ulm university universität ulm

Konzeption und Implementierung eines Echtzeitverfahrens zur nutzerintendierten Abstraktion von Quelltextveränderungen

Sebastian Mechelke

Universität Ulm

Fakultät für Ingenieurwissenschaften und Informatik

Institut für Programmiermethodik und Compilerbau

Mai 2014

Masterarbeit im
Studiengang Medieninformatik

Erstgutachter: Prof. Dr. H. Partsch
Zweitgutachter: Prof. Dr. F. Schweiggert

Inhalt

1 Einführung

Bei dieser Arbeit handelt es sich um eine Masterarbeit, die sich den Bereichen Compilerbau, Data Mining und Werkzeugentwicklung zuordnen lässt. Unter Anwendung des Methodenwissens aus dem Medieninformatik-Studium und anhand ausgiebiger Literaturrecherchen soll ein Software-Tool entwickelt werden, welches zu jeder Editierung an einem Quelltext die nutzerintendierten Veränderungen erkennt.

1.1 MOTIVATION UND PROBLEMSTELLUNG

Während seines Lebenszyklus, wird ein Software-Projekt häufig verändert. Bisherige Forschungsarbeiten versuchen die Veränderungen zu interpretieren, indem sie die Editierdistanz auf Syntaxbäume von aufeinanderfolgenden Coderevisionen anwenden. Die resultierenden Baum-Operationen sollen als Grundlage dienen, Schlüsse über die Nutzerintention zu ziehen. Im Rahmen dieser Masterarbeit soll ein Tool konzipiert und entwickelt werden, welche diese Baumoperationen extrahieren kann.

In einem Punkt soll sich das angestrebte Verfahren jedoch von allen anderen bestehenden Verfahren unterscheiden: Die Prüfung der Veränderungen soll in Echtzeit und nicht, wie sonst üblich, anhand von Revisionen erfolgen. Verglichen mit dem Revisionsvergleich können bei der Echtzeitüberprüfung zusätzlich alle Texteingaben, welche sich zeitlich zwischen den Revisionen befinden, verarbeitet werden.

Die hierdurch zusätzlich gewonnene Information ermöglicht eine stochastisch besser abgesicherte Extraktion der Code-Veränderungen als es beim Revisionsvergleich möglich ist. Ein Beispiel hierfür ist in Abbildung 1 dargestellt. Dort werden zwei strukturell äquivalente Syntaxbäume (oben links und oben rechts), die jeweils auf gleiche Weise verändert, so dass sie in zueinander kongruenten Bäumen (unten links und unten rechts) resultieren. Der Unterschied zwischen den beiden Varianten (bzw. Spalten) liegt jedoch darin, dass auf der linken Seite zusätzlich alle Zwischenschritte (mittlere Spalte, links) bekannt sind, wie es auch bei dem angestrebten Verfahren der Fall sein wird. Anhand dieser Information kann darauf geschlossen werden, dass die Kindknoten U und X modifiziert und verschoben worden sind. Ein solcher Schluss kann bei der rechten Spalte jedoch nicht zwingend abgeleitet werden. Wie an der Abbildung zu sehen ist, sind ohne die Zusatzinformation noch alternative Schlussfolgerungen (a-c) möglich, welche zwar ebenfalls korrekt erscheinen, die eigentliche Nutzerintention jedoch weniger treffend beschreiben.

Um dieser Ambiguität zu begegnen werden in vielen bisherigen Verfahren stochastische Mittel, wie etwa Ähnlichkeitsmaße über Funktionsbezeichnern, angewandt. Solche, auf Wahrscheinlichkeiten basierenden, Interpretationen sind jedoch stets einer gewissen Unsicherheit unterworfen. Weiterhin kann der Mangel an Informationen dazu führen, dass Interpretationen, egal ob sicher oder unsicher, gar nicht erst aufgestellt werden können. Die Verfahren aus den Quellen [**1**], [**2**], [**3**], [**4**], [**5**], [**6**] und [**7**] haben gemeinsam, dass sie einen Anteil an komplett fehlenden semantischen Veränderungen aufweisen, ohne dass die Qualität der korrekt erkannten semantischen Veränderungen hinterfragt wird.

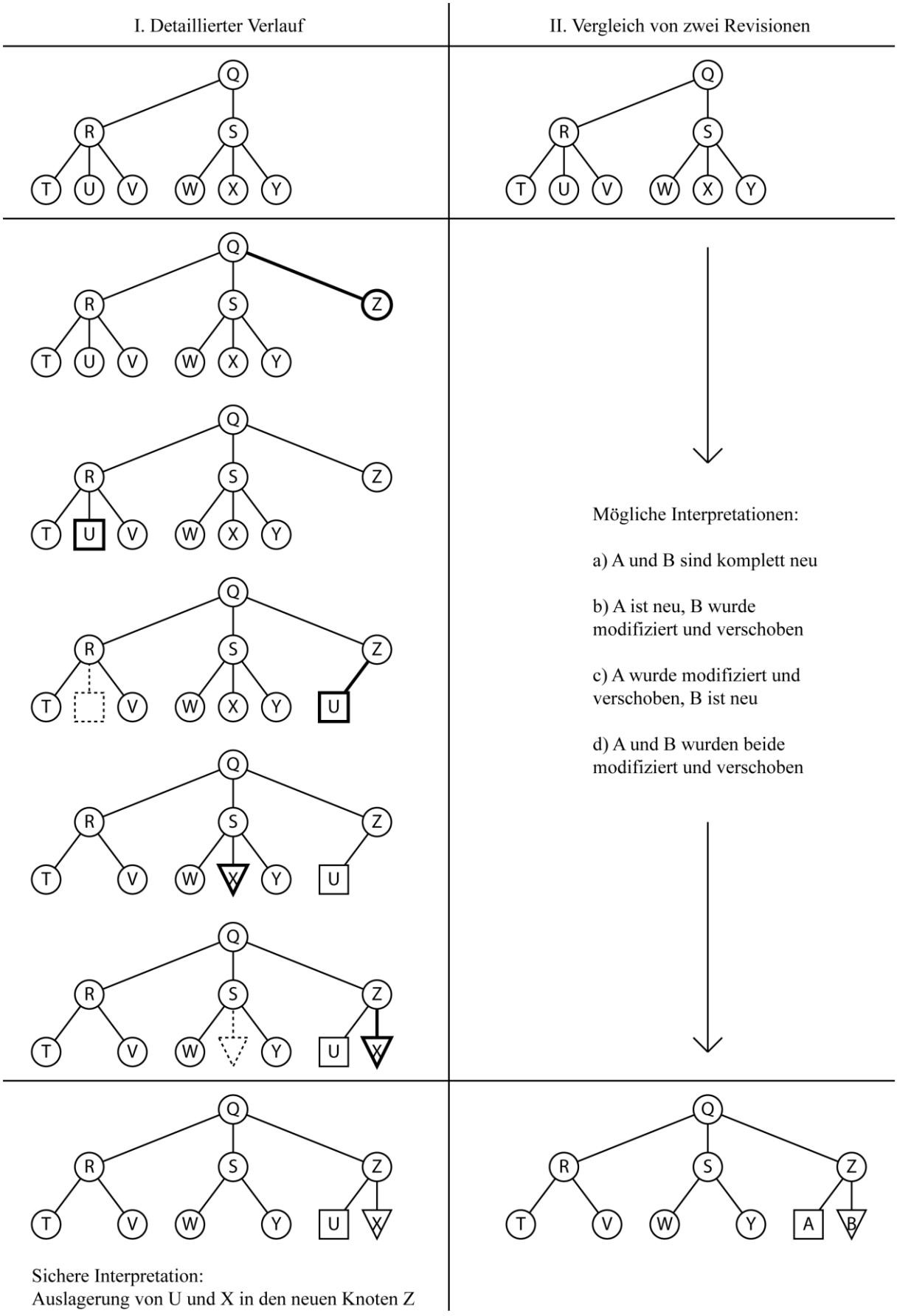


Abbildung 1 Sicherheitsgewinn durch eine detaillierte Historie

Um nun die zuvor beschriebene Überprüfung des Quellcodes zur Echtzeit zu realisieren, soll das angestrebte Softwareartefakt als Erweiterungsmodul für Visual Studio[1] realisiert werden. Bei der unterstützen Programmiersprache soll es sich um C#[2] handeln.
Um den Nutzerkomfort nicht zu beeinträchtigen, soll das Plug-in als Hintergrundprozess realisiert sein und keine Benutzereingaben erfordern. Es ist damit zu rechnen, dass diverse Optimierungsverfahren benötigt werden, damit eine verzögerungsfreier Ablauf ermöglicht wird. Verglichen mit den bisherigen Verfahren gibt es zwei zusätzliche Faktoren, die einen entscheidenden Einfluss auf die Performanz haben. Der erste Faktor, welcher sich negativ auf Performanz auswirkt, ist die erhöhte Informationsdichte. Der zweite Faktor, welcher sich positiv auswirkt, ist die Möglichkeit Veränderungen präziser lokalisieren zu können.
Wahlweise soll die Möglichkeit bestehen, ein Fenster einzublenden, welches die aktuell erkannten semantischen Veränderungen in sinnvollen und verständlichen Klassifikationen anzeigt. Sollten alle Forderungen erfüllt werden, so ist es erstrebenswert, dass klar strukturierte Change-Events geworfen werden, so dass auch andere Projekte davon profitieren können.

1.2 AUFBAU DER ARBEIT

Diese Arbeit ist in sieben Kapitel aufgeteilt. Auf dieses Einführungskapitel folgt Kapitel 2, in welchem ein Überblick über die Literatur und den aktuellen Forschungsstand gegeben wird. Hier werden grundlegende Begriffe geklärt und Ansätze ähnlicher Arbeiten vorgestellt, in denen ebenfalls die Extraktion semantischer Veränderungen im Quellcode untersucht werden.
Aufbauend auf den Erkenntnissen aus der Literaturrecherche wird in Kapitel 3 der eigene Lösungsweg vorgestellt. Hierbei werden zunächst die verschiedenen Lösungsansätze illustriert und beschrieben wie daraus das Endresultat entwickelt worden ist. Dieses Resultat ist ein regelbasiertes Lösungsverfahren aus mehreren Schritten, das im Laufe des dritten Kapitels erklärt wird.
Nachdem das Verfahren erklärt worden ist, wird Kapitel 4 auf dessen Eigenschaften eingegangen. Hier werden die Einschränkungen des Verfahrens erläutert und auf die Vollständigkeit und Korrektheit eingegangen. Abschließend wird angegeben, wie den aufgezeigten Schwächen entgegengewirkt wird.
Im Anschluss an die theoretische Beschreibung des Lösungsverfahrens wird in Kapitel 5 ausgeführt, wie diese realisiert und implementiert wird. Dabei wird die Wahl der technischen Tools begründet, die Software-Architektur vorgestellt und der Prototyp demonstriert.
Abschließend wird im sechsten Kapitel ein Ausblick darüber gegeben, welche Möglichkeiten sich anhand der Ergebnisse dieser Arbeit eröffnen und wie diese Möglichkeiten sinnvoll ausgeschöpft werden können. In Kapitel 7 werden die Ergebnisse dieser Arbeit reflektiert, diskutiert und den Ergebnissen ähnlicher Arbeiten vergleichend gegenübergestellt.

[1] Eine von der Microsoft Cooperation angebotene integrierte Entwicklungsumgebung.
[2] Ein von Microsoft, Hewlett Packard und Intel entworfene Programmiersprache. ISO/IEC 23270:2006

2 Forschungsstand

Wie in der Einleitung aufgeführt, wurde die Detektion von semantischen Veränderungen des Quellcodes bereits im Zuge unterschiedlicher Ausarbeitungen untersucht. In diesem Kapitel werden zunächst mögliche Anwendungsgebiete für die Detektion von Änderungsoperatoren aufgezeigt. Danach befasst sich dieses Kapitel mit den wichtigsten Konzepten, Datenstrukturen und Lösungsansätzen, auf die sich diese Ausarbeitungen stützt. Diese Information dient der Veranschaulichung des aktuellen Forschungsstands und der Positionierung dieser Arbeit. Weiterhin können anhand des hier recherchierten Wissens wertvolle Anhaltspunkte für den Lösungsansatz dieser Masterarbeit gewonnen werden.

2.1 ANWENDUNGSGEBIETE

Integrierte Entwicklungsumgebungen (IDEs) sind aus der modernen Softwareentwicklung nicht wegzudenken. Sie steigern die Produktivität des Entwicklers, indem sie eine Vielzahl an Tools in einem gemeinsamen Arbeitsbereich vereinen. Eine große Menge dieser Tools unterstützen den Benutzer auf semantischer Ebene, indem sie ihm Bedeutungsaspekte von Quellcode aufzeigen [**8**]. Sie verhelfen zu einem besseren Codeverständnis, welches für Softwareentwicklung essentiell ist. In der Literatur wird die Kritikalität des Code-Verstehens betont und es wird explizit die Zuhilfenahme von Tools bei den Feldern Refactoring [**9**], Wartung [**10**] und Wiederverwendbarkeit [**11**] [**12**] empfohlen.

Eine Teilmenge dieser semantischen Tools beschäftigt sich mit der Analyse der Veränderungen des Quellcodes. Hierbei handelt es sich um Verfahren, welche mehrere Coderevisionen betrachten, textuell vergleichen und bewerten, mit dem Ziel übergeordnete, abstraktere Codeveränderungen zu erkennen. Diese können beispielsweise Einfügen, Löschen, Modifizieren oder Verschieben sein [**1**].

Die automatisierte Identifizierung solcher semantischer Veränderungen hat diverse Vorteile. Sie kann zum Beispiel dafür genutzt werden, Veränderungen am Quellcode schneller zu begreifen [**1**]. Das kann zu einer Effizienzsteigerung bei der Wartung von Software führen, da hier der Verstehensprozess einen Großteil der zu investierenden Zeit benötigt [**10**] [**6**]. Der Verstehensprozess kann dabei auf unterschiedlichen Wegen unterstützt werden.

Eine Möglichkeit, das Verständnis von Quellcode zu unterstützen, besteht in der automatisierten Zusammenfassung und Interpretation von Veränderungen. Dies kann bei Softwareprojekten helfen, bei denen mehrere Entwickler an einem gemeinsamen Projekt arbeiten. Wird in einem solchen Projekt z.B. ein Programmierer während einer Urlaubsphase durch einen anderen vertreten, so kann eine Zusammenfassung aller Veränderungen, die während seiner Abstinenz getätigt worden sind, einen schnelleren Wiedereinstieg unterstützen.

Ein Beispiel für eine derartige zusammenfassende Interpretation ist die Umbenennung von mehreren Symbolen. Wird diese detektiert, so lässt das den Schluss zu, dass eventuell ein Refactoring stattgefunden hat. Die Arbeiten von Weißgerber et al. [**13**] und Krahn et al. [**14**] stellen hierzu Ansätze vor, wie solche Refactorings anhand von semantischen Codeveränderungen erkannt werden. In eine ähnliche Richtung forschen auch Hammad et al.

[**15**] und Delater et al. [**16**], die den Zusammenhang von semantischen Code-Veränderungen und dem zu Grunde liegenden Code Design untersuchen.
Neben dem Verständnis von Quellcode kann auch dessen Erstellung unterstützt werden. Wird eine Codeveränderung (wie z.B. Verschieben) erkannt, so lassen sich daraus Auswirkung auf die Quellcode-Semantik abschätzen. Zimmermann et al. [**17**] und Neamtiu et al. [**7**] stellen einige Verfahren vor, wie festgestellt werden kann, inwieweit sich eine erkannte Veränderung auf andere Quellcodestellen explizit auswirkt. Neben der Auswirkung auf andere Codestellen kann auch die Auswirkung auf zusammenhängende Requirements geprüft werden [**18**]. Kagdi [**19**] und Ying et al. [**20**] gehen noch einen Schritt weiter, indem sie Mechanismen vorstellen, die auf eine Veränderung möglicherweise sinnvolle Folgeveränderungen vorschlagen.
Allgemein lässt sich erwähnen, dass die Detektion von semantischen Veränderungen insbesondere bei der Zusammenarbeit von mehreren Programmierern Verwendung findet. Westfechel [**21**], Thione et al. [**22**] und Binkley et al. [**23**] führen z.B. an, wie das Wissen über semantische Veränderungen für ein leistungsfähigeres Mergen von Branches genutzt werden kann als es derzeit bei Apache Subversion (SVN) [**24**] möglich ist. Neben dem automatisierten Mergen eines Versionierungstools wie SVN, können auch Merge-Editoren wie diff [**25**] oder WinMerge [**26**] zur Codeverbesserung eingesetzt werden. Derartige Tools markieren alle Textzeilen, die sich zwischen zwei Textdateien unterscheiden. Für Quellcode ist dies jedoch nicht immer optimal, weil lediglich der rein textuelle Unterschied markiert wird. Horwitz [**3**] zeigt Möglichkeiten auf, wie diese Markierungen besser auf den Programmier-Kontext angepasst werden können. Nachdem ein Programmierer alle gewünschten Änderungen durchgeführt hat, und sie mit dem Repository-Server abgleichen will, ist es üblich, dass er eine knappe Erklärung dazu abgibt, worin die Änderungen bestehen. Da die Änderungen die Tätigkeit von einem kompletten Arbeitstag sein können, kann es dazu kommen, dass dem Programmierer einige Details entfallen. Auch hier kann die Erkennung von Änderungsoperationen hilfreich sein, da aus ihnen eine automatische Zusammenfassung generiert werden kann.
Nachdem eine Veränderung an einem Softwareprojekt getätigt wurde, wird in der Regel ein Regressionstest gestartet, um zu ermitteln, ob alle zuvor funktionierenden Testfälle immer noch dieselben Ergebnisse liefern. Da große Softwaresysteme sehr viele Funktionen aufweisen, kann die Zahl der Testfälle sehr hoch sein und sich deren Ausführung als sehr zeitintensiv erweisen. Um dem entgegenzuwirken, schlagen Binkley [**27**] und Vokolos et al. [**28**] in ihren Arbeiten Methoden vor, wie sich die Zahl der Testfälle, unter Zuhilfenahme der Informationen zu Codeveränderungen, signifikant reduzieren lässt.
Neben den praxisorientierten Anwendungsszenarien existiert auch die Möglichkeit, gefundene Codeveränderungen für theoretische Forschungsaspekte zu nutzen. Wenn die Codeveränderungen über einen Zeitraum abgespeichert werden, kann daraus eine Änderungshistorie entstehen, anhand derer diverse Rückschlüsse gezogen werden können. Eine Forschungsarbeit, in der sich mit der Auswertung derartiger Historien beschäftigt wird, stammt von Tu und Godfrey [**29**]. Sie erforschen die Gründe, warum zu bestimmten Zeitpunkten Änderungen getätigt werden, überlegen sich Wege, wie mit Codeveränderungen umgegangen werden kann, und was die weiterführenden Folgen einer Veränderung sind. Aus diesen Erkenntnissen versuchen sie Richtlinien abzuleiten, die zu einer erfolgreichen Software-Evolution beisteuern sollen.

Weitere Forschungsfragen sind die nach der Beschaffenheit und Wesensart typischen Code-Veränderungen [**1**], der Zusammenhang zwischen Codeveränderungen und Kopplung und Kohäsion [**7**], Rückschlüsse auf typisches Benutzerverhalten und Auswirkungen von Veränderungen auf die Programmstabilität und den gesamten Software-Evolutionsprozess [**30**].

2.2 GRUNDBEGRIFFE

2.2.1 Einordnung der Thematik

Einer der großen Forschungsbereiche der Informatik ist der *Compilerbau*. Dieser beschäftigt sich mit der theoretischen und praktischen Konzeption von Programmiersprachen und deren Compilern. Ein *Compiler* (oder Übersetzer) ist ein Computerprogramm, welches eine Programmiersprache in Befehle übersetzt, welche für den Computer "verständlich" sind.
Der Übersetzungsprozess besteht aus mehreren Schritten, welche für die Umwandlung in Maschinensprache als auch für die Interpretation von Quellcode notwendig sind. Informationen zu allen Schritten sind in Aho et al. [**31**] gegeben. Für diese Arbeit sind lediglich die ersten zwei Schritte von Interesse: die lexikalische Analyse und die syntaktische Analyse.
Die erste Phase des Compilers ist die *lexikalische Analyse*. In dieser Phase wird die Zeichensequenz des Quelltextes eingelesen und in Bedeutungseinheiten, den *Symbolen*, eingeteilt. Für jedes Symbol wird daraufhin ein *Token* erstellt. Ein Token wird definiert als Tupel aus Symbol und zugehöriger Symbolklasse. Gültige Tupel sind beispielsweise (1337, Integer) oder ("some entry", String).
Nach abgeschlossener lexikalischer Analyse werden die resultierenden Token der zweiten Compiler-Komponente, dem *syntaktischen Analysierer*, übergeben. Dieser prüft zunächst, ob der geschriebene Quellcode syntaktisch korrekt ist, also der Programmiersprachen-Grammatik entspricht. Um dies zu prüfen, werden die erhaltenen Token abgeleitet und in einen Syntaxbaum überführt. Die Blätter des Baums entsprechen den Terminalen der Grammatik, die Knoten entsprechen den Nicht-Terminalen der Grammatik. Der syntaktische Analysierer wird auch *Parser* genannt.
In den verbleibenden Phasen des Übersetzungsprozesses wird der Syntaxbaum weiterverarbeitet bis letztendlich der Maschinencode entsteht. Diese restlichen Phasen sind die semantische Analyse, die maschinenunabhängige Transformationen, die Adresszuordnung, die Erzeugung des Zielprogramms und die maschinenabhängige Codeverbesserung [**31**]. Im Rahmen dieser Arbeit interessieren jedoch nur die Ergebnisse der ersten zwei Schritte: die Token und die Syntaxbäume.
Neben dem Einsatz im Übersetzungsprozess kann der Syntaxbaum auch als Grundlage für die Interpretation von Quellcode und den dahinter liegenden Nutzerintentionen dienen. Im Rahmen dieser Arbeit interessiert insbesondere die Veränderung eines Syntaxbaums über die Zeit. Anhand der Baumveränderungen sollen Rückschlüsse auf die zu Grunde liegenden Nutzeraktionen gezogen werden.
Um eine Veränderung zwischen zwei Bäumen zu erkennen werden *Baumvergleiche* herangezogen. Algorithmen, die einen Baumvergleich durchführen, vergleichen die Knoten beider Bäume und finden die kürzeste Folge an Baumoperationen, um einen Baum a in einen

Baum b zu überführen. In Hein et. al. [**32**] werden eine Vielzahl an unterschiedlichen Baum-Vergleichsalgorithmen gegenübergestellt. Laut ihnen ist der Algorithmus nach [**33**] der derzeit schnellste Algorithmus mit einer Laufzeit von $O\left(n^{2+o(1)}\right)$.
Es gibt unterschiedliche Arbeiten, die Baumvergleiche auf eine Zeitreihe von unterschiedlichen Syntaxbäumen anwenden, um eine Reihe an Baumoperationen zu erhalten. Für derartige Verfahren gibt es jedoch (noch) keinen standardisierten Überbegriff. In der Literatur findet sich eine Fülle von Begriffen. Einige sind Difference Extractor [**1**], Source Code Difference Analysis [**6**], Mining Version Histories [**17**], Change Distilling [**18**] oder Evolutionary Code Extractor [**34**]. In dieser Arbeit wird diese Tätigkeit als *Extraktion von Änderungsoperationen* bezeichnet, oder kurz: Änderungsextraktion.

2.2.2 Beschreibungsebenen

Im vorangegangenen Abschnitt wurde erwähnt, dass aus der Veränderung des Quellcodes auf die *Nutzerintention* geschlossen werden soll. Um den Begriff der Nutzerintention zu präzisieren, kann er in die vier-Stufen-Skala nach Hassan et al. [**34**] eingeordnet werden.
Ihre Arbeit beschreibt ein Szenario, bei dem ein Programmierer eine Änderung an einer Revision getätigt hat. Auf die Frage hin, welche Änderungen von ihm getätigt worden seien, hat er folgende Antwortmöglichkeiten [**34**]:

1. *Ich habe 5 Codezeilen verändert.*
2. *Ich habe 3 Codezeilen in der Datei main.c verändert. Zusätzlich habe ich 2 Codezeilen aus der Datei layout.c verändert.*
3. *Ich habe 1 Codezeile in der main()-Methode hinzugefügt, 2 Codezeilen in der init() Methode hinzugefügt und 2 Zeilen aus der refreshLayout()-Funktion entfernt.*
4. *Ich habe die Main()-Methode dahingehend verändert, dass sie init() aufruft. Danach habe ich eine zusätzliche Abfrage in init() eingefügt um sicherzugehen, dass der Dateiname abgefragt wird, bevor refreshLayout() aufgerufen wird. Zum Schluss habe ich den Namenscheck aus refreshLayout() entfernt.*

Alle Aussagen beziehen sich auf diesselbe Veränderung. Dabei beschreibt die erste Aussage die Größenveränderung des Gesamtsystems und wird als *System-Level*-Veränderung bezeichnet. Der zweite, spezifischere Satz wird dem *Datei-Level* zugeordnet. Bei der dritten, nochmals spezifischeren, Variante werden die getätigten Veränderungen zu den zugehörigen Funktionen gemappt, und daher als *Funktions-Level*-Veränderung klassifiziert. Bei der letzten Aussage handelt es sich schließlich um eine *Funktions-Level*-Veränderung mit Aussagen über die *Aufruffolge* [**34**].
Die hier beschriebene Nutzerintention würde in der obigen Skala Punkt Vier entsprechen, da die Veränderungen C#-Schlüsselwörtern zugeordnet, und ihre Auswirkungen auf globale Zusammenhänge überprüft werden sollen.

2.2.3 Datenstrukturen & Visualisierungen

Bisher wurde der Syntaxbaum als Visualisierungsform des Quellcodes vorgestellt. Neben ihm existieren noch weitere Darstellungsformen, welche sich auf unterschiedliche Eigenschaften von Quellcode konzentrieren. Da in anderen Arbeiten einige von ihnen eingesetzt werden, um Quellcode Veränderungen zu interpretieren, werden sie an dieser Stelle kurz vorgestellt.

