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, 25. Oktober 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

1 Einleitung

Bei der Entwicklung von konfigurierbaren Softwaresystemen, wie zum Beispiel Clone-and-Own, oder Softwareproduktlinien, gibt es im Laufe des Lebenszyklus immer mehr Features. Es ist von Vorteil, eine Möglichkeit zu haben, die Features im Code zu unterscheiden und automatisch zu finden. Einige Möglichkeiten dazu wären Präprozessor-Annotationen, oder Build-Systeme [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 Ⓥ zu sehen (Abb. 1.1 St.Ⓥ). 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.Ⓥ).

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 variabilitätbezogenen Code (Abb.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 geparst. Das Parsen in DiffDetective funktioniert für Variation-Trees und für Variation-Diffs über einen einzigen gemeinsamen Algorithmus. Der Algorithmus arbeitet wie folgt: Er geht über alle Zeilen des Codes/Textes und schaut sich für jede Zeile an, wie diese Zeile manipuliert wurde, ob die Zeile unverändert geblieben ist, gelöscht wurde, oder neu ist [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.⑤). 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 textbasierten 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.⑤). 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.

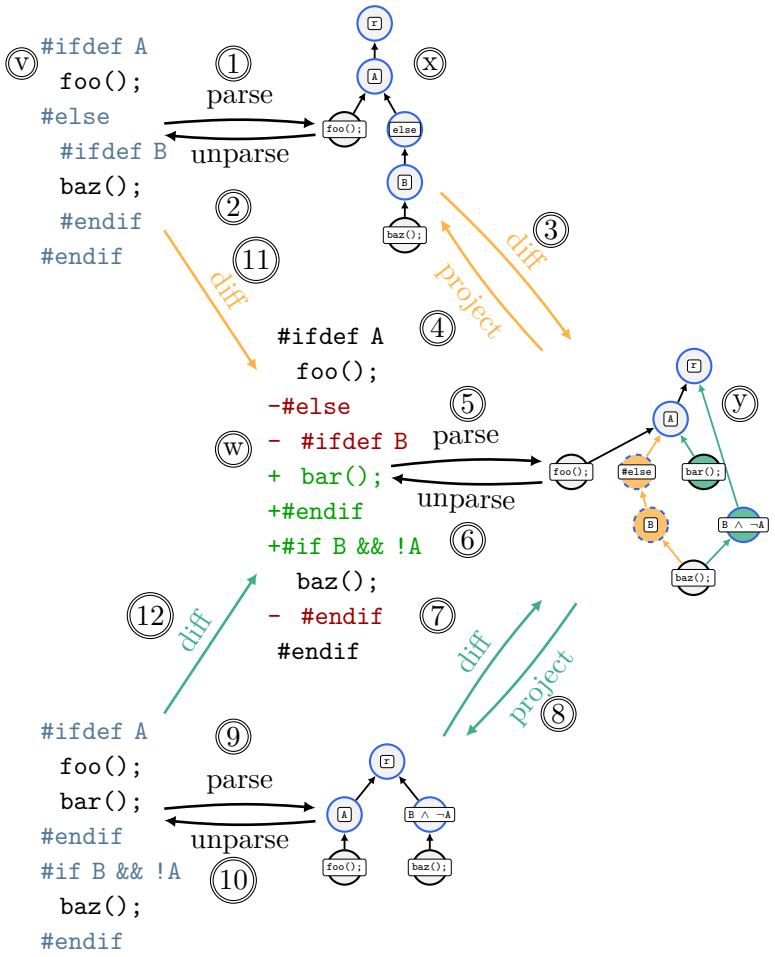


Abbildung 1.1: Überblick über Variabilität bezogene Konvertierungen

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.ⓧ) in C-Präprozessor-Annotierten Code (Abb. 1.1 St.ⓥ) und Variation-Diffs (Abb. 1.1 St.ⓨ) in textbasierte Diffs (Abb. 1.1 St.ⓦ) überführt.

Eine Einsatzmöglichkeit des Unparsers wäre, das Unparsen von Variation-Trees, bei denen die Variabilität mutiert wurde. Eine Möglichkeit zu Analyse von Softwareproduktlinien ist Mutation-Tests. Bei Mutation-Tests werden Mutation-Operatoren verwendet, welche aber nur auf der abstrakten Ebene, also auf Variation-Trees, angewandt werden können [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 $\tau(v) = \texttt{mapping}$ 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 Verglichene 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 Verglichene gleich sein muss, wenn man alle Zeichen, die Whitespace sind, entfernen würde. Bei der semantischen Gleichheit muss der Sinn gleich sein, was uns das letzte Beispiel der Abbildung 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.

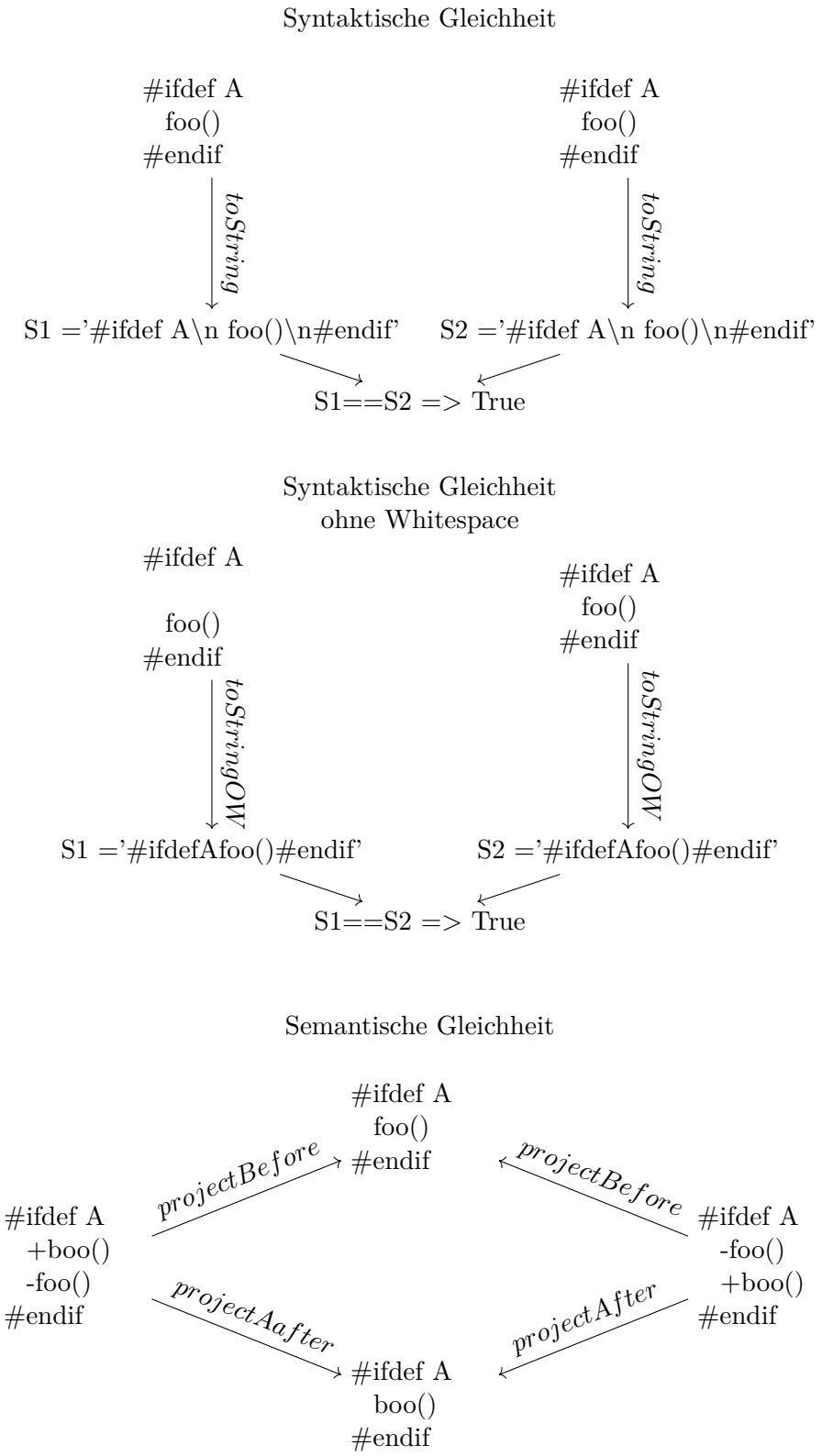


Abbildung 1.2: Beispiel für Metrik

2 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 Einsatzmöglichkeit. 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 Whihtespace-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, umdefiniert 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                 int i;
17                 puts("Hello world!");
18 #ifdef N
19                 for (i = 0; i < N; i++)
20                 {
21                         puts(A);
22                 }
23 #endif
24
25                 return 0;
26         }
```

Abbildung 2.1: Beispiel für C Code mit Präprozessor Anweisungen

2.2 Realisierung von Variabilität mit dem C-Präprozessor

Der C-Präprozessor ist eine Möglichkeit, Variabilität zu erzeugen [2]. Um Variabilität mithilfe des C-Präprozessor 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 und auch wie sie definiert sind, werden bestimmte Codefragmente entweder behalten oder entfernt. 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. Bei der Realisierung von Variabilität wird oft mit Features gearbeitet. Ein Feature ist dabei ein Merkmal oder ein für den Endbenutzer sichtbares Verhalten eines Softwaresystems. Ob eine C-Code Zeile im endgültigen Programm auftaucht oder nicht wird durch die dazugehörige Bedingung bestimmt. Diese Bedingungen werden durch C-Präprozessor-Annotation dargestellt. Es ist möglich, mit den C-Präprozessor Anweisungen eine große Menge an Bedingungen abzubilden [3]. Zur leichterer Pflege werden diese Bedingungen oft in eine Folge von mehreren Feature-Annotationen aufgeteilt. Beispiel dafür ist Zeile 7 zu sehen, wo seine Feature-Annotationen FEATURE_A && FEATURE_B und FEATURE_D sind, aber seine Bedingung FEATURE_A && FEATURE_B && FEATURE_D. In diesem Code-Beispiel ist auch die Abhängigkeit einiger Features von anderen zu erkennen, wie in Zeile 5, wo die Auswahl des Features D nur dann möglich ist, wenn auch die Features A und B ausgewählt sind. Die Zeile 17 in Abbildung 2.2 hat als Feature-Annotationen FEATURE_C und !FEATURE_B die Bedingung dabei aber ist FEATURE_C && !FEATURE_B. Nicht nur die Definition von Features, sondern auch die Nicht-Definition

```
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         sho();
18 #define SIZE 100
19 #endif
20         bazzz();
21 #else
22         boom();
23         broo();
24 #endif
25 #if SIZE > 180
26         long j;
27 #elif SIZE < 111
28         short j;
29 #else
30         int j;
31 #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          sho();
3          bazzz();
4          short j;
```

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

kann, einen Einfluss auf das Ergebnis haben, wie in der Zeile 16 zu sehen ist. In dem Beispiel wird, wie bei der Implementierung von 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. Diese Ergebnisse werden als Varianten bezeichnet. Dabei werden für die Abbildung 2.3 die Features A und B definiert und für die Abbildung 2.4 nur das Feature C. Eine Konfiguration ist eine Auswahl der Features welche ausgewählt und nicht ausgewählt werden. Es ist zu sehen, dass der generierter Code nur in dem Bezeichner j gleich ist und sonst nicht. Das veranschaulicht, wie unterschiedlich das Programm sein kann.

Die Abbildung 2.5 zeigt Beispiele für vier Patern, 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 defi-

niert. 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.

