



**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, 23. September 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>7</b>
2.1	C-Präprozessor . . . . .	7
2.2	Variabilität Umsetzung mit C-Präprozessor . . . . .	8
<b>3</b>	<b>Unparse Algorithmus</b>	<b>11</b>
3.1	Variation-Tree und Variation-Diff . . . . .	11
3.2	Parser . . . . .	13
3.3	Verlorengehende Informationen . . . . .	17
3.4	Modifizierte Variation-Tree und Variation-Diff . . . . .	17
3.5	Verlorengehende Informationen . . . . .	17
3.6	Unser Algorithmus . . . . .	22
3.7	Laufzeitanalyse . . . . .	24
3.7.1	Laufzeitanalyse für Unparsen von Variation-Trees . . . . .	24
3.7.2	Laufzeitanalyse für Unparsen von Variation-Diffs . . . . .	24
<b>4</b>	<b>Metrik</b>	<b>25</b>
<b>5</b>	<b>Implementierung</b>	<b>31</b>
5.1	Code . . . . .	31
5.1.1	Parser von Variation-Trees mit Heuristik . . . . .	31
5.1.2	Parser von Variation-Trees mit Speicherung . . . . .	31
5.1.3	Parser von Variation-Diffs . . . . .	31
5.2	Test . . . . .	31
	<b>Bibliography</b>	<b>32</b>





# Einleitung

Bei der Entwicklung von konfigurierbaren Softwaresystemen, wie zum Beispiel Clone-and-Own, oder Softwareproduktlinien, gibt es im Laufe des Lebenszyklus immer mehr Features. Es ist von Vorteil, eine Möglichkeit zu haben, die Features im Code zu unterscheiden und automatisch zu finden. Einige Möglichkeiten dazu wären Präprozessor-Annotationen, oder Build-Systeme [2]. Wie bei der Clone-and-Own-Entwicklung, wo für jede Variante der Software eine neue Kopie der gesamten Software angelegt wird und parallel entwickelt wird [3]. Dort müssen Features gefunden werden, um diese zu aktualisieren [3, 8, 10, 9, 11, 23].

Die Entwickler sind bei der Entwicklung von konfigurierbarer Software daran interessiert, zu verstehen, wie sich ihre Änderungen auf die Variabilität auswirken und wie die Variabilität von konfigurierbarer Software aussieht [3]. Sonst wenn man das Verständnis über die Auswirkungen der Änderung nicht hat, kann das zu Fehlern und Problemen bei der Entwicklung führen [3, 13, 14, 17, 20, 7]. Dies stellt einen Aspekt einer größeren Aufgabe dar, der Aufrechterhaltung und Weiterentwicklung von Informationen über Variabilität bei Quellcodeänderungen [3]. Für die Entwickler stellt diese Aufgabe eine große Herausforderung dar [3, 15, 16, 18].

Der C-Präprozessor ist eine Möglichkeit, Variabilität zu erzeugen [2]. Der C-Präprozessor ist ein Tool, das den Quellcode vor dem Kompilieren manipuliert [2]. Dieses Tool bietet Möglichkeiten zur Dateieinbindung, zu lexikalische Makros, und zur bedingte Kompilierung [2]. Wie ein mit dem C-Präprozessor annotierter Code aussehen kann, ist in der Abbildung 1.1 Stelle ⑤ zu sehen (Abb. 1.1 St.⑤). Um die Variabilität mithilfe des C-Präprozessors zu erzeugen, brauchen wir dessen Möglichkeit zur bedingten Kompilierung [2]. Dabei können beliebige Aussageformeln über Features im Quellcode mit den C-Präprozessor-Anweisungen `#if`, `#ifdef` und, `#ifndef` abgebildet werden [3] (Abb. 1.1 St.⑤).

Zur Unterstützung der Variabilitätsanalyse kann man Tools verwenden [19, 22], wie zum Beispiel DiffDetective. DiffDetective ist eine Java-Bibliothek [5]. Der Zweck von DiffDetective ist es, Änderungen im Quellcode und Änderungen der Variabilität darstellbar und den Zusammenhang zwischen ihnen analysierbar zu machen. DiffDetective stellt einen variabilitätsbezogenen Differencer [5, 3] zur Verfügung, der sich nur auf Aspekte im Code/Text bezieht, welche die Variabilität berücksichtigen. Diese Bibliothek ermöglicht auch die Analyse der Versionshistorie von Softwareproduktlinien [3] und bietet daher einen flexiblen Rahmen für großangelegte empirische Analysen von Git-Versionsverläufen statisch konfigurierbarer Software [5, 4].

Zentral für DiffDetective sind zwei formal verifizierte Datenstrukturen für Variabilität und Änderungen an dieser [3]. Das sind Variation-Trees (Abb. 1.1 St.ⓧ) für variabilitätsbezogenen Code (Abb. 1.1 St.Ⓥ) und Variation-Diffs (Abb. 1.1 St.Ⓨ) für variabilitätsbezogene Diffs (Abb. 1.1 St.Ⓦ). Diese Datenstrukturen sind generisch. Das bedeutet, dass die Datenstrukturen möglichst von der Umsetzung der Variabilität im Code abstrahieren. Also kann eine Umsetzungsmöglichkeit leicht durch eine andere ersetzt werden, zum Beispiel können C-Präprozessor-Annotationen durch Java-Präprozessor-Annotationen ohne oder geringer Änderungen an den Datenstrukturen selbst, ersetzt werden. Ein Variation-Tree ist ein Baum, welcher die Verzweigungen/Variationen eines annotierten Codes darstellt [5, 3, 4]. Ein Variation-Diff ist ein Graph, welcher die Unterschiede zwischen zwei Variation-Trees zeigt [5, 3, 4]. In beiden Fällen werden die Bedingungsknoten, welche Informationen zu Variabilität erhalten, und die Code-Knoten unterschieden. Beim Variation-Diff sind zudem die eingefügten Knoten, die gelöschten Knoten und, die unveränderten Knoten zu unterscheiden.

Das Parsen führt die Eingabe von der konkreten Syntax in die abstrakte Syntax um. In unserem Fall parst DiffDetective C-Präprozessor-Annotationen, dieses kann aber auch auf andere Präprozessor-Annotationen erweitert werden. Beim Parsen wird nur der C-Präprozessor-Annotierter Code in seine abstrakte Syntax überführt, der C- bzw. C++-Code wird als Text behandelt und wird nicht geparkt. Das Parsen in DiffDetective funktioniert für Variation-Trees und für Variation-Diffs über einen einzigen gemeinsamen Algorithmus. Der Algorithmus arbeitet wie folgt: Er geht über alle Zeilen des Codes/Textes und schaut sich für jede Zeile an, wie diese Zeile manipuliert wurde, ob die Zeile unverändert geblieben ist, gelöscht wurde, oder neu ist [21]. Dazu wird festgelegt von welchem Typ die Zeile ist, also ob diese Zeile C/C++ Code enthält oder eine C-Präprozessor-Kontrollstruktur. Als Nächstes wird geprüft, ob die Zeile eine #endif-Annotation enthält. Wenn ja, dann weist das darauf hin, dass ein Bedingungsblock zu Ende ist. Wenn die Zeile kein #endif enthält, dann wird ein neuer Knoten mit Informationen über Elternknoten, den Typ der Zeile und, wie die Zeile manipuliert wurde, erstellt. Wenn dieser Knoten kein Code-Knoten ist, wird er gemerkt und für die Angabe der Elternknoten verwendet. Der Algorithmus ist an sich für das Parsen von textbasierten Diffs in Variation-Diffs ausgelegt (Abb. 1.1 St.Ⓟ). An den Stellen ① und ⑨ wird anders vorgegangen, da wir als Eingabe ein C-Präprozessor Code (Abb. 1.1 St.Ⓥ) haben und als Ausgabe ein Variation-Tree (Abb. 1.1 St.ⓧ). Der gegebene Algorithmus ist für das direkte Parsen von C-Präprozessor Code nicht ausgelegt. Deshalb wurde dort Umwege verwendet, um diesen Algorithmus anwendbar zu machen und die benötigte Ausgabe zu erzielen. Ein Text kann in ein nicht verändertes, textbasiertes Diff umgewandelt werden, durch die Bildung eines Diffs mit sich selbst. Dadurch ist es möglich aus C-Präprozessor Code (Abb. 1.1 St.Ⓥ) ein textbasiertes Diff (Abb. 1.1 St.Ⓦ) zu erzeugen, also wurden die Stelle ⑪ oder ⑫ verwendet. Da jetzt ein textbasiertes Diff vorhanden ist, kann der Algorithmus darauf angewandt werden (Abb. 1.1 St.Ⓟ). Um aus dem erhaltenen Variation-Diff (Abb. 1.1 St.Ⓨ) ein Variation-Tree zu bekommen, muss man die Stelle ④ oder ⑧ aus der Abbildung 1.1 anwenden. So sieht man das die Stelle ① durch die Stellen ⑪,⑤,④ [ ① → ⑪,⑤,④ ] und die Stelle ⑨ durch die Stellen ⑫,⑤,⑧ [ ⑨ → ⑫,⑤,⑧ ] ersetzt werden kann.