Eine Erweiterung des Syntaxbaums ist der *abstrakte Syntaxbaum*, oder kurz AST (Abstract Syntax Tree). Er entspricht einem regulären Syntaxbaum mit dem Unterschied, dass auf nicht sinngebende Elemente, wie Kommentare oder Zeilenumbrüche, verzichtet wird. Abstrakte Syntaxbäume werden eingesetzt, wenn lediglich die Funktionalität des Quellcodes im Vordergrund steht. Um die ASTs von ihren detailbehafteten Gegenstücken begrifflich zu unterscheiden, werden erstere als *abstrakte* und letztere als *konkrete* Syntaxbäume bezeichnet.
In einigen Anwendungsgebieten werden Syntaxbäume mit zusätzlicher domänenspezifischer Information versehen. Derartig individualisierte Bäume werden häufig als *dekorierte* Syntaxbäume bezeichnet. Ein Beispiel für einen dekorierten Syntaxbaum ist der Abstract Semantic Graph (ASG) aus [**1**]. Der ASG erhält zusätzliche Kanten, welche die Knoten, die Variablenreferenzen repräsentieren, mit denen, welche ihren Variablendeklarationen repräsentieren, verbinden. So können Abhängigkeiten zwischen den Elementen eines Syntaxbaums besser verfolgt werden. ASGs werden im Tool zur Änderungsextraktion von Raghavan et al. [**1**] genutzt.
Eine weitere Darstellungsform für Quellcode ist der *Kontrollflussgraph* [**35**]. Ein Kontrollflussgraph ist ein gerichteter Graph mit exakt einem Wurzelknoten. Hierbei muss jeder Knoten des Graphs von der Wurzel aus erreichbar sein. Jeder Knoten repräsentiert einen Computerbefehl. Die ausgehenden Kanten eines Knotens zeigen auf, welche Befehle auf einen Befehl folgen können. Auf diese Weise drückt ein Kontrollflussgraph alle möglichen Programmabläufe einer Quellcodemenge aus. Im Änderungsextraktionstool von Horwitz [**3**] wird ein erweiterter Kontrollflussgraph genutzt, um semantische Codeveränderungen von rein textuellen Codeveränderungen zu unterscheiden.
Eine weitere Visualisierungsform für Quellcode, welche von Tools zur Änderungsextraktion genutzt wird, ist das UML-Klassendiagramm [**36**]. Ein Klassendiagramm beschreibt die statische Struktur eines Softwaresystems, indem es seine Klassen, deren Attribute und Methoden, und die Beziehungen der Klassen zueinander darstellt. Sie erlauben eine Sicht auf die globale Architektur des Softwaresystems, blenden jedoch detaillierte Informationen, wie z.B. den Methodenrumpf, aus. Daher eignen sich Klassendiagramme, um globale architekturspezifische Code Veränderungen zu detektieren. Für detaillierte Veränderungen im Kleinen sind sie jedoch ungeeignet. Ein Änderungsextraktions-Tool, welches auf Basis von Klassendiagrammen arbeitet, ist beispielsweise von Hammad et al. [**15**] entwickelt worden.

2.3 EXTRAKTION DER VERÄNDERUNGEN

Um ein Programm zur Extrahierung von semantischen Code Veränderungen zu entwickeln, müssen diverse Herausforderungen gemeistert werden. Die wichtigsten sind im Folgeabschnitt 2.3.1 erwähnt. Im Folgekapitel 2.3.2 werden daraufhin mögliche Ansätze aus der Literatur gezeigt, um diesen Herausforderungen zu begegnen.

2.3.1 Herausforderungen

Eine vollständige Aufzählung aller Herausforderungen für dieses Projekt bezogen auf bisherige Arbeiten und Literatur ist nicht möglich, da eine derartige Software bisher noch nicht entwickelt worden ist. Daher können lediglich Abschätzungen bezüglich des Aufwands und der auftretenden Probleme gegeben werden. Erfreulicherweise gibt es mit Hassan et al. [**34**] bereits eine Veröffentlichung, welche ähnliche Ziele wie auch diese Arbeit verfolgt.

Ihnen ist es nicht gelungen, ein derartiges Programm zu verwirklichen, sie konnten jedoch trotzdem eine Abschätzung für den Aufwand und die Schwierigkeiten eines derartigen Projekts machen. So versuchen sie anhand von Erfahrungen aus der Entwicklung diverser ähnlicher Projekte die voraussichtlich auftretenden Probleme vorherzusagen. Hierbei sehen sie fünf Problemfelder: Robustheit, Skalierbarkeit, Präzision, fehlende Code-Stabilität und Entwicklungsaufwand.
In ihrem Bericht weisen Hassan et al. [**34**] zunächst auf die Bedeutung der *Robustheit* hin. Diese sei insbesondere dann gefährdet, wenn eine Vielzahl an Erweiterungen für einen Standardcompiler existiert. In solchen Fällen könne es dazu kommen, dass unbekannte Codefragmente auftreten, deren Weiterverarbeitung problematische Auswirkungen nach sich ziehen könnten. Als Lösungsmöglichkeiten werden hier Parsing-Techniken wie island grammars, robust parsing, precise parsing [**37**] oder auch teilweises Aussetzen des Compilers empfohlen.

Ein weiteres Problem sehen Hassan et al. [**34**] in der Bewältigung der *Skalierbarkeit*. Diese ist vor allem dann von signifikanter Bedeutung, wenn das Tool innerhalb von größeren Softwaresystemen mit mehreren Millionen Codezeilen (LOC) eingesetzt wird. Neben der Größe des Softwareprojekts spielt die Häufigkeit der Analysevorgänge des Tools eine gravierende Rolle. Je kleiner die Zeitintervalle zwischen Prüfungsvorgängen sind, desto mehr nimmt die Komplexität zu. Daher sind elaborierte Techniken zur Bewältigung der Komplexität notwendig. So sind beispielsweise inkrementelle Kompilierungstechniken möglich.
Als nächstes erwähnen Hassan et al. [**34**] die Wichtigkeit der *Präzision*. Nach ihnen besteht bei Tools zur Erkennung von semantischen Veränderungen Tendenzen zum Übersehen von Veränderungen (false negatives) oder zur Ermittlung überflüssiger Veränderungen (false positives). Für ein präzises Verhalten benötige es komplexer Grammatiken und aufwendiger Techniken zur Identifizierung und, falls nötig, Fehlerbehandlung von Informationen. Weiterhin wird auf [**38**] und [**39**] verwiesen. Dort werden ebenfalls auf Präzisionsschwierigkeiten von revisionsbasierten Extraktionstools hingewiesen. Zusätzlich wird von ihnen vermutet, dass es zu Namenskonflikten kommen kann, wenn Entitäten, über einen längeren Zeitraum betrachtet, zufällig identische Namen haben könnten.
Eine weitere Problematik ist nach [**34**] die fehlende *Code-Stabilität*, da der Quellcode nicht jederzeit compilierbar ist. Beispielsweise könnten Funktionsaufrufe stattfinden, bevor sie definiert worden sind. Für die Bewältigung solcher Sonderfälle bräuchte es nach [**34**] wieder besonderer Techniken, die erneut nicht genauer spezifiziert werden.
Als letzter Punkt wird der *Entwicklungsaufwand* genannt. Es sei zwar möglich, das angestrebte Tool auf bereits vorhandenen Lösungen, wie z.B. revisionsbasierten Extraktionstools, aufzubauen, jedoch nur bedingt empfehlenswert. Es wird davon ausgegangen, dass das Stützen auf vorhandene Lösungen eine derartige Einschränkung bedeutet, dass die Flexibilität und die Performanz negativ beeinflusst werden würden.

2.3.2 Lösungsansätze

In der Literatur gibt es bereits diverse Verfahren, welche die Änderungshistorie von Quellcode untersuchen. Nicht alle verwenden Syntaxbaum-Vergleiche und keines von ihnen deckt alle gesteckten Ziele ab, die in der vorliegenden Arbeit angestrebt werden. Trotzdem lohnt ein Blick auf diese Veröffentlichungen mit ähnlichen Problemstellungen, zum Einen,

um einen Überblick über den Status quo zu erhalten und zum Anderen, um wertvolle Inspirationen für den eigenen Lösungsansatz zu erhalten.

(a) Metrik-basierte Ansätze

Softwaresysteme besitzen unterschiedlich messbare Eigenschaften, z.B. Lines of code. Um diesen Eigenschaften "greifbare" Zahlenwerte zuzuordnen, werden Metriken eingesetzt. Eine (Software-)Metrik ist eine Funktion, die Softwareeigenschaften auf einen Zahlenwert abbilden [**40**]. Ein Beispiel für eine Metrik ist (durchschnittliche Klassengröße) = (LOC des Gesamtprojekts) / (Anzahl der Klassen).

In den recherchierten Arbeiten zur Änderungsdetektion haben Metriken zwei primäre Anwendungszwecke. Sie dienen als Ähnlichkeitsmaß für potentiell zusammenhängende Vorgänger- und Nachfolgeknoten und geben eine Aussage über die Auswirkungen von Veränderungen.

Ein Beispiel für die Nutzung von Metriken, in der Untersuchung von Software-Evolution, ist in Godfrey und Tu [**29**] gegeben. Dort wird die *Bertillonage Analyse* genutzt, um Abhängigkeiten zwischen Syntaxknoten zu finden. Besagte Analyse besteht aus fünf Metriken (S-Komplexität, D-Komplexität, Zyklomatische Komplexität, Albrecht und Kafura [**41**]), die auf veränderte Codefragmente angewandt werden. So entsteht für jedes Codefragment ein Tupel in einem 5-dimensionalen Merkmalsraum. Anhand dieser Tupel werden die alten Codefragmente mit den neuen verglichen. Die euklidische Distanz dient dabei als Ähnlichkeitsmaß.

(b) Struktur-basierte Ansätze

In Abschnitt (a) wurden Metriken genutzt, um Vorgänger-Nachfolger-Paare zweier Syntaxbaumversionen zu ermitteln. Andere Ansätze versuchen die korrekte Zuordnung über strukturelle Eigenschaften von Syntaxknoten zu erreichen.

Der Unterschied zwischen metrik-basierten Vergleichen gegenüber struktur-basierten Vergleichen ist, dass bei Ersteren lediglich ein Zahlenwert ermittelt wird, bei Letzteren jedoch Referenzpunkte betrachtet werden. Bei Metriken werden nur die Eigenschaften von Syntaxknoten ermittelt, bei struktur-basiertem Vorgehen wird zusätzlich die Umgebung eines Syntaxknotens beachtet.

Ein Beispiel für ein struktur-basiertes Vorgehen wird in Godfrey und Tu [**29**] gegeben. Sie verwenden eine *Abhängigkeitsanalyse*, welche sinngemäß wie folgt ausgedrückt werden kann[1]:

1. Bestimme alle Codefragmente **G** aus der neuen Revision, welche nicht deckungsgleich mit den Codefragmenten **F** aus der alten Revision sind.
2. Bestimme zu jedem g ∈ **G** alle Vorgänger $\mathbf{V}_g$ und Nachfolger $\mathbf{N}_g$,
   mit $\mathbf{V}_x = \{v \in \mathbf{V} \mid v \text{ ruft } x \text{ auf}\}$, $\mathbf{N}_x = \{n \in \mathbf{N} \mid x \text{ ruft } n \text{ auf}\}$.
3. Das Tupel $(\mathbf{N}_g, \mathbf{V}_g)$ beschreibt die Abhängigkeitsstruktur eines Knoten g. Die Ähnlichkeit eines Knoten g mit einem Knoten f wird durch seine Abhängigkeitsstruktur bestimmt: $Diff: (N_f, V_f) \times (N_g, V_g) \rightarrow [0,1]$

[1] Die Notation ist eine starke Vereinfachung des originalen Algorithmus.

Wenn also zwei Knoten f und g die gleiche Abhängigkeitsstruktur aufweisen, dann steigt die Wahrscheinlichkeit, dass g eine Modifikation von f und nicht komplett neu entstanden ist.
Neben Godfrey und Tu [**29**] existieren noch weitere Forschungsarbeiten, die Knotenpaare anhand ihrer strukturellen Umgebung ermitteln. In Krinke [**4**] werden hierfür Abhängigkeitsgraphen genutzt, in Thione et al. [**22**] wird Datenflussanalyse und in Horwitz [**3**] werden Kontrollflussgraphen verwendet. Gall et al. [**42**] [**43**] nutzen ein Konstrukt aus logischen Zusammenhängen, welches sie CAESAR nennen. Klusener et al. [**36**] sowie Xing und Stroulia [**44**] verwenden UML-Klassendiagramme.
Neben der Möglichkeit zur Verfolgung von Syntaxtoken, ermöglicht der Einsatz der unterschiedlichen Modellen es auch, Aussagen über Bedeutungsaspekte zu tätigen. Werden z.B. Code-Änderungen auf ein UML-Klassendiagramm gemappt, so lässt dies Aussagen über die sich verändernde Software-Architektur zu.

(c) Syntaxbaum-basierte Ansätze
Ein naheliegender Ansatz zur Extraktion von Änderungssemantiken führt über den Syntaxbaum-Vergleich. Wie zuvor beschrieben, werden hier Syntaxbäume mit ihrem jeweiligen zeitlichen Nachfolger verglichen, um die kleinstmögliche Menge an Baumoperationen zu extrahieren, die den Vorgänger-Baum in den Nachfolger-Baum überführt. Die Menge dieser Operationen wird auch *Baum-Editierdistanz* genannt [**45**].
Eine der Arbeiten, in der in diese Richtung geforscht wird, stammt von Yang [**46**]. In seiner Veröffentlichung stellt er ein Tool vor, welches eine Weiterentwicklung des Linux Tools *diff* [**47**] ist. Mit diff werden zwei Textdateien miteinander verglichen und die Zeilen, die sich unterscheiden, markiert. Yangs Erweiterung besteht darin, dass zwei Quellcode-Dateien miteinander verglichen werden können, und die Unterschiede wortweise und nicht mehr zeilenweise aufgezeigt werden. Statt Codezeilen, geht Yang davon aus, dass Syntaxbäume miteinander verglichen werden müssen. Sein Vergleichsalgorithmus geht jedoch nur auf die Blätter der Bäume ein und lässt die Baumknoten aus. Dies begründet er damit, dass die Knoten eine Verallgemeinerung der Blätter seien. Für den Tokenvergleich nutzt er dann den Hirschberg-Algorithmus [**48**], welcher aus dem Feld der Sequenzanalyse stammt, und die längste gemeinsame Teilfolge zweier Folgen bestimmt. Zusammengefasst lässt sich sagen, dass in dieser Arbeit nur die Unterschiede zwischen den Token aufgezeigt werden, nicht jedoch, wie diese Unterschiede entstanden sind.
Eine Arbeit, die Syntaxbäume nutzt und mehr auf die Veränderungsoperationen eingeht, stammt von Raghavan et al. [**1**]. Im ersten Schritt ihres Lösungsansatzes wird das Tool CPPX [**49**] genutzt, um dekorierte Syntaxbäume namens ASG, wie in Abschnitt 2.2.3 vorgestellt, aus den Quellcode-Revisionen zu generieren. Aufbauend auf den zusätzlichen Informationen, die durch die Dekoration des Syntaxbaums entstehen, werden Kostenfunktionen definiert. Eine Kostenfunktion gibt an, wie unterschiedlich zwei Teilbäume zueinander sind. Je niedriger die Kosten sind, um von einem Knoten zum anderen zu gelangen, desto wahrscheinlicher ist es, dass es sich um ein Vorgänger-Nachfolger-Paar handelt. Diese Funktion fließt in den ASG-Vergleich ein. Er wird durchgeführt, indem iterativ immer mehr Knoten miteinander gematcht werden und vor jedem Matching-Schritt eine Kostenfunktion auf die beiden Restbäume angewandt wird.
Eine weitere Arbeit, die Änderungsextraktion durch Syntaxbaum-Vergleiche erzielt, stammt von Fluri et al. [**18**]. Ihr Baumvergleich wird in mehreren Schritten abgehandelt. Im

Initialschritt werden Ähnlichkeiten zwischen einzelnen Wörtern des alten und neuen Baums anhand einer *n-Gram basierten Text-Ähnlichkeitsfunktion* [**50**] berechnet. Hierbei handelt es sich um einen Textvergleich, in dem stets eine Gruppe aus n benachbarten Wörtern der alten Coderevision mit einer Gruppe aus n benachbarten Wörtern der Nachfolgerevision verglichen wird. Unter Verwendung der Wörtergruppen wird eine Robustheit gegenüber kleinen Veränderungen, wie das Umbenennen eines Bezeichners, geschaffen.
Im zweiten Schritt werden die ermittelten Ähnlichkeitswerte für die Wörter in die jeweiligen Token übertragen. Es folgt eine weitere Ähnlichkeitsfunktion für Syntaxknoten. Der neuen Funktion folgend, sind zwei Syntaxknoten genau dann einander ähnlich, wenn auch die Kindknoten einander ähneln. Auf diese Weise werden die initialen Wahrscheinlichkeitswerte in die Knoten hochtraversiert.
Abschließend werden die Knoten des alten Syntaxbaums den Knoten des neuen Syntaxbaums anhand der Ähnlichkeitswerte zugeordnet. Bei der Zuordnung werden dynamische anpassende untere Schranken für die Wahrscheinlichkeitswerte genutzt werden, damit falsche Zuordnungen vermieden werden.
Das letzte hier erwähnte Beispiel zur Syntaxbaum-basierten Änderungsdetektion ist der Algorithmus von Neamtiu et al. [**7**]. In diesem Algorithmus werden sowohl der alte als auch der neue Syntaxbaum parallel durchlaufen. Wenn bei der Baumtraversierung eine Stelle gefunden wird, in der ein Bezeichnername innerhalb einer Definition verändert worden ist (z.B. `int counter;` zu `int count;`), dann wird diese Namensveränderung in Form einer Bijektion abgespeichert. Bei jeder späteren Stelle, bei der dieser Name erneut auftritt, kann die Bijektion angewandt werden, um unterschiedliche Bezeichnernamen auf funktionale Gleichheit zu prüfen. Die Verwendung der Bijektionen führt dazu, dass *rename*-Operationen geworfen werden, die semantisch korrekter sind als eine *insert*-Operation, die auf eine *delete*-Operation folgt.

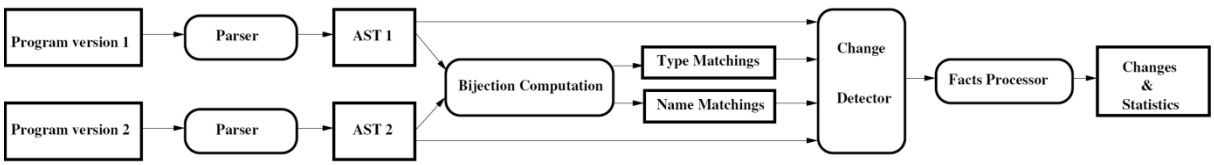


Abbildung 2 Der parallele AST-Vergleich nach Neamtiu et al. [7]

Zur besseren Veranschaulichung der Funktionsweise des zuvor erwähnten Algorithmus ist in Abbildung 2 ein Schaubild aus der ursprünglichen Arbeit aufgeführt. Es ist zu erkennen, dass zwei Typen von Bijektionen berücksichtigt werden. Dies sind zum Einen veränderte Namen und zum Anderen veränderte Variablentypen. Immer dann, wenn das parallele Durchtraversieren der Syntaxbäume stoppt, da die Folgeknoten unterschiedlich sind, wird eine passende Folge von *insert*- und *delete*-Baumoperationen für die Folgeknoten ausgegeben.

(d) XML-basierter Ansatz
Maletic und Collard [**6**] argumentieren, dass Änderungsextraktion über Syntaxbäume diverse Nachteile mit sich bringt. Sie behaupten, dass es aufgrund der Baumstruktur schwierig sei, unvollständigen Quellcode und nicht-funktionale Änderungen (wie die Eingabe von Leerzeichen) zu erkennen.

Aus diesem Grund führen sie eine XML-basierte Sprache namens srcML [**51**] ein. Diese Sprache ist so konzipiert, dass sie sowohl alle textuellen Informationen des Quellcodes als auch alle syntaktischen Informationen des Syntaxbaums beinhaltet. Weiterhin werden nicht kompilierbarer Code und Codefragmente beschrieben.
In ihrer eigenen Implementierung eines Änderungsextraktors, halten Maletic und Collard zu jedem Zeitpunkt eine srcML-Repräsentation des Quellcodes. Mit jeder Änderung des Quellcodes wird auch die srcML-Repräsentation verändert. Um nun die semantischen Veränderungen zu erlangen, werden die XML-Techniken XPath und XQuery eingesetzt, um aus dem srcML-Dokument eines von mehreren Modellen (UML Klassendiagramm, Abhängigkeitsgraph, Programmflussgraph oder ad-hoc Model) zu generieren. Anhand der Modelle können wie in (b) Aussagen über die sich veränderte Semantik getätigt werden.

(e) Weitere aussichtsreiche Ansätze
Neben den vier zuvor vorgestellten Klassen an Lösungsansätzen sind noch weitere Ansätze vorstellbar, welche jedoch in keiner vorliegenden Forschungsarbeit behandelt worden sind. Eine dieser zusätzlich möglichen Ansätze ist die Benutzung von Pattern Matching [**52**] um die Struktur zwischen Syntaxbäumen zu vergleichen. Weiterhin sind Ansätze denkbar, in denen der Programmierer eine graphische Schnittstelle benutzen muss, um seinen Quellcode zu verändern. Bei diesem Ansatz könnten die Änderungsoperationen durch die selektierten Bedienelemente der grafischen Oberfläche vorgegeben werden. Abschließend wäre noch ein regelbasierter Ansatz denkbar, bei dem eine Kette von Regeln feuern, wenn Editierungen am Quelltext getätigt werden.

3 Das SITCOM-Verfahren

Das in dieser Arbeit entwickelte Verfahren für die Erkennung von semantischen Codeveränderungen trägt den Namen SITCOM, ein Akronym für "*S*ymbol-*I*nitiated *T*ree *Com*parison". Die Namensgebung spielt auf die Tatsache an, dass das im Folgenden vorgestellte Verfahren auf der Detektion von kleinstmöglichen semantischen Einheiten, den Symbolen, basiert. Ausgehend von ihrer Detektion werden weitere Schlüsse über den Syntaxbaum gezogen, bis der Algorithmus an einer Zielvorgabe terminiert.
In diesem Kapitel werden die einzelnen Teilschritte von SITCOM mitsamt ihrer zu Grunde liegenden Ideen und Entwicklungsschritte vorgestellt.

3.1 GRUNDLEGENDE VORGEHENSWEISE

Um die semantischen Auswirkungen eines Tastendrucks auf den zugehörigen Quellcode nachvollziehen zu können, bietet sich ein Baumvergleich des *a priori*-Baums (vor der Eingabe) mit dem *a posteriori*-Baum (nach der Eingabe) an. Die Operationen, die benötigt werden, um von einem Baum zum anderen zu gelangen, werden im Allgemeinen als *Änderungsoperatoren* bezeichnet. Diese Änderungsoperatoren werden als die kleinstmögliche semantische Veränderung angesehen. Ziel des SITCOM-Verfahrens ist es, diese Änderungsoperatoren zu detektieren. Es folgen einige Definitionen zum formalen Fixieren der genannten Grundbegriffe.

Definition 1. Jede Veränderung des Quelltextes durch den Texteditor des Benutzers wird als *(Text-)Editierung* bezeichnet. Sie wird durch das Vier-Tupel (Editierungstyp, Inhalt, Startposition, Endposition) beschrieben. Der Editierungstyp ist entweder "Löschen" oder "Einfügen" und sagt aus, ob durch die Textveränderung ein Textabschnitt gelöscht oder eingefügt wird. Das Feld "Inhalt" gibt die eingefügte bzw. gelöschte Zeichenfolge an. Die Felder "Startposition" und "Endposition" geben die Zeichenposition des eingebetteten Inhalts innerhalb des Quelltextes an. *Lösch-Editierungen*, sind Editierungen, bei denen der Editiertyp "löschen" ist. Das Gegenstück wird als *Einfüge-Editierung* bezeichnet.

Definition 2. Jede Textänderung des Benutzers im Code-Editor führt dazu, dass sich der zu Grunde liegende Quellcode verändert. Wird der Quellcode zu jedem Zeitpunkt als Syntaxbaum betrachtet, so überführt die Texteingabe des Benutzers den zuvor aktuellen Syntaxbaum in einen neuen. Zukünftig wird der Syntaxbaum vor der Nutzereingabe als *a priori-Baum* bezeichnet. Das Pendant, welches nach der Nutzereingabe entsteht, wird als *a posteriori-Baum* definiert.

Definition 3. Ein *Änderungsoperator* ist eine atomare, strukturverändernden Aktion auf einem Syntaxbaum. Anhand von ihnen sollen Aussagen über die Semantik der durch den Nutzer getätigten Veränderung gemacht werden. Beispiele für Änderungsoperatoren sind z.B. Add Leaf, Split Leaf oder Remove Node. In den späteren Kapiteln 3.3.3 und 3.3.4 werden alle in SITCOM genutzten Änderungsoperatoren detailliert definiert.

Definition 4. Eine *Änderungsoperation* ist die explizite Anwendung eines Änderungsoperators auf einen gegebenen Baum.

3.1.1 Anforderungsprofil

In der Literatur gibt es ein Vielzahl an Arbeiten, in denen Syntaxbäume mit obiger Absicht verglichen werden. Einige von ihnen werden beispielsweise in [**53**] aufgezählt. Ein möglicher Ansatz zur Erreichung des gesteckten Ziels wäre es, solche bereits existierenden Verfahren nach jedem Editierungsschritt zu durchlaufen. Dieser Ansatz wurde zum Beispiel in dem semantischen Code-Evolutions-Tool von Robbes [**54**] verfolgt.