```
#ifdef WINDOWS
#include <windows.h>
#else
#include <unix.h>
#endif
...
```

```
#ifdef FOO
int foo(){...}
#else
#define foo(...) 0
#endif

int i = 429 + foo()
...
```

```
#ifdef FEAT_SELINUX
#define FEAT_LINUX 1
#undef  FEAT_WINDOWS
#endif

#ifdef FEAT_WINDOWS
...
```

```
#ifdef  A
#define SIZE 128
#else
#define SIZE 64
#endif

...allocate(SIZE)...
```

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

3 Unparse Algorithmus

In diesem Kapitel wird der Algorithmus zum Unparsen von Variation-Trees und Methode Variation-Diffs umzuparsen vorgestellt. Wir beschreiben den theoretischen Hintergrund, welche Bedingungen erfühlt werden müssen, damit der Algorithmus korrekt funktioniert und die Arbeitsweise des Algorithmus.

Wir beschäftigen uns mit der Definition von Variation-Tree und Variation-Diff, nehmen neue Definitionen für Variation-Tree und Variation-Diff aus anderer Arbeit und erweitern diese. Es werden auf Bedingungen aufgestellt, ohne die der Algorithmus nicht korrekt funktionieren kann. Der von uns entwickelte Algorithmus basiert auf der Tiefensuche und kann nur für das unparsen von Variation-Trees verwendet werden. Für das Unparsen von Variation-Diffs reduzieren wir das Problem. Indem wir statt Variation-Diff zu unparsen, den Variation-Diff projizieren, dann zwei Variation-Trees unparsen und schließlich ein Diff über das Ergebnis bilden.

Wir fangen an mit Definitionen von Variation-Tree und Variation-Diff in Kapitel 3.1. Danach im Kapitel 3.2 sehen wir den Parser, welcher Variation-Trees und Variation-Diffs erstellt. Im Kapitel 3.3 werden die, während des Parsens, verlorengehende Informationen bestimmt und Möglichkeiten vorgestellt diese zurückzubekommen. Aufbauend auf den vorherigen wird im Kapitel 3.4 unser Algorithmus zum Unparsen von Variation-Trees vorgestellt. Schlussendlich wird im Kapitel 3.5 eine Laufzeitanalyse des Algorithmus durchgeführt.

3.1 Variation-Tree und Variation-Diff

Um verstehen zu können, wie wir Variation-Trees und -Diffs unparsen können, müssen wir uns mit den Einschränkungen und Möglichkeiten des Parsers vertraut machen.

Um Variation-Trees und Variation-Diffs kennenzulernen, betrachten wir zunächst die ursprüngliche Definition der Datenstrukturen [3].

Definition 3.1. *Ein* VARIATION-TREE (V, E, r, τ, 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* $\tau(v) \in \{$ARTIFACT,MAPPING,ELSE$\}$ *identifiziert einen Knoten* $v \in V$ *entweder als Vertreter eines Implementierungsartefakts, einer Feature-Annotation oder eines else-Zweigs. Das Label* $l(v)$ *ist eine aussagenlogische Formel, wenn* $\tau(v) =$ MAPPING*, ein Verweis auf ein Implementierungsartefakt, wenn* $\tau(v) =$ 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 werden, dabei hat ein Knoten w von Typ $\tau(w) = \text{MAPPING}$ höchstens einen Knoten u von dem Typ $\tau(u) = \text{ELSE}$.

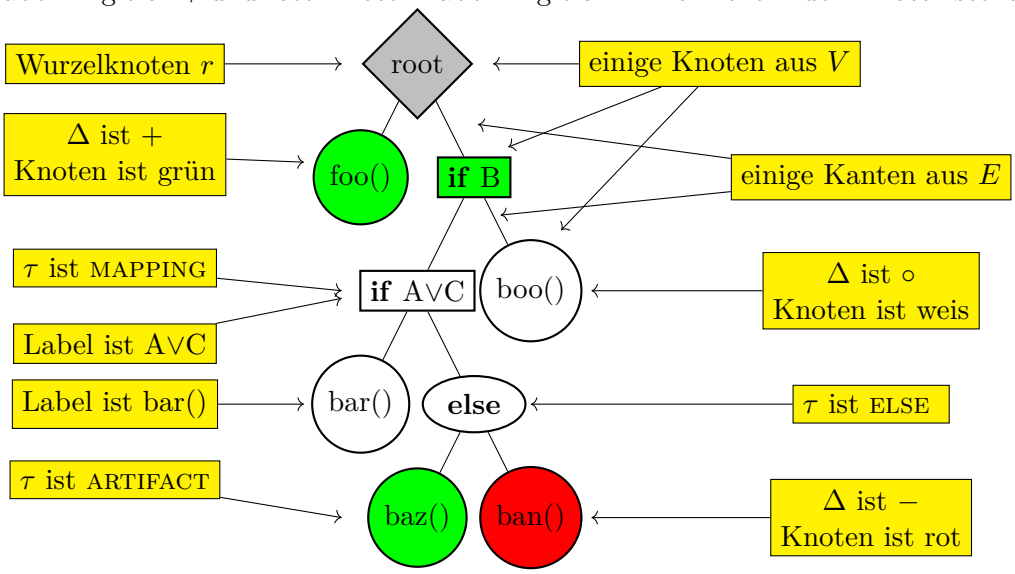
Definition 3.2. *Ein Variation-Diff ist ein gerichteter, zusammenhängender, azyklischer Graph $D = (V, E, r, \tau, l, \Delta)$, welcher einen Wurzelknoten hat, mit Knoten V, Kanten $E \subseteq V \times V$, Wurzelknoten $r \in V$, Knotentyp τ, Knotenlabel l und einer Funktion $\Delta : V \cup E \to \{+, -, \circ\}$, 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.*

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

Ob ein Knoten oder eine Kante zu einer gegebenen Zeit existiert oder nicht, wird durch EXISTS bestimmt. EXISTS ist hier genauso definiert wie in [3] und [12].

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

In der Abbildung unten ist ein Beispiel für ein Variation-Diff gegeben. Es hilft auch bei dem Verständnis von Variation-Trees da diese, ähnlich zu Variation-Diffs sind. Die Abbildung zeigt wie die unterschiedlichen Komponenten des Variation-Diffs visuell dargestellt werden können. Diese Darstellung wurde an der Darstellung eines Variation-Diffs aus DiffDetective [5] angelehnt. In der Abbildung ist ein Variation-Diff und gelbe Kasten mit Pfeilen, welche die Bestandteile des Variation-Diffs beschreiben. Die Form eines Knotens stellt seinen Knotentyp τ dar, runde Knoten haben ARTIFACT als τ, rechteckige Knoten haben MAPPING als τ und elliptische Knoten haben ELSE als τ. Die Farbe eines Knotens stellt sein Δ dar, wiese Knoten haben Δ gleich $\circ$, grüne Knoten haben Δ gleich $+$ und rote Knoten haben Δ gleich $-$. Der Text in den Knoten stellt



den Label dar.

Das ist aber nicht die einzige Möglichkeit Variation-Tree und Variation-Diff zu definieren. In [12] wurden die Variation-Tree und Variation-Diff etwas anders definiert. Obwohl dort auch Variation-Tree und Variation-Diff definiert werden, werden wir in dieser Arbeit die neuen Definitionen als geordneter Variation-Tree und als geordneter Variation-Diff bezeichnen. Diese Definitionen haben eine Eigenschaft, welche wir brauchen um das Unparsen zu bewerkstelligen. Das ist die Ordnung der Kinderknoten, ohne die wir nicht eindeutig wissen, wie der Inhalt der

Knoten einzuordnen ist. Genauer gehen wir darauf in Unterkapitel 3.3 ein. Die Definitionen von geordneten Variation-Trees und geordneten Variation-Diff sehen wie folgt aus:

Definition 3.5. *Ein geordneter Variation-Tree (V,E,r,τ,l,O) ist ein geordneter Baum, bei dem V, E, r, τ und l genauso definiert sind wie in der Definition 3.1 . Dazu hat ein geordneter Variation-Tree eine injektive Funktion $O : V \to \mathbb{N}$, die eine Ordnung für die Kinder eines jeden Knotens jedes Knotens definiert.*

Definition 3.6. *Ein geordneter Variation-Diff $D = (V, E, r, \tau, l, \Delta, O_{before}, O_{after})$ ist ein gerichteter, zusammenhängender, azyklischer Graph, bei dem V, E, r, τ, l, und Δ genauso definiert sind wie in der Definition 3.2 . Die Reihenfolge der Kinderknoten vor der Änderung O_{before} und nach der Änderung O_{after} sind eine injektive Funktion $O_{before}, O_{after} : V \to \mathbb{N}$. Die Projektionen $project_O(D, t)$ müssen für alle Zeiten $t \in \{after, before\}$ ein Variation-Tree mit demselben Wurzelknoten sein.*

Aus Gründen der Eindeutigkeit haben wir auch die Projektion umbenannt, da sich die Definition von $project_O(D, t)$ von der Definition der Projektion $\text{PROJECT}(D, t)$ aus der Definition 3.3 unterscheidet, wegen der zusätzlichen Informationen, die gespeichert werden.

Definition 3.7. *Die Projektion $project_O(D, t)$ eines geordneten Variation-Diffs $D = (V, E, r, \tau, l, \Delta, O_{before}, O_{after})$ zum Zeitpunkt $t \in \{after, before\}$ ist definiert als:*
$project_O(D, t) := (V', E', r, \tau, l, O_t)$, *wobei* $V' = \{v \in V \mid \text{EXISTS}(t, \Delta(v))\}$,
$E' = \{e \in E | \text{EXISTS}(t, \Delta(e))\}$ *und die Existenz von* $d \in \{+, -, \circ\}$, *zu der Zeit* $t \in \{after, before\}$ *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. Es ist so, da geordnete Variation-Tree bzw. Variation-Diff eine Erweiterung von normale Variation-Tree bzw. Variation-Diff sind. Aus diesem Grund können wir geordnete Variation-Tree bzw. Variation-Diff in normale Variation-Tree bzw. Variation-Diff umzuwandeln.

Definition 3.8. *$reduce_{OVT}$ wandelt ein geordnetes Variation-Tree (Definition 3.5) in ein normales Variation-Tree (Definition 3.1)*
$reduce_{OVT}((V, E, r, \tau, l, O)) := (V, E, r, \tau, l)$.

Definition 3.9. *$reduce_{OVD}$ wandelt ein geordnetes Variation-Diff (Definition 3.6) in ein normales Variation-Diff (Definition 3.2)*
$reduce_{OVD}((V, E, r, \tau, l, \Delta, O_{before}, O_{after})) := (V, E, r, \tau, l, \Delta)$.

Es bleibt uns nur noch zu zeigen, dass die Reihenfolge der Anwendung nicht von Bedeutung ist. Damit wir die Abbildung 3.1 bekommen, welche zeigt wie geordnete Variation-Trees, geordnete Variation-Diffs, normale Variation-Trees und normale Variation-Diffs transformiert werden können.

Lemma 3.10. *Für ein geordneten Variation-Diff D und eine Zeit $t \in \{before, after\}$ gilt $reduce_{OVT}(project_O(D, t)) = project(reduce_{OVD}(D), t)$.*

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

□

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. Dieser Unterschied ist die Ordnung der Kinderknoten, welche die geordneten Variation-Trees und Variation-Diffs haben und die normalen Variation-Trees und Variation-Diffs nicht. Diese Ordnung ist für das Unparsen notwendig. 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. Nachdem wir jetzt wissen was Variation-Trees und Variation-Diffs sind, ist es an der Zeit nachzuvollziehen wie diese gebildet werden.

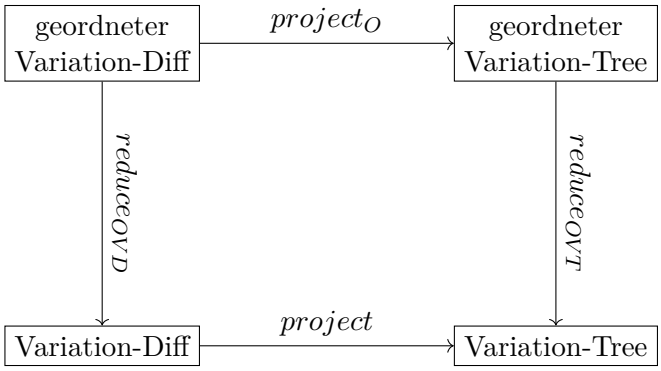


Abbildung 3.1: Transformationen von geordneten Variation-Diff , geordneten Variation-Tree, Variation-Diff und Variation-Tree

3.2 Parser

Jetzt beschäftigen wir uns mit den Parsen, also wie Variation-Trees bzw. Variation-Diffs aus mit C-Präprozessor-Direktiven annotiertem Code bzw. einem textbasierten Diff von solchem Code erstellt werden. Das Verständnis des Parsens ist, für das Verständnis des Unparsens von Bedeutung, da das Unparsen das Parsen invertiert. Dazu schauen wir uns den Parser-Algorithmus von Viegener [21] an, welcher das Parsen von Variation-Diffs aus textbasierten Diffs eingeführt hat.

Der unten stehender Algorithmus überführt einen textbasierten Diff in einen Variation-Diff. Dabei werden in dem Algorithmus einige Funktionen verwendet, die nicht so in der Definition vorkamen. Einer dieser Funktionen ist der Code-Typ dieser stellt dar, welche Rolle die Zeile in dem Diff hat. Es kann die Werte if, elif, else, code, oder endif haben. Bei dem Wert if ist gegeben, dass die Zeile eine der Präprozessor Anweisungen #if, #ifdef, oder #ifndef hat. Bei dem Wert else ist in der Zeile die Präprozessor Anweisungen #else, bei Wert elif ist die Präprozessor Anweisungen #elif und bei Wert endif ist die Präprozessor Anweisungen #endif gegeben. Wenn der Wert von Code-Typ code ist, dann enthält die Zeile keine Präprozessor Anweisungen, sondern normalen Code. Da Wert elif als Erweiterung betrachtet werden kann, wird auf sie nicht weiter in unserer Arbeit eingegangen. Der Code-Typ einer Zeile wird bei der Erstellung eines neuen Knotens in Knotentyp τ überführt. Der Code-Typ if wird in τ gleich MAPPING, der Code-Typ code wird in τ gleich ARTIFACT und der Code-Typ else wird in τ gleich ELSE überführt. Der Code-Typ endif hat keine Überführung in Knotentyp τ, diese Code-Type wird nur intern von dem Algorithmus verwendet. Eine andere Funktion ist der Diff-Typ, welcher für eine Zeile angibt, ob diese Zeile hinzugefügt wurde, entfernt wurde oder unverändert geblieben ist. Der Diff-Typ $+$ sagt, dass die Zeile hinzugefügt wurde, $-$ sagt, dass die Zeile entfernt wurde und $\circ$ sagt,

dass die Zeile unverändert geblieben ist. Der Diff-Typ wird, auch wie der Code-Typ, bei der Erstellung eines neuen Knotens in Δ überführt. Dabei wird + in +, − in− und ∘ in ∘ überführt. 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.

Algorithmus 1: Erstellung eines Variation-Diffs aus einem Patch