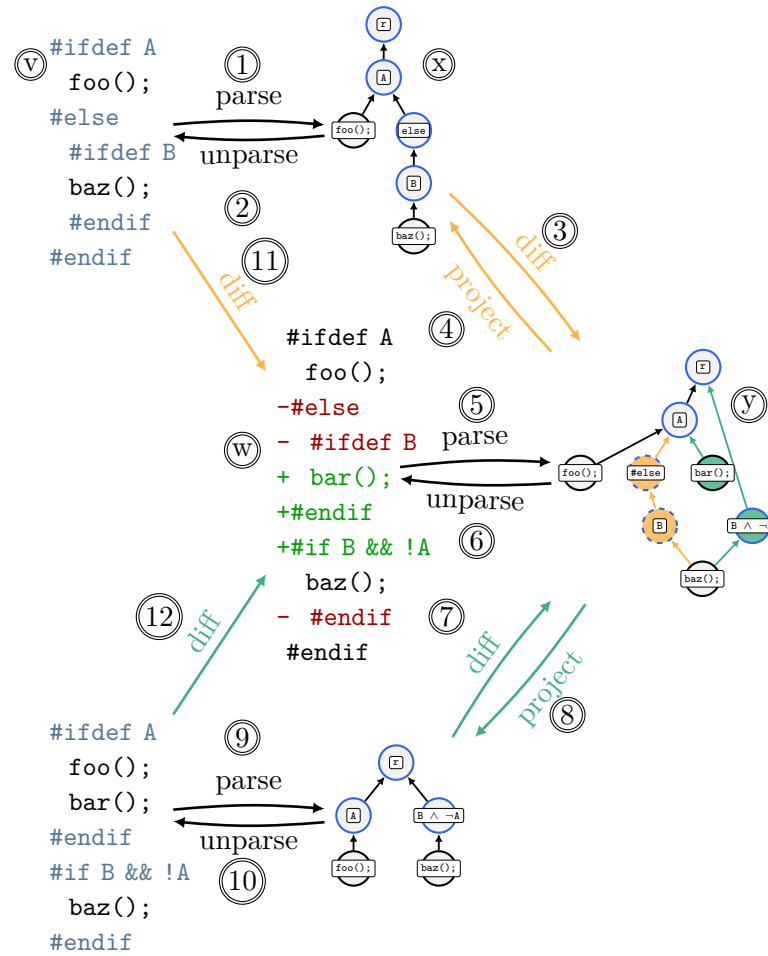


Abbildung 1.1: Überblick über Variabilität bezogene Konvertierungen

Obwohl DiffDetective Funktionen zum Parsen (Abb. 1.1 St. ①, ⑤, ⑨) hat, besitzt dieses Tool keine Funktion zum Unparsen (Abb. 1.1 St. ②, ⑥, ⑩) von Variation-Trees und Variation-Diffs. Das Unparsen ist die Überführung aus der abstrakten Syntax in die konkrete Syntax, also ist das Unparsen die Invertierung des Parsens. Unser Ziel ist es, das zu ändern. Dazu müssen wir einen Unparser entwickeln, welcher auf direktem oder indirektem Wege, Variation-Trees (Abb. 1.1 St. ⑩) in C-Präprozessor-Annotierten Code (Abb. 1.1 St. ⑤) und Variation-Diffs (Abb. 1.1 St. ⑥) in textbasierte Diffs (Abb. 1.1 St. ⑦) überführt.

Eine Einsatzmöglichkeit des Unparsers wäre, das Unparsen von Variation-Trees, bei denen die Variabilität mutiert wurde. Eine Möglichkeit zu Analyse von Softwareproduktlinien ist Mutation-Tests. Bei Mutation-Tests werden Mutation-Operatoren verwendet, welche aber nur auf der abstrakten Ebene, also auf Variation-Trees, angewandt werden können [1]. Um weiter in der Analyse vorzugehen, muss man von der abstrakten Ebene zu der konkreten Ebene übergehen und hier wird der Unparser angewandt. Eine andere Einsatzmöglichkeit wäre die Verwendung von Variation-Diffs als Patches. Wenn ein Patch modifiziert werden muss, um ihn für andere Versionen zu verwenden oder um Änderungen zu sehen, die nur ein bestimmtes Feature betreffen [4].

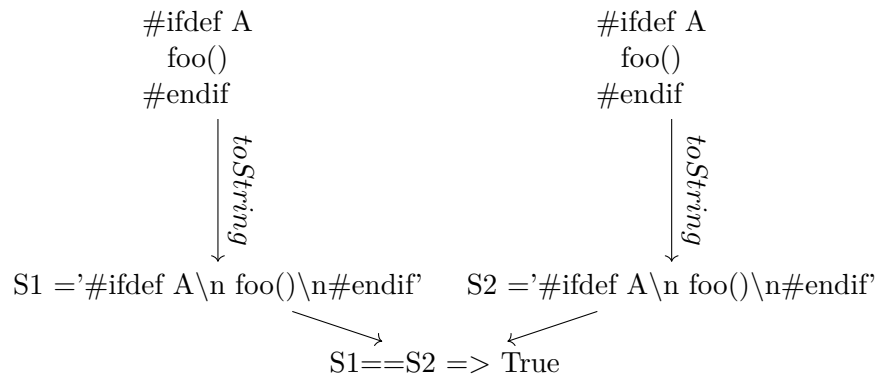
Für das Unparsen stellt das Fehlen einiger Informationen, die im annotierten Code vorhanden sein müssen, aber in Variation-Trees bzw. Variation-Diffs nicht vorhanden sind, das größte

Problem dar. Diese Informationen sind entweder durch das Parsen verloren gegangen oder waren von Anfang an nicht vorhanden, wenn Variation-Trees bzw. Variation-Diffs ohne Parsen gebildet wurden. Diese Informationen sind die exakte Formel, die ein Mapping-Knoten  $\tau(v) = \text{mapping}$  besitzt [3], die Position von `#endif` und deren Einrückung. Aus diesem Grund müssen wir entweder Annahmen treffen, oder DiffDetective so erweitern, dass er diese Information explizit speichert. Eine Annahme könnte sein, dass das `#endif` genauso eingerückt ist, wie die Bedingung, zu der es gehört.

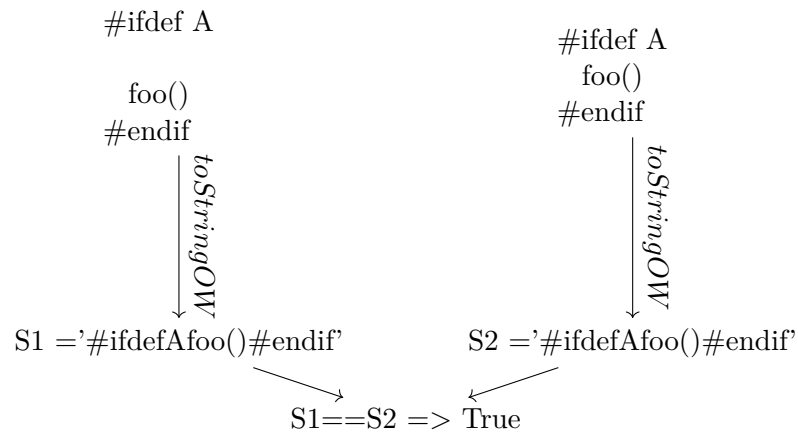
Eine Möglichkeit könnte sein, das Unparsen von Variation-Trees direkt umzusetzen. Dazu muss ein entsprechender Algorithmus entwickelt werden. Für das Unparsen von Variation-Diffs ziehen wir in Betracht indirekt vorzugehen, ähnlich wie bei dem Parsen von C-Präprozessor-Annotierten Code zu Variation-Trees.

Der Beitrag setzt sich aus Konzept, Implementierung und Auswertung zusammen. Beim Konzept wird ein Vorgehen zum Unparsen von Variation-Trees und Variation-Diffs in das ursprüngliche Textformat ausgearbeitet. In der Implementierung wird dieses Vorgehen in das DiffDetective-Tool eingebaut. In der Bachelorarbeit wird eine Metrik spezifiziert, anhand derer die Korrektheit bewertet wird. Zurzeit wird in Betracht gezogen, die Korrektheit, der Implementierung anhand folgender Kriterien festzustellen: syntaktische Gleichheit, syntaktische Gleichheit ohne Whitespace und semantische Gleichheit. Ein ähnliches Kriterium für die Gleichheit bezogen aber auf Variation-Trees bzw. Variation-Diffs ist im Konferenzbeitrag von zu finden [4]. Die syntaktische Gleichheit bedeutet, dass das Vergleichene in jedem Zeichen übereinstimmt, so wie das erste Beispiel in Abbildung 1.2. Das zweite Beispiel der Abbildung 1.2 zeigt die syntaktische Gleichheit ohne Whitespace, bei der das Vergleichene gleich sein muss, wenn man alle Zeichen, die Whitespace sind, entfernen würde. Bei der semantischen Gleichheit muss der Sinn gleich sein, was uns das letzte Beispiel der Abbildung 1.2 zeigt. Das ist wie folgt zu verstehen, zwei Diffs sind semantisch gleich, wenn ihre Projektionen syntaktisch gleich bzw. syntaktisch gleich ohne Whitespace sind. Am Ende der Auswertung wird anhand der vorher spezifizierten Metrik festgelegt, wie korrekt die Implementierung und somit das Vorgehen ist.