Derartige Vorgehen weisen jedoch diverse Schwächen auf. Zunächst wird in gebräuchlichen Baum-Vergleichs-Algorithmen davon ausgegangen, dass sich Veränderungen im gesamten Syntaxbaum befinden können. Dies ist ineffizient, da die durch eine einzelne Editierung vollzogene Änderung in der Regel auf einen deutlich kleineren Teilbaum reduziert werden kann. Auch wenn der beschriebene Teilbaum lokalisiert werden kann, so fehlen immer noch die teils wichtigen Informationen der Texteditierung.

In Abbildung 3 werden drei Szenarien gezeigt, bei denen das Wissen über die Editierung eine Mehrdeutigkeit bei der Detektion von Änderungsoperation entgegenwirken kann. Szenario eins beschreibt eine Situation, bei der eine Zeile gelöscht wird. Ohne das Wissen, an welcher Textstelle die Löschung stattfindet, lässt sich nicht ausmachen, welcher von den drei Knoten des zugehörigen Syntaxbaums gelöscht werden muss. Das zweite Szenario zeigt eine Modifikation der dritten Codezeile. Auch hier könnte ohne ein Wissen über die modifizierte Textstelle nicht lokalisiert werden, welcher dazugehörige Knoten modifiziert wird. Das letzte Beispiel zeigt ein Szenario, bei dem ohne das Wissen über die editierte Textstelle zwei Interpretationen möglich sind. Eine, bei der die obere Zeile modifiziert und die untere gelöscht wird und eine weitere, bei der lediglich die obere Zeile gelöscht wird.

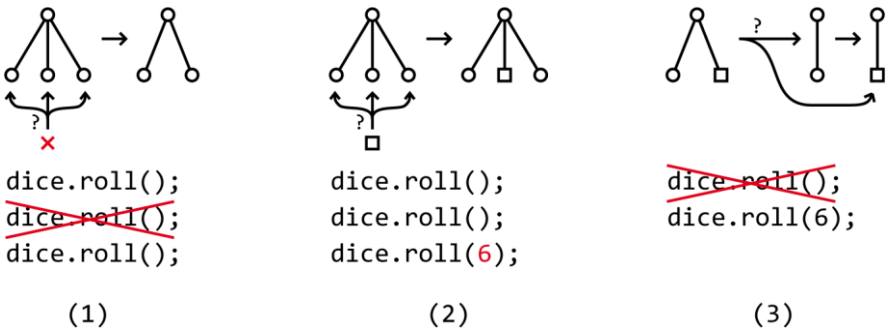


Abbildung 3 Informationsverlust durch Mangel der Editierungsinformation

Ein weiterer Schwachpunkt bei den bereits existierenden Vergleichstechniken besteht darin, dass abstrakte Syntaxbäume verwendet werden, bei denen auf Details wie etwa Kommentare verzichtet wird. Damit werden potentiell wichtige Informationen unterschlagen. So ist es beispielsweise semantisch treffender, zu erkennen, dass ein Teilstück der Syntax auskommentiert wird, anstatt das Auskommentieren mit dem Löschen eines Syntaxknotens zu begründen.

Hinzu kommt, das bei der Verwendung herkömmlicher Vergleichsverfahren implizite, zeitliche Ordnungsrelationen ausgelassen werden. Ein Beispiel hierfür wird in Abbildung 4 gegeben. Dort findet eine Lösch-Editierung statt, die auf zwei unterschiedliche Weisen

interpretiert werden kann. Die mit Baumvergleichs-Interpretation betitelte Interpretation, ist ungenauer als die mit SITCOM-Interpretation betitelte Variante. Dies führt insbesondere dann zu einem Problem, wenn der `static`-Token automatisiert verfolgt werden soll.

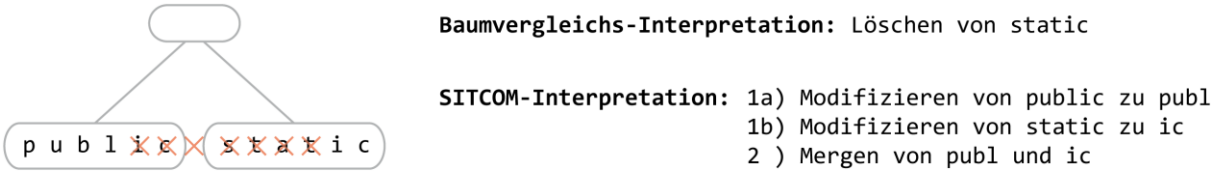


Abbildung 4 Auslassen zeitlicher Ordnungsrelationen bei der Verwendung herkömmlicher Baumvergleiche

Ein letzter Kritikpunkt an den bisher existierenden Baumvergleichen ist der mangelnde Umgang mit unfertigen bzw. fehlerhaften Bäumen. So wirft der *Evolutionary Extractor* aus [**54**] nur dann ein Event, wenn aus der umschließenden Methode ein vollständiger, fehlerfreier Syntaxbaum generiert werden kann.
Aus den zuvor genannten Schwächen lässt sich ein Anforderungsprofil für den SITCOM-Algorithmus ableiten. So muss beim Lösungsansatz darauf geachtet werden, dass der kleinstmögliche Teilbaum lokalisiert, Trivialinformationen berücksichtigt, mehrstufige Veränderungen erkannt, und unfertige, sowie fehlerbehaftete Parsebäume berücksichtigt werden.

3.1.2 Konzipierung des Lösungsansatzes

Im Folgenden wird die methodische Herangehensweise geschildert, bei der eine Folge von verworfenen Lösungsansätzen zum Endresultat, dem SITCOM Verfahren, geführt hat.

(a) Auswahl möglicher Ausgangspositionen
Ein initialer Ansatz ist es, die Summe alle Möglichkeiten zu ermitteln, mit welchen ein Editor den Quelltext beeinflussen kann. Dies kann als sinnvoller Ausgangspunkt für weitere Schritte gelten, da ausgehend von diesen Interpretationsbedingungen eine sinnvolle funktionale Unterteilung gegeben ist. Die Menge der ermittelten Interpretationsbedingungen ist Abbildung 5 zu entnehmen.

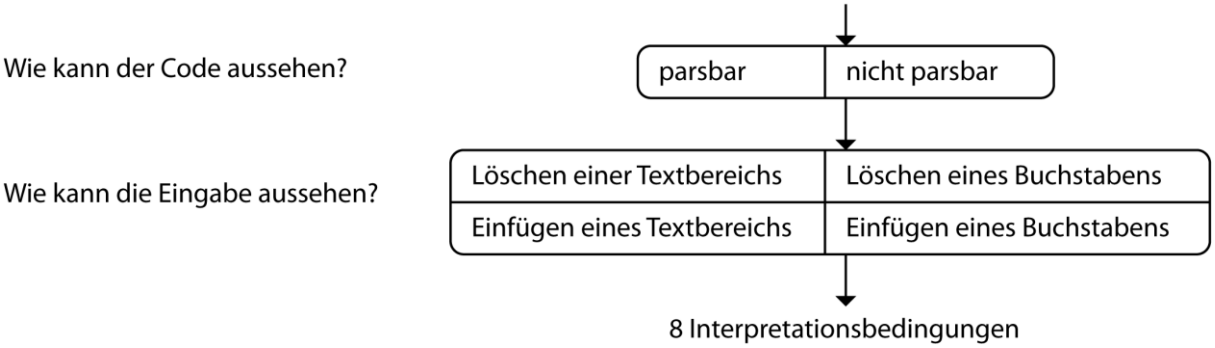


Abbildung 5 Acht mögliche Ausgangsbedingungen Quelltext zu beeinflussen

Mit diesen acht Ausgangsbedingungen werden alle Eingabemöglichkeiten des Texteditors abgedeckt. Eine Eingabeaktionen wie Überschreiben kann auf eine Folge von Löschen und Einfügen reduziert werden kann.

Im späteren Verlauf ist jedoch festgestellt worden, dass die Unterscheidung von parsbarem und nicht parsbarem Quellcode nicht notwendig ist, da der später verwendete Compiler für beide Zustände einen Syntaxbaum generieren kann. Weiterhin ließ sich das Löschen eines Buchstabens als Spezialfall der Flächenlöschung ansehen. Abschließend ist während der SITCOM Entwicklung erkannt worden, dass es vorteilhaft ist, Einfügeoperationen zu Löschungen zu invertieren. So wurden sämtliche Interpretationsbedingungen aus Abbildung 5 zu einem einzigen Fall zusammengefasst und die Fallunterscheidung war somit obsolet.

(b) Merkmalsvektor als Entscheidungshilfe
Eine auf dieser Vorüberlegung aufbauende Idee ist die eines Merkmalsvektors für jeden Knoten des jeweils aktuellen Syntaxbaums (siehe Abbildung 6, links). Dieser Vektor wird in Verbindung mit einem Entscheidungsbaum genutzt (siehe Abbildung 6, rechts). Hierbei wird beim Entscheidungsbaum an jedem Entscheidungspunkt auf die Informationen des Merkmalsvektors zurückgegriffen. Im Grunde ähnelt dieser Ansatz dem vorherigen, mit dem Unterschied, dass der Baum lediglich Informationen abfragt und dessen Aufbau nicht durch selbige vorgegeben wird.

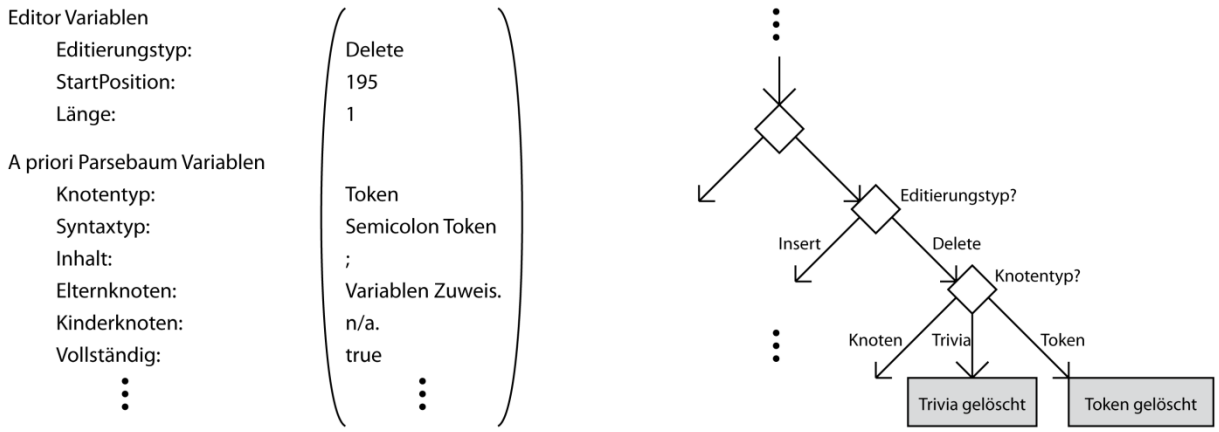


Abbildung 6 Merkmalsvektor mit Entscheidungsbaum

Die Idee des Merkmalsvektors ist in das spätere SITCOM-Verfahren eingeflossen, mit dem einzigen Unterschied, dass die Informationen nur On-Demand ermittelt werden, um Rechenaufwand einzusparen. Demgegenüber wurde das Flussdiagramm nicht in die weiteren Überlegungen eingebunden, da dessen Anwendung zu starr ist. Dies ist darauf zurückzuführen, dass sich die Menge der unterschiedlichen Zustände nur sehr schwer skalieren lässt und jeder einzelne Auswertungspfad nur eine einzige Operatorfolge beschreibt. Eine generischere Lösung scheint angebrachter.

(c) Behaviors statt der statischen Entscheidungsbaumstruktur
Eine gedankliche Weiterführung des Merkmalvektors ist angelehnt an die Subsumptionshierarchie nach Brooks [**55**] [**56**]. In seinen Arbeiten beschreibt er eine verhaltensbasierte Architektur zur Steuerung von reaktiven Robotern. Die Architektur wird aus einer Anzahl unterschiedlicher *Behaviors* zusammengesetzt. Ein Behavior ist definiert als:

behavior: (c,a)	$c \subseteq P$	Menge von Perzepten (→Bedingung)
	a	Aktion

Wenn alle Bedingungen aus c erfüllt werden, dann *feuert* (c,a) und löst die Aktion a aus [**56**]. Hierbei können auch mehrere Behaviors zeitgleich feuern. Angewandt auf den SITCOM-Kontext würde (c,a) wie in Abbildung 7 definiert sein. Dort entspricht jede Spalte einer Beschreibungsdimension, so dass 192 mögliche Interpretationsschritte entstehen können. Durch eine probate Umsortierungen der Dimensionen kann ein Geschwindigkeitszuwachs erzielt werden. Durch Zusammenfügen mehrerer Behaviors kann die Gesamtzahl der Regeln reduziert werden.

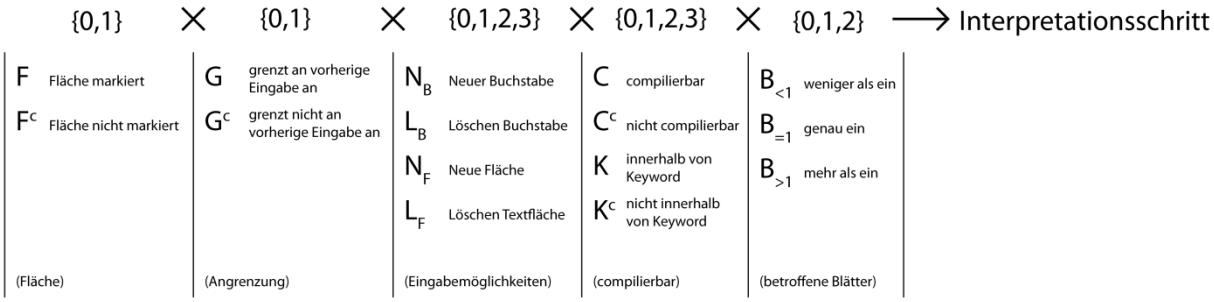


Abbildung 7 Ein Bedingungsraum für ein auf Parsebaum-Vergleich angepasstes Behavior

Das Konzept der Subsumtionshierarchie birgt mehrere Vorteile, die im späteren Verlauf in das SITCOM-Verfahren eingeflossen sind. Zum einen ist dies die Idee eines regelbasierten Entwurfs, da auf diese Weise eine gewisse Generik geschaffen wird. Eine Regel feuert immer dann, wenn ihre Bedingungen erfüllt sind. Hierbei ist es irrelevant, wie es zu der Erfüllung dieser Bedingungen gekommen ist. Damit ist eine Regel im Gegensatz zum Flussdiagramm kontextfrei und lässt sich allgemeiner einsetzen. So entsteht eine Wiederverwendbarkeit der Regeln und Robustheit gegenüber unbekannten Situationen.
Eine weitere konzeptionelle Stärke lässt sich aus Brooks Architekturidee ableiten: die Idee, dass unterschiedliche Regeln voneinander abhängen und eine Hierarchie bilden können. So verwendet das später vorgestellte SITCOM-Verfahren übergeordnete Regeln, die erst gefeuert werden, wenn passende untergeordnete Regeln feuern.

3.1.3 Tatsächlicher Lösungsansatz

Die Vorüberlegungen haben gezeigt, dass ein hierarchisches, regelbasiertes System günstige Eigenschaften bereitstellt. Im Folgenden wird darauf eingegangen, wie ein solches System realisiert werden kann.
Zunächst müssen die initialen Auslöser für das angestrebte Regelwerk festgelegt werden. Hierfür sind zwei unterschiedliche Mengen denkbar: Die Menge der hierarchisch höchsten veränderten Vaterknoten (siehe Abbildung 8, links) und die Menge der veränderten Token (siehe Abbildung 8, rechts). Aufbauend auf beiden Varianten können jeweils erste Regeln feuern, welche wiederum Folgeregeln anstoßen können, bis die Gesamtheit aller veränderter Knoten interpretiert ist.

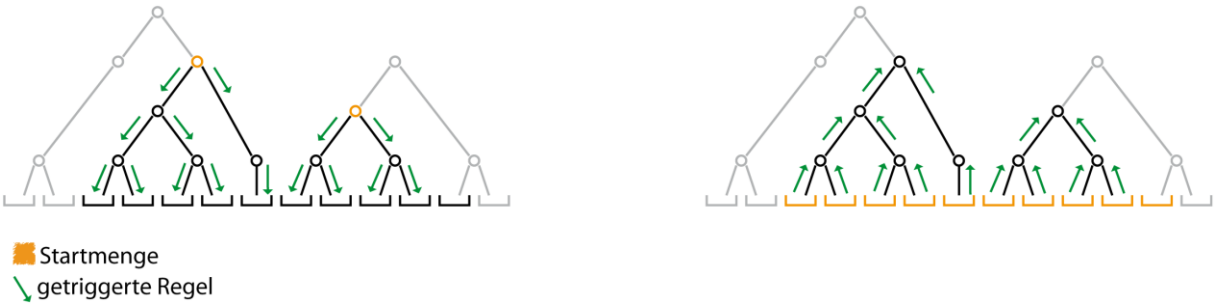


Abbildung 8 Zwei Möglichkeiten für die initiale Auslösermenge

Aus den beiden möglichen Startmengen wurde die tokenbasierte Variante gewählt. Diese Entscheidung resultiert aus zwei Überlegungen. Die erste Überlegung besteht darin, dass ein signifikant geringerer Aufwand entsteht, wenn Token statt Vaterknoten detektiert werden. Sollten die Vaterknoten ermittelt werden, so müsste, beginnend an der Wurzel, der gesamte Syntaxbaum nach veränderten Knoten durchsucht werden. Wird jedoch bei den Blättern des Baums gestartet, so ließe sich die Menge der veränderten Blätter anhand der Editierfläche herleiten. Der Algorithmus hierzu wird in Abschnitt 3.2 vorgestellt.
Der zweite Entscheidungsgrund für die tokenbasierte Vorgehensweise geht einher mit dem *divide and conquer*-Paradigma. Ein Folgerung aus diesem Paradigma ist es, dass Probleme häufig leichter zu lösen sind, wenn sie in ihre Unterprobleme aufgeteilt werden. In diesem Zuge liegt die Vermutung nahe, dass ein Ansatz beginnend bei den Blättern leichter realisierbar ist. Ein Vaterknoten lässt sich besser analysieren, wenn seine Kinder zuvor untersucht worden sind.
Die Regeln, die in SITCOM eingesetzt werden, sind in Blattoperationen und Knotenoperationen aufgeteilt. Blattoperationen erkennen die Veränderungen an den Blättern des Parsebaums. Ihre Auslösebedingungen enthalten mindestens eine Bedingung bzgl. der editierten Fläche am Textcursor. Knotenoperationen erkennen alle Veränderungen an Syntaxknoten, die keine Blätter sind. In ihrer Auslösemenge ist mindestens ein Ergebnis einer vorangehenden Operation. Blattoperationen werden in Abschnitt 3.4 genauer erläutert, Knotenoperationen in Abschnitt 3.5 . Eine Regel wird wie folgt definiert:

Definition 5. Eine (*Detektions-)Regel* wird durch ein 3-Tupel (Auslöser, Operation, Effekt) beschrieben. Das Feld Auslöser beschreibt eine Menge an Bedingungen, bei der jede einzelne Bedingung zutreffen muss, damit die Regel angewandt wird. Trifft dies zu, so wird die Operation erkannt und ausgegeben. Nach erfolgreicher Anwendung tritt der Effekt ein. Dabei handelt es sich um eine Menge von Zuständen, die nach der Regelanwendung hinzukommen und die Auslöser möglicher Folgeregeln beeinflussen können.

Neben der reinen Erkennung von Knotenoperationen wurde im Abschnitt 3.1.1 zusätzlich die Berücksichtigung von zeitlichen Aspekten bei der Modifikationserkennung von Syntaxknoten gefordert. Um dies zu berücksichtigen müssen die vorhandenen Regeln zusätzlich erweitert werden, wie dies realisiert wird, wird in Kapitel 3.6 geschildert.

3.2 ERMITTELN DER STARTMENGE

Wie zuvor geschildert, wird eine Menge an Tokens benötigt, die als Auslöser für die Blattoperationen dient. Hierbei interessieren allein diejenigen Token, die sich im *a priori*- und *a posteriori*-Baum unterscheiden. Es wird also eine "Vorher"-Symbolfolge X und eine "Nachher"-Symbolfolge Y benötigt werden.

Definition 6. Sei X_{Gesamt} die Folge aller Token eines *a priori*-Baums und Y_{Gesamt} die Folge aller Token des zugehörigen *a posteriori*-Baums. Wird X_{Gesamt} mit Y_{Gesamt} verglichen, so resultiert genau eine *Unterfolge* $X \subset X_{Gesamt}$ für die $X \cap Y_{Gesamt} = \emptyset$ gilt. Wird hingegen Y_{Gesamt} mit X_{Gesamt} verglichen, so resultiert genau eine *Unterfolge* $Y \subset Y_{Gesamt}$ für die $Y \cap X_{Gesamt} = \emptyset$ gilt. Beide Unterfolgen werden zukünftig nur noch mit *X* respektive *Y* bezeichnet.

3.2.1 Lösungsheterogenität

Auf den ersten Blick kann es als trivial erscheinen, die Menge der sich unterscheidenden Blätter zu detektieren. Die Vermutung liegt nahe, dass lediglich die Token, die die editierte Textstelle berühren für X und Y in Frage kommen. Für einfache Fälle, wie den Fall (a) aus Abbildung 9, trifft dies auch zu. Lediglich das eingefügte Semikolon-Zeichen wird als veränderter Token erkannt.

In anderen Fällen, in der Abbildung mit (b) - (d) beschriftet, trifft dies jedoch nicht zu, da stets ein Token "c" in den Symbolfolgen berücksichtigt werden muss, obwohl er keine Editierstelle unmittelbar berührt. Es bedarf also eines Verfahrens, das alle veränderten Token berücksichtigt, auch wenn sie die Editierfläche nicht unmittelbar berühren. Wie dies geschehen kann wird im Folgeabschnitt gezeigt.

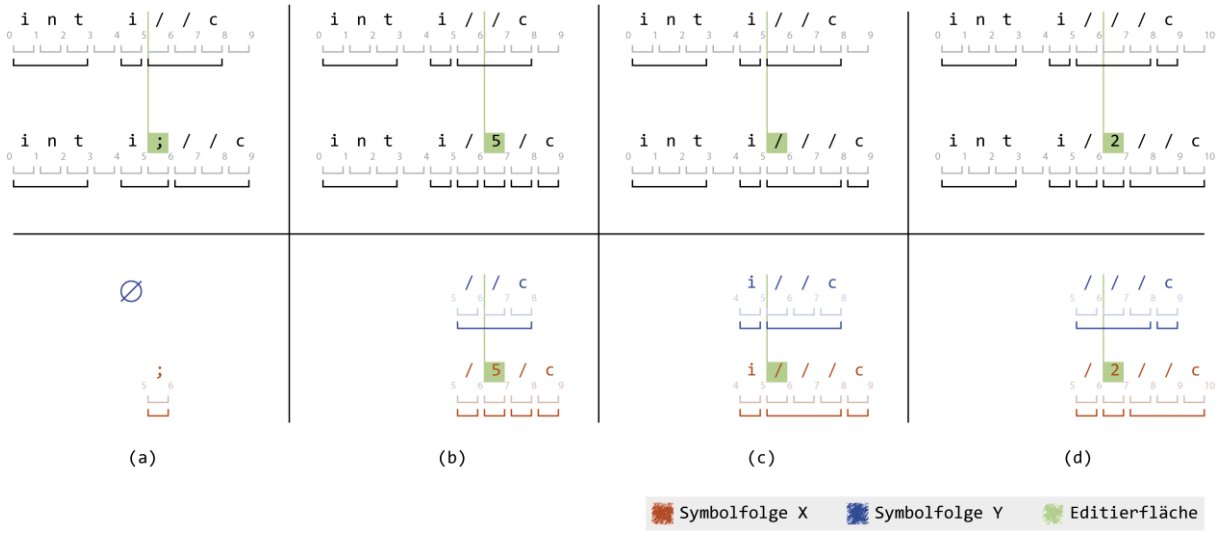


Abbildung 9 Beispiele für unterschiedliche X-Y-Startmengen Paare

3.2.2 Startmenge

Der Algorithmus zur Erkennung der Symbolfolgen X und Y ist in Abbildung 10 als Pseudocode angegeben. Er beschreibt, wie im Vorher-Baum (X_{All}) und Nachher-Baum (Y_{All}) jeweils beginnend bei der Editierung so lange Symbole markiert werden, bis der kleinste gemeinsame Zeichenabstand des am weitesten rechtsliegenden markierten Symbols zum

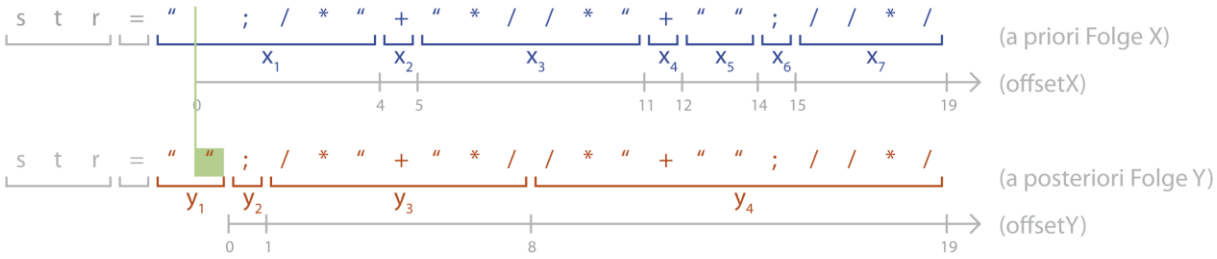
Editierungsende gefunden wird. Der Algorithmus beschreibt nur das Fortschreiten in Leserichtung nach rechts. X und Y müssen zusätzlich noch entgegen der Leserichtung nach links expandiert werden. Hierfür verhält sich das Verfahren jedoch analog und wurde daher ausgelassen. Anmerkend gilt noch zu beachten, dass sich ein Token über eine Menge von Zeichenpositionen innerhalb des gesamten Quellcodes erstreckt. Er berührt die Editierung genau dann, wenn beide mindestens eine Zeichenposition gemeinsam haben. Der im Pseudocode verwendete Befehl *dist* beschreibt den 1-dimensionalen euklidischen Abstand zwischen zwei Zahlen.

```
procedure FindMaxDiffSets(X_All : [Token], Y_All : [Token], edit : TextRange)
   returns X : [Token], Y : [Token]
{
   offsetX : int
   offsetY : int

   X := { x∈X_All | x touches edit }
   Y := { y∈Y_All | y touches edit }
   offsetX := dist(edit.End, (X.Last).End)
   offsetY := dist(edit.Ent, (Y.Last).End)

   while(offsetX != offsetY)
   {
      if(offsetX > offsetY) X.AppendNextTokenFrom(X_All)
      if(offsetX < offsetY) Y.AppendNextTokenFrom(Y_All)
      offsetX := dist(edit.End, (X.Last).End)
      offsetY := dist(edit.Ent, (Y.Last).End)
   }

   return X, Y
}
```

Abbildung 10 Der Algorithmus zum Auffindern von X und Y (in Rechtsrichtung)

Zum besseren Verständnis des aufgeführten Algorithmus wird in Abbildung 11 ein konkretes Beispiel gegeben. Der Quellcode-Ausschnitt ist unrealistisch, da er derart gewählt worden ist, dass die Funktionsweise des Algorithmus im Vordergrund steht. Im Beispiel wird durch die Eingabe eines einzelnen Anführungszeichens die Token x_1 bis x_7 in die Token y_1 bis y_4 überführt.
Da sowohl X- als auch Y-Menge für das Auslösen der initialen Regeln benötigt werden, wird ihre Wertesammlung als Startmenge bezeichnet:

Definition 7. Das Zwei-Tupel aus X und Y wird als *Startmenge* bezeichnet.