```
   Data: ein textbasierter Diff
   Result: ein Variation-Diff
 1 erstelle den Wurzelknoten
 2 initialisiere ein Stack/Keller before mit dem Wurzelknoten
 3 initialisiere ein Stack/Keller after mit dem Wurzelknoten
 4
 5 foreach Zeile in dem Patch/Diff do
 6 |   δ ← identifiziere Diff-Typ der Zeile
 7 |   γ ← identifiziere Code-Typ der Zeile
 8 |   σ ← identifiziere relevante Stacks mithilfe von δ
 9 |
10 |   if γ = endif then
11 |   |   Entferne, solange Knoten von allen Stacks in σ, bis γ = if entfernt wurde
12 |   else
13 |   |   erstelle einen neuen Knoten mit δ, γ und gerichtete Kanten von Elternknoten
   |   |    aus σ zu dem neu erstellten Knoten
14 |   |   if γ ≠ code then
15 |   |   |   füge den neuen Knoten σ hinzu
16 |   |   end
17 |   end
18 end
```

Der Algorithmus arbeitet wie folgt. Ganz am Anfang wird der Wurzelknoten in Zeile 1 erstellt. Danach werden zwei Stacks erstellt und jeweils mit dem Wurzelknoten initialisiert, was in Zeilen 2 und 3 des Algorithmus 1 zu sehen ist. Die Stacks speichern dabei die Elternknoten. Ein Stack speichern die Elternknoten in davor Zustand und der anderer im danach Zustand. Beide Stacks werden mit Wurzelknoten befühlt, welcher den ganzen Diff repräsentiert und hat deshalb als einziger Knoten in Variation-Diff keinen Elternknoten. In Zeile 5 ist eine Schleife zu sehen, welche über alle Zeilen des textbasierten Diffs geht. Dabei wird für jede Zeile zuerst der Diff-Typ δ in Zeile 6 und dann der Code-Typ γ in Zeile 7 ermittelt. In Zeile 8 werden die relevanten Stacks σ anhand von Diff-Typ δ bestimmt, und zwar wie folgt:

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

Diese Informationen werden zum einen dazu gebraucht für Algorithmus interne Berechnungen und zum anderen zur Erstellung von Knoten gebraucht. Danach in Zeile 10 kommen wir zu einer if-Abfrage. Wenn der Code-Typ der bearbeiteten Zeile endif entspricht, dann wird aus den relevanten Stacks in σ solange Knoten entfernt bis man ein Knoten mit dem Code-Type γ if entfernt hat. Falls beide Stacks relevant sind, muss der if-Knoten in beiden Stacks gefunden werden. Dieses Vorgehen ist Notwendig, da wenn eine endif-Anweisung kommt, muss der dazugehörige if-Block oder if-else-Block zu Ende sein. Das führt mit sich das die dazugehörigen if-Knoten und else-Knoten keine Elternknoten mehr sein können und aus den Stacks entfernt

werden müssen. Wenn der Code-Typ nicht endif entspricht, kommen wir in den else-Teil ab Zeile 12 des Algorithmus 1. Dort wird zuerst ein neuer Knoten erstellt, dazu unter anderem wird Diff-Typ δ und Code-Typ γ verwendet. Es werden auch Kanten von Elternknoten aus den Stacks von σ zu diesen neuen Knoten erstellt. Als Nächstes wird in Zeile 14 überprüft, ob der erstellter Knoten nicht von Code-Typ code ist. Diese Abfrage ist nötig da nur solche Knoten ein Elternknoten sein können. Den Code-Typ endif kann dieser Knoten nicht haben, wegen der if-Abfrage in Zeile 10, welche nicht zulässt, dass 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 σ hinzugefügt, sonst wenn der Knoten, den Code-Type code hat, wird nichts gemacht.

Der vorgestellter Algorithmus ist für das Parsen von textbasierten Diffs, welche aus mit C-Präprozessor-Annotierten Code entstanden sind, zu Variation-Diffs ausgelegt aber es ist auch möglich den Algorithmus zum Parsen von C-Präprozessor-Annotiertem Code zu einem Variation-Tree zu verwenden. Wir reduzieren das Problem ein Variation-Tree zu parsen auf das Problem ein Variation-Diff zu parsen. Um das anstellen zu können, müssen wir zwei Sachen beachten. 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 einen 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 unserem C-Präprozessor-Annotierten Code als Davor-Zustand und Danach-Zustand. Danach bekommen ein textbasiertes Diff in dem jede Zeile als unverändert markiert ist. 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, welcher in ein Variation-Tree umgewandelt werden muss. Um dies anzustellen, bilden wir eine Projektion des Variation-Diffs auf den Davor- bzw. Danach-Zustand und bekommen einen Variation-Tree. Es ist irrelevant welcher von den beiden Zuständen genommen wird, da der Davor-Zustand gleich dem Danach-Zustand sein soll. Mit den gezeigten Zwischenschritten 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 das untenstehende, beispielhafte Stück C-Code mit C-Prüprozessor-Annotationen an und veranschaulichen das Vorgehen in der Abbildung 3.3. Diese Abbildung zeigt Kasten, welche den Variation-Diff und relevanten Stacks für angegebene Zeile aus dem C-Code unten, nachdem der Algorithmus diese Zeile verarbeitet hat, enthalten. Hier ist ein C-Code gegeben. Wir brauchen aber ein textbasiertes Diff. Es wird wie in dem Abschnitt davor vorgegangen und dieser C-Code 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 des C-Codes, dort befindet sich eine normale Codezeile, welche nicht annotiert ist. Es ergibt sich, dass diese Zeile den Code-Typ code und den Diff-Typ none hat. 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-Typs für jede Zeile sein. Dasselbe gilt auch für die relevanten Stacks in σ, 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 erstellt und dem Variation-Diff hinzugefügt, wie das aussieht, ist in Abbildung 3.3 in dem Kasten „Nach Z.1“ zu sehen. Der erstellter Knoten hat als Elternknoten den Wurzelknoten, wie es in den Stacks zu sehen ist. In Abbildung 3.3 im Kasten „Nach Z.2“ ist der Variation-Diff und die relevanten Stacks nach der