### Syntaktische Gleichheit



### Syntaktische Gleichheit ohne Whitespace



### Semantische Gleichheit

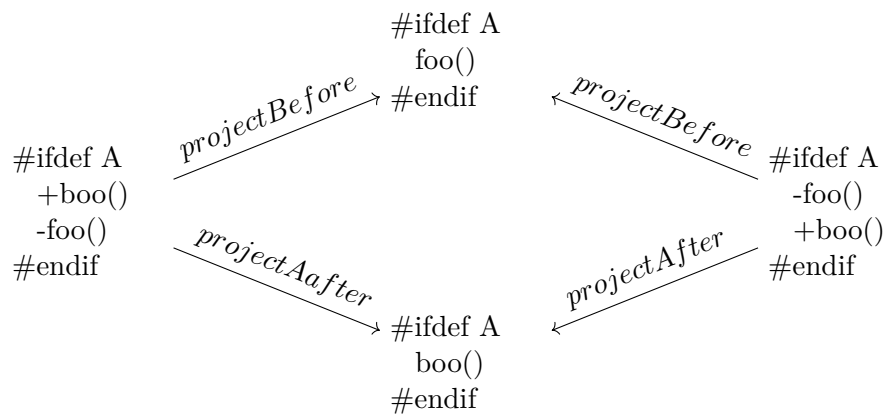


Abbildung 1.2: Beispiel für Metrik



## 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 [2]. Dieses Tool bietet Möglichkeiten zur bedingte Kompilierung, zur Dateieinbindung und zur Erstellung lexikalische Makros [2]. 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 [2]. Um die Variabilität mithilfe des C-Präprozessors zu erzeugen, brauchen wir dessen Möglichkeit zur bedingten Kompilierung [2]. 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 [3]. 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 Patern

## Unparse Algorithmus

### 3.1 Variation-Tree und Variation-Diff

Um verstehen zu können wie wir Variation-Trees und Variation-Diffs unparsen können, müssen wir uns näher mit dehnern befassen. Damit wir dessen Einschränkungen und Möglichkeiten verstehen. Um beurteilen zu können ob und wie die Variation-Trees und Variation-Diffs überhaupt für unseres vorgehen zu gebrauchen sind.

Um Variation-Trees und Variation-Diffs kennenzulernen, betrachten wir zunächst die Definition der Datenstrukturen aus dem Schreiben Classifying Edits to Variability in Source Code [3]. 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.

Wie die Projektion zu verstehen ist, ist hier definiert:

**Definition 3.3.** Die Projektion  $\text{PROJECT}(D, t)$  für das Variation-Diff aus Definition 3.2 ist das Entfernen von  $\Delta$  und den Knoten und Kanten, welche zu der Zeit  $t$  nicht vorhanden sind.  $\text{PROJECT}((V, E, r, \tau, l, \Delta), t) := (\{v \in V \mid \text{EXISTS}(t, \Delta(v))\}, \{e \in E \mid \text{EXISTS}(t, \Delta(e))\}, r, \tau, l)$

Ob ein Knoten oder eine Kante zu einer gegebenen Zeit existiert oder nicht, definieren wir für alle Definitionen die das verwenden Gleich. Unsere Definition von EXISTS entspricht der Definitionen aus dem Schreiben Classifying Edits to Variability in Source Code [3] und der Bachelorarbeit Constructing Variation Diffs Using Tree Diffing Algorithms [12]

**Definition 3.4.** *Ob ein Knoten oder eine Kante zu einer gegebenen Zeit existiert oder nicht, stellt EXISTS für  $x \in V \cup E$  das folgendermaßen fest  $\text{EXISTS}(t, \Delta(x)) := (t = \text{b} \wedge \Delta(x) \neq \text{+}) \vee (t = \text{a} \wedge \Delta(x) \neq \text{-})$*

Das ist aber nicht die einzige Möglichkeit Variation-Tree und Variation-Diff zu definieren. In der Bachelorarbeit Constructing Variation Diffs Using Tree Diffing Algorithms [12] wurden die Variatio-Tree und Variation-Diff etwas anders definiert. Obwohl dort auch Variation-Tree und Variation-Diff definiert werden, werden wir in dieser Arbeit die Definitionen aus Constructing Variation Diffs Using Tree Diffing Algorithms als geordneter Variation-Tree und als geordneter Variation-Diff bezeichnen, die wie folgt aussehen:

**Definition 3.5.** *Ein GEORDNETER VARIATION-TREE  $(V, E, r, \tau, l, O)$  ist ein geordneter Baum mit einem Wurzelknoten  $r \in V$ , mit Knoten  $V$ , Kanten  $E \subseteq V \times V$ , Knotentypen  $\tau : V \rightarrow \{\text{ARTIFACT}, \text{MAPPING}, \text{ELSE}\}$ , Label  $l : V \rightarrow A \cup P$ , wobei  $A$  die Menge aller Artefakte ist und  $P$  die Menge aller aussagenlogischer Formeln und eine injektive Funktion  $O : V \rightarrow \mathbb{N}$ , die eine Ordnung für die Kinder eines jeden Knotens jedes Knotens definiert. Der Wurzelknoten  $r$  muss den Typ  $\tau(r) = \text{MAPPING}$  und Label  $l(r) = \text{true}$  haben. Ein Knoten  $v$  mit  $\tau(v) = \text{ARTIFACT}$  wird als Artefaktknoten und muss ein Artefakt  $a \in A$  als Label  $l(v) = a$  haben. Analog dazu wird ein Knoten  $v$  mit  $\tau(v) = \text{MAPPING}$  als Mappingknoten bezeichnet und muss eine Merkmals Abbildung, eine propositionale Formel  $p \in P$  als Label  $l(v) = p$  haben. Im Gegensatz dazu hat ein Knoten  $v$  mit  $\tau(v) = \text{ELSE}$  ein leeres Label, wird als else-Knoten bezeichnet und muss der einzige else-Knoten eines if-Knotens sein.*

**Definition 3.6.** *Ein GEORDNETER VARIATION-DIFF  $D = (V, E, r, \tau, l, \Delta, O_{\text{before}}, O_{\text{after}})$  ist ein gerichteter, zusammenhängender, azyklischer Graph, welcher einen Wurzelknoten hat, mit Knoten  $V$ , Kanten  $E \subseteq V \times V$ , Wurzelknoten  $r \in V$ , Knotentypen  $\tau : V \rightarrow \{\text{ARTIFACT}, \text{MAPPING}, \text{ELSE}\}$ , Label  $l : V \rightarrow A \cup P$ , wobei  $A$  die Menge aller Artefakte ist und  $P$  die Menge aller aussagenlogischer Formeln, die Zeit der Existenz  $\Delta : V \cup E \rightarrow \{+, -, \circ\}$ , die definiert, ob ein Knoten oder eine Kante hinzugefügt  $+$  wurde, entfernt  $-$  wurde oder unverändert  $\circ$  geblieben ist, die Kinderknoten vor der Änderung  $O_{\text{before}}$  und nach der Änderung  $O_{\text{after}}$  sind eine injektive Funktion  $O_{\text{before}}, O_{\text{after}} : V \rightarrow \mathbb{N}$ . Die Projektionen  $\text{project}_O(D, t)$  müssen für alle Zeiten  $t \in \{\text{after}, \text{before}\}$  ein Variation-Tree mit dem selben Wurzelknoten sein.*

Für die Eindeutigkeit halber haben wir auch die Projektion umbenannt, da sich die Definition von  $\text{project}_O(D, t)$  von der Definition der Projektion  $\text{PROJECT}(D, t)$  aus der Definition 3.3 unterscheidet, wegen den zusätzlichen Information, die gespeichert werden.