Iterationsschritt	offsetX	X	offsetY	Y
1	4	$\{x_1\}$	0	$\{y_1\}$
2	4	$\{x_1\}$	1	$\{y_1,y_2\}$
3	4	$\{x_1\}$	8	$\{y_1,y_2,y_3\}$
4	5	$\{x_1,x_2\}$	8	$\{y_1,y_2,y_3\}$
5	11	$\{x_1,x_2,x_3\}$	8	$\{y_1,y_2,y_3\}$
6	11	$\{x_1,x_2,x_3\}$	19	$\{y_1,y_2,y_3,y_4\}$
7	12	$\{x_1,x_2,x_3,x_4\}$	19	$\{y_1,y_2,y_3,y_4\}$
8	14	$\{x_1,x_2,x_3,x_4,x_5\}$	19	$\{y_1,y_2,y_3,y_4\}$
9	15	$\{x_1,x_2,x_3,x_4,x_5,x_6\}$	19	$\{y_1,y_2,y_3,y_4\}$
10	19	$\{x_1,x_2,x_3,x_4,x_5,x_6,x_7\}$	19	$\{y_1,y_2,y_3,y_4\}$

Abbildung 11 Extraktion der X und Y Menge anhand eines Beispiels

3.2.3 Erweiterung der Startmenge

Mit dem vorgestellten Algorithmus können alle sichtbaren Token ermittelt werden, die beim Übergang zwischen zwei Syntaxbäumen verändert werden. Eine Klasse von Token, die jedoch nicht zwingend erfasst wird, sind sogenannte *MissingTokens*.

Immer wenn unvollständiger Quelltext auftritt und ein Parser trotzdem einen Syntaxbaum generieren will, werden *MissingTokens* eingesetzt. Hierbei handelt es sich um Token mit der Wortlänge null, die die Stellen ausfüllen, an denen der Parser Text erwarten würde. Würde beispielsweise die Buchstabensequenz *int i =* durch den Benutzer eingegeben werden, würde der zugehörige Syntaxbaum einen Identifier-Ergänzungstoken und einen Semikolon-Ergänzungstoken erhalten (siehe Abbildung 12).

Da solche Ergänzungstoken keinen Text besitzen, werden sie nicht zwingend von der Startmengendetektion erfasst. Weil sie jedoch trotzdem berücksichtigt werden müssen, muss die existierende Startmenge um sie erweitert werden. Folglich wird ein zweiter Algorithmus benötigt.

Dieser zweite Algorithmus besteht aus gerichteten Abwärtstraversierungen beginnend bei der Baumwurzel des modifizierten Teilbaums und endend bei den neu modifizierten Ergänzungstoken, falls vorhanden. Hierbei werden stets diejenigen Knoten verfolgt, welche mit Fehlerdiagnosen markiert sind, in der Abbildung mit einem gelben Blitz gekennzeichnet. Das Ergebnis dieses Algorithmus wird als Erweiterungsmenge bezeichnet.

Definition 8. Es werde die Menge der Ergänzungstoken, die ausschließlich im *a priori*-Baum und nicht im *a posteriori*-Baum auftritt, als X' bezeichnet. Weiterhin werde die Menge der Ergänzungstoken, die ausschließlich im *a posteriori*-Baum und nicht im *a priori*-Baum

auftritt, als Y' bezeichnet. Dann wird das Zwei-Tupel aus X' und Y' als *Erweiterungsmenge* bezeichnet.

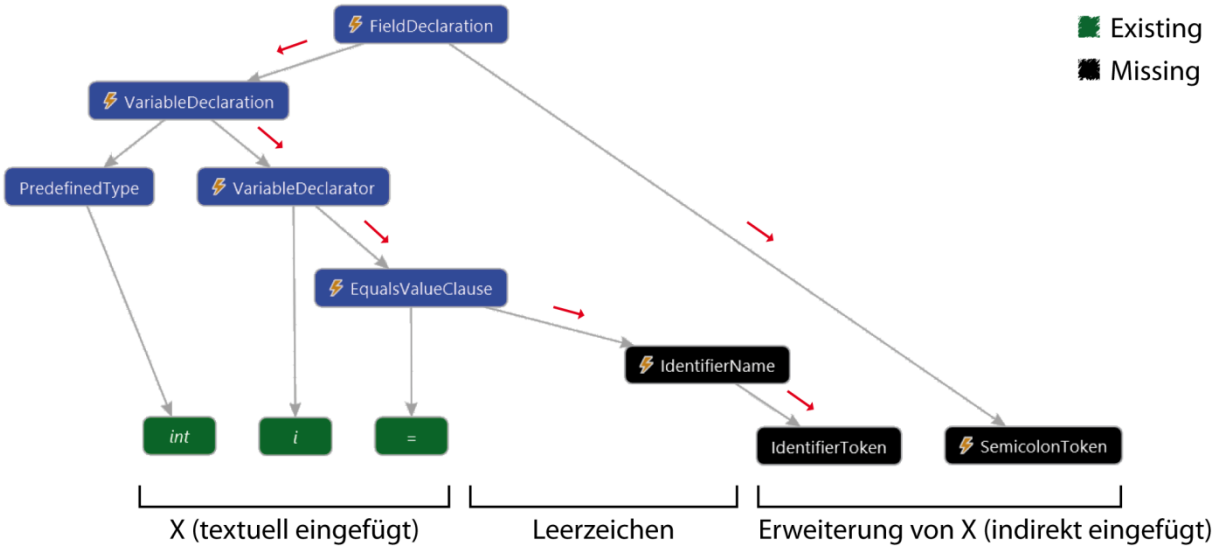


Abbildung 12 Detektion von Ergänzungstoken

3.3 DEFINITION DER OPERATOREN

In den vorangegangenen Kapiteln wurde festgehalten, dass in SITCOM Regeln zur Detektion von Blatt- und Knotenoperationen eingesetzt werden. Weiterhin wurde die Startmenge definiert, die als Auslöser für die Folge an Regelanwendungen fungiert. Jede ausgelöste Regel erkennt einen *(Änderungs-)Operator*. Ein Operator ist eine atomare, strukturändernde Aktion auf einem Syntaxbaum. Anhand von ihnen sollen Aussagen über die Semantik der durch den Nutzer getätigten Veränderung gemacht werden.

3.3.1 Nachteile vorhandener Operatoren

Auch in vergleichbaren Arbeiten wurden solche atomaren Operatoren definiert. So beschreibt beispielsweise Raghavan et al. [**1**] ein System, welches aus den Operatoren "insert", "delete", "update" und "move" besteht. Ähnliche Änderungsoperatoren nutzt auch Robbes [**54**], welcher seinen Algorithmus auf "Addition", "Removal", "Insertion", "Deletion", "Destruction" und "Creation" stützt.

Derartige Operationen sind jedoch nur bedingt hilfreich für das SITCOM-Verfahren. Zunächst ist es erwünscht, dass der Syntaxbaum bei jeder Operatoranwendung eine zusammenhängende Struktur beibehält. Operationen wie "Cut" oder "Create" [**54**] sollen daher nicht berücksichtigt werden.

Weiterhin ermöglicht das angestrebte SITCOM Verfahren Aussagen in einem Detailgrad, der zuvor nicht möglich war. Daher können zusätzliche Operationen wie "Split" und "Merge" implementiert werden, was in ähnlichen Arbeiten nicht möglich war.

Ein dritter Grund für die Abweichung von vorangegangenen Arbeiten besteht im Wunsch nach unterschiedlichen Operatoren für Blätter und Knoten. Dies liegt daran, dass in SITCOM-Blattoperationen und Knotenoperationen einen äußerst unterschiedlichen Charakter haben. Beide unterscheiden sich in Anzahl der Kindknoten, Triggertypen, Sichtbarkeit im Code, Stabilität und weiteren Eigenschaften.

Aus diesen und weiteren Gründen wurde für SITCOM eine eigene Menge von Operatoren entworfen.

3.3.2 Wichtige benötigte Eigenschaften

Trotz erwähnter Einschränkungen lassen sich dennoch nützliche Eigenschaften aus den Operatoren anderer Arbeiten ableiten. Vor allem sind zwei Gemeinsamkeiten zu erkennen. Zum Einen *decken* alle Operatoren den gesamten Transformationsraum *ab*, d.h. dass mit der Summe aller Operatoren jede mögliche Veränderung am Quellcode berücksichtigt werden kann.
Die zweite Gemeinsamkeit ist die *paarweise Disjunktheit* der Urbildmenge X, womit gemeint ist, dass es keine zwei Operationen geben kann, die gemeinsame Auslöser haben. Diese Eigenschaft führt dazu, dass Lösungen eindeutig sind und dass für eine wiederkehrende Veränderung im Quellcode stets die gleichen Operationen erkannt werden.
Eine dritte wichtige Eigenschaft ist die *Invertierbarkeit* der Operationen. Jede Operation benötigt eine Gegenoperation. Die Gegenoperation von Add ist beispielsweise Delete, die Gegenoperation von Merge ist Split. Die Invertierbarkeit spielt in Kapitel 3.6 eine zentrale Rolle, weil sie zu einer erheblichen Reduzierung der Regelmenge führt.

3.3.3 Blattoperatoren

Obige Betrachtungen haben zu dem Schluss geführt, eigene, auf SITCOM angepasste, Operationen zu entwerfen, welche die genannten Probleme vorhandener Operatoren beseitigen und die Stärken vorheriger Verfahren übernehmen. Die Blattoperatoren lauten wie folgt.

- **Add Leaf.** Die Bezeichnung für einen Operator, der ein Tokens $t \in X$ komplett neu einfügt. Bei der Y-Token Generierung darf keiner der bereits vorhandenen Buchstaben in t einfließen.
- **Delete Leaf.** Die Invertierung von Add, bei der ein Token $t \in X$ durch die Editierung komplett entfernt wird. Keines der Zeichen von t darf in der Y-Menge verbleiben.
- **Modify Leaf.** Das Einfügen oder Entfernen von Zeichen bei einem bestehenden Token t. Der einzige Wortunterschied zwischen $t \in X$ und $t' \in Y$ ist die entfernte/eingefügte Zeichenkette.
- **Move Leaf.** Die Bezeichnung für das Bewegen eines Blatts von einem Vaterknoten zu einem anderen.
- **Split Leaf.** Die Aufspaltung eines Tokens $t \in X$ in mehrere Token $t_i \in Y$ mit $t'_i \notin$ edit und $|t'_i| \geq 2$.
- **Merge Leaf.** Die Invertierung von Split und damit die Überführung von $t_i \in Y$ mit $t'_i \notin$ edit und $|t'_i| \geq 2$ nach $t \in X$.

Die Eigenschaften der derart definierten Operatoren werden in Abbildung 13 nochmals veranschaulicht. In der Abbildung sind die zwei Editierungsoperationen Einfügen (links) und Löschen (rechts) in allen möglichen Anordnungen zum zu interpretierenden Symbol dargestellt. Daraus, dass für jede Anordnung mindestens eine Operation vorhanden ist, lässt sich ableiten, dass für jede Gegebenheit eine Interpretation gefunden werden kann. Daraus, dass jede einzelne Anordnung nur genau eine Operation auslöst, wird die paarweise

Disjunktheit gezeigt. So entspricht z.B. die Überführung der Zeichenkette "in" nach "int" einem Modify Operator und keiner Operatorfolge aus Added("t") und Merged("in","t").
Bei weiterer Betrachtung der Abbildung fällt zudem die *InterstageOperation* auf. Auf diese wird im Abschnitt 3.4.2 eingegangen.

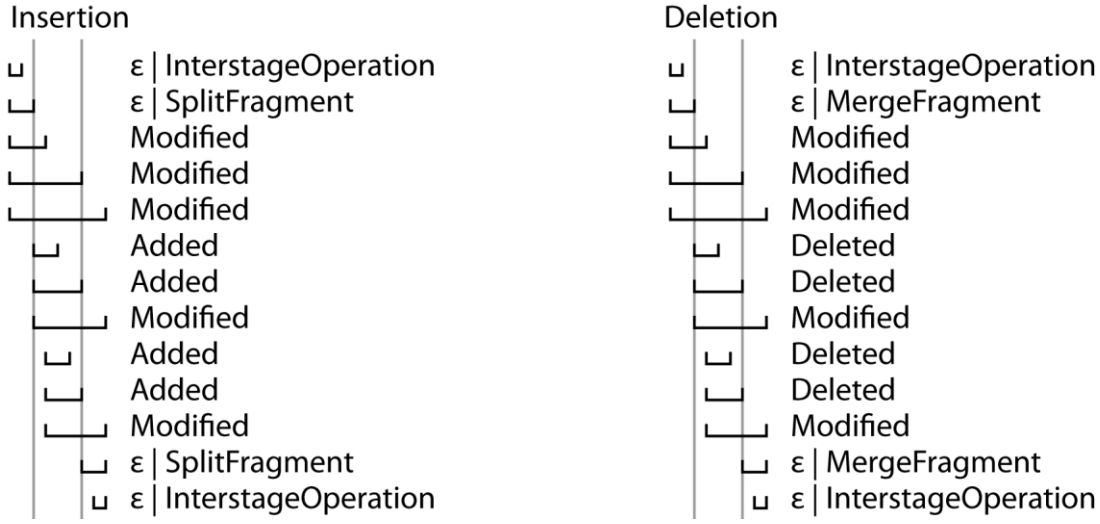


Abbildung 13 Mögliche Symbolverändungen und ihre resultierenden Operatoren

3.3.4 Knotenoperatoren

Die zuvor vorgestellten Blattoperatoren werden von Knotenoperatoren komplettiert. Dabei treten alle sechs Operatoren erneut auf, stehen jedoch im Kontext der Syntaxknoten. Zusätzlich werden die Operatoren Insert und Remove eingeführt.

- **Add Node.** Das Einfügen eines vor der Editierung nicht vorhandenen Knotens K. Der gesamte Textbereich von K muss innerhalb der Editierungsspanne liegen.
- **Delete Node.** Das Löschen eines nach der Editierung nicht mehr vorhandenen Knotens K. Die gesamte Textspanne von K muss innerhalb der Editierungsspanne liegen.
- **Modify Node.** Jede Aktion, bei der der Syntaxtyp eines Knotens K verändert wird, wird als *Modify Node* bezeichnet, gdw. mindestens ein Kind und mindestens ein Vater von K unverändert bleiben.
- **Move Node.** Die Bezeichnung für das Bewegen eines Knotens von einem Vaterknoten zu einem anderen.
- **Split Node.** Die Aufspaltung eines Knotens in mehrere Knoten und/oder Blätter.
- **Merge Node.** Die Invertierung der *Split Node*-Operation, bei die mehrere Syntaxknoten und/oder Blätter zu einem gemeinsamen Vaterknoten zusammengefasst werden.
- **Insert Node.** Das Einfügen eines Syntaxknotens in den bestehenden Baum. Im Gegensatz zur semantisch ähnlichen *Add Node*-Operation muss hier jedoch mindestens ein Kind sowohl im *a priori*- als auch im *a posteriori*-Baum erhalten bleiben.
- **Remove Node.** Die Invertierung der *Insert Node*-Operation, bei der ein Syntaxknoten gelöscht wird und mindestens ein Kindelement bestehen bleiben muss.

3.4 REGELN ZU DETEKTION VON BLATTOPERATIONEN

In Kapitel 3.2 wurden die X und Y Mengen vorgestellt, welche den Blattdetektionsregeln als mögliche Auslöser dienen. Weiterhin wurde in Kapitel 3.3.3 die Menge der möglichen Blattoperationen vorgestellt, die das Ergebnis einer Regelanwendung sind. Nachdem nun Auslöser und Ergebnis festgelegt sind, wird die Logik der Blattregeln vorgestellt.
Die Menge der Blattregeln ist aus einer Vielzahl an unterschiedlichen Regeln aufgebaut. Eine einzelne systematische Strategie zur Erstellung aller möglichen Regeln konnte nicht gefunden werden. Daher wurden unterschiedliche Strategien verfolgt und deren resultierenden Regeln aufsummiert, mit dem Ziel eine möglichst gute Gesamtabdeckung zu erzielen. Die vier prominentesten Strategien zur Regelfindung werden in den vier Folgeabschnitten vorgestellt.

3.4.1 Triviale Detektionsregeln

Die Strategie, die zu den meisten Regeln geführt hat, ist die triviale. Hier werden alle Token $x \in X$ in Beziehung zur Texteditierung gesetzt und mit den a posteriori Token $y \in Y$ verglichen. Ein x Token, dessen Spanne komplett innerhalb einer Lösch-Editierung platziert ist, wird beispielsweise als *Deleted Leaf* klassifiziert. Abbildung 13 bietet eine gute Übersicht über diese Klasse von Blatt Detektionsregeln.

3.4.2 Regeln basierend auf relationaler Algebra

Eine weitere Strategie zur Ermittlung von Blatt-Regeln stützt sich auf Methoden der relationalen Algebra. Die Idee besteht darin, die Startmengen X und Y als Mengen von Relationen anzusehen, bei der jedes $x \in X$ und jedes $y \in Y$ jeweils eine Relation aus Tokeninhalt und Textspanne bildet. Auf den so konstruierten Mengen lassen sich Blatt-Regeln anhand von Mengenoperationen generieren.

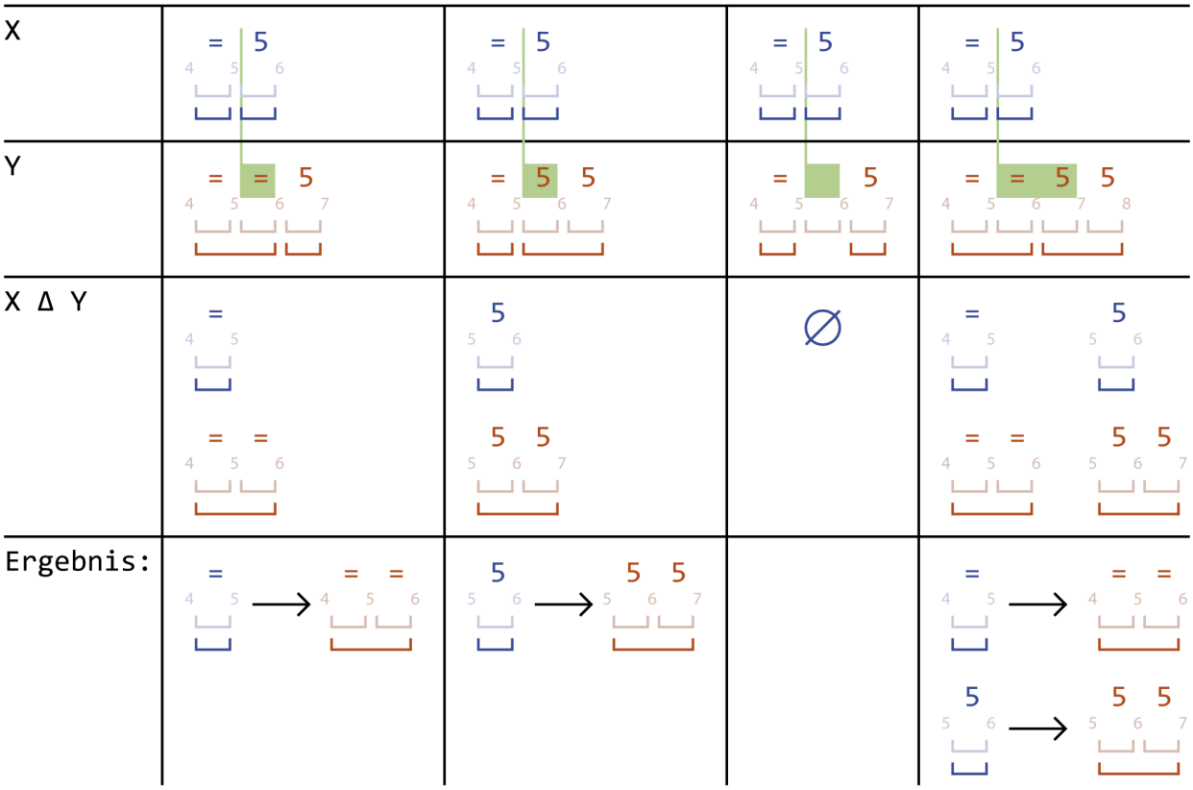


Abbildung 14 Erhalt von *Modify Leaf*-Operationen anhand des symmetrischen Schnitts

In Abbildung 14 wird beispielsweise beschrieben, wie eine *Modify Leaf*-erzeugende Regel anhand des symmetrischen Schnitts aus X und Y generiert werden kann. Die Idee besteht darin, dass der Schnitt all diejenigen Token entfernt, die in X und Y gemeinsam vorkommen. Wenn nach der Entfernung der Schnittmenge genau ein $x \in X$ und genau ein $y \in Y$ bestehen bleiben, muss daraus folgen, dass eine Modifizierung von x nach y stattgefunden hat.
Neben den *Modify Leaf*-generierenden Regeln, kann der symmetrische Schnitt auch für weitere Regeln genutzt werden. So kann beispielsweise bei einer 1:n-Beziehung von x zu y von einer *Split Leaf*-Operation ausgegangen werden.

3.4.3 Regeln basierend auf Zwischenzuständen

In einigen Fällen reichen die bisher vorgestellten Detektionsregeln nicht aus. Ein Beispiel hierfür ist im oberen Abschnitt von Abbildung 15 zu finden. Dort ist zu beobachten, dass sich X nicht ohne Weiteres in Y überführen lässt. Dies liegt daran, dass bei der Auflösung von Mergern, in diesem Fall m(**";/*"**), mehrere Modifikationsschritte implizit stattfinden, welche einer zeitlichen Ordnungsrelation folgen. So müsste m(**";/*"**) zunächst in die drei Fragmente a("), b(**;**) und c(**/*"**) aufgeteilt werden. Erst danach dürften weitere Schritte folgen, wie etwa das Feststellen einer Modifizierung von a(") nach a'("").
Um dieser zeitlichen Ordnung gerecht zu werden wird ein "Divide and Conquer"-Ansatz verfolgt. Die Idee basiert darauf, einen Zwischenzustand *Interstage I* als gemeinsame Oberfolge von X und Y einzuführen. Da X ⊂ I ⊃ Y gilt, existiert eine Operatormenge, die X nach I transformiert und eine zweite, die I nach Y überführt, siehe Abbildung 15.