```
1 f();
2 #if(A)
3     #if(B||C)
4         g();
5     #else
6         z();
7     #endif
8     x();
9 #endif
```

Abbildung 3.2: Beispiel für mit C-Präprozessor-Annotierten Code

Bearbeitung der Zeile 2 zu sehen. Es wurde ein neuer Knoten erstellt, welcher eine if-Anweisung enthält und in beide Stacks wurden dieser Knoten hinzugefügt. Die Schleife wurde fast gleich wie im vorherigen Fall durchgelaufen, außer an der letzten if-Abfrage. Diese Abfrage war bei der Zeile 1 false dieses Mal, da wir keinen Code-Typ code haben, wird diese Abfrage ausgeführt und der neu erstellter 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 der Erstellung 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. 3.3 „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 3.3 „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, welcher sich in der Zeile 11 des Algorithmus 1 befindet. Der Algorithmus entfernt nun von beiden Stacks jeweils so lange Knoten, bis ein if-Knoten entnommen wurde. Dabei werden aus den Stacks die Knoten mit else und if(B||C) entnommen und übrig bleiben der if(A) Knoten und der Wurzelknoten. Dieser Schritt verändert den Variation-Diff nicht. Im nächsten Schritt treffen wir wieder auf eine Codezeile und erstellen einen Knoten und fügen den dem Variation-Diff hinzu, welcher in Abbildung 3.3 „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(A) 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. Wir brauchen einen Variation-Tree statt eines Variation-Diffs welchen wir bekommen habe. Dazu müssen wir noch eine Projektion auf den Zustand davor oder danach bilden. Schlussendlich erhalten wir einen Variation-Tree, welcher genauso aufgebaut ist, wie der Variation-Diff aus der Abbildung 3.3 Kasten „Nach Z.8“.

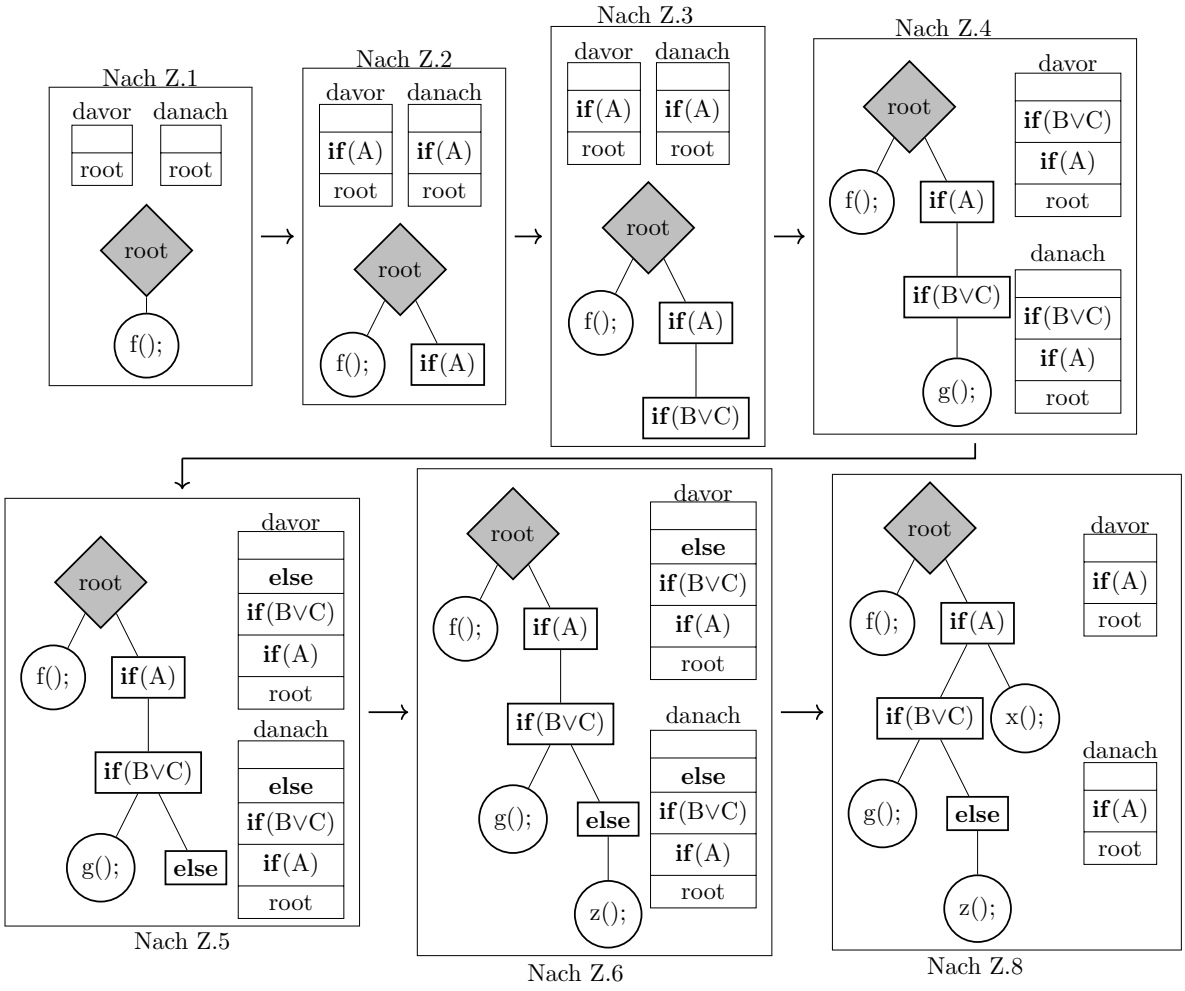


Abbildung 3.3: Beispiel für den Parsen Algorithmus von Viegener

Dieser Algorithmus kann sowohl gewöhnliche Variation-Diffs nach Definition 3.2 als auch geordneten Variation-Diffs nach Definition 3.5 erstellen. Das ist möglich, da die Reihenfolge der Kindknoten in Zeile 12 festgelegt wird.

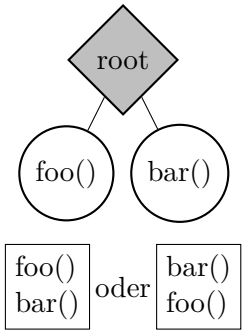
Jetzt wissen wir wie der Parser von Viegener funktioniert und der Parser auch für mehr als nur eine Definition von Variation-Diff zu gebrauchen ist. Dies können wir uns zunutze machen, wenn wir eine eigene Definition von Variation-Diff und Variation-Tree ausarbeiten werden. Es ist nun an der Zeit zu sehen, welche Informationen durch das Parsen und der Definitionen nach verloren gehen und wie wir diese Informationen zurückbekommen können.

3.3 Verlorengehende Informationen und deren Wiederherstellung

Nachdem wir uns mit dem Beschäftigt haben, was Variation-Trees und Variation-Diffs sind und wie dieser aus mit C-Präprozessor-Annotierten Code und textbasierten Diffs erzeugt werden. Werden wir uns damit auseinandersetzen welche Informationen während des Parsens verloren gehen. Also welche Informationen sind im mit C-Präprozessor-Annotierten Code bzw. textbasierten Diff erhalten, aber nach dem Parsen nicht in Variation-Tree bzw. Variation-Diff zu finden sind. Und wie wir diese Informationen zurückbekommen können, um das Unparsen zu bewerk-

stelligen. Die verlorengehende Informationen sind die Ordnung der Zeilen, die Position von Zeilen mit einem #endif innerhalb von textbasierten Diff bzw. mit C-Präprozessor-Annotierten Code und der Inhalt aller Zeilen inklusive der Zeilen mit #endif und #else.

Der Definition von Variation-Tree bzw. Variation-Diff nach haben, die Kinder eines Knotens keine Reihenfolge. Deshalb können wir nicht wissen, bei dem Unparsen welcher Knoten zuerst kommt und welcher danach. Zum Beispiel ein Variation-Tree kann mehrdeutig verstanden wäre, was für uns nicht zu gebrauchen ist.



Um das Problem umzugehen, wie wir schon erwähnt haben, verwenden wir statt der normalen Definition von Variation-Tree und Variation-Diff die Definition von geordneten Variation-Tree und geordneten Variation-Diff. Diese Definition hat eine Ordnung bei den Kinderknoten. Deshalb kann sie uns eine eindeutige Reihenfolge geben, so das keine Mehrdeutigkeiten in Bezug auf das wie die Knoten eingeordnet sind vorkommt. Das Umsetzen dieser Definition von dem Parser stellt für uns keine Schwierigkeiten dar. Die Definitionen von normalen Variation-Tree bzw. Variation-Diff unterscheiden sich von geordneten Variation-Tree bzw. geordneten Variation-Diff nur um die Ordnung O. Was zu Folge hat das alles andere so wie gewohnt umgesetzt werden kann. Die Ordnung selbst muss dann gesetzt werde, wenn ein neuer Knoten entsteht und er seine Eltern bekommt. Dies findet in der Zeile 12 des Algorithmus 1. Dort stehen keine genaueren Angaben zu Erstellung des Knotens, also kann diese Zeile auch um die Setzung der Ordnung erweitert werden. Damit haben wir unseren ersten Informationsverlust beseitigt.

Als nächstes Beschäftigen wir ums mit dem Verlust der Position von #endif. Es gibt keinen Knoten innerhalb von Variation-Trees und Variation-Diffs , welcher #endif repräsentiert. Deshalb wissen wir nicht, wo sich die Zeilen mit #endif befinden müssen wenn wir das gegebene unparsen. Zu Beschaffung diese Information muss die Position von #endif aus der gegebener Information rekonstruiert werden. In dieser Beschreibung wird einfachheitshalber so gehandelt, als gäbe es ein Knoten für #endif. Um zu bestimmen, ob ein Knoten als sein letztes Kinderknoten ein #endif besitzen wird, müssen wir prüfen ob der Knoten τ gleich mapping hat. Wenn das zutrifft, wird ein Dummyknoten als neuer letzter Kindknoten zu diesem Knoten hinzugefügt. Nach dieser Handlung wissen wir, dass die Zeile mit dem #endif nach dem Inhalt allen anderen Kinderknoten und dessen Unterbäumen kommt. Damit haben wir auch eine Möglichkeit diesen Information-Verlust zu beseitigen.

Als letztes müssen wir uns damit beschäftigen wie wir den Inhalt aller Zeilen aus Variation-Tree oder Variation-Diff bekommen können. Bei Zeilen mit #endif und #else wäre es möglich mit Heuristiken zu arbeiten, da die Einrückung von #endif und #else im Falle von C-Präprozessor nicht dazu führt, das Code fehlerhaft wird. Es wäre möglich eine Heuristik, wie z.B. das #endif und #else so weit eingerückt sind, wie der dazugehörige if oder das #else und #endif immer am Zeilenanfang sind, zu verwenden. Bei der Verwendung einer Heuristik, können wir nicht die

exakte Gleichheit garantieren, erfordert aber keine extra Aufwände. Dazu könnten Kommentare in diesen Zeilen nicht wiederbekommen werden. Bei Knoten mit τ gleich artifact, könnte man so vorgehen, das man in Label die Zeile abspeichert. Der Definition von Label nach werden dort entweder ein Implementierungsartefakt oder eine aussagenlogische Formel gespeichert. Ein Implementierungsartefakt kann dabei mehr als nur die Codezeile sein. Ein Implementierungsartefakt ist dabei eine identifizierbare Einheit mit beliebiger Granularität innerhalb eines Softwareprojekts [3]. Unsere Arbeit ist darauf ausgelegt, dass wir einen Unparser bereitstellen, welcher aus Variation-Trees bzw. Variation-Diffs mit C-Präprozessor-Annotierten Code bzw. textbasiertes Diff erstellt. Wir konnten fordern, dass das Label als Implementierungsartefakt die Codezeile abspeichert. Für Knoten mit τ gleich else oder mapping würde das aber nicht funktionieren. Für Knoten mit τ gleich else wird der Definition nach überhaupt kein Label gespeichert. Für Knoten mit τ gleich mapping wird das Label eine aussagenlogische Formel speichern, aber die Bedingungen in der C-Präprozessor-Annotation sind nicht immer eine aussagenlogische Formel und muss deshalb durch boolean abstraction [3] in solche umgewandelt werden. Das kann z.B. so aussehen: #if A(x) $>$ 3 ist gegeben und nach boolean abstraction sieht es dann so aus #if A___LB___x___RB___GT___3. Dabei ist eine zurück Umwandlung nicht garantiert, da wir nicht sicher sein können das z.B A___LB___x___RB___GT___3 nicht als ein Variablenname gewählt wurde und damit keine Umwandlung benötigt. Es ergibt sich, dass das Label nur für τ gleich artifact gut die Zeile speichern kann, bei den anderen Zeilen gibt es Schwierigkeiten. Um diese Informationen auch im Variation-Tree und Variation-Diff zu haben, müssen wir diese Informationen explizit speichern. Die Definitionen von Variation-Tree, Variation-Diff, geordneten Variation-Tree und geordneten Variation-Diff sehen aber dafür nichts vor. Deshalb müssen wir die Definition von Variation-Diff bzw. geordneten Variation-Tree und Variation-Diff bzw. geordneten Variation-Diff erweitern. Obwohl wir für τ gleich artifact die Zeile in dem Label speichern können, werden wir wegen der Einheitlichkeit alle Zeile in der Erweiterung speichern und die nicht unterscheiden. Wir erweitern die Definitionen wie folgt:

Definition 3.11. *Ein* speichernder Variation-Tree (V, E, r, τ, l, M) *ist für V, E, r, τ und l so definiert wie in der Definition 3.1 und M : V → (String ,String) speichert in der ersten Stelle des Tupels die Zeile, welche der Knoten darstellt und an der zweiten Stelle wird für Knoten mit τ gleich mapping die dazugehörige Zeile mit #endif gespeichert, für die restlichen Knoten wird dort Null gespeichert.*

Definition 3.12. *Ein* speichernder Variation-Diff $(V, E, r, \tau, l, \Delta, M)$ *ist für V, E, r, τ, l und Δ so definiert wie in der Definition 3.2 und M : V → (String ,String) speichert in der ersten Stelle des Tupels die Zeile, welche der Knoten darstellt und an der zweiten Stelle wird für Knoten mit τ gleich mapping die dazugehörige Zeile mit #endif gespeichert, für die restlichen Knoten wird dort Null gespeichert.*

Definition 3.13. *Ein* geordneter, speichernder Variation-Tree (V, E, r, τ, l, O, M) *ist für V, E, r, τ, l und O so definiert wie in der Definition 3.5 und M : V → (String ,String) speichert in der ersten Stelle des Tupels die Zeile, welche der Knoten darstellt und an der zweiten Stelle wird für Knoten mit τ gleich mapping die dazugehörige Zeile mit #endif gespeichert, für die restlichen Knoten wird dort Null gespeichert.*

Definition 3.14. *Ein* geordneter, speichernder Variation-Diff $(V, E, r, \tau, l, O_{before}, O_{after}, M)$ *ist für V, E, r, τ, l, O_{before} und O_{after} so definiert wie in der Definition 3.6 und M : V → (String ,String) speichert in der ersten Stelle des Tupels die Zeile, welche der Knoten darstellt und an der zweiten Stelle wird für Knoten mit τ gleich mapping die dazugehörige Zeile mit #endif gespeichert, für die restlichen Knoten wird dort Null gespeichert.*

Dazu nachdem wir die neuen Variation-Trees und Variation-Diffs definiert haben, müssen wir noch entsprechende Projektionen definieren.

Definition 3.15. *Die Projektion $project_M(D,t)$ für das speichernde Variation-Diff ist das Entfernen von Δ, M und der Knoten und Kanten, welche zu der Zeit t nicht vorhanden sind.*
$project_M((V,E,r,\tau,l,\Delta,M),t) := (\{v \in V | \textsc{exists}(t,\Delta(v))\},\{e \in E | \textsc{exists}(t,\Delta(e))\},r,\tau,l,M)$

Definition 3.16. *Die Projektion $project_{OM}(D,t)$ für das geordnete, speichernde Variation-Diff ist das Entfernen von Δ, M und der Knoten und Kanten, welche zu der Zeit t nicht vorhanden sind. Dazu wird nur der Zeit entsprechende Ordnung O_t behalten.*
$project_{OM}((V,E,r,\tau,l,\Delta,O_{before},O_{after},M),t)$
$:= (\{v \in V | \textsc{exists}(t,\Delta(v))\},\{e \in E | \textsc{exists}(t,\Delta(e))\},r,\tau,l,O_t,M)$

Da wir die Definitionen erweitert haben, ist es möglich die speichernden Variation-Trees bzw. Variation-Diffs in normale Variation-Trees bzw. Variation-Diffs und geordnete, speichernden Variation-Trees bzw. Variation-Diffs in geordnete Variation-Trees bzw. Variation-Diffs umzuwandeln.

Definition 3.17. *$reduce_{MVT}$ wandelt ein speicherndes Variation-Tree (Definition 3.11) in ein normales Variation-Tree (Definition 3.1)*
$reduce_{MVT}((V,E,r,\tau,l,M)) := (V,E,r,\tau,l)$.

Definition 3.18. *$reduce_{MVD}$ wandelt ein speicherndes Variation-Diff (Definition 3.12) in ein normales Variation-Diff (Definition 3.2)*
$reduce_{MVD}((V,E,r,\tau,l,\Delta,M)) := (V,E,r,\tau,l,\Delta)$

Definition 3.19. *$reduce_{MOVT}$ wandelt ein geordnetes, speicherndes Variation-Tree (Definition 3.13) in ein geordnetes Variation-Tree (Definition 3.5)*
$reduce_{MOVT}((V,E,r,\tau,l,O,M)) := (V,E,r,\tau,l,O)$

Definition 3.20. *$reduce_{MOVD}$ wandelt ein geordnetes, speicherndes Variation-Diff (Definition 3.14) in ein geordnetes Variation-Diff (Definition 3.6)*
$reduce_{MOVD}((V,E,r,\tau,l,\Delta,O_{before},O_{after},M)) := (V,E,r,\tau,l,\Delta,O_{before},O_{after})$

Es bleibt uns nur noch zu zeigen das die Reihenfolge der Anwendung von *project* und *reduce* irrelevant ist. Dann zusammen mit den gezeigten aus Kapitel 3.1 ist die Abbildung 3.4 gegeben, welche eine Erweiterung der Abbildung 3.1 ist. Die Abbildung 3.4 veranschaulicht die Transformationen von geordneten speichernden Variation-Trees, geordneten speichernden Variation-Diffs, speichernden Variation-Trees, speichernden Variation-Diffs, geordneten Variation-Trees, geordneten Variation-Diffs, normalen Variation-Trees und normalen Variation-Diffs.

Lemma 3.21. *Für einen speichernden Variation-Diff D und eine Zeit $t \in \{before, after\}$ gilt $reduce_{MVT}(project_M(D,t)) = project(reduce_{MVD}(D),t)$.*

Beweis. $reduce_{MVT}(project_M(D,t))$
$= reduce_{MVT}(project_M((V,E,r,\tau,l,\Delta,M),t))$
$= reduce_{MVT}((\{v \in V | \textsc{exists}(t,\Delta(v))\},\{e \in E | \textsc{exists}(t,\Delta(e))\},r,\tau,l,M))$
$= (\{v \in V | \textsc{exists}(t,\Delta(v))\},\{e \in E | \textsc{exists}(t,\Delta(e))\},r,\tau,l)$
$= project((V,E,r,\tau,l,\Delta),t)$
$= project(reduce_{MVD}(V,E,r,\tau,l,\Delta,M),t)$
$= project(reduce_{MVD}(D),t)$

□

Lemma 3.22. *Für einen geordneten, speichernden Variation-Diff D und eine Zeit $t \in \{before, after\}$ gilt $reduce_{MOVT}(project_{OM}(D,t)) = project_O(reduce_{MOVD}(D),t)$.*

Beweis. $reduce_{MOVT}(project_{OM}(D,t))$
$= reduce_{MOVT}(project_{OM}((V,E,r,\tau,l,\Delta,O_{before},O_{after},M),t))$
$= reduce_{MOVT}((\{v \in V|\text{EXISTS}(t,\Delta(v))\},\{e \in E|\text{EXISTS}(t,\Delta(e))\},r,\tau,l,O_t))$
$= (\{v \in V|\text{EXISTS}(t,\Delta(v))\},\{e \in E|\text{EXISTS}(t,\Delta(e))\},r,\tau,l,O_t)$
$= project_O((V,E,r,\tau,l,\Delta,O_{before},O_{after}),t)$
$= project_O(reduce_{MOVD}(V,E,r,\tau,l,\Delta,O_{before},O_{after},M),t)$
$= project_O(reduce_{MOVD}(D),t)$

□

Nach dem Ganzen ergibt sich, ein folgendes Schaubild:

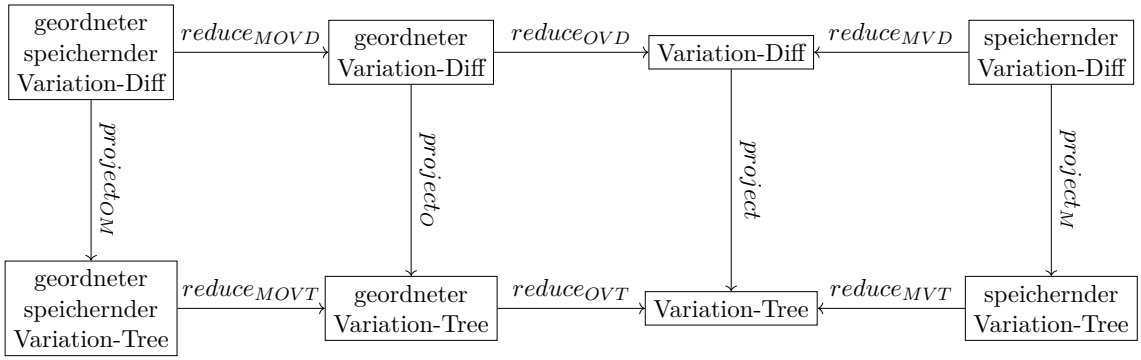


Abbildung 3.4: Transformationen von verschiedenen Variation-Trees und Variation-Diffs

Nachdem wir uns mit den neuen Definitionen beschäftigt haben, müssen wir noch den Parsen anpassen. Damit der Parser auch diese Definitionen umsetzen kann. Dazu muss man die Zeile 11 des Algorithmus etwas erweitern, damit der Algorithmus den Knoten mit $\gamma =$ if sich merkt. Dazu muss man noch eine neue Zeile zu dem Algorithmus zufügen und das ist die Zeile 12 des Algorithmus 2. Dort wird ein Tupel mit dazugehöriger Zeile des Knotens und der Zeile, mit dem #endif, erstellt. Dann wird diese Tupel in M für den gemerkten Knoten gespeichert. Alle anderen Zeilen werden für ihre Knoten jeweils in Zeile 15 des Algorithmus 2 gespeichert. Dort wird ein Tupel mit der Zeile und Null erstellt und für den neu erstellten Knoten gespeichert. Damit sind wir in der Lage die neuen Definitionen auch aus mit C-Präprozessor-Annotierten Code und textbasierten Diffs zu erzeugen.

Algorithmus 2: Erstellung eines Variation-Diffs, welcher sich #else und #endif merkt, aus einem Patch

