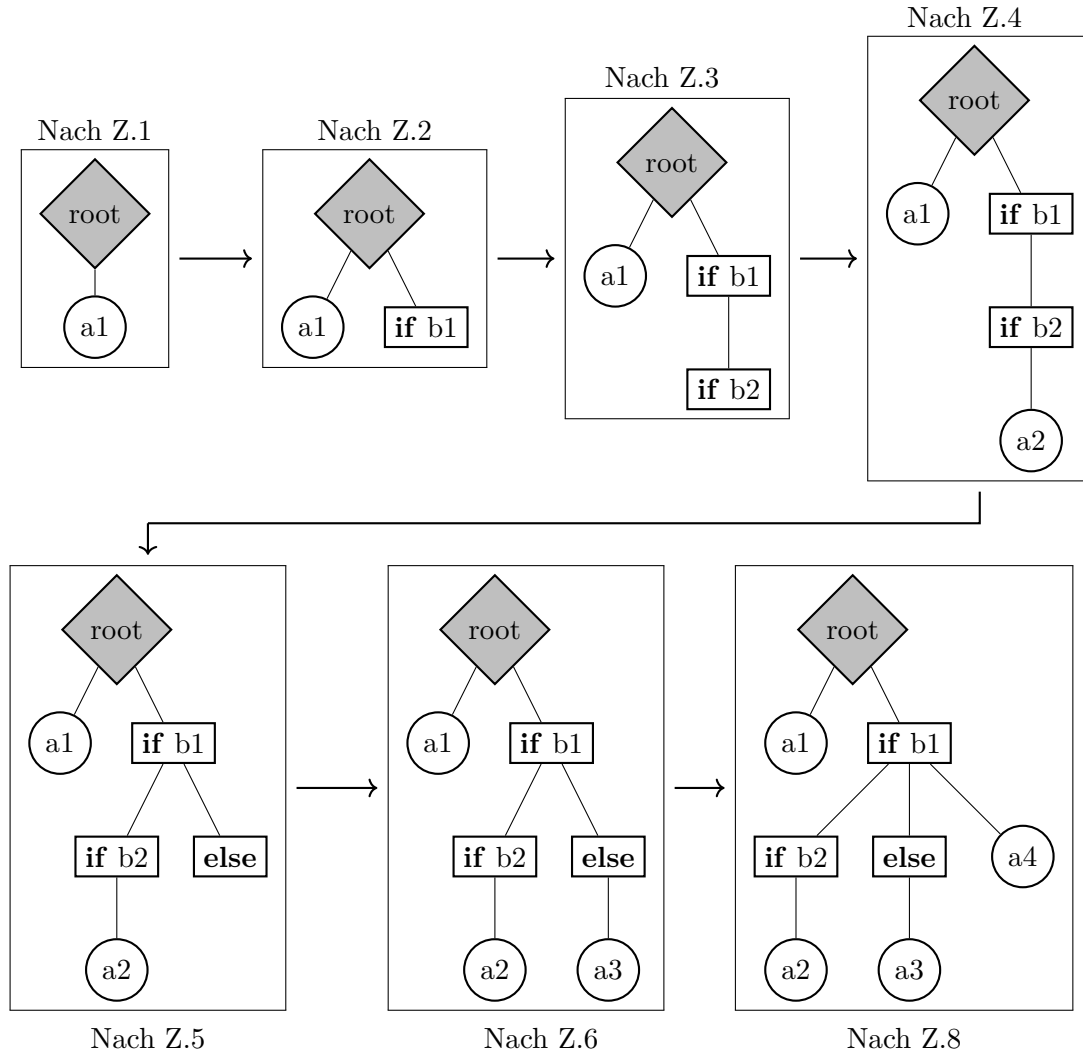


Abbildung 3.1: Beispiel für den Parsen Algorithmus von Viegner

Um weitere Informationen zu finden, die wehrend des Parse Vorgangs verloren gehen, müssen wir die Definition von Variation-Tree und Variation-Diff aus dem Schreiben Classifying Edits to Variability in Source Code Bittner et al. [2] betrachten. Variation-Tree wird wie folgt definiert:

**Definition 3.1.** Ein VARIATION-TREE  $(V, E, r, \tau, l)$  ist ein Baum mit Knoten  $V$ , Kanten  $E \subseteq V \times V$  und Wurzelknoten  $r \in V$ . Jede Kante  $(x, y) \in E$  verbindet einen Kinderknoten  $x$  mit seinem Elternknoten  $y$ , bezeichnet mit  $p(x) = y$ . Der Knotentyp  $\tau(v) \in \{\text{ARTIFACT}, \text{MAPPING}, \text{ELSE}\}$  identifiziert einen Knoten  $v \in V$  entweder als Vertreten einer Implementierungsartefakts, einer Merkmalszuordnung oder eines else-Zweig. Der Label  $l(v)$  ist eine aussagenlogische Formel wenn  $\tau(v) = \text{MAPPING}$ , ein Verweis auf ein Implementierungsartefakt, wenn  $\tau(v) = \text{ARTIFACT}$ , oder leer wenn  $\tau(v) = \text{ELSE}$  ist. Der Wurzelknoten  $r$  hat den Typ  $\tau(r) = \text{MAPPING}$  und das Label  $l(r) = \text{TRUE}$ . Ein Knoten  $e$  von Typ  $\tau(e) = \text{ELSE}$  kann nur unterhalb eines Nichtwurzelknotens  $v$  mit dem Typ  $\tau(v) = \text{MAPPING}$  platziert, dabei hat ein Knoten  $w$  von Typ  $\tau(w) = \text{MAPPING}$  höchstens einen Knoten  $u$  von dem Typ  $\tau(u) = \text{ELSE}$ .

Der Variation-Diff wird dabei wie folgt definiert:

**Definition 3.2.** Ein VARIATION-DIFF ist ein gerichteter, zusammenhängender, azyklischer Graph  $D = (V, E, r, \tau, l, \Delta)$ , welcher einen Wurzelknoten hat, mit Knoten  $V$ , Kanten  $E \subseteq V \times V$ ,

Wurzelknoten  $r \in V$ , Knotentyp  $\tau$ , Knotenlabel  $l$  und einer Funktion  $\Delta : V \cup E \rightarrow \{+, -, \circ\}$ , die definiert, ob ein Knoten oder eine Kante hinzugefügt  $+$  wurde, entfernt  $-$  wurde oder unverändert  $\circ$  geblieben ist, so das  $\text{PROJECT}(D, t)$  ein Variation-Tree für alle Zeiten  $t \in \{a, b\}$  ist. Dies ist wie folgt definiert  $\text{PROJECT}((V, E, r, \tau, l, \Delta), t) := (\{v \in V \mid \text{EXISTS}(t, \Delta(v))\}, \{e \in E \mid \text{EXISTS}(t, \Delta(e))\}, r, \tau, l)$ . Dabei bedeutet  $\text{EXISTS}$  für  $x \in V \cup E$  folgendes,  $\text{EXISTS}(t, \Delta(x)) := (t = b \wedge \Delta(x) \neq +) \vee (t = a \wedge \Delta(x) \neq -)$ .

Aus der gegebenen Information, lässt sich herausfinden, welche Informationen während des Parsens verlorengehen. Aus der Definition von Variation-Tree dass beim Mapping Informationen verloren gehen. Der Informationsverlust geschieht wie folgt, die Ausgangsbedingung (z.B.  $!A \ \&\& \ B$ ) aus Bedingungsanweisungen ( $\#if \ !A \ \&\& \ B$ ) in aussagenlogische Formel (z.B.  $\neg A \wedge B$ ) umgewandelt wird und somit die Ausgangsbedingung in Bedingungsanweisung verlorengeht. In dem Knoten des Variation-Trees ist dann die modifizierte Bedingungsanweisung (z.B.  $\#if \ \neg A \wedge B$ ) enthalten. Die ursprüngliche Information ist dann nicht mehr in dem Variation-Tree enthalten und muss beim Unparsen wiederhergestellt werden. Für Variation-Diff verhält es sich gleich, da der Definition nach dort das mapping genauso wie in Variation-Trees verhält. Aus dem Parse-Algorithmus lässt sich entnehmen, dass die Position und die Einrückung ausgehend von Zeilenbeginn, der  $\#endif$  Anweisungen verlorengeht. Das geschieht, da der Algorithmus sich nichts zu dieser Anweisung merkt im Variation-Diff/Tree und auch keine Knoten gibt, welcher diese Anweisung darstellt. Das führt uns dazu dass diese Informationen auch von uns wiedergewonnen werden müssen. Damit haben wir herausgefunden, welche Informationen während des Parsens verlorengehen und die wir wiederherstellen müssen um das Unparsen zu bewerkstelligen.