**Definition 3.7.** *Die Projektion  $\text{project}_O(D, t)$  eines geordneten Variation-Diff  $D = (V, E, r, \tau, l, \Delta, O_{\text{before}}, O_{\text{after}})$  zum Zeitpunkt  $t \in \{\text{after}, \text{before}\}$  ist definiert als :  $\text{project}_O(D, t) := (V', E', r, \tau|_V, l|_V, O_V)$ , wobei  $V' = \{v \in V \mid \text{EXISTS}(t, \Delta(v))\}$ ,  $E' = \{e \in E \mid \text{EXISTS}(t, \Delta(e))\}$  und die Existenz von  $d \in \{+, -, \circ\}$ , zu der Zeit  $t \in \{\text{after}, \text{before}\}$  ist in der Definition 3.4 gegeben.*

Die Definitionen von normalen Variation-Tree und Variation-Diff sind sehr ähnlich zu den Definitionen von geordneten Variation-Tree und Variation-Diff. Wir sehen hier die Möglichkeit geordnete Variation-Tree bzw. Variation-Diff in normale Variation-Tree bzw. Variation-Diff umzuwandeln. Dies wird wie folgt angestellt: für die Umwandlung von geordneter Variation-Tree zu Variation-Tree  $\text{reduce}_{OVT}((V, E, r, \tau, l, O)) := (V, E, r, \tau, l)$  und die Umwandlung von

geordneter Variation-Diff zu Variation-Diff ist  $reduce_{OVD}((V, E, r, \tau, l, \Delta, O_{before}, O_{after})) := (V, E, r, \tau, l, \Delta)$ . Es bleibt uns nur noch zu zeigen dass die Reihenfolge der Anwendung nicht von Bedeutung ist. Es muss also für ein geordneten Variation-Diff  $D$  und eine Zeit  $t \in \{before, after\}$  gelten  $reduce_{OVT}(project_O(D, t)) = project(reduce_{OVD}(D), t)$ . Wir werden zeigen das diese Aussage gilt. Der Beweis ist unten zu sehen.

$$\begin{aligned}
 & reduce_{OVT}(project_O(D, t)) = \\
 & reduce_{OVT}(project_O((V, E, r, \tau, l, \Delta, O_{before}, O_{after}), t)) = \\
 & reduce_{OVT}((\{v \in V | \text{EXISTS}(t, \Delta(v))\}, \{e \in E | \text{EXISTS}(t, \Delta(e))\}, r, \tau, l, O_t)) = \\
 & (\{v \in V | \text{EXISTS}(t, \Delta(v))\}, \{e \in E | \text{EXISTS}(t, \Delta(e))\}, r, \tau, l) = \\
 & project((V, E, r, \tau, l, \Delta), t) = \\
 & project(reduce_{OVD}(V, E, r, \tau, l, \Delta, O_{before}, O_{after}), t) = \\
 & project(reduce_{OVD}(D), t)
 \end{aligned}$$

□

Jetzt haben wir uns mit dem Beschäftigt wie Variation-Tree und Variation-Diff zu verstehen sind. Dabei haben wir zwei Definitionen von Variation-Tree bzw. Variation-Diff kennengelernt, die sich sehr ähnlich sind aber auch einen Unterschied haben. Was zu Folge hat das sich gewisse Eigenschaften von den unterscheiden. Dazu haben wir gezeigt das es möglich geordnete Variation-Tree bzw. Variation-Diff in Variation-Tree bzw. Variation-Diff überführen und projizieren in beliebiger Reihenfolge anzuwenden, was die Abbildung 3.1 ergibt. Im späteren Verlauf können wir bei Bedarf diese Unterschiede für unsere Zwecke verwenden. Um zu verstehe welche Definition für uns die Angemessenere ist, also welche sich besser für das Unparse eignet.

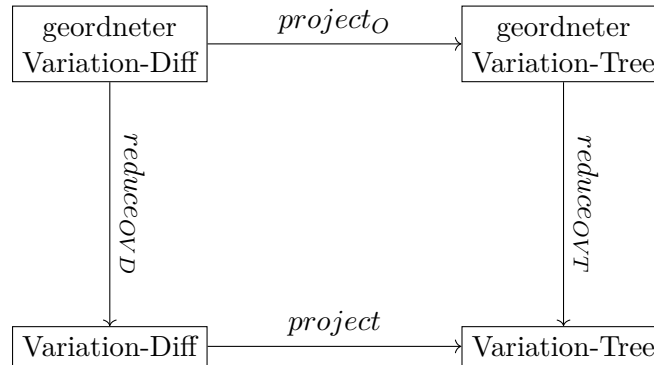


Abbildung 3.1: Transformationen von geordneten Variation-Diff , geordneten Variation-Tree, Variation-Diff und Variation-Tree

## 3.2 Parser

Jetzt beschäftigen wir uns damit wie Variation-Trees bzw Variation-Diffs aus mit C-Präprozessor-Annotierter Code bzw. textbasierter Diff erstellt werden. Nachdem wir uns mit dem Beschäftigt haben was wir genau unter Variation-Tree und Variation-Diff verstehen. Dazu schauen wir und den Parser Algorithmus von Viegner [21] an.

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 Zeil entnommen, die der Knoten repräsentiert und gelten damit auf für Zeilen eines textbasierten Diffs. Der Diff-Type 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, dass 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 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 das 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

```

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

```

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 hinzugefügt, 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.



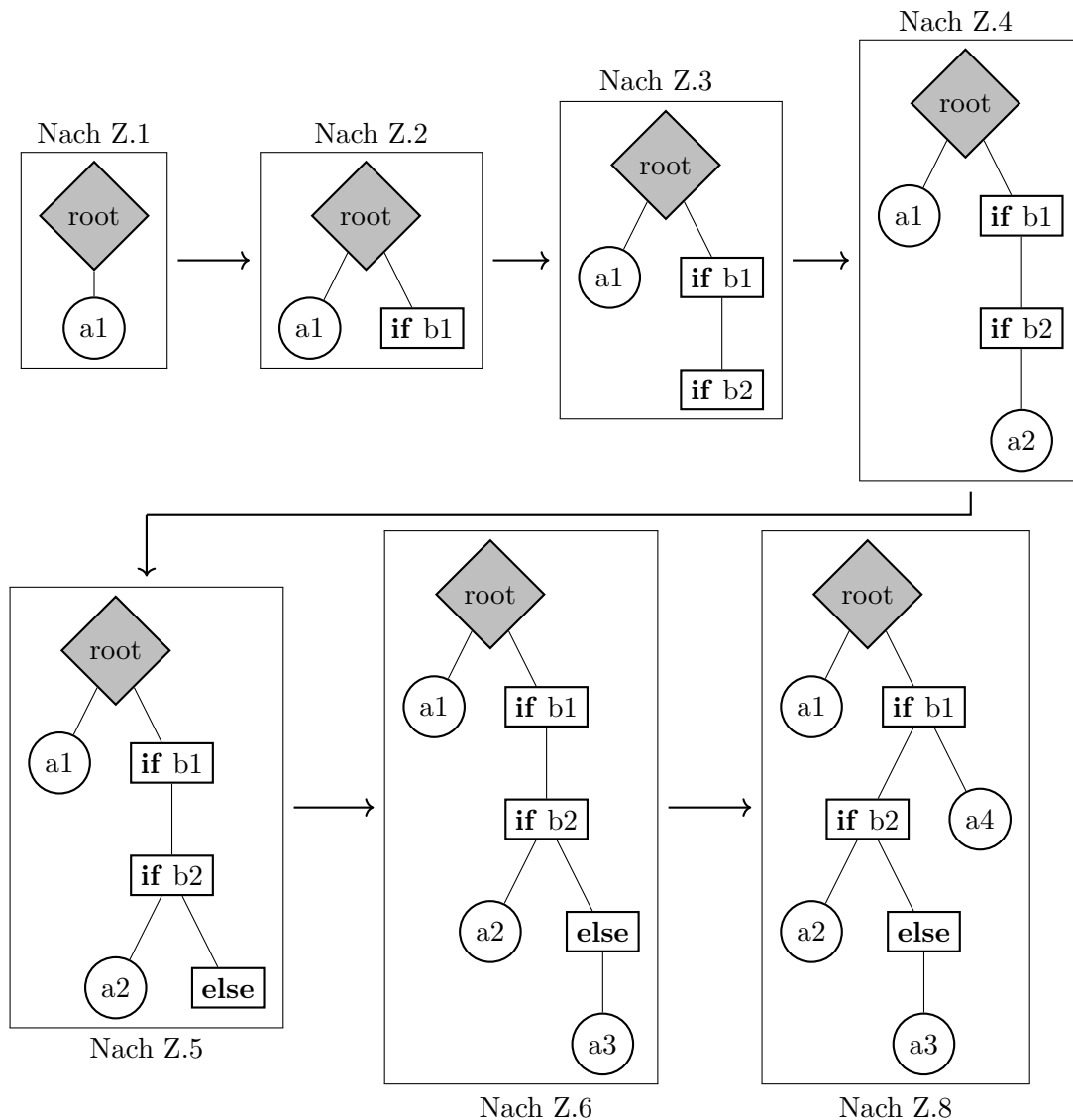


Abbildung 3.2: Beispiel für den Parsen Algorithmus von Viegner

### 3.3 Verlorengelungende Informationen

### 3.4 Modifizierte Variation-Tree und Variation-Diff

### 3.5 Verlorengelungende Informationen

Um zu verstehen was für Informationen verloren werden und der Unparser wiederherstellen muss, betrachten wir im folgenden den Parser Algorithmus von Viegner [21]. 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-Type 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, dass 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 an 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 2:** 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 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 das 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

```

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

```

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 hinzugefügt, 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.