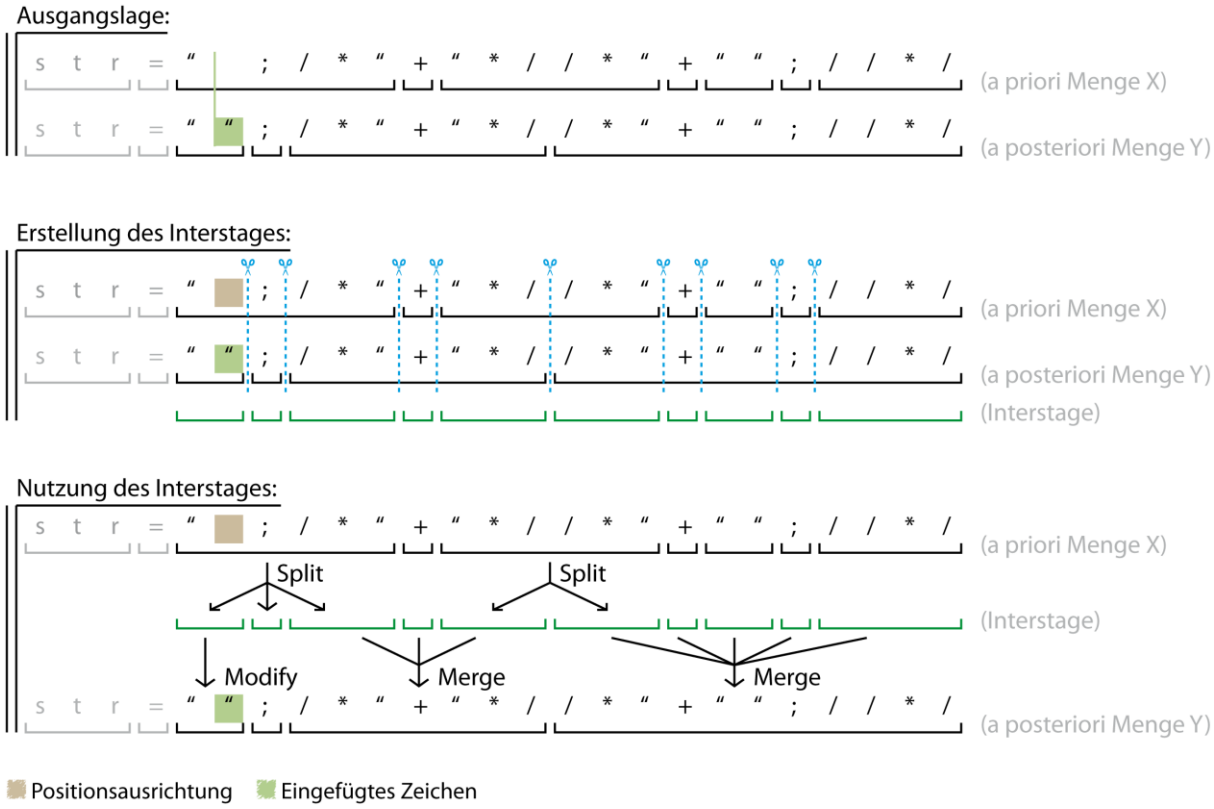


Abbildung 15 Erstellung und Nutzung des Interstages

Die Generierung des Interstages geschieht in mehreren Schritten, siehe Abbildung 15 (Mitte). Zunächst werden die Elemente von X und Y derart aneinander ausgerichtet, so dass die durch die Editierung bedingte Zeichenverschiebung berücksichtigt wird. Symbole links von der Editierung werden nicht verschoben, Symbole rechts von der Editierung hingegen schon. Für Symbole, die die Editierung schneiden gelten unterschiedliche Regeln; sie können unter anderem aufgebläht, zerlegt oder belassen werden. Da es hier sehr viele Fallunterscheidungen gibt, wird an dieser Stelle nicht weiter darauf eingegangen.
Die Tokengrenzen der neu ausgerichteten Menge X werden dazu genutzt die Token aus Y zu zerlegen und vice versa. Derart geteilte Token werden in SITCOM als *partielle* Token bezeichnet. Abschließend wird der Interstage I aufgefüllt, indem für jedes i die passende "Ausstanzung" aus X und Y genommen wird. Darauf, welches der jeweils passende Kandidat ist, wird an dieser Stelle ebenfalls nicht eingegangen.

3.4.4 Regeln basierend auf Fehlerbehandlungen

In Kapitel 3.2.3 wurde gezeigt, dass die X und Y Startmengen um Ergänzungstoken erweitert werden müssen. Diese Ergänzungstoken müssen bei der Detektionsregel-Generierung gesondert betrachtet werden.
Sie können eingefügt und gelöscht werden, auch wenn sie außerhalb der Texteditierung platziert sind. Hierzu müssen die Erweitungsmengen zu X und Y verglichen werden.
Weiterhin können sie dazu führen, dass eine erkannte *Added Leaf*- oder *Deleted Leaf*-Operation in *Modify Leaf* umgewandelt werden muss. Dies passiert immer dann, wenn ein unsichtbarer Ergänzungstoken auf derselben Position befindet. So kann z.B. ein Missing-Semikolon-Token in ein tatsächliches Semikolon-Token modifiziert werden.
Zudem kann eine *Modify Leaf*-Operation zu *Move Leaf* verändert werden. Eine solche Umwandlung ist genau dann notwendig, wenn eine Modifizierung einen Token derart verfälscht hat, dass der Compiler ihn nicht mehr in den syntaktischen Kontext einordnen kann. Da der Token sich nicht mehr zuordnen lässt wird er in ein sogenanntes SyntaxTrivia-Blatt *verschoben*, ein angehängtes Label, welches nicht mitkompiliert wird.

3.5 REGELN ZUR DETEKTION VON KNOTENOPERATIONEN

Aufbauend auf den Ergebnissen der *Blatt Operation*-Detektionsregeln, können nun Regeln zur Detektion von Knoten Operationen entworfen werden. Auch hier ist es nicht gelungen eine allgemeingültige Strategie zu finden, mit der alle möglichen Detektionsregeln erstellt werden können. Daher wird sich auch hier auf die Präsentation der drei ertragreichsten Strategien beschränkt.

3.5.1 Triviale Detektionsregeln

Der ersichtlichste Zustand tritt dann auf, wenn mehrere Blattoperationen des gleichen Typs unmittelbar nebeneinander aufgefunden werden. Alle Knoten, die sich innerhalb dieser Bereiche befinden, erhalten die Knoten Variante des Bereich-Kinderoperators. Wenn z.B. eine Nachbarschaft von Token existiert, in der alle Token eingefügt worden sind, so sind auch alle Knoten, deren Spanne sich komplett innerhalb dieser Nachbarschaft befindet, eingefügt.
Das Schema dieser Detektionsregel-Erstellungsstrategie ist in Abbildung 16 gegeben. Würde beispielsweise ein größerer Textbereich gelöscht werden, so würden auch die enthaltenen Token entfernt werden (siehe Abbildung 16, links). Von dort an bedarf es *Delete Node*-

Detektionsregeln, so dass sukzessive alle Vaterknoten innerhalb des Bereichs gelöscht werden. Diese Regeln können so lange angewandt werden, bis der Bereich verlassen wird. Ist ein Bereich verlassen, so ist ein *Randknoten* erreicht und die trivialen Detektionsregeln lassen sich nicht weiter anwenden.

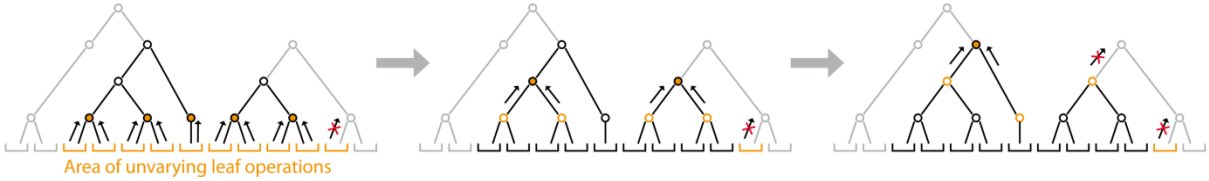


Abbildung 16 Funktionsweise trivialer Detektionsregeln

3.5.2 Regeln basierend auf Randknoten

Nachdem triviale Detektionsregeln angewandt worden sind, muss die Menge der übrig gebliebenen Randknoten untersucht werden. Dies geschieht anhand unterschiedlicher Heuristiken.

Gilt z.B. dass kein Nachfolgerknoten für einen Randknoten gefunden wird, dieser mindestens ein Kindknoten beibehält, mindestens einen Vater beibehält und mindestens ein Kindknoten gelöscht wird, so wird eine *Remove Node*-Operation detektiert.

Werden alle Knoten eines Trivialbereichs gemergt, so müssen alle Randknoten daraufhin untersucht werden, ob sie nach der Editierung weiterhin bestehen bleiben. Bleibt kein Randknoten bestehen, so wird die gesamte gemergte Menge *verschoben*.

Neben den Randknoten müssen auch deren Väter anhand von Detektionsregeln untersucht werden. Wird an einem Randknoten ein *Modify Node* erkannt, muss untersucht werden, ob gleiches für seinen Vater gilt. Wird ein Randknoten gelöscht, so muss untersucht werden, ob dessen Vater ein neues Kind erhält. Erhält der Vater kein neues Kind, so muss untersucht werden, ob er zu einem *IncompleteMember* verändert worden ist, usw...

Ein Spezialfall von Randknoten sind *Randblätter*, siehe Abbildung 16 rechts. Hierbei handelt es sich um Blätter, die keinen Nachbarn mit gleichartigem Detektionsergebnis haben. Die Logik hinter allen Regeln mit Randknoten als Auslösern gilt ebenfalls für Randblätter.

3.5.3 Regeln basierend auf Fehlerbehandlungen

In der Erweiterungsmenge zu X und Y befinden sich Ergänzungstoken und auskommentierte fehlerhafte Token. Die Detektionsregeln, die Token aus diesen Erweiterungsmengen zu ihren Auslösern zählen, unterscheiden sich in ihrer Funktionsweise nur unwesentlich zu bisher vorgestellten Aufwärtstraversierungen.

3.6 ANORDNUNG DER OPERATIONEN

In Kapitel 3.1.3 wurde festgelegt, dass SITCOM auf Regeln basieren soll. Weiterhin wurde die Generierung der Regeln schematisch beschrieben. Um sich nun von der textuellen Beschreibung zu lösen, und die Regeln in eine syntaktisch greifbare Notationen einzubetten, muss abschließend der Aspekt der Reihenfolge beleuchtet werden.

3.6.1 Motivation

Die Operatoren, die von Regeln erstellt werden, können voneinander abhängig sein. Beispielsweise kann ein Vaterknoten erst dann gelöscht werden, wenn seine Kinder gelöscht werden, da der Syntaxbaum ansonsten seine zusammenhängende Struktur verliert. Ein weiteres Beispiel ist ein Blatt das verschoben werden muss, bevor dessen Vater gelöscht werden kann, da der alte Vaterknoten Teil der *Move Leaf*-Parameter ist. Zur besseren Veranschaulichung solcher temporalen Abhängigkeiten ist ein Beispiel in Abbildung 17 gegeben.

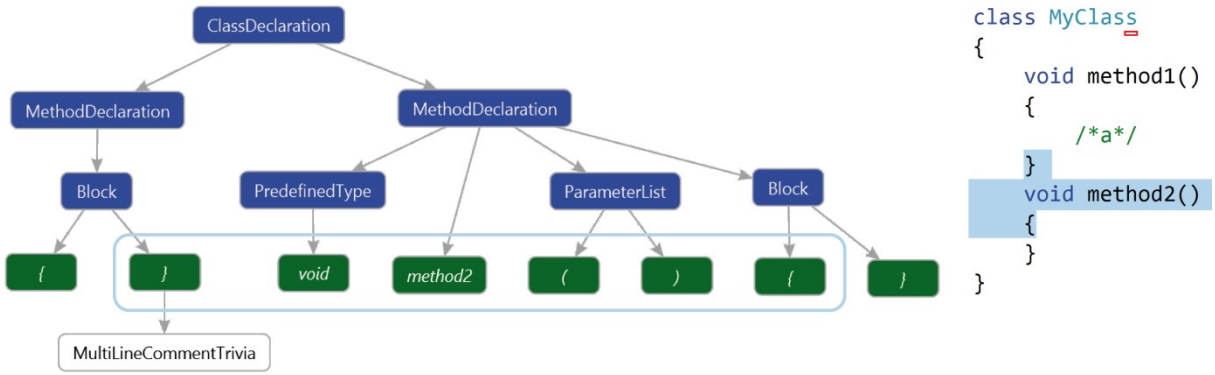


Abbildung 17 Geplante Löschung einer Textstelle

In diesem Beispiel wird eine Textspanne gelöscht. Die Löschung löst insgesamt elf Detektionsregeln mit zugehörigem Operator aus, siehe Abbildung 18. Doch wie zuvor angedeutet können nicht alle elf Detektionsregeln zeitgleich feuern. Die linke geschweifte Klammer kann erst gelöscht werden, nachdem der *MultiLineCommentTrivia* verschoben worden ist. Der *PredefinedType*-Knoten darf erst entfernt werden, wenn das *void*-Blatt nicht mehr existiert, usw.

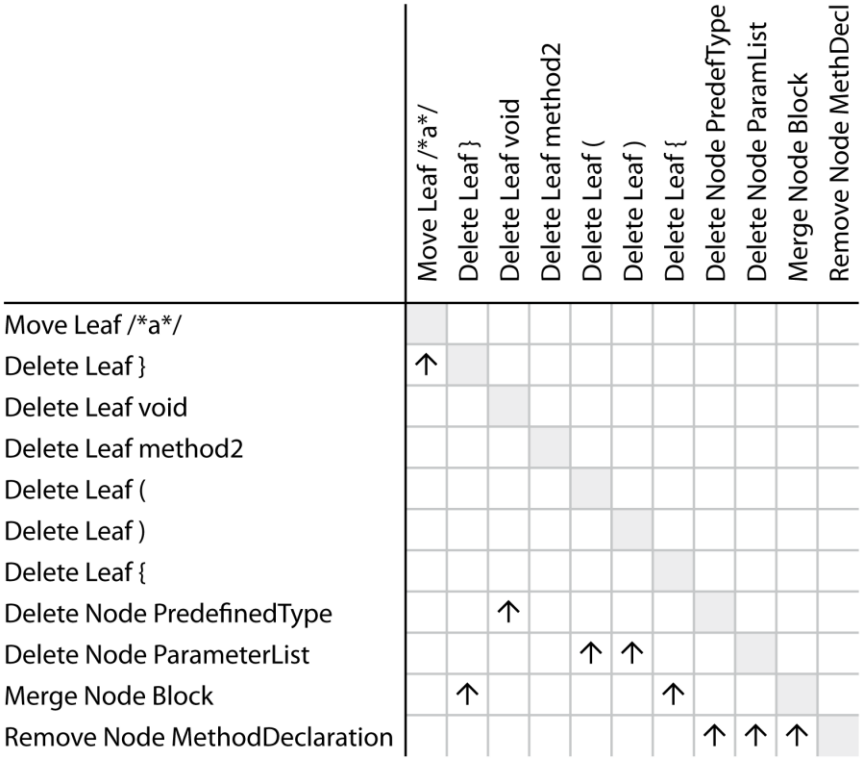


Abbildung 18 Abhängigkeitstabelle zum Beispiel aus Abbildung 17

Alle Abhängigkeiten zwischen den Operatoren, die in diesem Beispiel berücksichtigt werden müssen, sind in Abbildung 18 aufgezeigt. Jede Zeile beschreibt eine Operation. Immer wenn ein Pfeil-Symbol gezeigt ist, besteht eine Abhängigkeit zwischen der Zeilen-Operation mit der Spalten-Operation. Die *Merge Node* Block-Operation ist beispielsweise von *Delete Leaf }* und *Delete Leaf {* abhängig.
Sollen nun alle Operationen ausgeführt werden, so gilt es diverse Punkte zu beachten. Zum einen stellt sich die Frage, welche Operation(en) zuerst gefeuert werden soll. Weiterhin wäre es erstrebenswert möglichst viele Operationen parallel abzuarbeiten. Weiterhin kommen Punkte wie optimale Ausführungsreihenfolge, kausale Zusammenhänge, Terminierung, usw. hinzu. Eine geeignete Visualisierungsform zur Klärung dieser Punkte ist ein Abhängigkeitsgraph (siehe Abbildung 19). Dessen Erstellung wird im Folgeabschnitt beschrieben.

3.6.2 Partial-order Planer

Wie zuvor erwähnt soll ein Abhängigkeitsgraph erstellt werden, um die kausalen Zusammenhänge zwischen den Operationen zu visualisieren. Hierfür bietet sich ein Partial-order Plan aus dem Forschungsbereich der Intelligenten Handlungsplanung an [**57**]. Dabei ist ein Plan die Lösung zu einem *Planungsproblem*, welches aus einer Ausgangssituation, einer Zielsituation und einer Menge möglicher Aktionen besteht [**58**].

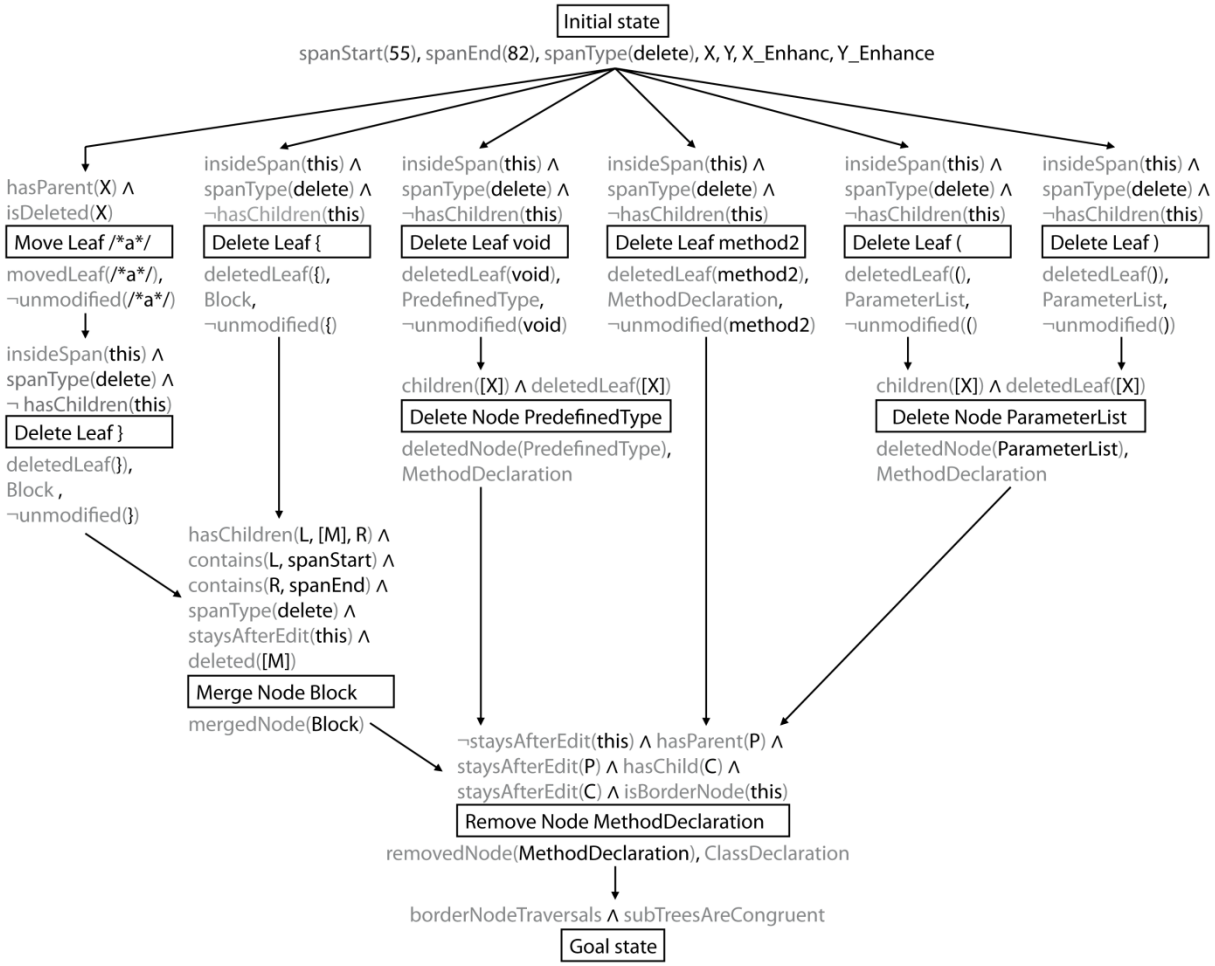


Abbildung 19 Der vereinfachte Partial-Order Plan zum Beispiel aus Abbildung 17

(a) Das Planungsproblem

Es folgt die formale Bestimmung der drei Bestandteile des zuvor definierten Planungsproblems.

Definition 9. Die *Ausgangssituation* ist die initiale Weltsituation, der Startzustand der Suche. Dieser wird meist durch eine Faktenbasis, bestehend aus unterschiedlichen Formeln beschrieben. Im SITCOM Kontext ist die Ausgangssituation durch die Summe aus TextEditierung, X,Y, X_Erweiterung und Y_Erweiterung gegeben.

Definition 10. Die *Zielsituation* ist der angestrebte Zustand der Welt. Ob dieser erreicht ist, kann durch eine Testfunktion ermittelt werden. In SITCOM gibt die Testfunktion wahr zurück, wenn alle Randknoten genügend hochtraversiert wurden und die minimalen Teilbäume kongruent sind. Die Definition der minimalen Teilbäume wird in Kapitel 3.7 gegeben, die Korrektheit der Zielbedingung wird in Kapitel 4.2 gezeigt.

Definition 11. Die *möglichen Aktionen* des Planungsproblems sind Operationen, die einen Einfluss auf die Welt haben. In SITCOM werden sie durch die bisher besprochenen Regeln gegeben und durch die formale Sprache STRIPS [**59**] formuliert. Der STRIPS Formalismus gibt vor, dass eine Aktion neben der Aktionsbeschreibung aus einer Menge von Vorbedingungen und Nachbedingungen bestehen muss. In Abbildungen werden

Aktionsnamen üblicherweise eingerahmt, ihre Vorbedingungen stehen textuell vor diesem Rahmen, ihre Nachbedingungen stehen dahinter. In Abbildung 19 sind elf Aktionen mit Vor- und Nachbedingung abgebildet.

(b) Funktionsweise des Planers

Das zuvor definierte Planungsproblem gibt Startzustand, Zielzustand und mögliche Operationen an. Es wird gelöst, indem, beginnend bei dem Startzustand, so lange Aktionen angewandt werden, bis der gegebene Zielzustand erreicht wird. Die Aktionskette, die von Start- zum Zielzustand führt, wird in einem Graphen festgehalten, welcher als *Plan* für das gegebene Problem definiert ist. Der Plan entspricht dem geforderten Abhängigkeitsgraph und ist damit das gewünschte Endergebnis für einen SITCOM Auswertungsschritt.

Das Erstellen der Aktionskette ist nicht immer trivial und bedarf häufig einer expliziten Strategie. Die Komponente, die anhand einer solchen Strategie Planschritte auswählt, wird als Planer bezeichnet. Häufig treten Fälle auf, in denen ein Planer auf eine Sackgasse stößt, und eine rekursives Zurückspringen erforderlich ist. Häufig sind Pläne auch nicht eindeutig, da es mehrere unterschiedliche Lösungen für ein Planungsproblem gibt. In SITCOM werden solche Probleme jedoch vermieden, indem die Aktionen entsprechend definiert werden. Dadurch, dass die Blatt- und Knotenoperatoren paarweise disjunkt sind, ist nur eine eindeutige Lösung möglich. Dadurch, dass die Vorbedingungen ausreichend präzise und detailliert sind, ist jede angewandte Aktion zwingend korrekt.

Der eigentliche SITCOM Planer nutzt Heuristiken, um die Menge der anwendbaren Regeln zu minimieren, beispielsweise können alle *Add Node*- und *Add Leaf*-Regeln ausgelassen werden, wenn die Formel *spanType(delete)* gilt. Danach wird die Restmenge entsprechend ihrer Anwendungswahrscheinlichkeit sortiert. Abschließend werden alle Regeln auf den jeweils aktuellen Zustand getestet und ggf. aktiviert, bis das Zielkriterium erreicht ist. Die ersten Schritte des SITCOM Planers werden in Abbildung 20 gezeigt.

```
spanStart(55), spanEnd(82), spanType(delete), x(/*a*/), x(}), x(void), x(method2), x((), x()),
x({), y(/*a*/)

try _Delete Leaf(L)_ ... failed.                    L := /*a*/, insideSpan(/*a*/) failed
try _Delete Leaf(L)_ ... applied.                   L := }

spanStart(55), spanEnd(82), spanType(delete), x(/*a*/), ¬x(}), x(void), x(method2), x((), x()),
x({), y(/*a*/), deletedLeaf(})

try _Delete Leaf(L)_ ... applied.                   L := void

spanStart(55), spanEnd(82), spanType(delete), x(/*a*/), ¬x(}), ¬x(void), x(method2), x((), x()),
x({), y(/*a*/), deletedLeaf(}), deletedLeaf(void)

try _Delete Leaf(L)_ ... applied.                   L := method2

                                (weitere Anwendungen...)
```

```
_Delete Leaf (L)_
Preconditions:   insideSpan(L), spanType(delete), ¬hasChildren(L)
Add List:        deletedLeaf(L), L.Parent
Delete List:     L
                                (weitere Regeln...)
```

Abbildung 20 Die erste Regelanwendungen des SITCOM-Planers am Beispiel aus Abbildung 17

Weiterhin lässt sich in der Abbildung zum ersten Mal die genaue Syntax einer SITCOM-Regel erkennen. Statt einer Postcondition wird eine Addlist und eine Deletelist genommen. Alle Elemente der Addlist sind nach der Regelanwendung wahr, alle Elemente der Deletelist sind nach der Regelanwendung falsch. Formeln, die nicht in den beiden Listen enthalten sind, bleiben unverändert.

3.6.3 Invertierung

Der initiale Startzustand des SITCOM Verfahrens besteht aus Syntaxtoken bzw. Blättern. Im Syntaxbaum bilden sie die unterste Schicht. Mit jeder Regelanwendung werden die Blätter nach und nach verlassen und immer weiter übergeordnete Knoten geprüft. Die Auswertungsreihenfolge ist aufwärts gerichtet, siehe Abbildung 21 (links).

(a) Idee

Im Falle einer Textlöschung funktioniert diese aufwärtsgerichtete Regelanwendung ohne Probleme. Zunächst werden die Blätter gelöscht, danach werden deren Vaterknoten gelöscht, usw. Zu jedem Zeitpunkt bleibt die zusammenhängende Struktur des Syntaxbaums erhalten.

Wird nun aber von einer Texteinfügung ausgegangen, so kann dies zu Problemen führen. Würde hier bei den Blättern gestartet werden, so würden die Blätter zwar erstellt aber nicht an den Restbaum angefügt werden, da deren Vaterknoten noch nicht existieren. Eine Alternative hierzu wäre es, bei einer Texteinfügung von den obersten Vätern auszugehen und von dort aus abwärts zu traversieren, siehe Abbildung 21 (rechts). Diese Top-Down Vorgehensweise würde sich jedoch nicht mit dem bisherigen Token-initiierten Ansatz decken. Eine elegante Lösung zu dem gegebenen Problem wird in Abbildung 21 beschrieben. Hier wird illustriert, wie eine Top-Down Knotentraversierung durch eine invertierte Bottom-Up Variante ersetzt werden kann. Die Idee besteht darin, dass bei einer Text*löschung*, die einen Baum A in einen Baum B überführt, im Grunde genau dasselbe passiert wie beim *Einfügen* des selben Textes, worauf Baum B zurück in Baum A überführt wird.