## 3.2 Unser Algorithmus

### 3.3 Laufzeitanalyse

#### 3.3.1 Laufzeitanalyse für Unparsen von Variation-Trees

#### 3.3.2 Laufzeitanalyse für Unparsen von Variation-Diffs

### 3.4 Metrik

	Variation-Tree	Variation-Diff
Syntaktische Gleichheit	$C$ = C-Präprozessor- Annotierter Code $C_p = \text{parse}(C)$ $C_{pu} = \text{unparse}(C_p)$ $\text{stringEquals}(C, C_{pu}) == \text{True}$	$D$ = Textbasierter Diff $D_p = \text{parse}(D)$ $D_{pu} = \text{unparse}(D_p)$ $\text{stringEquals}(D, D_{pu}) == \text{True}$
Syntaktische Gleichheit ohne Whitespace	$C$ = C-Präprozessor- Annotierter Code $C_p = \text{parse}(C)$ $C_{pu} = \text{unparse}(C_p)$ $C_w = \text{deleteWhitespace}(C)$ $C_{puw} = \text{deleteWhitespace}(C_{pu})$ $\text{stringEquals}(C_w, C_{puw}) == \text{True}$	$D$ = Textbasierter Diff $D_p = \text{parse}(D)$ $D_{pu} = \text{unparse}(D_p)$ $D_w = \text{deleteWhitespace}(D)$ $D_{puw} = \text{deleteWhitespace}(D_{pu})$ $\text{stringEquals}(D_w, D_{puw}) == \text{True}$
Semantische Gleichheit	Out of Scope unentscheidbar für C exponentielles Wachs- tum für CPP	$D$ = Textbasierter Diff $\text{SynGl} = \text{Syntaktische Gleichheit}$ $\text{SynGLOW} = \text{Syntaktische Gleichheit ohne Whitespace}$ $D_p = \text{parse}(D)$ $D_{pu} = \text{unparse}(D_p)$ Für $\forall t \in \{a, b\}$ $p_1 = \text{textProject}(D, t)$ $p_2 = \text{textProject}(D_{pu}, t)$ $\text{SynGl}(p_1, p_2) == \text{True} \vee$ $\text{SynGLOW}(p_1, p_2) == \text{True}$



Nachdem wir eine algorithmische Lösung für das Problem ausgearbeitet haben, müssen wir entscheiden ob unsere Lösung korrekt ist. Um die Kriterien an denen die Korrektheit festgelegt wird, wird es in folgenden gehen. Wir stellen Ihnen unsere Metrik für die Korrektheit des Unparsens. Wir haben uns für drei mögliche Korrektheitsstufen entschieden, an denen wir die Korrektheit entscheiden. Diese Stufen sind syntaktische Gleichheit, syntaktische Gleichheit ohne Whitespace und semantische Gleichheit. In der Tabelle 4.1 ist eine Zusammenfassung aufgeführt, wie die

In diesem Abschnitt sprechen wir über die syntaktische Korrektheit. Syntaktische Korrektheit bedeutet das der zu vergleichender Text in jedem Zeichen identisch ist. Der Vergleich auf syntaktische Korrektheit sieht für mit C-Präprozessor-Annotierter Code und textbasierte Diffs gleich aus. Hierfür muss der ausgangs C-Präprozessor-Annotierter Code bzw. der textbasierter Diff mit dem Ergebnis nach dem Parse und Unparse Schritt in jedem Zeichen übereinstimmen.