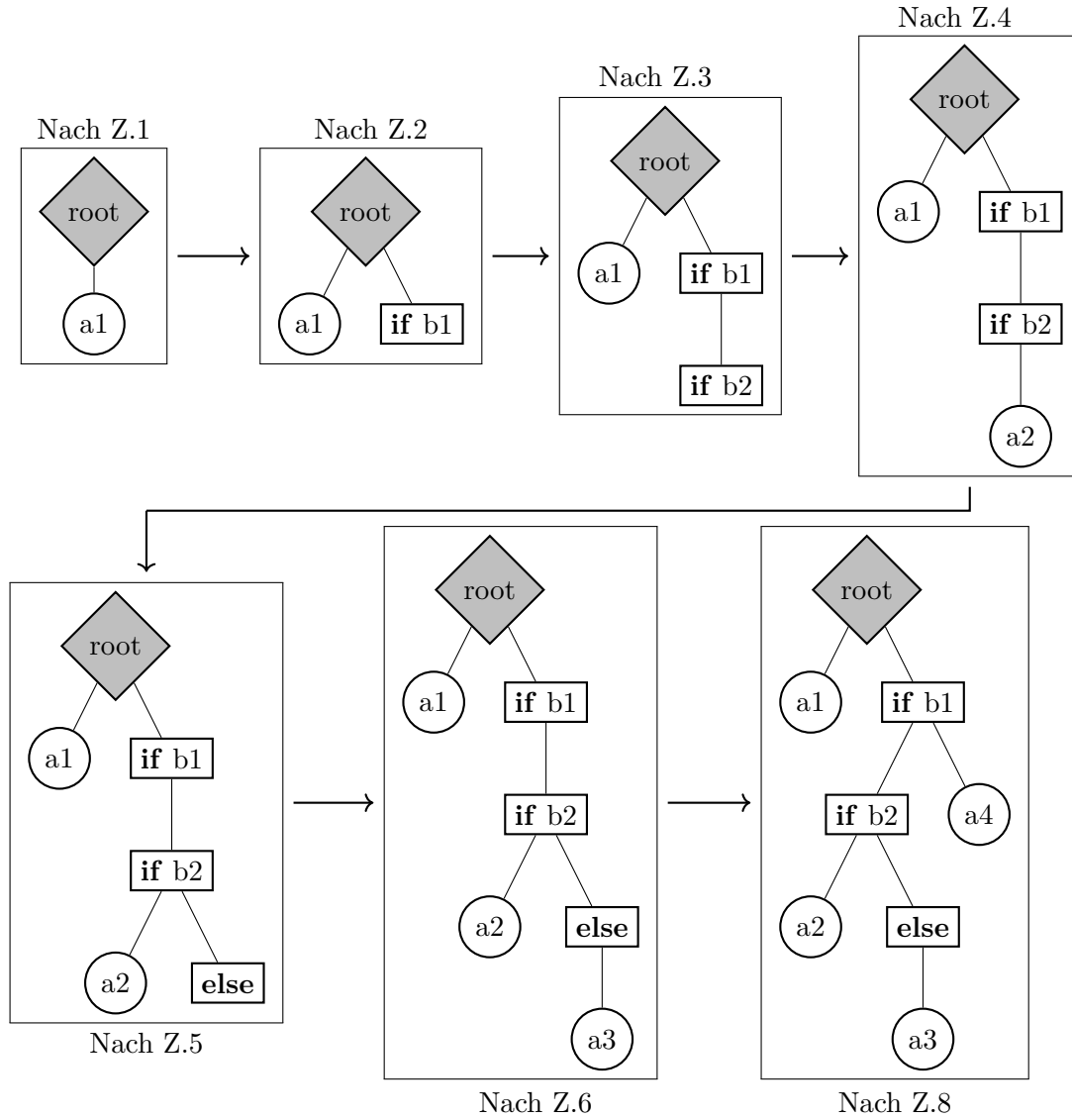


Abbildung 3.3: Beispiel für den Parsen Algorithmus von Viegner

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 [3] betrachten. Variation-Tree wird wie folgt definiert:

**Definition 3.8.** 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 3.9.** 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 dass  $\text{PROJECT}(D, t)$  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.

## 3.6 Unser Algorithmus

Nachdem wir festgestellt haben welche Informationen während des Parsens verlorengehen, müssen wir einen Weg finden diese Informationen zurück zu bekommen und das Unparsen zu bewerkstelligen. Darüber geht es in dem folgenden Unterkapitel. Wir stellen unseren Algorithmus für das Unparsen von Variation-Trees und ein Vorgehen zum Unparsen von Variation-Diffs.

Damit unser Algorithmus korrekt funktioniert müssen paar Sachen beachtet werden. Als erstes gehen wir davon aus dass die Reihenfolge der Knoten auch die Reihenfolge der Zeilen in dem Ergebnis des Unparsers widerspiegelt. Wenn ein Kindknoten A vor dem Kindknoten B aufgelistet wird bedeutet, dass das der Inhalt alle Knoten die einen Teilbaum mit Kindknoten A als Wurzel bilden vor dem Inhalt des Kindknoten B kommt. Als zweites muss man beachten dass Variation-Trees und Variation-Diffs keine Speicherung der Information über  $\#endif$  vorsehen. Unser Algorithmus kann zwar bestimmen an welchen Stellen ein  $\#endif$  kommen soll, aber er kann nicht die genauere Entfernung von Zeilenanfang wissen. Aus diesem Grund muss man hier entweder mit einer Heuristik arbeiten oder die Implementierung von Variation-Tree und Variation-Diff so modifizieren dass diese Informationen gespeichert werden und wenn benötigt abgerufen.

---

**Algorithmus 3:** Ein Variation-Tree zum mit C-Präprozessor-Annotierten Code unparsen

---

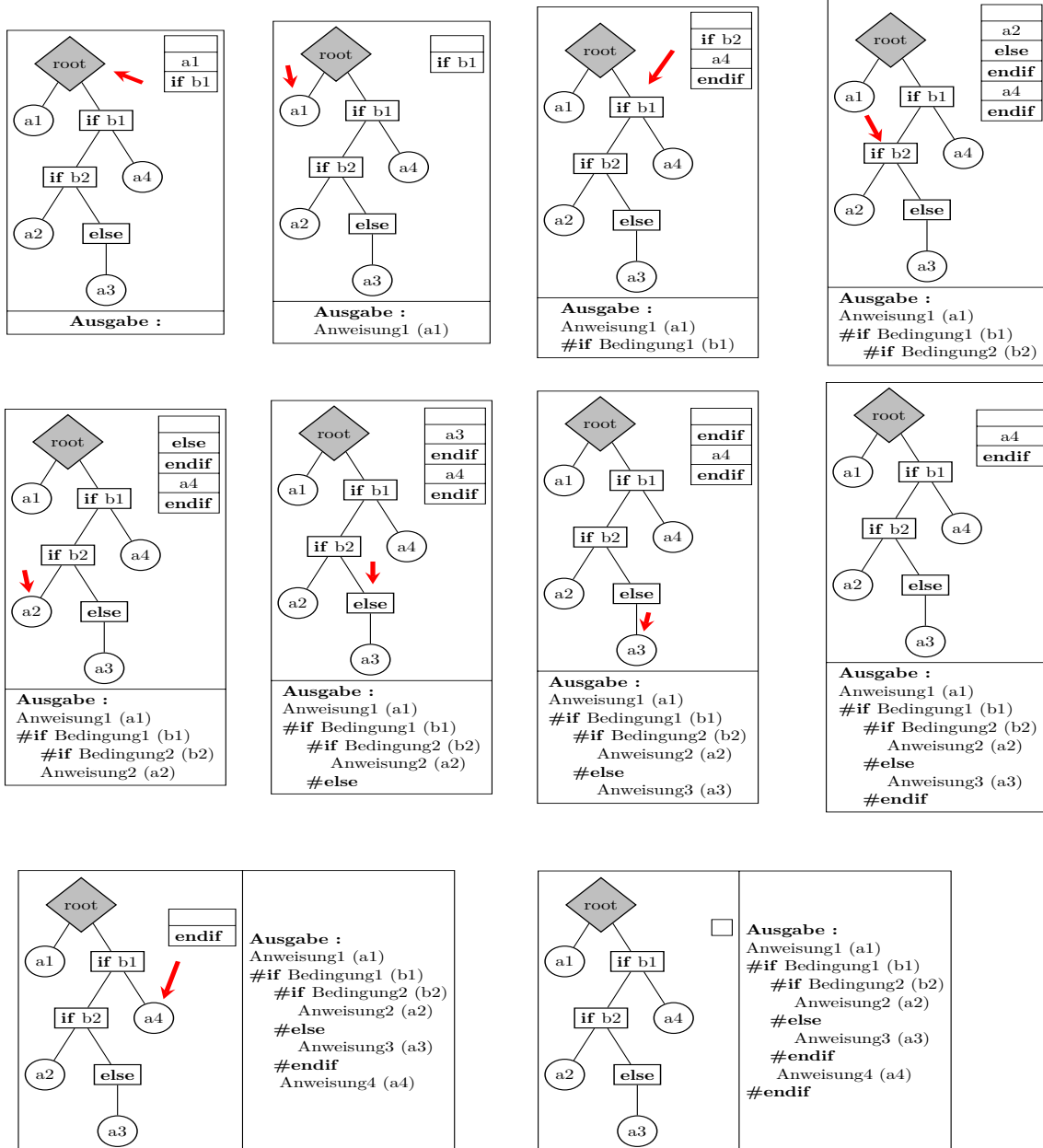
**Data:** ein Variation-Tree  
**Result:** ein mit C-Präprozessor-Annotierten Code

```

1 initialisiere einen leeren Stack/Keller stack
2 initialisiere einen String ergebniss root  $\leftarrow$  Variation-Tree gibt Wurzelknoten aus
3 kinder  $\leftarrow$  root gibt seine Kinderknoten aus
4 for  $i = n \rightarrow 1$  do
5   | lege den Knoten aus kinder[i] auf den Stack stack
6 end
7 while stack nicht leer ist do
8   | knoten  $\leftarrow$  nehme das oberste Element aus dem Stack stack
9   if Knoten von Typ if oder elif then
10    | modifiziere die gespeicherte Zeile aus knoten so das in der Bedingung logische
    | Zeichen durch äquivalente aus der Programmiersprache ersetzt werden und füge
    | die den ergebniss hinzu
11  else
12    | füge die gespeicherte Zeile aus knoten den ergebniss hinzu
13  end
14  if wenn knoten nicht von Typ Artefact ist und sein letztes Kindknoten nicht von Typ
    | else oder elif ist then
15    | erstelle einen Dummyknoten welcher die endif-Anweisung beinhaltet
16    | füge diesen Knoten den Stack hinzu
17  end
18  kinder  $\leftarrow$  knoten gibt seine Kinderknoten aus
19  for  $i = n \rightarrow 1$  do
20    | lege den Knoten aus kinder[i] auf den Stack stack
21  end
22 end