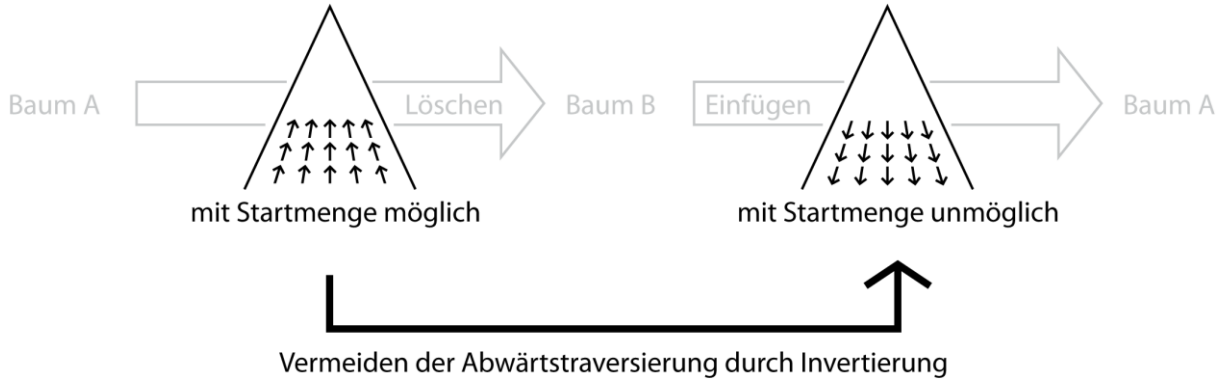


Abbildung 21 Erhalt des gesuchten Baums durch Invertierung

(b) Definition

Sei $Plan_{Del}$ der Gesamtplan, der einen Baum A in einen Baum B durch eine Textlöschung überführt. Sei weiterhin $Plan_{Ins}$ der Gesamtplan, der Baum B zurück in Baum A überführt. Für jede Aktion $Action_{Del}$ aus $Plan_{Del}$ gibt es genau eine Gegenaktion $Action_{Ins}$ in $Plan_{Ins}$, für die gilt: {Postconditions von $Action_{Del}$} = {Preconditions von $Action_{Ins}$} ∧ {Preconditions von $Action_{Del}$} = {Postconditions von $Action_{Ins}$}. Für jede gerichtete Kante die von $Action_{Ins1}$ auf

$Action_{Ins2}$ zeigt, existiert genau eine Gegenkante, welche von $Action_{Del2}$ auf $Action_{Del1}$ zeigt. Wird $Plan_{Del}$ als Graph (V,E) betrachtet, so gilt:

- $(Plan_{Del})^{-1}$ = ({Gegenkanten von $Plan_{Del}$}, {Gegenaktionen von $Plan_{Del}$})
- $(Plan_{Del})^{-1}$ = $Plan_{Ins}$.

(c) Konkretisierung
Zum besseren Verständnis der vorangegangenen Definition ist in Abbildung 22 ein Plan zusammen mit seiner Invertierung abgebildet. Die Gegenoperation von *Delete Leaf /*b*/* ist *Insert Leaf /*b*/*, die Gegenoperation von *Merge Leaf /*a*/,/*c*/* ist *Split Leaf/*a /*c*/*, usw. Die Gegenkante von (Delete Leaf, Merge Leaf) ist (Split Leaf, Insert Leaf), usw.
Neben der Vermeidung von Abwärtstraversierung bietet die Planinvertierung einen weiteren großen Vorteil. Dieser liegt in dem reduzierten Aufwand zur Erstellung der Regeln. Dies folgt daraus, dass es nicht mehr notwendig ist, Regeln, wie etwa für das Einfügen von Blättern, zu erstellen, da die Einfügeoperation ohnehin über das Planinverse erreicht wird.

3.7 ERMITTELN DER ZIELMENGE

Bisher wurde dem SITCOM-Planer ein initialer Weltzustand und eine Menge von Aktionen zugewiesen. Beginnend bei dem Startzustand werden die Aktionen nach und nach angewandt. Damit die Aktionsanwendung terminiert, muss nun abschließend der Zielzustand definiert werden.

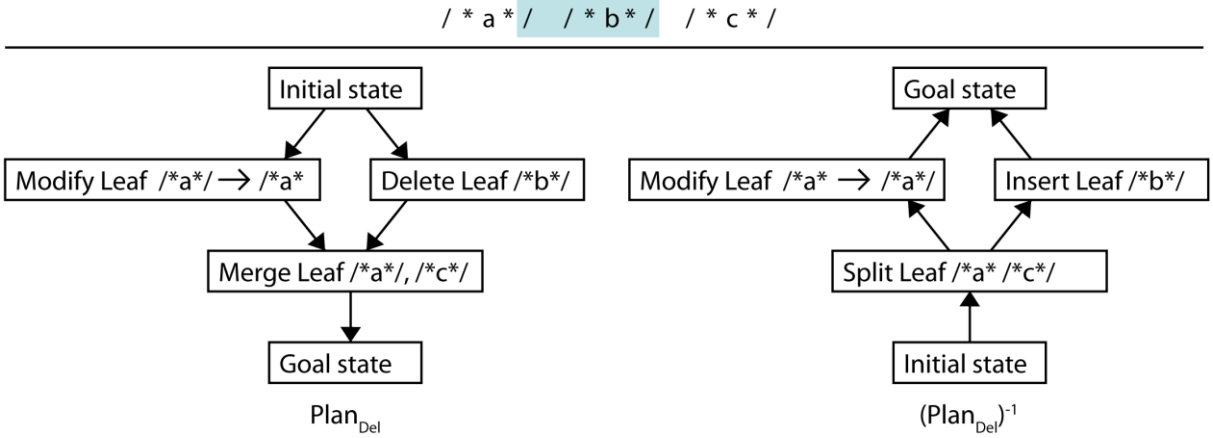


Abbildung 22 Ein Plan und seine Invertierung

Hierfür wird derjenige Syntaxknoten gesucht, der im *a priori*- und *a posteriori*-Syntaxbaum gleich ist, alle Änderungen enthält und minimal ist. Bezogen auf die veränderten Token ließe sich der gesuchte Knoten als kgV, *kleinster gemeinsamer Vaterknoten*, bezeichnen. Wenn die sukzessive, aufwärtstraversierende Regelanwendung beim kgV angelangt ist, kann davon ausgegangen werden, dass eine weitere Hochtraversierung keinen Sinn ergibt, da der Restbaum ohnehin identisch ist.
Für die Detektion des kgV findet eine gerichtete Suche statt. Die Suche startet bei dem gemeinsamen Baumwurzelknoten des *a priori*- und des *a posteriori*-Baums. Beginnend bei dieser Wurzel wird immer derjenige Kindknoten expandiert, welcher X respektive Y komplett

umspannt. Die Suche bricht dann ab, wenn der aktuelle Knoten sich in den zwei Bäumen unterscheidet, oder wenn keine Kinderknoten existieren, die X bzw. Y umspannen.
Neben der Nutzung als Abbruchkriterium hat der kgV noch weiteren Nutzen. Es ist beispielsweise ein besserer Ausgangspunkt zur Ermittlung der Erweiterungsmengen gegenüber dem Baumwurzelknoten. Dies liegt an der signifikant geringeren Menge der zu durchsuchenden Knoten mit Fehlerdiagnosen. Weiterhin wird der kgV bei der Fehlerunterdrückung, wie in Kapitel 4.3 beschrieben, genutzt, da ein Kongruenztest zwischen Teilbäumen signifikant effizienter ist als ein Kongruenztest über die gesamten Bäume.
Die Erreichung des kgV kann als absolutes Abbruchskriterium für die SITCOM Regelanwendung angesehen werden. Es gibt jedoch ein Abbruchskriterium, das schon früher greift, so dass nicht unnötig viele Regeln angewandt werden müssen. Aus Effizienzgründen wird daher dieses früher greifende Abbruchskriterium als Zielmenge für das SITCOM-Verfahren hergenommen. Hierbei handelt es sich um die Erreichung der *Randknoten*.

Definition 12. Ein *Randknoten* ist ein Syntaxknoten, der sowohl im *a priori*- als auch im *a posteriori*-Baum vorhanden ist, in beiden Bäumen derselben Knotenklasse angehört, in beiden Bäumen den gleichen Vater besitzt und mindestens einen Kindknoten besitzen, der verändert wurde

Wie später in Kapitel 4.2.2 gezeigt wird, sind alle Syntaxknoten, die durch eine Texteditierung verändert werden können, eine Teilmenge der Vereinigungsmenge über alle Nachfahren der Randknoten.

4 Wichtige Eigenschaften des SITCOM-Verfahrens

Im vorangegangenen Kapitel wurde das SITCOM Verfahren und seine Funktionsweise vorgestellt. Ergänzend dazu sollen in diesem Kapitel wichtige Aspekte dargestellt werden, die es bei der Anwendung von SITCOM zu beachten gilt.

4.1 EINSCHRÄNKUNGEN

Auch bei korrekter Ausführung muss bei SITCOM mit Einschränkungen gerechnet werden. Dies ist darin begründet, dass atomare Baumoperatoren verwendet werden und deren Einsatz zu Problemen führen kann.

4.1.1 Syntaxvalidität

Prinzipiell lässt sich jeder Syntaxbaum A anhand von Änderungsoperationen in einen Syntaxbaum B überführen. Vereinzelt können dabei jedoch derartige Konstellationen entstehen, dass eine Änderung nicht möglich ist, ohne die Grammatik des Baums zu verletzen.

Ein Beispiel hierfür ist in Abbildung 23 zu sehen. Dort werden zwei Varianten gezeigt, wie sich die Änderung eines Quellcodebeispiels von `i=5/4` nach `i=5//4` interpretieren lässt. Die Änderung hat zur Folge, dass *NumericLiteralExpression* der neue Kindknoten von *EqualsValueClause* wird. Dies kann erreicht werden, indem *DivideExpression* von *EqualsValueClause* und *NumericLiteralExpression* getrennt und *NumericLiteralExpression* anschließend an *EqualsValueClause* angefügt wird (siehe Abbildung 23, links). Ein wesentlicher Nachteil dieser Variante ist es jedoch, dass die zusammenhängende Struktur des Syntaxbaums aufgebrochen wird. Die Alternative hierzu befindet sich auf der rechten Seite der Abbildung. Dort wird beschrieben, wie *NumericLiteralExpression* von *DivideExpression* zu *EqualsValueClause* verschoben wird und *DivideExpression* daraufhin entfernt wird. Das Problem hierbei ist, dass *EqualsValueClause* direkt nach der Verschiebung zwei Kindknoten hat und somit die Grammatik verletzt.

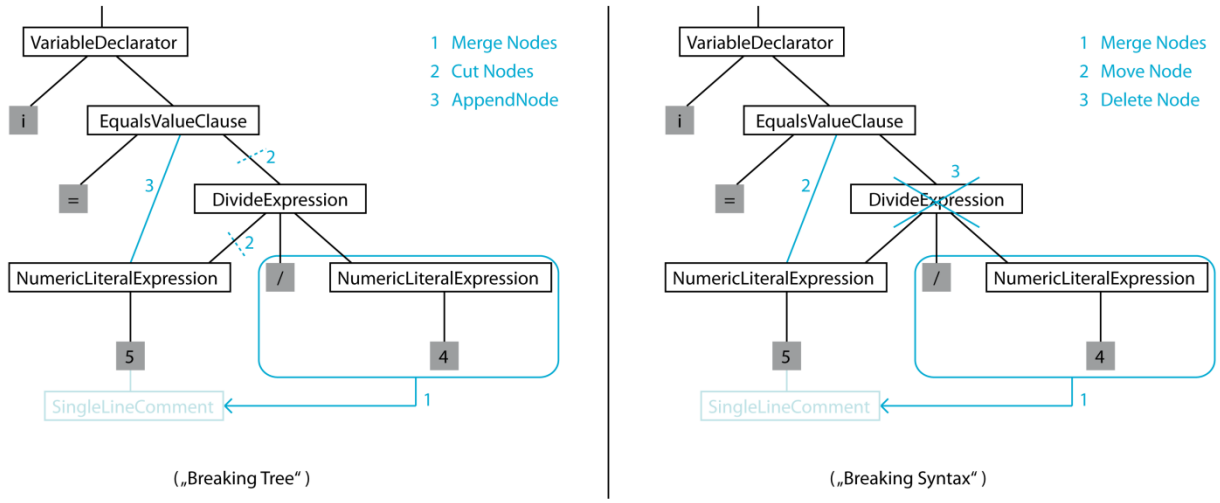


Abbildung 23 Breaking tree vs. breaking syntax

Egal für welche dieser zwei Varianten sich entschieden wird, die Lösung ist suboptimal. Da ein zusammenhängender Baum die Implementierung vereinfacht, wird im SITCOM-Verfahren die rechte Variante ausgeführt.
Ein weiterer Problempunkt für die Syntaxvalidität ist die Modifizierung von mehreren Elementen einer Vater-Kind-Kette, siehe Abbildung 24. In dem Beispiel aus der Abbildung wird ein Szenario gezeigt, bei dem zwei Modifikationen a und b getätigt werden müssen. Die *modify a*-Operation ändert den *IdentifierName*-Knoten zu *PredefinedType*, die *modify b*-Operation ändert den *"strin"*-Token zu *string*. Wird zuerst die a-Modifikation getätigt, so entsteht die Grammatik-verletzende Konstellation, dass *"strin"* ein *PredifinedType* ist. Wird hingegen mit der b-Modifikation gestartet, so ist *string* illegalerweise Kind eines *IdentifierNames* statt eines *PredifinedTypes*.

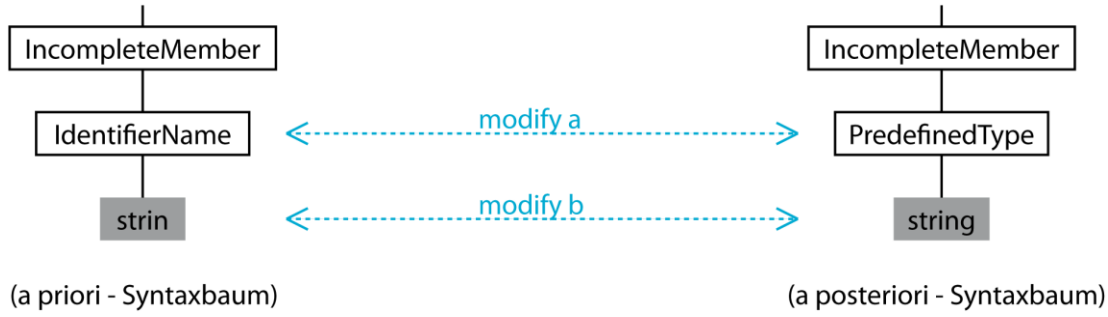


Abbildung 24 Problem des ersten Modify

Beide zuvor erwähnten Beispiele zeigen, dass in bestimmten Situationen der Erhalt der Syntaxbaum-Validität, unter Nutzung von atomaren Änderungsoperatoren, nicht möglich ist.

4.1.2 Knoten-Verfolgbarkeit

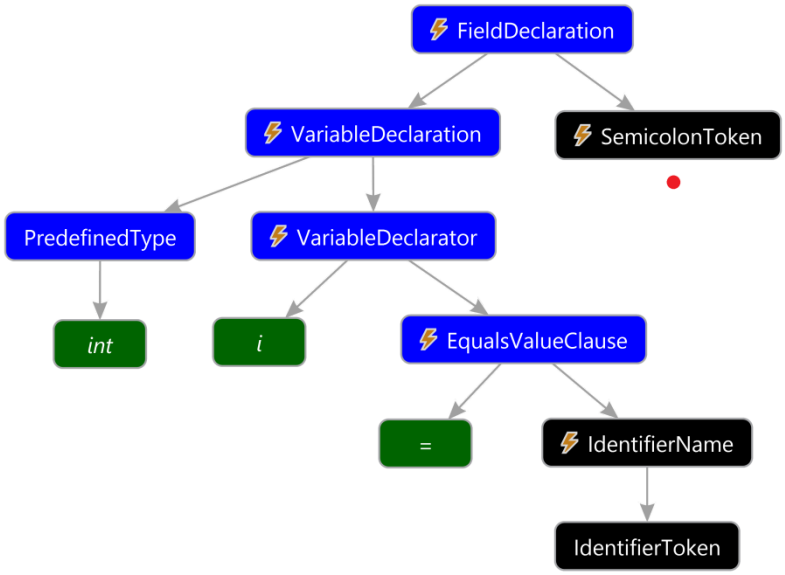
In einem Großteil der SITCOM Regeln werden Klauseln wie *staysAfterEdit(N)* verwendet, siehe Abbildung 19. Sie gibt wahr zurück, wenn der übergebene Syntaxknoten N auch nach der Editierung bestehen bleibt. Um die Funktionalität dieser Formeln umzusetzen, muss der markierte Knoten N ∈ *a priori*-Baum im *a posteriori*-Baum gefunden werden. Diese Lokalisierung kann nur geschehen, wenn die korrekte Startposition von N ∈ *a posteriori*-Baum gefunden wird. Der Nachfolger eines Knotens wird wie folgt definiert:

Definition 13. Sei a ein Syntaxtoken oder -knoten des *a priori*-Baums. Sei weiterhin b ein Syntaxtoken oder -knoten des *a posteriori*-Baums. b ist genau dann ein *Nachfolger* von a, wenn sich a in b überführen lässt. Zwingende Bedingungen für eine Überführung ist, dass der Abstand zwischen a.Start und b.Start allein auf die Texteditierung zurückzuführen ist und dass (a.Inhalt ⊆ b.Inhalt) ∨ (a.Inhalt ⊇ b.Inhalt) gilt.

In Abbildung 24 ist *PredefinedType* der Nachfolger von *IdentifierName* und *string* der Nachfolger von *strin*.

(a) Offsets
In der Definition wird der Begriff Abstand aufgeführt. Dabei wird der Abstand zwischen a.Start und b.Start als Offset *bezeichnet*. Es gilt: neue Startposition = alte Startposition + Offset. Im Beispiel aus Abbildung 25 wird ein *a priori*-Syntaxbaum angegeben, bei dem der

Semicolon-Ergänzungstoken auf mögliche Offsets geprüft werden soll. Bei unterschiedlichen Eingaben verschiebt sich der Nachfolgeknoten um unterschiedlich viele Stellen. Untere Tabelle in Abbildung 25 stellt ausgewählte Änderungsfälle und die daraus resultierenden Offsets für den Nachfolgeknoten dar.
Es ist festzustellen, dass es für den einfachen angegebenen *a priori*-Baum mindestens die 14 angegebenen unterschiedlichen Möglichkeiten gibt, den Offset zu beeinflussen. Bei der Tabelle wurden diverse Fälle ausgelassen, beispielsweise alle Fälle, in denen vorhandener Text gelöscht wird. Dieses Beispiel und die Tatsache, dass es sich nur um ein Beispiel von vielen möglichen handelt, lassen den Schluss zu, dass die Bestimmung des richtigen Offsets eine nicht triviale Aufgabe ist. In SITCOM wird die Offset-Bestimmung über ein Modul gesteuert, welches auf einen Datensatz an Fallunterscheidungen beruht. Die Summe der dort abgelegten Regeln deckt den Großteil der gebräuchlichsten Code Veränderungen ab. Das Regelwerk konnte jedoch nicht auf Vollständigkeit geprüft werden, da keine Strategie gefunden worden ist, mit der sukzessive alle möglichen Offset-Fälle ermittelt werden können.



Einfüge Editierung	Offset
[any] int␣i␣=␣	[any].Length
int␣i␣=␣ [any]	0
int␣i␣=␣5	1
int␣i␣=␣5␣	2
int␣i␣=␣-5	2
[int␣i2␣=␣	1
int␣i␣==␣	n/a.

Einfüge Editierung	Offset
int␣i␣=␣;	0
int␣i␣=␣␣	1
int␣i␣=;␣	-1
int␣i␣=;;␣	-1
int␣i␣=␣;;	0
int␣i␣=\n␣	-1
//int␣i␣=␣	n/a.

Abbildung 25 Verfolgung des Semicolon-Ergänzungstokens

(b) Beispielszenario zur Nachfolgerermittlung

Im Beispiel aus Abbildung 25 wird weiterhin angedeutet, dass Fehler bei der Verfolgung von Knoten auftreten können. Da *SemicolonToken*, *IdentifierName* und *IdentifierToken* alle dieselbe Startposition besitzen, kann es zu fehlerhaften Zuordnungen kommen. Um diese Verwechslungen zu vermeiden wird in SITCOM eine Ähnlichkeitsfunktion P aus Knotenbezeichnung, Name des Vaters und Anzahl Kinderknoten genutzt:

$$P(\text{Knoten}_{Pre}, \text{Knoten}_{Post}) = \frac{(1\text{-}d_1(\text{Knoten}_{Pre}, \text{Knoten}_{Post})) + (1\text{-}d_2(\text{Knoten}_{Pre}, \text{Knoten}_{Post})) + (1\text{-}d_3(\text{Knoten}_{Pre}, \text{Knoten}_{Post}))}{4}$$

$$d_1(a, b) = \begin{cases} 1 & , a.\text{Bezeichner} = b.\text{Bezeichner} \\ 0 & , \text{sonst} \end{cases}$$

$$d_2(a, b) = \begin{cases} 1 & , |a.\text{Kinder}| = |b.\text{Kinder}| \\ 0 & , \text{sonst} \end{cases}$$

$$d_3(a, b) = \begin{cases} 1 & , (a.\text{Inhalt} \subseteq b.\text{Inhalt}) \vee (a.\text{Inhalt} \supseteq b.\text{Inhalt}) \\ 0 & , \text{sonst} \end{cases}$$

Ein mögliches Szenario, das eine Anwendung dieser Funktion demonstriert, tritt dann auf, wenn der Nachfolger des *IdentifierNames* bestimmt werden soll, während im Editor eine "5" eingegeben wird, siehe linke Tabelle, dritte Spalte von oben. Der entstehende *a posteriori*-Baum sieht nahezu identisch aus, mit dem Unterschied, dass *IdentifierName* zu *NumericLiteralExpression* und *IdentifierToken* zu *SyntaxToken 5* modifiziert wird.

Zunächst werden die Offset-Regeln dazu verwendet, alle Syntaxknoten und -token des *a posteriori*-Baums zu finden, deren Startwerte dem Startwert des alten *IdentifierNames* gleichen. Dies trifft zunächst auf die *a posteriori*-Elemente *SemicolonToken*, *NumericLiteralExpression* und *Syntaxtoken* zu. Bei ihnen handelt es sich um die möglichen Nachfolger zu *IdentifierName*. Als nächstes wird die Ähnlichkeitsfunktion P genutzt, um den richtigen Kandidaten zu ermitteln. P(*IdentifierName*, *SemicolonToken*) ergibt 0, woraufhin der Offset des SemicolonTokens korrigierend um 1 erhöht wird. Der nächste Kandidat ist *IdentifierToken* und erhält den Wert $P(IdentifierName, IdentifierToken) = 0{,}66$. Als letztes wird die *NumericLiteralExpression* getestet und wird mit $P(IdentifierName, NumericLiteralExpression) = 1$ evaluiert. Damit ist *NumericLiteralExpression* der Nachfolger von *IdentifierName*.

(c) Vergleich mit anderen Arbeiten

Vorherige Punkte haben aufgezeigt, dass die Verfolgbarkeit von Syntaxknoten nicht trivial ist. Eine vollständig korrekte Lösung zum Verfolgbarkeitsproblem konnte nicht gefunden werden. Es kann lediglich angeführt werden, dass sich die Verfolgbarkeit in allen Testfällen in Rahmen von SITCOM als korrekt erwiesen hat. Daher kann derzeit auch für die *staysAfterEdit(N)* Formel keine absolute Korrektheit ausgesprochen werden, womit auch das allgemeine SITCOM Regelwerk nicht mathematisch beweisbar vollständig korrekt ist.

Im Vergleich zu Verfahren aus ähnlichen Arbeiten, kann jedoch bei dieser Art von Token-Tracing die vermutete Position für den Nachfolger anhand des Offsets genau ausgemacht werden. Damit ist der Suchraum auf eine einzige Zeichenposition beschränkt, statt auf die Zeichenspanne des gesamten Syntaxbaums. Auf diese Weise wird die Zahl der *a posteriori*-Nachfolgerkandidaten für einen *a priori*-Knoten reduziert, so dass die Wahrscheinlichkeit für

ein korrektes Suchergebnis signifikant erhöht wird. In [**1**] wird beispielsweise eine Wahrscheinlichkeitsfunktion zum Knoten-Tracing verwendet, bei der, ähnlich wie in SITCOM, Informationen zu Kindknoten und Knotenbezeichnung einfließen. Diese Wahrscheinlichkeitsfunktion wird aber global auf viele unterschiedliche Nachfolgekandidaten des *a posteriori*-Baums angewandt. Weitere Beispiele für global angewandte Ähnlichkeitsmaße sind in
[**18**], [**15**] und [**5**] zu finden. In diesen drei Arbeiten wird versucht, den Nachfolger eines Knotens anhand von funktionalen Aspekten, wie z.B. deren Abbildung im Klassendiagramm oder Kontrollflussdiagramm, abzuleiten.
Überdies hinaus setzen einige Arbeiten wie [**46**] oder [**18**] voraus, dass ein Nachfolger eines Knotens textuell gleich sein muss. In diesen Arbeiten kann beispielsweise eine *VariableDeclaration int i = 55* kein Nachfolger der *VariableDeclaration int i = 5* sein. In SITCOM sind solche Vorgänger-Nachfolger-Paare jedoch gewollt und möglich.
Der Ansatz zur Knotenverfolgbarkeit, der SITCOM am ähnlichsten ist, wird in [**6**] beschrieben. Dort werden alle textuellen Unterschiede zwischen zwei Quellcode Dateien anhand des Unix-Programms *diff* [**47**] ermittelt. Daraufhin werden alle textuellen Unterschiede einzeln daraufhin untersucht, auf welche Weise sie den *a priori*-Baum in den *a posteriori*-Baum überführen und daraus die Änderungsoperationen abgeleitet. Ebenso wie in SITCOM werden textuelle Unterschiede genutzt, um den Übergang zwischen unterschiedlichen Knoten herzuleiten. Verglichen mit SITCOM kann mit dem Ansatz aus [**6**] jedoch nicht mit genauen Offsets hantiert werden, da *diff* zeilenweise und nicht buchstabenweise agiert.