```
   Data: ein textbasierter Diff
   Result: ein Variation-Diff
 1 erstelle den Wurzelknoten
 2 initialisiere ein Stack/Keller before mit dem Wurzelknoten
 3 initialisiere ein Stack/Keller after mit dem Wurzelknoten
 4
 5 foreach Zeile in dem Patch/Diff do
 6 |  δ ← identifiziere Diff-Typ der Zeile
 7 |  γ ← identifiziere Code-Typ der Zeile
 8 |  σ ← identifiziere relevante Stacks mithilfe von δ
 9 |
10 |  if γ = endif then
11 |  |  Entferne, solange Knoten von allen Stacks in σ, bis ein Knoten v γ = if entfernt
   |  |   wurde
12 |  |  M(v) ← (M(v)[0],Zeile)
13 |  else
14 |  |  erstelle einen neuen Knoten v mit δ, γ und gerichtete Kanten von Elternknoten
   |  |   aus σ zu dem neu erstellten Knoten
15 |  |  M(v) ← (Zeile,Null)
16 |  |  if γ ≠ code then
17 |  |  |  füge den neuen Knoten σ hinzu
18 |  |  end
19 |  end
20 end
```

Mit der Erweiterung der Definitionen können wir den Inhalt der Zeile mit #endif oder #else bekommen und damit ist dieser Informationsverlust auch beseitigt.

Nachdem wir herausgefunden haben, welche für das Unparsen relevante Information verloren geht und wie man diese Information Zurück bekommen kann, sind wir in der Lage das alles zu nutzen, um einen Algorithmus zum Unparsen zu entwickeln.

3.4 Unparsing

Nachdem wir festgestellt haben welche Informationen während des Parsens verloren gehen und wie diese zurückzubekommen sind, sind wir in der Lage das Unparsen zu bewerkstelligen. Darüber geht es in diesem Unterkapitel. Wir stellen unseren Algorithmus für das Unparsen von speichernden, geordneten Variation-Trees und ein Vorgehen zum Unparsen von speichernden, geordneten Variation-Diffs.

Unser Algorithmus ist der Algorithmus 3 und ist nur für das Unparsen von speichernden, geordneten Variation-Tree zu einem mit C-Präprozessor-Annotierten Code bestimmt. Wenn man M aus dem Algorithmus durch eine Heuristik ersetzt, ist es auch möglich mithilfe unseres Algorithmus geordnete Variation-Trees umzuparsen. Der Algorithmus basiert auf der Tiefensuche. Ein Variation-Tree ist ein Baum, dessen Knoten die Zeilen des mit C-Präprozessor-Annotierten Codes enthalten. Dabei wenn ein Knoten Kinderknoten hat, bedeutet es das dieser Knoten τ gleich mapping oder τ gleich else hat. Dazu wissen wir dadurch, dass die Anweisungen im Kinder-

knoten von der Anweisung des Elternknotens eingeschlossen werden. Wegen so einer Anordnung ist die Verwendung von Tiefensuche von Vorteil. Die Tiefensuche besucht alle Knoten effizient und dazu geht die Tiefensuche zunächst ein Pfad vollständig in die Tiefe, bevor abzweigende Pfade beschritten werden. Die Auswahl des nächsten Knotens welcher besucht wird muss etwas geändert werden, damit die Pfade der Reihenfolge nach beschritten werden. Dazu werden die Kinderknoten eines Knotens den Stack in umgedrehter Reihenfolge hinzugefügt. Es ergibt sich, dass der letzte Kinderknoten auch als letzter drankommt und der erster als erster. Das zusammen ergibt, das unser Algorithmus die Knoten so besucht wie die in den Knoten enthaltene Zeilen angeordnet werden müssen.

Jetzt schauen wir und den Algorithmus an, nachdem wir sein Konzept besprochen haben. In Zeile 1 des Algorithmus 3 wird ein Stack initialisiert, so wie in der Tiefensuche. Danach in Zeile 2 wird ein String initialisiert, in dem am Ende der gesamte annotierter Code gespeichert wird. Von Zeile 3 bis Zeile 10 werden die Kinderknoten des Wurzelknotens auf den Stack gelegt. Das muss extra gemacht werden, da der Wurzelknoten keine Zeile enthält als Label, sondern true. Damit würde er das Ergebnis verfälschen. Genauer betrachtet wird folgendes gemacht, in der Zeile 3 werden die Kinderknoten des Wurzelknotens bestimmt und als Menge abgespeichert. Dann wird ein Array erstellt, dessen Länge gleich der Anzahl der Kinderknoten des Wurzelknotens ist. Als Nächstes in Zeile 5 wird über alle Kinderknoten iteriert und die Kinderknoten gemäß ihrer Ordnung im Array abgespeichert. Zum Schluss werden die Kinderknoten in umgekehrter Reihenfolge den Stack hinzugefügt. In Zeile 11 beginnt eine while-Schleife, welche so lange läuft bis der Stack leer wird. In der Schleife wird als Erstes der oberste Knoten von Stack genommen und sich gemerkt, das ist in Zeile 12. Als Nächstes in Zeile 13 wird, geprüft ob τ des Knotens gleich mapping ist. Wenn das der Fall ist, dann wird zuerst ein Dummyknoten, welcher die Information zu #endif aus M[1] enthält, erstellt, dann wird dieser Dummyknoten dem Stack hinzugefügt. Der Dummyknoten hat τ gleich artifact und im M[0] diesen Dummyknotens wird der Inhalt aus M[1] gespeichert. Da dieser Knoten nur intern im Algorithmus verwendet wird, kann er solche Werte annehmen, die die Definition nicht vorsieht. Wenn das nicht der Fall ist, wird nichts gemacht und die Abfrage übersprungen. In Zeile 17 wird der Inhalt von M[0] String „ergebniss“ hinzugefügt. Ab Zeile 18 bis Zeile 25 werden die Kinderknoten des gerade bearbeiteten Knotens dem Stack in umgedrehter Reihenfolge hinzugefügt. Dort wird genauso vorgegangen wie in den Zeilen 3 bis 10. Damit ist der Inhalt der Schleife abgearbeitet und wir kommen zur Zeile 27, welche den String „ergebniss“ zurückgibt in dem der mit C-Präprozessor-Annotierter Code enthalten ist.

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

Data: ein Vatiation-Tree (V,E,r,τ,l)
Result: ein mit C-Präprozessor-Annotierten Code