```

---



## 3.7 Laufzeitanalyse

### 3.7.1 Laufzeitanalyse für Unparsen von Variation-Trees

### 3.7.2 Laufzeitanalyse für Unparsen von Variation-Diffs



# 4

## Metrik

In diesem Kapitel stellen wir die Metrik vor, an der wir die Korrektheit unserer Lösung betrachten wollen. Dieser Kapitel beschäftigt sich nur mit Arten der Korrektheit und wie diese als Konzept für textbasierte Diffs und mit C-Präprozessor-Annotierten Code funktionieren soll. Die drei Arten der Korrektheit für textbasierte Diffs und mit C-Präprozessor-Annotierten Code, sind syntaktische Korrektheit, syntaktische Korrektheit ohne Whitespace und semantische Korrektheit, werden in diesem Kapitel erläutert.

Nachdem wir eine algorithmische Lösung für das Problem ausgearbeitet haben, müssen wir entscheiden, ob unsere Lösung korrekt ist. Um die Kriterien, an den 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 Arten der Korrektheit entschieden, an denen wir die Korrektheit entscheiden. Diese Arten sind syntaktische Gleichheit, syntaktische Gleichheit ohne Whitespace und semantische Gleichheit. Wenn die ausgangs Eingabe und das Ergebnis der ausgangs Eingabe nach Parsen und Unparsen eine dieser Gleichheiten erfüllen gilt das Unparsen für diesen Fall als Korrekt. In der Tabelle 4.1 ist kurz zusammengefasst wie jeweils die Art der Korrektheit bezogen auf C-Präprozessor-Annotierter Code oder textbasierte Diffs zu verstehen sind.

	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

In diesem Abschnitt sprechen wir über die syntaktische Korrektheit, die zweite Zeile aus der Tabelle 4.1. Syntaktische Korrektheit bedeutet, dass der zu vergleichende Text in jedem Zeichen identisch ist. Der Vergleich auf syntaktische Korrektheit sieht für mit C-Präprozessor-Annotierter Code und textbasierte Diffs gleich aus, was in der Abbildung 4.1 zu sehen ist. Hierfür muss der ausgangs C-Präprozessor-Annotierter Code bzw. der textbasierter Diff mit dem Ergebnis nach dem Parsen und Unparzen Schritt in jedem Zeichen übereinstimmen. Wie in der Abbildung 4.1 wird ein C-Präprozessor-Annotierter Code bzw. der textbasierte Diff genommen, dann darauf Parser und Unparser angewendet. Das Ergebnis und der C-Präprozessor-Annotierter Code bzw. der textbasierte Diff wird dann jeweils in ein String umgewandelt und diese dann auf Gleichheit geprüft. So wird die syntaktische Gleichheit von den C-Präprozessor-Annotierten Code bzw. den textbasierten Diff und dem Ergebnis von Parser und Unparser überprüft.

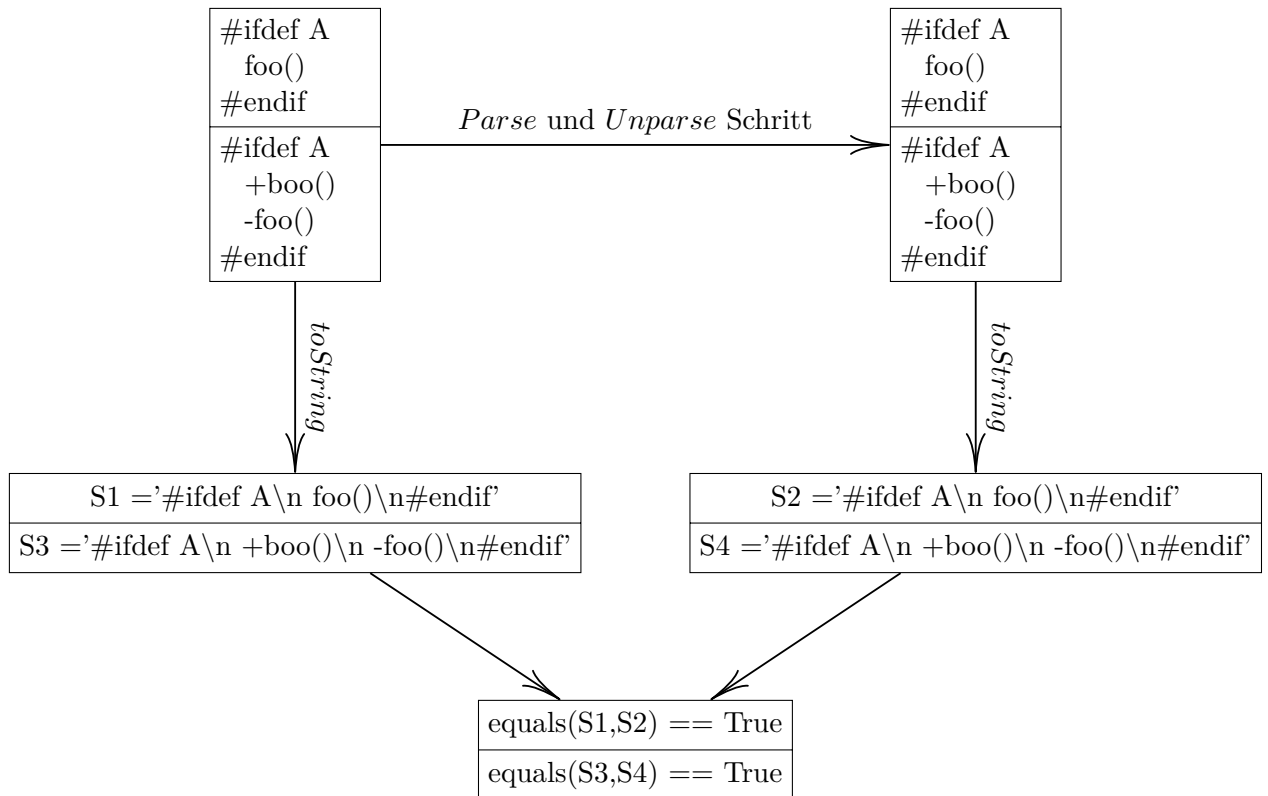


Abbildung 4.1: Beispiel für Syntaktische Gleichheit

Der syntaktischen Korrektheit ohne Whitespace aus der dritten Zeile der Tabelle 4.1 widmen wir uns in diesem Abschnitt. Analog zu syntaktischer Gleichheit ist syntaktische Gleichheit ohne Whitespace für den C-Präprozessor-Annotierten Code und textbasierte Diffs gleich zu verstehen, wie in Abbildung 4.2 zu sehen ist. Bei dieser Art von Korrektheit muss auch wie in vorherigen Fall der Ausgangs C-Präprozessor-Annotierter Code bzw. der textbasierter Diff mit dem Ergebnis nach dem Parsen und Unparsen Schritt in jedem Zeichen übereinstimmen, aber nur nachdem alle Zeichen, die zu Gruppe der Whitespace-Zeichen gehören, entfernt wurden. Die Abbildung 4.2 veranschaulicht das. Dort sind der Ausgangs C-Präprozessor-Annotierter Code bzw. der textbasierte Diff gegeben. Links von den ist das Ergebnis von Parse und Unparse Schritt. Danach werden die alle in Strings umgewandelt. Als Nächstes werden alle Whitespace-Zeichen aus den Strings entfernt und anschließend die auf Äquivalenz geprüft. So wird der C-Präprozessor-Annotierter Code bzw. der textbasierte Diff und das Ergebnis von Parse und Unparse Schritt auf syntaktische Gleichheit ohne Whitespace überprüft.