	Variation-Tree ↓ C-Präprozessor- Annotierter Code	Variation-Diff ↓ textbasierter Diff
Syntaktische Gleichheit	$C$ = C-Präprozessor- Annotierter Code $C_p = \text{parse}(C)$ $C_{pu} = \text{unparse}(C_p)$ $\text{stringEquals}(C, C_{pu}) == \text{True}$	$D$ = Textbasierter Diff $D_p = \text{parse}(D)$ $D_{pu} = \text{unparse}(D_p)$ $\text{stringEquals}(D, D_{pu}) == \text{True}$
Syntaktische Gleichheit ohne Whitespace	$C$ = C-Präprozessor- Annotierter Code $C_p = \text{parse}(C)$ $C_{pu} = \text{unparse}(C_p)$ $C_w = \text{deleteWhitespace}(C)$ $C_{puw} = \text{deleteWhitespace}(C_{pu})$ $\text{stringEquals}(C_w, C_{puw}) == \text{True}$	$D$ = Textbasierter Diff $D_p = \text{parse}(D)$ $D_{pu} = \text{unparse}(D_p)$ $D_w = \text{deleteWhitespace}(D)$ $D_{puw} = \text{deleteWhitespace}(D_{pu})$ $\text{stringEquals}(D_w, D_{puw}) == \text{True}$
Semantische Gleichheit	Out of Scope unentscheidbar für C exponentielles Wachs- tum für CPP	$D$ = Textbasierter Diff SynGl = Syntaktische Gleich- heit SynGLOW = Syntaktische Gleichheit ohne Whitespace $D_p = \text{parse}(D)$ $D_{pu} = \text{unparse}(D_p)$ Für $\forall t \in \{a, b\}$ $p_1 = \text{textProject}(D, t)$ $p_2 = \text{textProject}(D_{pu}, t)$ $\text{SynGl}(p_1, p_2) == \text{True} \vee$ $\text{SynGLOW}(p_1, p_2) == \text{True}$