4.1.3 Offene Merger

Für das Verständnis der im Folgenden beschriebenen Einschränkung, muss der Begriff des *Mergers* definiert werden:

Definition 14. Ein *Merger* ist eine Zeichenfolge, die mit einem *Opening Tag* beginnt und mit einem *Closing Tag* endet. Der Inhalt eines Mergers wird nicht interpretiert. Ein lexikalischer Scanner erkennt einen Merger als eine einzige Einheit an. Die Opening und Closing Tags eines Mergers werden von der Grammatik vorgegeben. In der C# Grammatik sind mögliche Merging Paare beispielsweise: Anführungszeichen und Anführungszeichen, Doppelter Slash und Zeilenumbruch oder Schrägstrich+Stern und Stern+Schrägstrich.

In SITCOM kann die Erstellung eines OpeningTags ohne die Existenz eines dazugehörigen ClosingTags zu Problemen führen. Dies tritt beispielsweise in einem Szenario auf, in dem der Benutzer einen mehrzeiligen Codeabschnitt auskommentieren will. Startet er die Auskommentierung mit der Eingabe einer OpeningTag-Zeichenfolge, so werden alle Folgetoken bis zum Dateiende, oder dem ClosingTags eines anderen Mergers, gemergt. Dieser ungewollte Merge birgt mehrere Nachteile in sich. Zunächst wird damit der Nutzerintention nicht korrekt entsprochen, da dieser lediglich alle Token bis zu einem gewissen Punkt (dem noch nicht eingegebenen ClosingTag) zusammenfassen wollte. Weiterhin werden Token gemergt, die mit der geplanten Situation in keinem Zusammenhang

stehen, so kann die Verfolgbarkeit dieser außenstehenden Token mitunter unnötigerweise nicht gewährleistet werden.
Um die genannten Schwierigkeiten zu vermeiden, werden in SITCOM *vorläufige* Events geworfen. Hierbei handelt es sich um Events, bei denen ein *"unfertig"*-Attribut auf *wahr* gesetzt wird. Das unfertige Merge-Event dient zur Notifizierung, dass ein Closing Tag noch folgen wird, und hilft der internen SITCOM Architektur, die ungewollt gemergten Token korrekt zu behandeln.

4.2 KORREKTHEIT

Ein wesentlicher Punkt zur Bewertung des SITCOM-Verfahrens ist seine Korrektheit. Kann diese bewiesen werden, so ist garantiert, dass SITCOM der gestellten Spezifikation aus 3.1.1 genügt.
In Abschnitt 3.6.2(a) wurde das Planungsproblem zu SITCOM definiert. Es beschreibt alle Elemente, die für eine vollständige Lösung benötigt werden, namentlich Ausgangssituation, mögliche Aktionen und Zielsituation. Wird gezeigt, dass Initial- und Zielzustand in jeglicher Situation korrekt sind und weiterhin bewiesen, dass die Regelmenge vollständig und ebenfalls korrekt ist, so kann daraus abgeleitet werden, dass auch das gesamte SITCOM-Verfahren fehlerfrei läuft.

4.2.1 Korrektheit der Ausgangssituation

Die Ausgangssituation des SITCOM-Verfahrens ist zusammengesetzt aus Startmenge und Erweiterungsmenge. Kann gezeigt werden, dass ihre Konkatenation alle Token enthält, die sich in *a priori*- und *a posteriori*-Baum unterscheiden, dann ist die Ausgangssituation vollständig und korrekt. Token, die ohnehin in beiden Bäumen gleich sind, müssen nicht von Regeln ineinander überführt werden.
Der Nachweis der Korrektheit geschieht in zwei Schritten. Zunächst wird die Vollständigkeit der Startmenge gezeigt, danach wird die Vollständigkeit der Erweiterungsmenge gezeigt. Für erstere muss noch der Begriff des Delimiters definiert werden:

Definition 15. Die Abgrenzung zwischen zwei Token wird als *Delimiter* bezeichnet. Übliche Beispiele hierfür sind Leerzeichen, Punkte, Semikolons, etc.

Hypothese. FindMaxDiffSets(...) aus Abschnitt 3.2.2 findet die minimalen, vollständigen Folgen X und Y aller Symbole, deren Vergleich als Basis für die Extraktion von Änderungsoperatoren genutzt werden kann.

Schlussfolgerung.

1. Lediglich die Token, deren Textinhalt sich ändert, sind relevant für eine Untersuchung auf semantische Änderungen.
2. Es müssen ausschließlich Token gefunden werden, deren Textinhalt sich ändert.
   2.1. Zunächst müssen alle Token, die die Editierstelle berühren, überlappen und/oder von ihr eingeschlossen werden, als Änderungskandidaten angesehen werden.
   2.2. Es bleiben lediglich die Symbole übrig, deren Textinhalt sich ändert, obwohl sie die Editierstelle nicht berühren.

2.2.1. Ohne direkten Einfluss der Editierung ändert ein Token seinen Textinhalt nur dann, wenn die Bedeutung mindestens eines seiner Delimiter aufgehoben wird.
2.2.2. Ohne direkten Einfluss kann die Bedeutung eines Delimiters nur aufgehoben werden, wenn er durch einen Merger außer Kraft gesetzt wird.
2.2.3. Wenn das Token, dessen Delimiter außer Kraft gesetzt wird, wiederum ein Merger ist, so können neue Token freigesetzt werden, welche zuvor gemergt waren.
2.2.4. Ist eines der freigelegten Token wiederum ein Opening Tag für einen weiteren Merger, so können wiederum weitere Merger entstehen und Delimiter außer Kraft gesetzt werden. Es entsteht ein Domino Effekt.

3. Alle Token, die den Vergleichsmengen X und Y angefügt werden sollen, sind im Intervall R mit:

$$R := \begin{cases} [Edit.End, Max(RightTouchingToken_X.End, LeftTouchingToken_Y.End)] \\ \qquad\qquad ,Merger\ Sequenz\ nicht\ vorhanden \\ [Edit.End, Max((LastMergerSequence_X.Last).End, (LastMergerSequence_Y.Last).End)] \\ \qquad\qquad ,Merger\ Sequenz\ vorhanden \end{cases}$$

4. Aus der while-Schleifenbedingung von FindMaxDiffSets(...) folgt, dass die erste Übereinstimmung von offsetX und offsetY bei R.End stattfinden muss, damit der Algorithmus korrekt terminiert. Dies gilt es zu zeigen.
   4.1. Wenn keine Merger Sequenz auftritt gilt trivialerweise RightTouchingSymbol$_X$.End == RightTouchingSymbol$_Y$.End == E.End
   4.2. Falls diese auftritt gilt, dass für jedes entstandene Merge M in X, dass Y so lange mit Symbolen aufgefüllt werden muss, bis M.End erreicht ist.
   4.3. Der gleiche Fall gilt auch vice versa.
   4.4. Wenn ein Merge M komplett aufgefüllt ist, ohne dass ein neuer Merge entsteht, so gibt es keine Möglichkeiten mehr dafür, dass weitere Token rechts von M.End durch die Editierung verändert werden können.
5. Der Algorithmus füllt X und Y mit allen ihren jeweiligen Token auf und terminiert korrekt bei R.End.
6. Für die linksgerichtete Rekursion gilt der Algorithmus analog.

Q.E.D.

Damit wurde die Vollständigkeit der Startmenge gezeigt. Wenn der Quellcode syntaktisch fehlerfrei ist, dann entspricht die Startmenge der gesamten Ausgangssituation für das Planungsproblem. Ist der Quellcode jedoch fehlerhaft, so entstehen Ergänzungstoken mit der Buchstabenbreite null. Ergänzungstoken werden nicht durch den Nachweis abgedeckt, da sie wegen ihrer Wortlänge keine echte Struktur aufweisen und daher nicht von Punkt Eins betroffen sind.
Der Algorithmus zur Ermittlung der Erweiterungsmenge wurde in Kapitel 3.2.3 kurz vorgestellt, jedoch nur sehr knapp beschrieben. Dies lag an den noch fehlenden Definitionen für kgV aus Kapitel 0und *staysAfterEdit*(N) aus Kapitel 4.1.2. Aufbauend auf den zwei Definitionen kann nun angefügt werden, dass bei der Erweiterungsmengenermittlung zwei Abwärtstraversierungen stattfinden. Die erste beginnt beim kgV des *a priori*-Baums und ermittelt alle Ergänzungstoken E_X. Die zweite beginnt beim kgV des *a posteriori*-Baums und ermittelt alle Ergänzungstoken E_Y. Anhand der Tracing-Mechanismen aus Kapitel 4.1.2 kann

der gemeinsame Schnitt von E_Y und E_X entfernt werden und es entsteht die Erweiterungsmenge ($E_X \backslash E_Y$, $E_Y \backslash E_X$).

Hypothese. Der Algorithmus zu Ermittlung der Erweiterungsmenge erfasst die Menge aller Ergänzungstoken, die sich in *a priori*- und *a posteriori*-Baum unterscheiden. Dabei muss die Menge vollständig und minimal sein.

Schlussfolgerung.

1. Die Erweiterungsmenge ist *vollständig*, wenn alle gesuchten Ergänzungstoken dem kgV hierarchisch untergeordnet sind und die Abwärtstraversierung bei allen Ergänzungstoken endet.
   - 1.1. Alle gesuchten Ergänzungstoken sind genau dann dem kgV untergeordnet, wenn es keine Möglichkeit gibt, Ergänzungstoken außerhalb der kgV Spanne zu beeinflussen.
     - 1.1.1. Damit ein Ergänzungstoken erstellt/gelöscht/modifiziert werden kann, muss dessen Vater verändert werden.
     - 1.1.2. Der Vaterknoten kann nur verändert werden, wenn mindestens eines seiner Kindknoten strukturell verändert wird.
     - 1.1.3. Strukturell veränderte Kindknoten werden bereits von FindMaxDiffSets(...) abgefangen. Damit ist der veränderte Vaterknoten dem kgV untergeordnet.
   - 1.2. Die Abwärtstraversierung endet bei allen Ergänzungstoken, wenn zwischen dem kgV und jedem Ergänzungstoken ein Pfad führt, dessen Knoten ausschließlich mit Fehlerdiagnosen markiert sind.
     - 1.2.1. Ergänzungstoken sind Token, die vom Compiler erwartet werden und noch fehlen.
     - 1.2.2. Ihr Fehlen führt dazu, dass ihrem Vaterknoten ein Kind fehlt und dadurch eine Fehlerdiagnose Markierung erhält.
     - 1.2.3. Da der Vaterknoten eine Fehlerdiagnose erhält, muss auch dessen Vaterknoten eine Markierung erhalten.
     - 1.2.4. Daraus folgt, dass zwischen jedem Ergänzungstoken und dem kgV ein Fehlerdiagnosen-Pfad bestehen muss.
2. Die Erweiterungsmenge ist *minimal*, wenn lediglich diejenigen Token erkannt werden, die sich im *a priori*- und *a posteriori*-Baum unterscheiden.
   - 2.1. Zuvor wurde die Vollständigkeit von E_X und E_Y bewiesen.
   - 2.2. Für die Minimalität muss die Vollständigkeit des Schnitts der beiden Mengen gezeigt werden.
   - 2.3. Der Schnitt ist genau dann vollständig, wenn alle Token, die in E_X und E_Y identisch sind erkannt werden.
   - 2.4. Die Identifikation der unveränderten Token geschieht über den Tracing-Mechanismus aus Kapitel 4.1.2.
   - 2.5. Ist der Tracing-Mechanismus korrekt, so ist die Erweiterungsmenge auch minimal.

Q.E.D.

Die Ermittlung der Erweiterungsmenge ist demnach vollständig und unter Einschränkungen minimal. Hierbei ist die Minimalität jedoch lediglich von formaler Bedeutung, weil es semantisch eleganter ist, sich auf Token zu beschränken, die sich tatsächlich verändern. Wird

jedoch ein Token hinzugenommen, der sich nicht verändert, so hat dies keine Auswirkung auf das tatsächliche Implementierungsergebnis. In SITCOM gibt es keine Blattregeln, die fehlerhafterweise von unveränderten Tokens ausgelöst werden.
Abschließend lässt sich noch sagen, dass die Konkatenation von Startmenge und Erweiterungsmenge die gesamte Ausgangssituation abdecken. Alle möglichen Token lassen sich zwei Klassen zuordnen. Entweder sie haben eine Struktur oder sie haben keine. Erstere werden in der Startmenge berücksichtigt, letztere in der Erweiterungsmenge. Damit wurde die Korrektheit der Ausgangssituation gezeigt.

4.2.2 Korrektheit der Zielsituation

Die Zielsituation beschreibt den Weltzustand, bei dem eine weitere Regelanwendung weder sinnvoll noch möglich ist. In SITCOM ist dies genau dann erreicht, wenn alle Randknoten abgearbeitet und interpretiert worden sind. Der Beweis zur Vollständigkeit der Randknoten erfolgt über *Paraknoten*:

Definition 16. Ein *Paraknoten* ist ein Syntaxknoten, der von der Ausgangsituation nicht mit SITCOM-Regeln zu erreichen ist und sich trotzdem mit der Editierung verändert.

Hypothese. Es existieren keine Paraknoten.

Schlussfolgerung.

1. Ein Paraknoten kann nur existieren, wenn mindestens einer seiner Kindknoten verändert wird.
2. Sein Kindknoten kann wiederum nur verändert werden, wenn einer seiner Kindknoten verändert wird.
3. Auf diese Weise führt ein Pfad vom veränderten Knoten vom Paraknoten hinab zu einem veränderten Syntaxtoken. Jeder Knoten auf dem Pfad muss verändert worden sein.
4. Die Vollständigkeit der Startmenge hat jedoch gezeigt, dass es keine veränderten Token geben kann, die nicht in der Startmenge enthalten sind.
5. Daher führt mindestens ein Weg von veränderten Knoten von der Startmenge bis zum Paraknoten.
6. Damit ist der Paraknoten mit SITCOM-Regeln zu erreichen und ist somit kein Paraknoten.

Q.E.D.

Aus dem Beweis lässt sich ableiten, dass sich jeder veränderte Knoten von der Ausgangssituation aus erreichen lässt. Damit sind alle potentiellen Veränderungen überprüft worden, wenn alle Randknoten erreicht worden sind.

4.2.3 Korrektheit der Menge der möglichen Aktionen

Nachdem gezeigt worden ist, dass die Ausgangs- und die Zielsituation korrekt sind, fehlt als letzter Schritt die Überprüfung der Regelmenge auf Korrektheit. Kann gezeigt werden, dass für jedes mögliche Szenario eine passende Regel feuert, so ist die Korrektheit der Regelmenge und somit auch die Korrektheit des gesamten SITCOM-Verfahrens gezeigt.

(a) Korrektheit der Planoperatoren

Bevor die Korrektheit der Regelmenge gezeigt werden kann, muss bewiesen werden, dass die Operatoren, die durch die Regeln ausgelöst werden, korrekt sind. Hierzu müssen sie sowohl *vollständig* als auch *paarweise* disjunkt sein.

Die *Vollständigkeit* beschreibt, dass anhand der Operatoren jeder Syntaxbaum in jeden beliebigen anderen Syntaxbaum überführt werden kann. Sie garantiert, dass immer die Möglichkeit besteht einen *a priori*- in einen *a posteriori*-Baum zu überführen. Die Vollständigkeit kann durch die Teilmenge (*Add Leaf*, *Delete Leaf*, *Add Node*, *Delete Node*) gezeigt werden. Mit den Operatoren dieser Teilmenge lassen sich stets alle Syntaxelemente, die im *a priori*-Baum und nicht im *a posteriori*-Baum enthalten sind löschen und daraufhin alle Elemente, die im *a posteriori*-Baum und nicht im *a priori*-Baum enthalten sind, einfügen. Dieses Vorgehen würde zwar nicht immer den naheliegendsten Schritt beschreiben, stellt jedoch stets eine Möglichkeit dar.

Diese Eigenschaft der paarweisen Disjunktheit führt dazu, dass Lösungen eindeutig sind und dass für eine wiederkehrende Veränderung im Quellcode stets die gleichen Operationen erkannt werden. Die paarweise Disjunktheit ist genau dann gegeben, wenn unterschiedliche Operatoren stets unterschiedliche Auslöser haben. Werden die Definitionen aus Kapitel 3.3.3 und 3.3.4 betrachtet, so kann beobachtet werden, dass dies stets der Fall ist.

(b) Vollständigkeit der Blattregeln

Die gesamte SITCOM-Regelmenge besteht aus Blattregeln und Knotenregeln. In diesem Abschnitt wird die Korrektheit ersterer, im Folgeabschnitt die Korrektheit letzterer gezeigt. Eine Regelmenge ist genau dann korrekt, wenn für jede gegebene Situation der korrekte Operator erkannt wird. Im Blattregel-Kontext muss demnach gezeigt werden, dass für jede mögliche SITCOM-Ausgangssituation die jeweils korrekte Menge an Blatt-Operatoren erkannt wird.

Eine Ausgangssituation besteht Definition 9 folgend aus X, X_{Enhanc}, Y, Y_{Enhanc} und Editierung. Es ist zu zeigen, dass jede Konkatenation $X \cup X_{Enhanc}$ anhand der Blattregeln in $Y \cup Y_{Enhanc}$ überführt werden kann. Dieser Umstand wurde in Abbildung 26 dargestellt. Dort wird $X \cup X_{Enhanc}$ $Y \cup Y_{Enhanc}$ gegenübergestellt, wobei X in X_{Edit} und X $\neg_{Edit}$ aufgeteilt und Y in Y_{Edit} und Y $\neg_{Edit}$ aufgeteilt wurde. Es ist zu erkennen, dass für jeden Teilbereich der ersten Konkatenation passende Regeln vorhanden sind, um ihn in die Nachfolge-Konkatenation umzuwandeln.

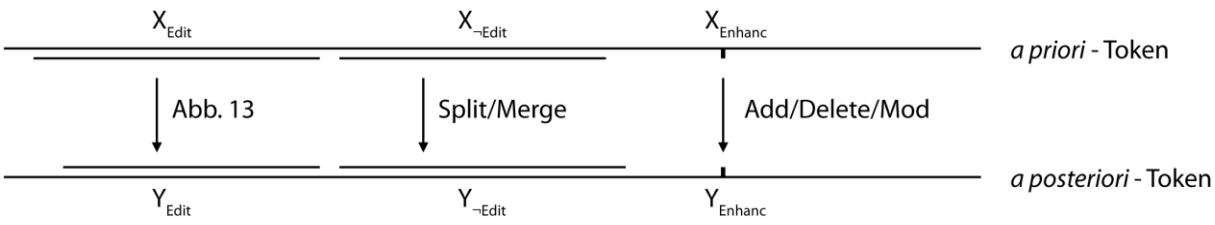


Abbildung 26 Gesamtheit der Tokenregeln

Wie auch der Abbildung 26 zu entnehmen ist, sind alle möglichen Tokenkonstellationen, welche die Texteditierung berühren, durch die Regeln in Abbildung 13 abgedeckt. Hinzu kommt die Menge aller potentiellen Interstagetoken, in der Abbildung mit X $\neg_{Edit}$ und Y $\neg_{Edit}$

bezeichnet. Abschließend gilt es die Menge der Ergänzungstoken X_{Enhanc} und Y_{Enhanc} zu betrachten. Die einzigen Blattregeln, die für diese Menge benötigt werden, sind drei Regeln, die jeweils eine Add Leaf-, eine Delete Leaf- und eine Modify Leaf-Operation auslösen.
Somit wurde aufgezeigt, dass für jede mögliche Ausgangssituation die passenden Blattregeln existieren, d.h. dass SITCOM vollständig korrektes Token-Tracing bietet.

(c) Vollständigkeit der Knotenregeln
Die letzte Hürde besteht darin, die Vollständigkeit und Korrektheit der Regeln zur Erkennung von Knotenoperatoren zu beweisen. Der hier anfallende Arbeitsaufwand ließ sich jedoch nicht im Rahmen der Masterarbeit bewältigen. Dies ist in der hohen Zahl an benötigten Regeln begründet. Ein erster Einblick hierfür wird in Abbildung 27 gegeben. Dort wird ein Spezialfall gezeigt, dessen Abdeckung bereits die Überprüfung von 147 Situationen erfordert. Ausgehend von diesem Fall, wird die tatsächlich benötigte Gesamtzahl an Knotenregeln abgeschätzt.

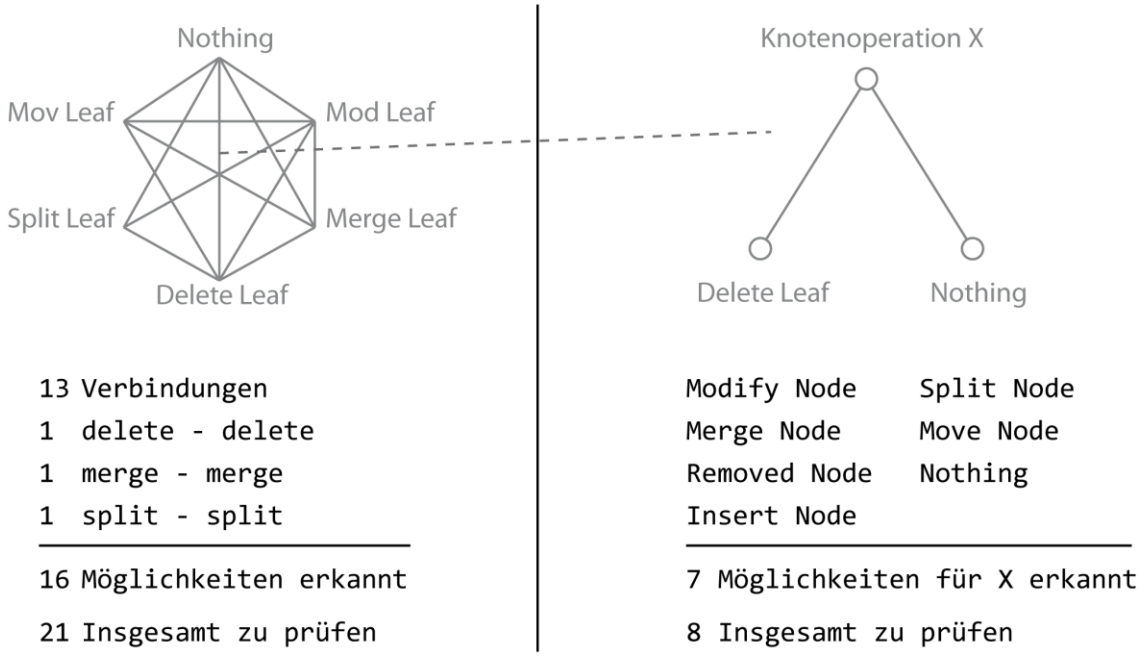


Abbildung 27 Mögliche Knotenoperationen für einen Knoten mit zwei Kindblättern

In der Abbildung wird eine Strategie aufgezeigt, mit der versucht wird, alle Knotenoperationen, die durch genau zwei Blattoperationen ausgelöst werden, herzuleiten. Es sollen nur Syntaxknoten betrachtet werden, die genau zwei Kindblätter besitzen.
Auf der linken Seite der Abbildung wurden alle möglichen Blattoperatoren notiert und im Kreis angeordnet. Eine Verbindungslinie zwischen zwei Operatornamen besagt, dass diese im Quellcode nebeneinander auftreten können. Ist beispielsweise ein *Delete Leaf* mit einem *Modify Leaf* verbunden, so heißt das, dass im Quellcode neben einem Token, der modifiziert worden ist, ein Token stehen kann, welcher gelöscht wurde. In der Abbildung sind alle Namen miteinander verbunden, bis auf die Paare *Mov Leaf - Split Leaf* und *Split Leaf - Merge Leaf*. Das heißt, dass in SITCOM für alle zweistelligen Blatt-Operator-Kombinationen mindestens eine darauf basierende Knotenoperation gefunden worden ist, mit Ausnahme der besagten Paare.

Jede Verbindungslinie sagt aus, dass die jeweilig markierte Kombination *mindestens* eine Knotenregel feuern kann. Im Falle einer Verbindung zwischen einem gleichbleibenden und einem gelöschten Token können in SITCOM sieben unterschiedliche Knotenregeln feuern (siehe Abbildung, rechts).