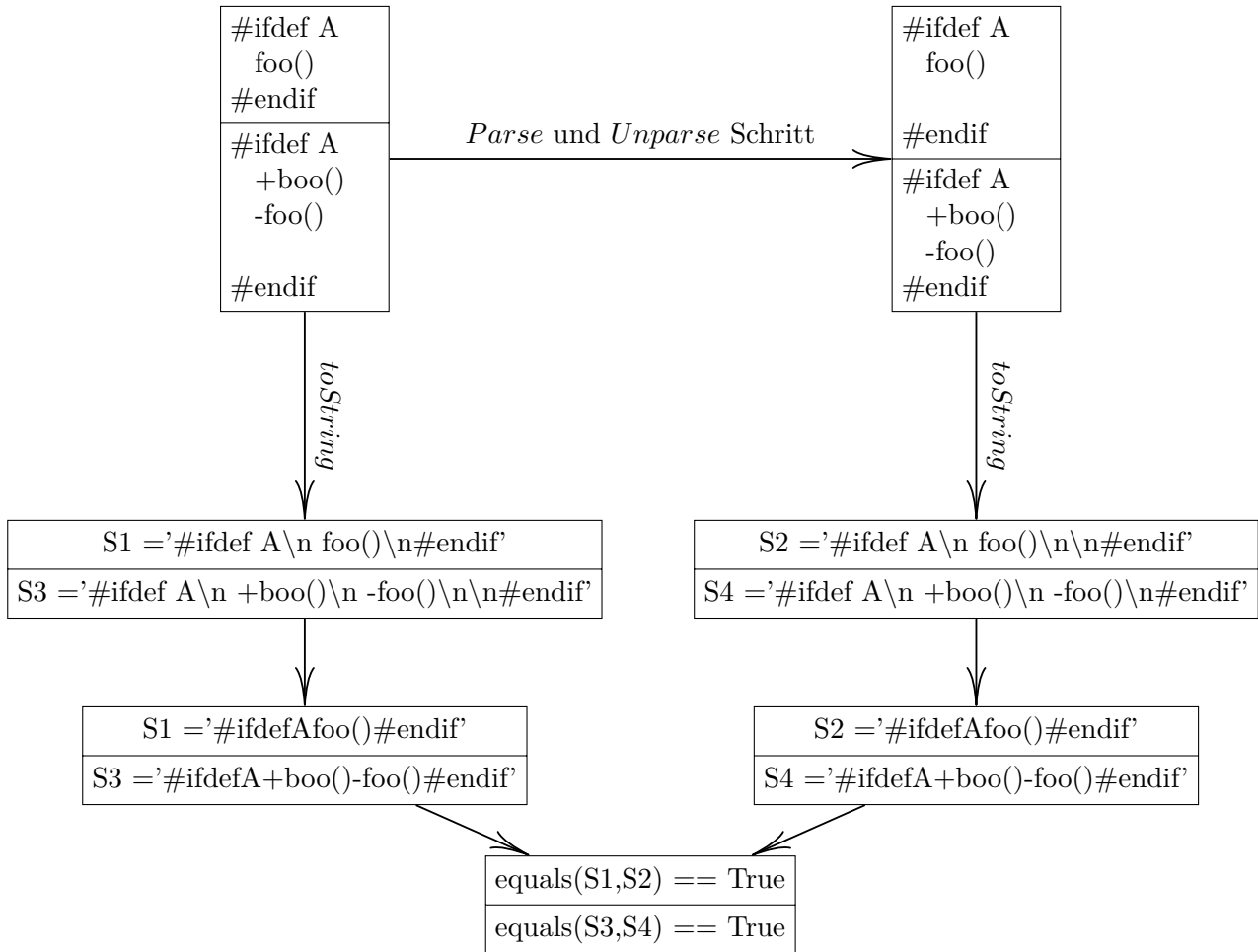


Abbildung 4.2: Beispiel für Syntaktische Gleichheit ohne Whitespace

Die semantische Gleichheit von mit C-Präprozessor-Annotierten Code werden wir nicht betrachten, da dafür wir entscheiden müssen ob zwei Programmen äquivalent sind. Das geht über den Rand unserer Möglichkeiten, da diese Fragestellung unentscheidbar ist und als das Äquivalenzproblem bekannt [6]. Mit den C-Präprozessor-Annotationen geht es auch über den Rand unserer Möglichkeiten, da C-Präprozessor-Annotationen hier für Erzeugung der Variabilität verwendet werden. Dabei hat so eine Softwareproduktlinie  $n$  Features und im Worst-Case muss  $2^n$  Varianten der Software betrachtet werden[2], welches eine exponentielle Laufzeit bedeutet und über den Rand unserer Möglichkeiten geht.

Um die semantische Gleichheit für textbasierte Diffs geht es in diesem Abschnitt. Wie die semantische Gleichheit für textbasierte Diffs zu verstehen ist, ist nicht eindeutig festgelegt. Unsere Interpretation der semantischen Gleichheit für textbasierte Diffs ist an der Gleichheit für Variation-Diffs [4] orientiert. Wir verstehen die semantische Gleichheit wie folgt, zwei textbasierte Diffs sind semantisch gleich, wenn ihre Projektionen auf den Zustand davor bzw. danach syntaktisch gleich oder syntaktisch gleich ohne Whitespace sind. In der Abbildung 4.3 ist dies dargestellt. Dabei ist die Projektion für textbasierte Diffs wie folgt zu verstehen: Ein textbasierter Diff hat Zeilen von drei Typen unverändert gebliebene Zeilen, gelöschte Zeilen und eingefügte Zeilen. Bei der Projektion werden einige dieser Typen der Zeilen entfernt einige beibehalten und so entsteht eine Projektion von textbasierten Diff auf ein mit C-Präprozessor-Annotierten Code. Dabei wird für die Projektion auf den Zustand davor, die unveränderten und gelöschten Zei-

len beibehalten und die eingefügten entfernt und für die Projektion auf den Zustand danach, die unveränderten und eingefügten Zeilen beibehalten und die gelöschten Zeilen entfernt. Dies verläuft analog zu der Projektion von Variation-Diff zu Variation-Tree.