```
1  initialisiere einen leeren Stack/Keller stack
2  initialisiere einen String ergebniss
3  kinder ← {v ∈ V | (r,v) ∈ E}
4  initialisiere ein Array array der Länge |kinder|
5  for ∀ v ∈ kinder do
6  |  array[O(v)] ← v
7  end
8  for i = |kinder| → 1 do
9  |  stack.push(kinder[i])
10 end
11 while stack nicht leer ist do
12 |  knoten ← stack.pop()
13 |  if τ(knoten) = mapping then
14 |  |  erstelle einen Dummyknoten welcher M(knoten)[1] beinhaltet
15 |  |  stack.push(Dummyknoten)
16 |  end
17 |  füge M(knoten)[0] ergebniss hinzu
18 |  kinder ← {v ∈ V | (knoten,v) ∈ E}
19 |  initialisiere ein Array array der Länge |kinder|
20 |  for ∀ v ∈ kinder do
21 |  |  array[O(v)] ← v
22 |  end
23 |  for i = |kinder| → 1 do
24 |  |  stack.push(kinder[i])
25 |  end
26 end
27 return ergebniss
```

Nachdem ihr mehr über unser Algorithmus erfahren habt, wollen wir seine Arbeitsweise in der Abbildung 3.3 verdeutlichen. In diesem Beispiel werden wir das erhaltene Variation-Tree aus der Abbildung 3.3 unparsen. Obwohl dort der Parser ein Variation-Tree erstellt, aber wir ein speichernden, geordneten Variation-Tree brauchen. Ein speichernder, geordneter Variation-Tree ist von dem Aussehen identisch, dem Variation-Tree aus Abbildung 3.3, wenn derselbe mit C-Präprozessor-Annotierter Code mithilfe von Algorithmus 2 geparst wird. Die Abbildung 3.5 zeigt in kleineren Bildern, entweder den Zustand nach Bearbeitung des Wurzelknotens oder eines Schleifendurchlaufs, und die Reihenfolgen, in der die Bilder betrachtet werden sollen, beginnend mit dem Bild ganz links oben. Ein Bild enthält das speichernde, geordnete Variation-Tree, den Stack, die Ausgabe und welcher Knoten des speichernden, geordneten Variation-Trees bearbeitet wurde. Am Anfang im Bild ganz links oben der Abbildung 3.4 sind wir außerhalb der Schliefe und es wird der Wurzelknoten abgearbeitet. Die Kinderknoten des Wurzelknotens werden dem Stack hinzugefügt. Die Ausgabe verändert sich nicht, da der Wurzelknoten keine Zeile enthält. In nächsten Bild sind wir in der Schleife. Der oberste Knoten wird von Stack genommen, der roter Pfeil zeigt auf den. Das ist ein Knoten mit τ gleich artifact, also wird die if-Abfrage in Zeile 13 des Algorithmus 3 mit falsch beantwortet und übersprungen. Danach in Zeile 17 wird M[0] des Knotens der Ausgabe hinzugefügt, was auch bei der Ausgabe im Bild zu sehen ist. Der

Inhalt des Knotens ist gleich der ersten Zeile der Ausgabe. Der Knoten ist ein Blattknoten und hat keine Kinderknoten, welche den Stack hinzugefügt werden können. Im Bild danach, wird wieder der oberste Knoten aus dem Stack genommen, deshalb ist der Knoten auf den der rote Pfeil zeigt im Stack des vorherigen Bildes vorhanden und nicht in diesem. Für den betrachteten Knoten gilt τ ist gleich mapping. Aus diesem Grund gehen wir in die if-Abfrage aus der Zeile 13 des Algorithmus 3. Dort wird ein Dummyknoten mit #endif erstellt und dem Stack hinzugefügt was wir auch im Bild sehen. Außerhalb der Abfrage wird M[0] des Knotens wider der Ausgabe hinzugefügt. Die Kinderknoten werden dem Stack hinzugefügt. Im Bild ganz rechts oben wird wie immer der oberste Knoten aus dem Stack genommen welcher mit dem roten Pfeil markiert ist. Dieser Knoten hat τ gleich mapping. Es wird gleich wie mit den vorherigen Knoten vorgegangen. Es wird ein Dummyknoten erstellt und dem Stack hinzugefügt. Außerhalb der Abfrage wird wie immer der M[0] des Knotens der Ausgabe hinzugefügt. Die Kinderknoten werden dem Stack hinzugefügt. Jetzt sind wir im Bild ganz links in der Mitte. Dieser Knoten hat τ gleich artifact, deshalb wird hier genauso wie im oberen, zweiten Bild von links vorgegangen. Im nächsten Bild wird ein Knoten mit τ gleich else bearbeitet. Dieser Knoten wurde aus dem Stack genommen. Der Knoten entspricht nicht τ gleich mapping, deshalb wird die if-Abfarge in Zeile 13 des Algorithmus 3 übersprungen. Danach wird der Inhalt von M[0] für diesen Knoten der Ausgabe hinzugefügt. Als Nächstes wird der einzige Kinderknoten dem Stack hinzugefügt. Im nächsten Bild wird wieder ein Knoten mit τ gleich artifact bearbeitet. Diese Knoten wird, genauso behandelt wie die anderen Knoten mit τ gleich artifact. Wir sind jetzt bei dem Bild in der Mitte ganz rechts. Dort wird zum erstem Mal ein Dummyknoten mit #endif bearbeitet. Der Knoten wird von Stack genommen. Es ist ein Dummyknoten und ist deshalb nicht in den gezeigten speichernde, geordneten Variation-Tree zu finden und es kann kein roter Pfeil auf den zeigen. Es wird geprüft ob der Dummyknoten τ gleich mapping hat, das ist nicht der Fall, da dieser Knoten τ gleich artifact hat. Wir überspringen die Abfrage aus Zeile 13 und gehen zu der Zeile 17 des Algorithmus 3. Dort wird der Inhalt von M[0] für diesen Knoten der Ausgabe hinzugefügt. Dieser Knoten hat keine Kinderknoten, deshalb wird dem Stack auch nichts hinzugefügt. In dem Linken unteren Bild der Abbildung 3.5 wird der mit dem roten Pfeil markierter Knoten bearbeitet. Dieser Knoten hat auch τ gleich artifact und ist kein Dummyknoten. Deshalb wird dort auch wie in anderen Fällen vorgegangen und wir sehen nichts Neues. Im letzten Bild wird wieder ein Dummyknoten bearbeitet. Das Vorgehen ist in beiden Fällen dasselbe. Damit wären wir mit dem Unparsen fertig. Wir haben das Variation-Tree aus der Abbildung ungeparst und es ist zu sehen, dass die Ausgabe im letzten Bild der Abbildung 3.5 gleich dem mit C-Präprozessor-Annotierten Code ist aus der Abbildung 3.3, die zum Parsen Variation-Tree genutzt wurde.

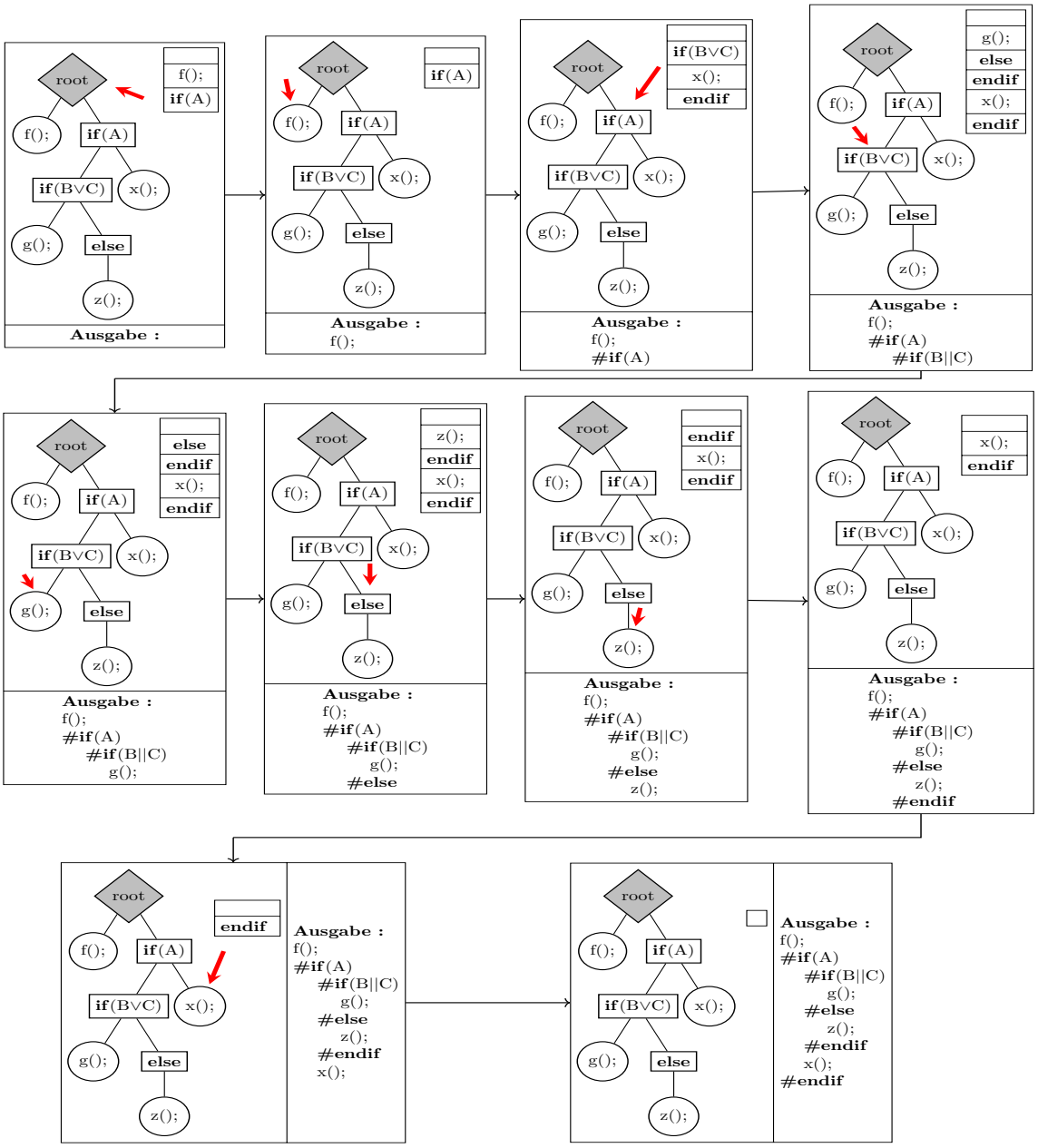


Abbildung 3.5: Beispiel für das Unparsen mithilfe unseren Algorithmus

Wir haben eine Möglichkeit gefunden speichernde, geordnete Variation-Trees umzupasren. Es bleibt uns noch eine Möglichkeit zum Unparsen von Variation-Diffs zu finden. Eine Möglichkeit zum Unparsen von Variation-Diffs bezieht sich auf speichernde, geordnete Variation-Diffs oder wenn eine Heuristik verwendet wird dann auf geordnete Variation-Diffs. Wir beschreiben das Vorgehen für speichernde, geordnete Variation-Diffs für geordnete Variation-Diffs geht es analog. Zuerst werden zwei Projektionen gebildet, jeweils auf den Zustand-Davor und den Zustand-Danach. Dadurch werden zwei speichernde, geordnete Variation-Trees erstellt. Auf diese Variation-Trees wird unser Algorithmus angewandt. Als Ergebnis erhalten wir zwei mit C-Präprozessor-Annotierte Codes. Zum Schluss wird ein Algorithmus, welcher aus Codes mit C-Präprozessor-Annotation ein textbasiertes Diff erstellen kann, auf das Ergebnis unseren Algorithmus angewandt. Damit haben wir ein Vorgehen zum Unparsen von speichernden, geordneten Variation-Diffs.

Jetzt wissen wir wie man speichernde, geordnete bzw. geordnete Variation -Trees und speichernde, geordnete bzw. geordnete Variation-Trees unparsen kann. Als Nächstes wollen wir eine Komplexitätsanalyse der Laufzeit für unseren Algorithmus durchführen.

3.5 Komplexitätsanalyse der Laufzeit

In diesen Teil des Kapitels wollen wir eine Komplexitätsanalyse der Laufzeit für unseren Algorithmus durchführen. Damit wir eine obere Schranke für seine Effizienz in Abhängigkeit von der Knotenmenge setzen können.

Bei unserer Komplexitätsanalyse der Laufzeit gehen wir von folgenden Laufzeiten für einige Anweisungen aus. Die Zeitkomplexität von Operationen mit dem Stack ist konstant. Das Zugreifen oder Setzen von Inhalten des speichernden, geordneten Variation-Trees wird als Konstanten Zeitaufwand betrachtet, da man τ, l, O, M auch als Variablen eines Knotens sehen kann und der Zugriff oder Setzung von diesen einen konstanten Aufwand hat. Die Konstante Zeitkomplexität hat auch die Verbindung von zwei Strings. Das Erstellen eines Dummyknotens wird auch als konstanten Zeitaufwand betrachtet, da ein Dummyknoten nur als Platzhalter im Stack für die #endif Zeile dient und nur M[0] und τ sich merkt. Das Setzen dieser Information ist wie schon angegeben in konstanter Zeit möglich. Wir gehen davon aus, da wir keine komplexen Datenstrukturen haben, dass die Initialisierung in konstanter Zeit möglich ist. In Zeilen 3 und 18 des Algorithmus 4 werden die Kinderknoten bestimmt. Der dazugehörige Zeitaufwand hängt davon ab, wie die Speicherung, der Kinderknoten realisiert ist. Wir gehen davon aus das die Speicherung von Kinderknoten in Form von Adjazenzliste erfolgt. Diese Annahme ergibt, dass das Erhalten der Kinderknoten eines Knotens eine konstante Zeitkomplexität hat. In Algorithmus 4 ist die Zeitkomplexität, in der O-Notation, für die einzelnen Anweisungen unseren Algorithmus zu sehen.

Algorithmus 4: Komplexitätsanalyse der Laufzeit für Algorithmus 3