Tabelle 4.1: Metrik für die Korrektheit



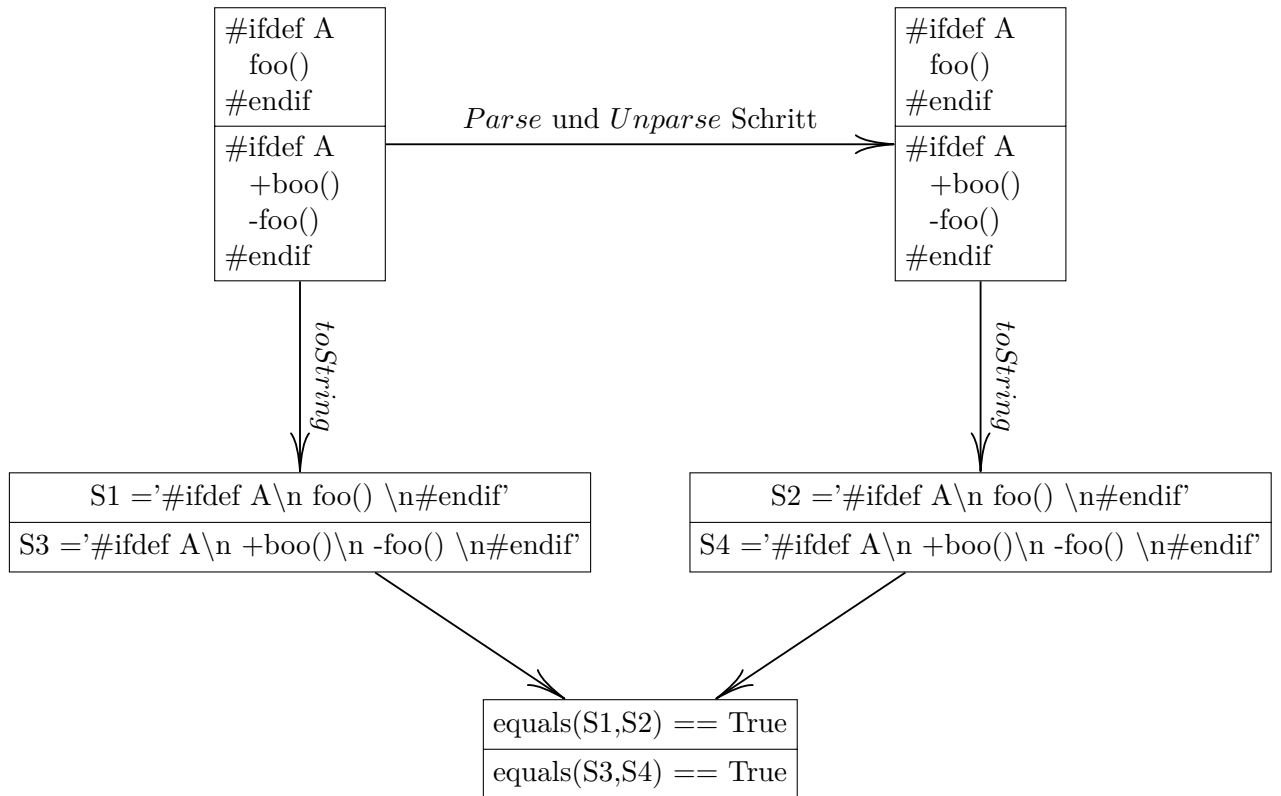


Abbildung 4.1: Beispiel für Syntaktische Gleichheit

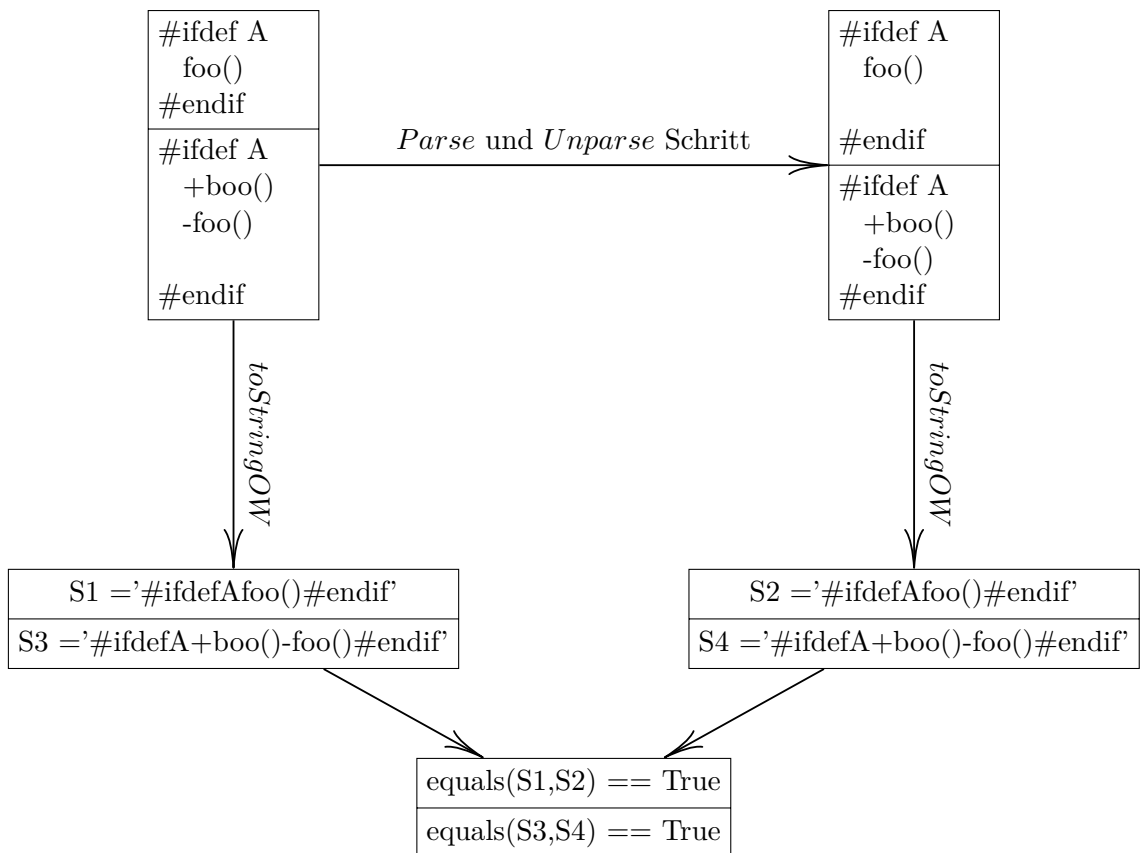


Abbildung 4.2: Beispiel für Syntaktische Gleichheit ohne Whitespace

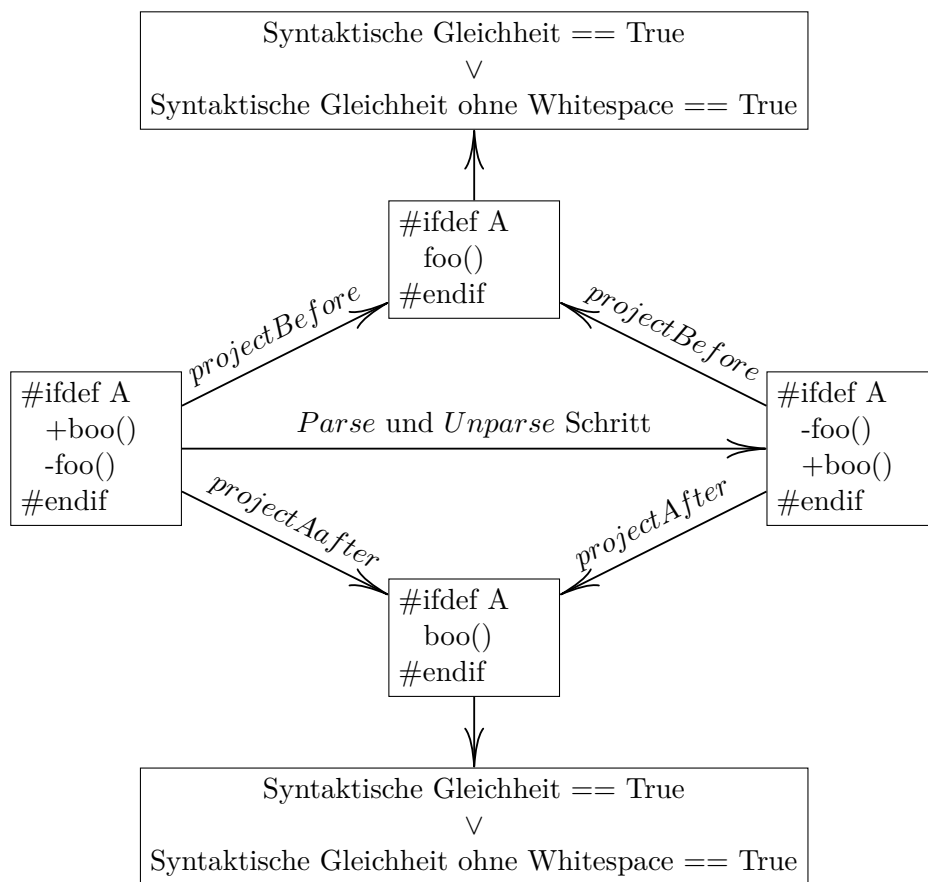


Abbildung 4.3: Beispiel für Semantische Gleichheit

## Implementierung

5.1 Code

5.2 Test



# Literaturverzeichnis

- [1] S. Apel, D. Batory, C. Kästner, and G. Saake, *Feature-Oriented Software Product Lines*. Berlin, Heidelberg: Springer, 2013.
- [2] P. M. Bittner, C. Tinnes, A. Schultheiß, S. Viegner, T. Kehrer, and T. Thüm, “Classifying Edits to Variability in Source Code,” in *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*. New York, NY, USA: ACM, Nov. 2022, pp. 196–208.
- [3] S. Viegner, “Empirical Evaluation of Feature Trace Recording on the Edit History of Marlin,” Bachelor’s Thesis, University of Ulm, Germany, Apr. 2021. [Online]. Available: [https://oparu.uni-ulm.de/xmlui/bitstream/handle/123456789/38679/BA\\_Viegner.pdf](https://oparu.uni-ulm.de/xmlui/bitstream/handle/123456789/38679/BA_Viegner.pdf)