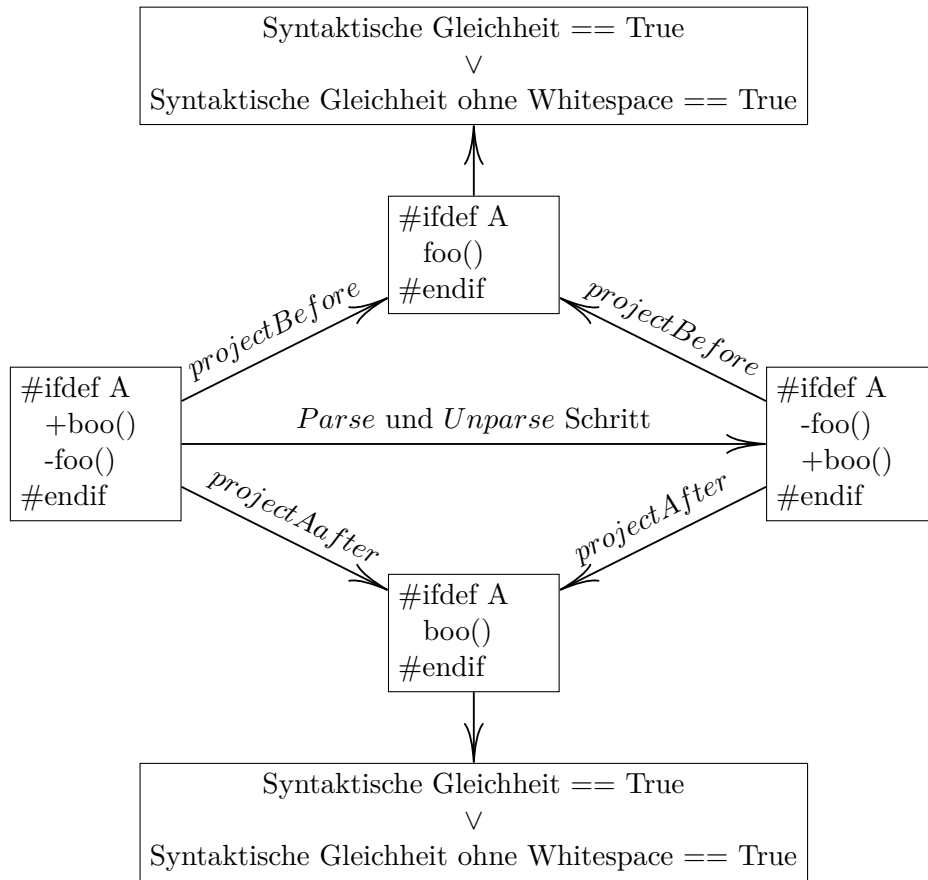


Abbildung 4.3: Beispiel für Semantische Gleichheit



## Implementierung

### 5.1 Code

#### 5.1.1 Parser von Variation-Trees mit Heuristik

#### 5.1.2 Parser von Variation-Trees mit Speicherung

#### 5.1.3 Parser von Variation-Diffs

### 5.2 Test





# Literaturverzeichnis

- [1] M. Al-Hajjaji, F. Benduhn, T. Thüm, T. Leich, and G. Saake. Mutation Operators for Preprocessor-Based Variability. In *Proc. Int'l Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*, pages 81–88, New York, NY, USA, Jan. 2016. ACM. ISBN 978-1-4503-4019-9. doi: 10.1145/2866614.2866626.
- [2] S. Apel, D. Batory, C. Kästner, and G. Saake. *Feature-Oriented Software Product Lines*. Springer, Berlin, Heidelberg, 2013. ISBN 978-3-642-37520-0. doi: 10.1007/978-3-642-37521-7.
- [3] 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)*, pages 196–208, New York, NY, USA, Nov. 2022. ACM. ISBN 9781450394130. doi: 10.1145/3540250.3549108.
- [4] P. M. Bittner, A. Schultheiß, S. Greiner, B. Moosherr, S. Krieter, C. Tinnes, T. Kehrer, and T. Thüm. Views on Edits to Variational Software. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*, pages 141–152, New York, NY, USA, Aug. 2023. ACM. ISBN 9798400700910. doi: 10.1145/3579027.3608985.
- [5] P. M. Bittner, A. Schultheiß, B. Moosherr, T. Kehrer, and T. Thüm. Variability-Aware Differencing with DiffDetective. In *Proc. Int'l Conference on the Foundations of Software Engineering (FSE)*, New York, NY, USA, July 2024. ACM. To appear.
- [6] M. J. Fischer. *Efficiency of Equivalence Algorithms*, pages 153–167. Springer US, Boston, MA, 1972. ISBN 978-1-4684-2001-2. doi: 10.1007/978-1-4684-2001-2\_14. URL [https://doi.org/10.1007/978-1-4684-2001-2\\_14](https://doi.org/10.1007/978-1-4684-2001-2_14).
- [7] C. Kästner, S. Apel, T. Thüm, and G. Saake. Type Checking Annotation-Based Product Lines. *Trans. on Software Engineering and Methodology (TOSEM)*, 21(3):14:1–14:39, July 2012. ISSN 1049-331X. doi: 10.1145/2211616.2211617.
- [8] T. Kehrer, T. Thüm, A. Schultheiß, and P. M. Bittner. Bridging the Gap Between Clone-and-Own and Software Product Lines. In *Proc. Int'l Conf. on Software Engineering (ICSE)*, pages 21–25, Piscataway, NJ, USA, May 2021. IEEE. ISBN 978-1-6654-0140-1. doi: 10.1109/ICSE-NIER52604.2021.00013.
- [9] J. Krüger and T. Berger. An Empirical Analysis of the Costs of Clone- and Platform-Oriented Software Reuse. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*, pages 432–444, New York, NY, USA, 2020. ACM. ISBN 9781450370431. doi: 10.1145/3368089.3409684.

- [10] J. Krüger and T. Berger. Activities and Costs of Re-Engineering Cloned Variants Into an Integrated Platform. In *Proc. Int'l Working Conf. on Variability Modelling of Software-Intensive Systems (VaMoS)*, New York, NY, USA, 2020. ACM. ISBN 9781450375016. doi: 10.1145/3377024.3377044.
- [11] E. Kuiter, J. Krüger, S. Krieter, T. Leich, and G. Saake. Getting Rid of Clone-and-Own: Moving to a Software Product Line for Temperature Monitoring. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*, pages 179–189, New York, NY, USA, Sept. 2018. ACM. ISBN 9781450364645. doi: 10.1145/3233027.3233050.
- [12] B. Moosherr. Constructing Variation Diffs Using Tree Diffing Algorithms. Bachelor's thesis, University of Ulm, Germany, Apr. 2023. URL [https://oparu.uni-ulm.de/xmlui/bitstream/handle/123456789/50184/BA\\_Moosherr.pdf](https://oparu.uni-ulm.de/xmlui/bitstream/handle/123456789/50184/BA_Moosherr.pdf).
- [13] L. Neves, P. Borba, V. Alves, L. Turnes, L. Teixeira, D. Sena, and U. Kulesza. Safe Evolution Templates for Software Product Lines. *J. Systems and Software (JSS)*, 106:42–58, 2015. doi: 10.1016/j.jss.2015.04.024. URL <https://www.sciencedirect.com/science/article/pii/S0164121215000801>.
- [14] M. Nieke, G. Sampaio, T. Thüm, C. Seidl, L. Teixeira, and I. Schaefer. Guiding the Evolution of Product-Line Configurations. *Software and Systems Modeling (SoSyM)*, 21:225–247, Feb. 2022. doi: 10.1007/s10270-021-00906-w.
- [15] L. Passos, K. Czarnecki, S. Apel, A. Wąsowski, C. Kästner, and J. Guo. Feature-Oriented Software Evolution. In *Proc. Int'l Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*, pages 1–8, New York, NY, USA, 2013. ACM. doi: 10.1145/2430502.2430526.
- [16] L. Passos, L. Teixeira, N. Dintzner, S. Apel, A. Wąsowski, K. Czarnecki, P. Borba, and J. Guo. Coevolution of Variability Models and Related Software Artifacts. *Empirical Software Engineering (EMSE)*, 21(4), 2016. doi: 10.1007/s10664-015-9364-x.
- [17] G. Sampaio, P. Borba, and L. Teixeira. Partially Safe Evolution of Software Product Lines. *J. Systems and Software (JSS)*, 155:17–42, 2019. ISSN 0164-1212. doi: 10.1016/j.jss.2019.04.051.
- [18] C. Seidl, F. Heidenreich, and U. Aßmann. Co-Evolution of Models and Feature Mapping in Software Product Lines. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*, pages 76–85, New York, NY, USA, 2012. ACM. doi: 10.1145/2362536.2362550.
- [19] J. Sprey, C. Sundermann, S. Krieter, M. Nieke, J. Mauro, T. Thüm, and I. Schaefer. SMT-Based Variability Analyses in FeatureIDE. In *Proc. Int'l Working Conf. on Variability Modelling of Software-Intensive Systems (VaMoS)*, New York, NY, USA, Feb. 2020. ACM. ISBN 9781450375016. doi: 10.1145/3377024.3377036.
- [20] C. Sundermann, M. Nieke, P. M. Bittner, T. Heß, T. Thüm, and I. Schaefer. Applications of #SAT Solvers on Feature Models. In *Proc. Int'l Working Conf. on Variability Modelling of Software-Intensive Systems (VaMoS)*, New York, NY, USA, Feb. 2021. ACM. ISBN 9781450388245. doi: 10.1145/3442391.3442404.
- [21] S. Viegner. Empirical Evaluation of Feature Trace Recording on the Edit History of Marlin. Bachelor's thesis, University of Ulm, Germany, Apr. 2021. URL [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).

- [22] B. Zhang and M. Becker. Variability code analysis using the vital tool. In *Proceedings of the 6th International Workshop on Feature-Oriented Software Development*, FOSD '14, page 17–22, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450329804. doi: 10.1145/2660190.2662113. URL <https://doi.org/10.1145/2660190.2662113>.
- [23] S. Zhou, Ș. Stănciulescu, O. Leßenich, Y. Xiong, A. Wąsowski, and C. Kästner. Identifying Features in Forks. In *Proc. Int'l Conf. on Software Engineering (ICSE)*, pages 105–116, New York, NY, USA, May 2018. ACM. doi: 10.1145/3180155.3180205. URL <https://dl.acm.org/citation.cfm?id=3180205>.