```
   Data: ein Vatiation-Tree (V,E,r,τ,l)
   Result: ein mit C-Präprozessor-Annotierten Code
 1 initialisiere einen leeren Stack/Keller stack                          O(1)
 2 initialisiere einen String ergebniss                                   O(1)
 3 kinder ← {v ∈ V | (r,v) ∈ E}                                           O(1)
 4 initialisiere ein Array array der Länge |kinder|                       O(1)
 5 for ∀ v ∈ kinder do
 6 |   array[O(v)] ← v                                                   O(1)
 7 end
 8 for i = |kinder| → 1 do
 9 |   stack.push(kinder[i])                                             O(1)
10 end
11 while stack nicht leer ist do
12 |   knoten ← stack.pop()                                              O(1)
13 |   if τ(knoten) = mapping then
14 |   |   erstelle einen Dummyknoten welcher M(knoten)[1] beinhaltet     O(1)
15 |   |   stack.push(Dummyknoten)                                       O(1)
16 |   end
17 |   füge M(knoten)[0] ergebniss hinzu                                  O(1)
18 |   kinder ← {v ∈ V | (knoten,v) ∈ E}                                  O(1)
19 |   initialisiere ein Array array der Länge |kinder|                   O(1)
20 |   for ∀ v ∈ kinder do
21 |   |   array[O(v)] ← v                                               O(1)
22 |   end
23 |   for i = |kinder| → 1 do
24 |   |   stack.push(kinder[i])                                         O(1)
25 |   end
26 end
27 return ergebniss                                                      O(1)
```

Es ist zu sehen das alle einzelnen Anweisungen in unserem Algorithmus eine Laufzeit von $O(1)$ haben. Es bleib uns nur noch zu bestimmen wie viel man die Schleifen durchgelaufen werden. Es ist zu sehen das die Schleife in Zeilen 5 bis 7 und die Schleife in Zeilen 8 bis 10 dieselbe Laufzeit haben. n ist die Anzahl aller Knoten. Die Schleiche in Zeilen 5 bis 7, wird so viel mal durchgelaufen wie der Wurzelknoten Kinderknoten hat, das bezeichnen wir mit w. Dabei gilt w $<$ n oder noch genauer w $\leq$ n-1, da alle Knoten an dem Wurzelknoten hängen könne und dabei können es alles Blattknoten sein. Deshalb können wir die Laufzeit der Schleife wie folgt angeben $O(1)$ * w = $O(1$ * w $) = O$(w) = O(n-1) = O(n). Dasselbe gilt auch für die Schleife in Zeilen 8 bis 10. Damit ist die Laufzeit der Schleifen zusammen O(n) + O(n) = O(n + n) = O(2n) = O(n). In Zeilen 20 bis 22 haben wir eine Schleife, die über die Kinderknoten des Knotens v geht. Die Schleife wird k_v mal durchlaufen und k_v gibt die Anzahl der Kinderknoten des Knotens v. Unser Algorithmus geht über alle Knoten des Baumes also werden von jedem Knoten die Kinderknoten ermittelt, dessen Anzahl k_v für Knoten v angibt. Bei einem Baum hat ein Knoten nur einen einzigen Elternknoten und wird deshalb auch nur ein mal durchlaufen, weil es auch nur ein einziges Mal den Stack hinzugefügt wird. Es ergibt sich das w + $\sum^{v \in V} k_v$ = n-1 gilt. Es gilt, da ein Knoten v k_v Kinderknoten hat, dabei ist w die Anzahl der Kinderknoten des Wurzelknotens. Der Baum ist zusammenhängend also haben alle Knoten außer den Wurzelknoten einen Elternknoten und sind damit auch die Kinderknoten eines anderen Kno-

tens. Deshalb muss die Aufsummierung alle Kinderknoten die Anzahl alle Knoten außer den Wurzelknoten ergeben, was auch n-1 ist. Damit ergibt es sich das die Schleife in Zeilen 20 bis 22 genau n-1-w mal insgesamt durchlaufen wird unabhängig davon wie viel mal die äußere Schleife in Zeile 11 durchgelaufen wird. Dasselbe gilt auch für die Schleifen in Zeilen 23 bis 25. Es ergibt sich das die Laufzeit einer Schleife $O(1)$ * n-1-w = $O(1$ * (n-1-w)) = O(n-1-w) = O(n). Beide Schleifen zusammen haben eine Laufzeit von O(n) + O(n) = O(n + n) = O(2n) = O(n). Diese Laufzeit gilt nicht für einen Durchlauf der äußeren Schleife, sondern für die Arbeitszeit des gesamten Algorithmus. Es bleibt uns nur noch zu schauen wie viel mal die Schleife von Zeile 11 bis Zeile 26 durchlaufen wird. Diese Schleife wird so lange durchlaufen bis der Stack leer wird, dabei wird bei jedem Schleifendurchlauf ein Element aus dem Stack genommen. Wir müssen herausfinden wie viel Elemente insgesamt den Stack hinzugefügt werden. Wir wissen das jeder Kinderknoten des Baumes den Stack hinzugefügt wird, das sind schon n-1 Elemente. Es gibt aber noch eine Stelle im Algorithmus, welche Knoten den Stack zufügt und das ist die Zeile 15 des Algorithmus 4. Damit man herausfinden kann wie viel Knoten durch diese Stelle hinzugefügt werden, müssen wir herausfinden wie oft maximal kann die Bedingung in Zeile 13 wahr sein. Das gilt so, da es für jeden Knoten mit τ gleich mapping ein Dummyknoten erstellt wird und dem Stack hinzugefügt wird. Wir müssen herausfinden wie viel der n-1 Knoten τ gleich mapping haben können. Es kann vorkommen das alle n-1 Kinderknoten τ gleich mapping haben, welches ergibt, dass auch n-1 Dummyknoten erstellt und den Stack hinzugefügt werden. In diesem Fall wurden den Stack insgesamt 2(n-1) Elemente hinzugefügt und die äußere Schleife wird auch 2(n-1) Mal durchlaufen. Die inneren Schleifen werden wie schon ober erwähnt unabhängig von der äußeren Schleife durchlaufen. Deshalb hat auch deren Zeitkomplexität keinen Einfluss auf die Zeitkomplexität der äußeren Schleife. Die if-Anweisung aus Zeilen 13 bis 16 hat eine Laufzeit von max($O(1)$+$O(1)$,0) + $O(1)$ = max($O(1)$),0) + $O(1)$ = $O(1)$ + $O(1)$ = $O(1)$. Alle anderen Anweisungen haben auch eine Laufzeit von $O(1)$. Es folgt ($O(1)$+$O(1)$+$O(1)$+$O(1)$+$O(1)$)*2(n-1) = $O(1)$* 2(n-1) = $O(1$*2(n-1)) = O(2(n-1)) = O(n-1) = O(n). Eine Laufzeit von O(n) haben alle Schleifen und sind dabei voneinander unabhängig. Eine Laufzeit von $O(1)$ haben die Zeilen 1 bis 4. Diese Zeilen wurden nicht in Schleifen oder anderswo berücksichtigt. Die gesamte Laufzeit unseren Algorithmus ist folgende $O(1)$ + $O(1)$ + $O(1)$ + $O(1)$ + O(n) + O(n) + O(n) = O(1+1+1+1+n+n+n) = O(3n+4) = O(n). Damit konnten wir zeigen, dass die asymptotische Laufzeit unseren Algorithmus bei O(n) liegt.

Die Komplexitätsanalyse der Laufzeit für das Unparsen von speichernden, geordneten Variation-Diffs so wie es gezeigt wurde, muss über drei Algorithmen geschehen. Wir müssen eine Komplexitätsanalyse der Laufzeit für die Projektion durchführen, eine für unseren Algorithmus, was wir auch in oberen Abschnitt gemacht haben, und eine für den Diff-Algorithmus. Die Projektion kann Unterschiedlich umgesetzt werden, was sich auf die Komplexitätsanalyse der Laufzeit auswirkt. Es können auch verschiedene Diff-Algorithmen gewählt werden, welche unterschiedliche Laufzeiten haben können. All das zusammen macht es für uns schwierig eine Komplexitätsanalyse der Laufzeit für das Unparsen von speichernden, geordneten Variation-Diffs durchzuführen.

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

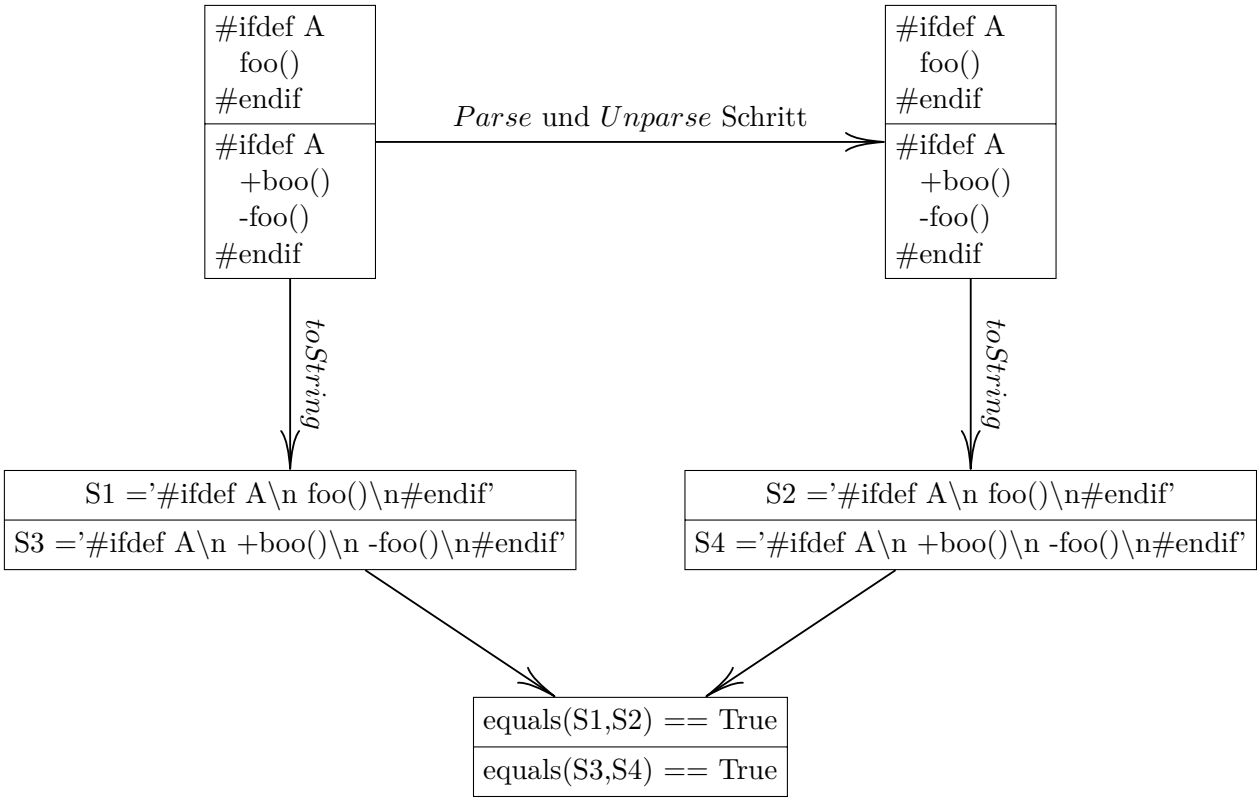


Abbildung 4.1: Beispiel für Syntaktische Gleichheit

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.

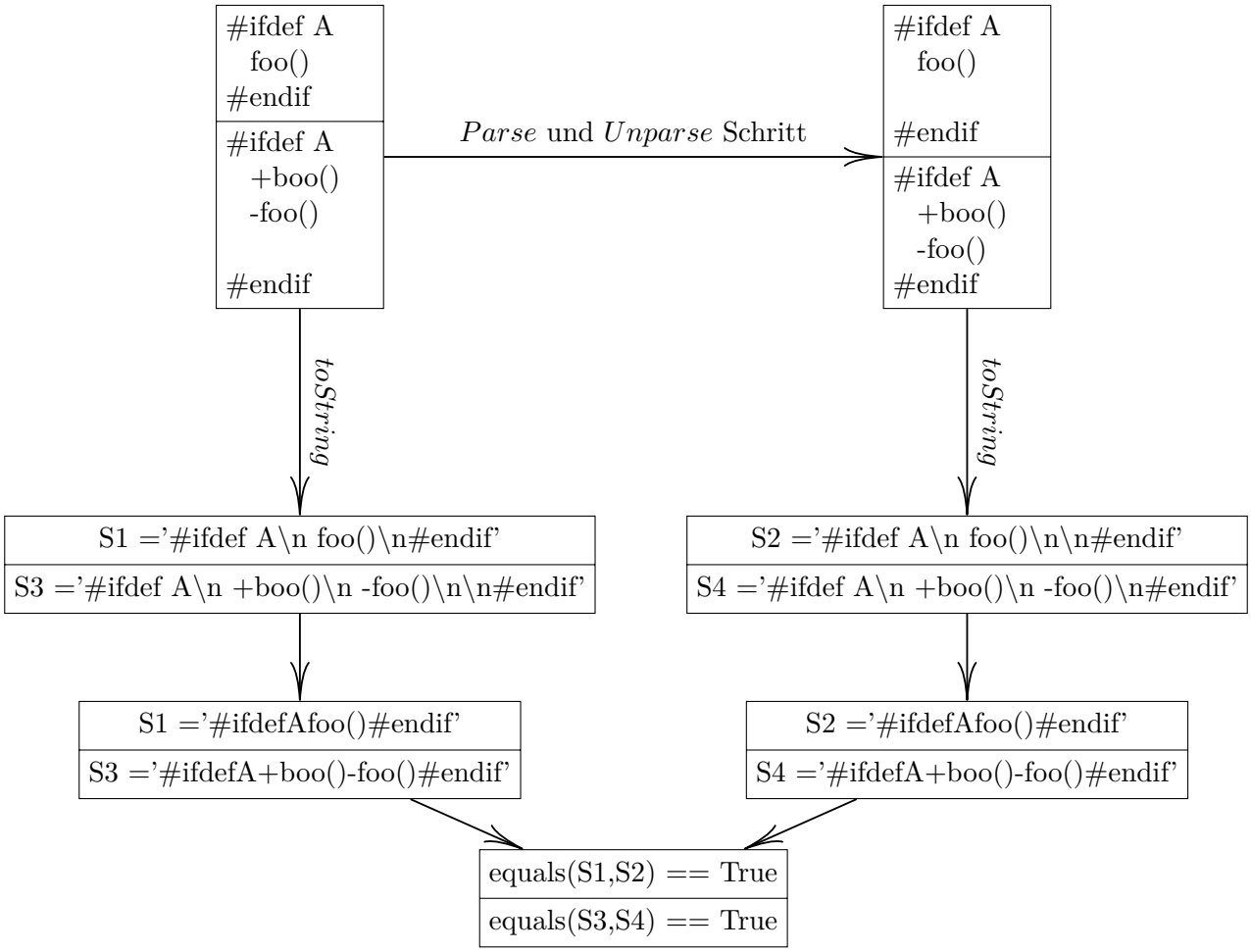


Abbildung 4.2: Beispiel für Syntaktische Gleichheit ohne Whitespace

Die semantische Gleichheit von mit C-Präprozessor-Annotierten Code werden wir nicht betrachten, da dafür wir entscheiden müssen ab 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.

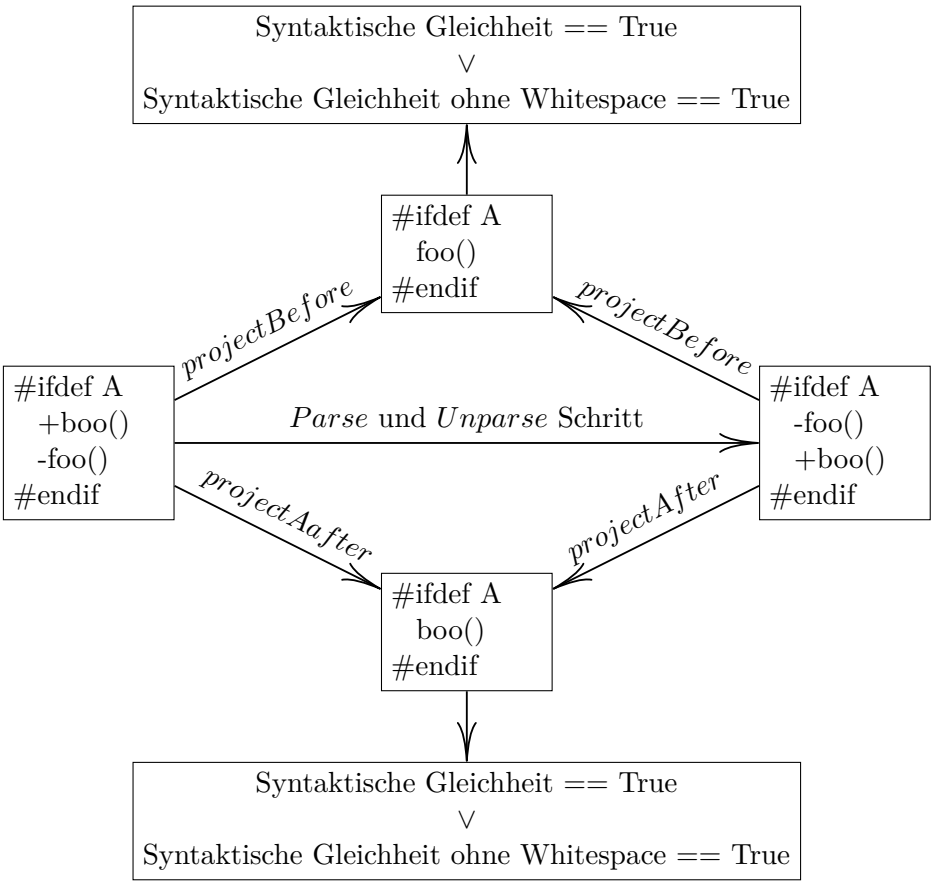


Abbildung 4.3: Beispiel für Semantische Gleichheit

5 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. Viegener, 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`.

[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`.

[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. Viegener. 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_Viegener.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, Ş. 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.