Der Abbildung lässt sich entnehmen, dass für den Spezialfall eines Knotens mit zwei Kindblättern insgesamt $[\binom{6}{2}$ (Anzahl der Verbindungslinien) + 6 (Reflexive Verbindungen)] * 8 (Anzahl der Knotenoperationen) $= [15 + 6] * 8 = 168$ Situationen für eine Regelerstellung in Frage kommen. Ein weiterer Spezialfall wäre ein Knoten, mit einem Kindblatt und einem Kindknoten, hier würden zusätzlich (6 * 8) / 2 (Anzahl der Verbindungen) * 8 (Anzahl der Knotenoperationen) = 192 zu berücksichtigende Situationen entstehen. Würden zusätzlich noch Knotenoperationen berücksichtigt werden, die von zwei Kindknoten-Operationen ausgelöst werden, so würden zusätzlich nochmals $[\binom{8}{2} + 8$] * 8 = 288 Konstellationen hinzukommen. In der Summe würde eine Anzahl von 648 zu berücksichtigten mögliche Regelauslöser entstehen. Um die Vollständigkeit zu beweisen, müsste jede dieser Situationen daraufhin überprüft werden, ob eine Regel entstehen kann, und falls nicht, ein Gegenbeweis dafür aufgestellt werden. Dies gilt alles unter der Berücksichtigung, dass *Add Leaf* und *Add Node* bereits aus den Betrachtungen ausgenommen worden sind, da sie wegen der Invertierung (siehe Kapitel 3.6.3) ausgelassen worden sind.
Eine wesentliche Optimierung lässt sich erzielen, wenn angenommen wird, dass zwei gleichartigen Blattoperationen nur genau eine Vaterknotenoperation zulassen. Beispielsweise muss der Vaterknoten von zwei gelöschten Blättern zwingend ein gelöschter Knoten sein, die anderen 6 möglichen Knotenoperationen in dem Fall können ausgeschlossen werden. Auf diese Weise lassen sich die reflexiven Verbindungskanten ausklammern und die zuvor 648 Prüfsituationen werden auf $[6+(15*8)] + [((6*8)/2)*8] + [8+(\binom{8}{2}*8)] = 550$ reduziert.
Die bisherigen Betrachtungen zur Anzahl möglicher Auslösekombinationen sind auf zwei Kindelemente, 6 Blattoperationen und 7 Knotenoperationen beschränkt. Werden diese statischen Werte mit den Variablen N (für Anzahl der Kindelemente), K_{Op} (für Anzahl der Knotenoperatoren) und B_{Op} (für Anzahl der Blattoperatoren) beschrieben, so lässt sich folgende Funktion aufstellen:

$$\text{Combinations}(N, B_{Op}, K_{Op}) = B_{Op} + K_{Op} + \left(\frac{(B_{Op} + K_{Op} + N - 1)!}{(B_{Op} + K_{Op})!\ N!} - B_{Op} - K_{Op} \right) K_{Op}$$

N Anzahl Kindknoten/-token
B_{Op} Anzahl Blattoperatoren
K_{Op} Anzahl Knotenoperatoren

Sie beruht auf der Formel der Anzahlsberechnung für Kombinationen mit Zurücklegen aus dem Bereich der Abzählenden Kombinatorik.
Basierend auf dieser Formel lassen sich Knoten mit beliebig vielen Kindern untersuchen, beispielsweise eine Methode mit beliebig vielen Ausdrücken. Damit die Anzahl der zu berücksichtigenden Kindknoten nicht gegen unendlich strebt, wird eine obere Schranke benötigt. Für dessen Bestimmung wird zunächst betrachtet, wie viele unterschiedliche Blattknoten-Operationen maximal nebeneinander stehen können, siehe Abbildung 28. Da diese Anzahl acht beträgt, und die Anzahl der unterschiedlichen nebeneinanderstehenden Operationen mit höherwerdender Knotenhierarchie nicht zunehmen, sondern nur abnehmen

kann, wird von einer oberen Schranke von maximal acht möglichen Kindern mit unterschiedlichen Operationen ausgegangen.

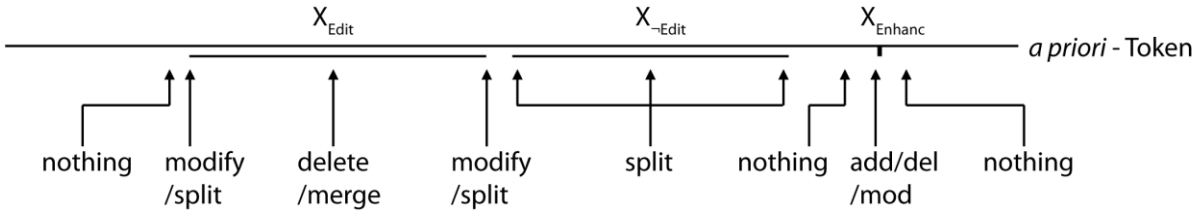


Abbildung 28 Maximale Anzahl an unterschiedlichen nebeneinanderstehenden Blattoperatoren

Anhand von *Combinations(N,B_{Op},K_{Op})* und der oberen Schranke von 8 lassen sich die folgenden drei Tabellen berechnen. In ihnen werden jeweils die unterschiedlichen Kombinationsmöglichkeiten, abhängig von der Kinderanzahl N, beschrieben. Die linke Spalte entspricht den SITCOM Eigenschaften mit 5 Blattoperationen und 8 Knotenoperationen. Die Summe aller Funktionswerte entspricht der oberen Schranke für die Anzahl benötigter Regeln in SITCOM. Es ist zu erkennen, dass die Summe von ≈1,6 Millionen Situationen zu groß ist, um für jeden Einzelfall zu prüfen, ob eine Knotenoperation ausgelöst werden kann.

N	Combinations (N,5,8)
1	13
2	637
3	3549
4	14469
5	49413
6	148421
7	403013
8	1007669

N	Combinations (N,5,6)
1	11
2	341
3	1661
4	5951
5	17963
6	47993
7	116633
8	262493

N	Combinations (N,4,5)
1	9
2	189
3	789
4	2439
5	6399
6	14979
7	32139
8	64314

Aus diesem Grund muss die Anzahl der möglichen Regelsituationen drastisch reduziert werden. Eine Möglichkeit wäre es zu zeigen, dass ein Syntaxknoten nicht mehr als beispielsweise vier Kindelemente mit unterschiedlichen Operatoren besitzen kann. So könnten die Fälle des unteren Tabellenabschnitts entfernt werden.
Eine weitere Möglichkeit bestünde darin, die Anzahl der Operatoren zu reduzieren. Würde beispielsweise die Knotenoperatoren *Remove Node* und *Insert Node* entfernt werden, so würde die mittlere der drei Tabellen entstehen. Die rechte der drei Tabellen entsteht, wenn zusätzlich noch *Move Leaf* und *Move Node* aus der Operatormenge gelöscht werden würde.
Zusammenfassend lässt sich resümieren, dass die Anzahl der zu überprüfenden Situationen zu groß ist, um für jede einzelne die Existenz einer Regel zu beweisen. Daher kann die Korrektheit der SITCOM-Regelmenge nicht gezeigt werden, womit auch das gesamte SITCOM Verfahren nicht vollständig nachgewiesen werden kann. Weitere Kürzungsmechanismen bergen das Potential, die Anzahl der zu kontrollierenden Situationen nochmals drastisch zu reduzieren, so dass eventuell doch die Korrektheit der Regelmenge gezeigt werden kann. In dieser Masterarbeit werden diese Aspekte jedoch nicht weiter verfolgt.

4.2.4 Korrektheit des gesamten SITCOM-Verfahrens

Um die Korrektheit des gesamten SITCOM-Verfahrens zu zeigen, müssen die Ausgangssituation, die Zielmenge und die Menge der Regeln auf Korrektheit bewiesen werden. Die ersten beiden sind korrekt, bei der letzten Menge ist die Teilmenge der Blattregeln korrekt. Für den Nachweis der Korrektheit der Knotenregeln muss die Anzahl der Situationen, die eine Regel erfordern, weiter stark reduziert werden. Ein weiterer Einfluss auf die Korrektheit des SITCOM-Verfahrens ist die Stabilität der Tracing-Funktion, welche in der Ausgangssituation und in einigen Regeln verwendet wird. Insgesamt konnten große Teilbereiche auf Korrektheit bewiesen werden. Für Ausnahmefälle, in denen SITCOM aufgrund von nicht vorhandenen Regeln nicht terminiert, wird eine Fehlerbehandlungsroutine gestartet. Auf diese wird im Folgeabschnitt eingegangen.

4.3 FEHLER-UNTERDRÜCKUNG

In Kapitel 4.2.3(b) wurde gezeigt, dass die Menge der Blattregeln in SITCOM vollständig ist, und für jede mögliche Texteditierung über einen Quelltext eine passende Regel existiert. Da eine korrekte Definition noch keine korrekte Implementierung ausmacht, wurden 102 automatisierte Unittests erstellt. Jeder Testfall simuliert eine kurze Texteingabe eines Benutzers auf dem Editor und prüft die resultierenden Blattoperationen.
Die Testfälle wurden Äquivalenzklassen zugeordnet und prüfen die vermutet fehleranfälligsten Eingabeszenarien. Am Ende der Entwicklungszeit liefen alle geschriebenen Tests fehlerfrei durch. In Abbildung 29 wird ein Ausschnitt des Testing Tools gezeigt, mit dem die Benutzereingaben auf Quellcode simuliert worden ist. Somit wurde eine stabile Grundlage für den Prozess der Knotenoperation-Erkennung geschaffen.

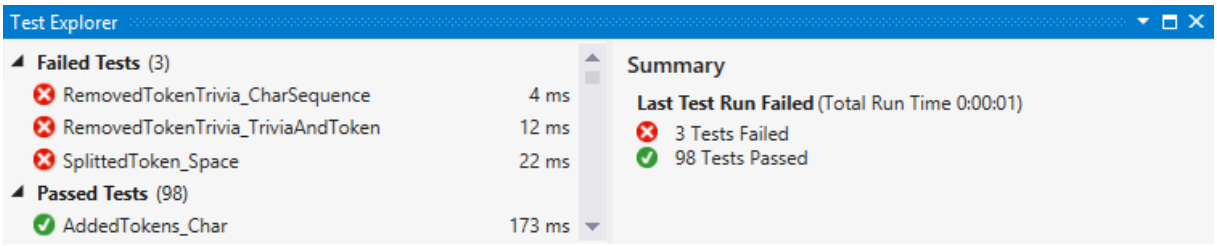


Abbildung 29 Das Testprogramm mit Testfällen zu Blattregeln

Für die Knotenerkennung wurden keine Testfälle mehr geschrieben. Dies ist auf die kurze Bearbeitungszeit der Masterarbeit zurückzuführen und darauf, dass neben der Erstellung der Testfälle ebenfalls eine Schnittstelle geschaffen werden muss, die vom Testprogramm aus Abbildung 29 genutzt werden kann.
Die Testfälle helfen der Fehlererkennung während des Entwicklungsprozesses. Trotzdem können noch Fehler im Endprodukt entstehen. Solche Fehler sind darauf zurückzuführen, dass für einige Ausnahmezustände Situationen entstehen können, für die keine passende SITCOM-Regel gefunden wird. In solchen Fällen wird eine Fehler-Unterdrückung-Routine gestartet. Tritt ein Deadlock auf, in dem keine Regel mehr gefeuert wird, und noch keine Kongruenz zwischen *a priori*- und *a posteriori*-Baum geschaffen worden ist, dann werden die noch nicht überführten Syntaxknoten der Fehler-Unterdrückung-Routine übergeben. Diese fügt so lange *Remove Node*- und *Insert Node*-Operationen auf beiden Bäumen aus, bis die Kongruenz

erreicht wird. Abschließend werden die nachträglich eingefügten Operationen in den SITCOM-Gesamtplan passend eingefügt.

4.4 LAUFZEIT

Üblicherweise werden bei algorithmischen Verfahren Laufzeiten in Landau-Notation angegeben. Für SITCOM wurde jedoch darauf verzichtet. Dadurch, dass in SITCOM stets nur ein kleiner Teilbereich des gesamten Syntaxbaums untersucht werden muss, und in der Implementierung diverse Best Practices beachtet worden sind, ist für den Benutzer keine spürbare Verzögerung des SITCOM-Programms wahrnehmbar.
Für die Weltzustände wurden passende Datenstrukturen verwendet. Die Regelanwendung ist derart gerichtet, so dass nur geeignete Regeln bei Problemstellungen abgefragt werden. Aufwändige Methoden, wie die Ermittlung eines Token-Nachfolgers, werden nur on-demand ausgelöst, usw.

Abbildung 30 Ausschnitt aus dem Performanzexplorer von Visual Studio

Die Summe dieser Einzeloptimierungen führt dazu, dass die Regelausführung, verglichen mit anderen rechenaufwändigen Aufgaben des SITCOM-Plugins, einen verschwindend geringen Teil der Prozessorressourcen einnimmt. In Abbildung 30 wird ein Abschnitt des Visual Studio Performance-Analyse-Tools gezeigt. Es ist zu sehen, dass das Rendern der Grafiken den Großteil der gesamten Rechenleistung einnimmt und die Logik-Schicht nicht in der Liste der fünf prozessorlastigsten Aufgaben aufgeführt ist. Somit benötigt die Ausführung des Baumvergleichs ~1% der Rechenleistung eines ohnehin nicht für den Benutzer spürbaren Prozesses.

Neben der in Abbildung 30 gezeigten Messung fanden neun weitere Messungen auf jeweils unterschiedlichen Codedateien statt. Alle Ergebnisse decken sich mit dem oben gezeigten. Hierbei spielt es auch keine Rolle, wie viele Codezeilen eine Datei besitzt.

5 Technische Umsetzung

Nachdem die theoretischen Konzepte des SITCOM-Verfahrens in Kapitel 3 und 4 vorgestellt worden sind, werden in diesem Kapitel die implementierungsrelevanten Themen beschrieben. Zunächst werden die verwendeten Entwicklungstools vorgestellt und kurz erörtert, warum sie sich besser für die gegebene Problemstellung eigenen als ihre Alternativen. Nachdem die passende Software ausgewählt worden ist, wird darauf aufbauend die entwickelte SITCOM-Softwarearchitektur vorgestellt. Abschließend folgt eine kurze Präsentation des fertigen Programms.

5.1 VERWENDETE TOOLS

Der Implementierungsanteil dieser Masterarbeit besteht aus einem Programm, welches die Eingabe von Quellcode überwacht und auswertet. In der heutigen Softwareentwicklung findet die Quellcode-Eingabe typischerweise in einem Editor statt, der Teil einer Entwicklungsumgebung ist. Innerhalb einer Entwicklungsumgebung sind mehrere unterschiedliche Werkzeuge in einer einheitlichen Oberfläche vereint. Daher ist es sinnvoll, den gegebenen Normen zu entsprechen, und das SITCOM-Programm möglichst nahtlos in die Entwicklungsumgebung zu integrieren.
Einige der verbreitesten Entwicklungsumgebungen, und damit Kandidaten für das Hosten des SITCOM-Tools, sind Eclipse [**60**], NetBeans [**61**], Visual Studio [**62**] und Xcode [**63**]. Jede dieser Werkzeugsammlungen besitzt unterschiedliche Stärken und Schwächen, wobei sich im Rahmen dieser Arbeit für Visual Studio entschieden worden ist. Gründe hierfür sind die Verbreitung, die Interoperabilität mit einer anderen Arbeit des Instituts, der mögliche Zugriff auf alle Visual Studio-Komponenten und das Compilerprojekt Roslyn [**64**].

5.1.1 Der Roslyn Compiler Service

Das von der Microsoft Corporation entwickelte Roslyn Projekt dient dem erleichterten Zugriff auf den .NET-Compiler und befindet sich derzeit in der offenen Beta Phase. Ein Compiler für Programmiersprachen läuft mehrere Phasen durch. In der ersten Phase wird der Quelltext anhand eines lexikalischen Scanners in Tokens eingeteilt. In der zweiten Phase wird aus den Tokens ein Syntaxbaum generiert. Spätere Phasen untersuchen globale Abhängigkeiten von Symbolen oder führen Optimierungsprozesse durch.
Die Besonderheit bei Roslyn besteht darin, dass auf die Ergebnisse jeder dieser Phasen zugegriffen werden kann. So kann zwischen den vom Tokenizer produzierten Tokens navigiert werden, Suchanfragen auf Syntaxbäumen getätigt werden oder die globalen Abhängigkeiten von Symbolen verfolgt werden.
Für die Ermittlung der Startmenge erweist sich die Navigation zwischen den Token als besonders hilfreich. Für die Generierung von Syntaxbäumen nach einer Texteditierung sind inkrementelle Parsefunktionen von großem Nutzen. So kann ein neuer Baum auf Basis des alten erstellt werden, so dass die performancelastige komplette Neugenerierung nach jedem Schritt umgangen werden kann.
Eine letzte Stärke des Roslyn Services ist es, dass mit der Öffnung des .NET-Compilers gleich zwei unterschiedliche Programmiersprachen, namentlich C# und Visual Studio.NET,

gleichzeitig unterstützt werden. So kann das entwickelte SITCOM-Tool von Benutzern unterschiedlicher Programmiersprachen genutzt werden und die Programmiersprachen-Unabhängigkeit des SITCOM-Ansatzes besser überprüft werden.

5.1.2 Verwendung von Paketen

Wie zuvor erwähnt, besteht eine integrierte Entwicklungsumgebung aus einer Vielzahl an unterschiedlichen Werkzeugen, die in ein einheitliches Gesamtprogramm eingebettet sind. In Visual Studio wird dieses Paradigma technisch umgesetzt, indem jedes Werkzeug in ein VSPackage eingebettet wird. Hierbei bezeichnet VSPackage ein Programmmodul, welches Methoden bereitstellt und Methoden anderer VSPackages verwenden kann. Abbildung 31 ist aus der offiziellen Visual Studio Dokumentation [**65**] entnommen und veranschaulicht diesen Umstand. Auf der linken Seite der Abbildung ist ein selbst zu erstellendes VSPackage abgebildet. Auf der rechten Seite der Abbildung ist das von Visual Studio vorimplementierte System an fertigen VSPackages, wie z.B. SVsUIShell für den Zugriff auf Toolfenster Funktionalität, abgebildet. Alle Pakete können sowohl Dienste anbieten (engl. proffer) als auch Dienste anderer VSPackages nutzen (engl. consume).

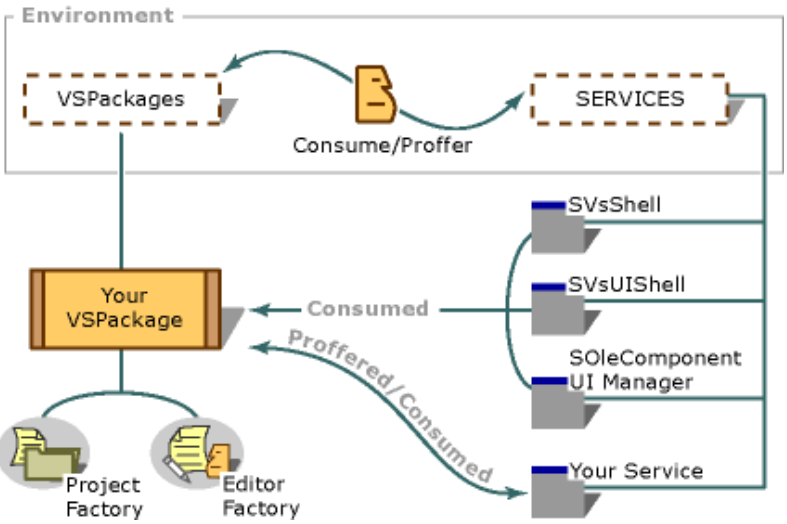


Abbildung 31 Die Funktionsweise von VSPackages [65]

Für die Entwicklung des SITCOM-Systems werden Dienste des Texteditor-Pakets genutzt, um die Veränderungen des Benutzers am Quellcode abzufangen. Weiterhin werden Fenster-Dienste für die Darstellung des Syntaxbaum-Vergleichs genutzt. Zudem kommen Services für eine vereinfachte Installation und eine vereinfachte Baumvergleich-Ergebnispropagierung zum Einsatz.
Das SITCOM-System besteht aus mehreren Tools, die jeweils in VSPackages eingebettet sind. Die kurze Vorstellung dieser Tools findet in Kapitel 5.2 statt.

5.1.3 GraphSharp

Zur Visualisierung des Abhängigkeitsgraphs wird die GraphSharp [**66**] Bibliothek verwendet. Durch sie können visuelle Steuerelemente sinnvoll automatisch angeordnet und durch Kanten verbunden werden. Für die Anordnung kann aus einem Spektrum an unterschiedlichen

Layoutalgorithmen ausgewählt werden. Von ihnen wurde sich für den *Efficient Sugiyama* [**67**] Algorithmus entschieden, da er den Abhängigkeitsgraph am besten darstellt.

5.2 GLOBALE ARCHITEKTUR

Wie zuvor erwähnt besteht Visual Studio aus einem Gerüst an unterschiedlichen Paketen, die jeweils Services anbieten und benutzen können. Im Rahmen von SITCOM wurden ebenfalls sieben Pakete entwickelt. Sie sind in Abbildung 32 aufgelistet.

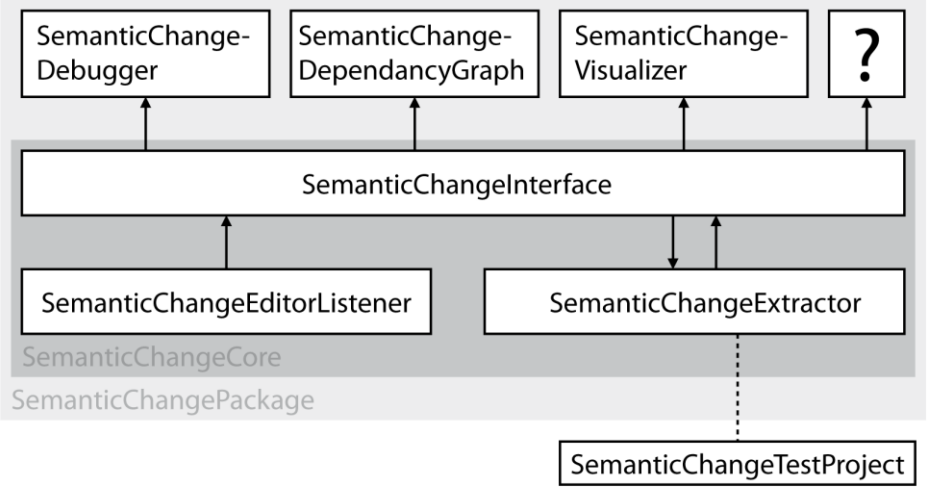


Abbildung 32 Die globale Paketarchitektur

5.2.1 SemanticChangeCore

Eines der Pakete ist das *SemanticChangeInterface*. Es dient der geordneten Kommunikation zwischen den Paketen. Zum einen dient es der internen Kommunikation zwischen Editor-*Listener* und *ChangeExtractor*, zum anderen öffnet es unter Nutzung von Methoden des gegebenen Paketmanager eine Schnittstelle des *SemanticChangeCores* nach außen. Die äußere Schnittstelle kann dann von Tools wie dem Debugger, dem Dependancy Graph, dem Visualizer oder beliebigen weiteren genutzt werden, um die Ergebnisse der SITCOM-Analyse zu erhalten (siehe Abbildung 32, oben).

Das *SemanticChangeEditorListener*-Paket ist Teil des *SemanticChangeCores* und hört alle Informationen ab, die mit der Quelltext-Modifikation des Texteditor-Fensters in Verbindung stehen. Hierbei handelt es sich um Informationen darüber, welche Quellcode-Klasse verändert wird, an welcher Stelle ein Text eingetragen oder gelöscht wird. Jede Textmanipulation, wie etwa "überschreiben", wird in Add- und Delete-Events aufgeteilt. Diese werden dann mit Information gefüllt und über die SemanticChangeInterface-Schnittstelle an den Semantic-ChangeExtractor versendet.

Bei dem *SemanticChangeExtractor* handelt es sich um das Paket, welches die SITCOM-Logik implementiert. Hier werden die Syntaxbäume modelliert, Detektionsregeln angewandt. Nach erfolgreichem Durchlauf werden die Analyseergebnisse an das Interface-Paket weitergeleitet, so dass jedes daran interessierte Paket mit neuen Ereignissen versorgt wird.

5.2.2 Anwendungen

Derzeit nutzen drei Anwendungen die SITCOM Ergebnisse. Dies sind *SemanticChange-Debugger*, *SemanticChangeDependancyGraph* und *SemanticChangeVisualizer*. In Abbildung 32 sind sie in der obersten Zeile zu finden.

Bei dem *SemanticChangeDebugger* handelt es sich um ein Fenster, mit welchem die wichtigsten Systemvariablen, wie z.B. erkannte Startmengen, ausgegeben werden. Dieses Fenster ist insbesondere während der Programmierung am SITCOM-SemanticChangeCore wichtig. Zum einen können auf diese Weise einige Bugs der sich im Beta-Stadium befindenden Software Roslyn umgangen werden, zum anderen lassen sich damit auch Wertebereiche grafisch ausgeben, so dass eine schnellere Aussage über die Korrektheit eines Interpretationsschritts getätigt werden kann.

Der *SemanticChangeDependancyGraph* ist eine Komponente, welche das Ergebnis eines Baumvergleichs visuell als Graph in einem Toolfenster darstellt. Er ist in Abbildung 35 des Folgekapitels vorgestellt. Der SemanticChangeVisualizer ähnelt dem SemanticChangeDependancyGraph in seiner Funktionsweise. Auch hier werden die Ergebnisse des SITCOM-Vergleichs ausgegeben, jedoch in schriftlicher Form.

In dieser Arbeit haben alle Anwendungen das Ziel, die Entwicklung des SITCOM Tools zu unterstützen. Jedoch soll das Tool nicht allein dem Selbstzweck dienen. Neben den vorgestellten drei Anwendungen, können noch beliebig viele weitere Anwendungen die SITCOM-Ergebnisse empfangen. In Abbildung 32 wurde die potentielle Existenz von weiteren Anwendungen mit einem Fragezeichen versehen. Einige Beispiele und Anregungen für solche Anwendungen werden im Motivationskapitel und im späteren Ausblickskapitel gegeben.

5.2.3 SemanticChangeTestProject

In Abbildung 32 sind alle Elemente im Paket SemanticChangePackage eingebettet mit Ausnahme des *SemanticChangeTestProjects*. Dies liegt daran, dass bei der Installationsroutine lediglich das SemanticChangePackage und alle seine Komponenten installiert werden sollen. Anwendungen wie der *SemanticChangeDependancyGraph* können sich als hilfreich für Benutzer erweisen und sind deshalb innerhalb des Packages platziert.

Die Testfälle, die in dem Testprojekt erfasst sind, sind jedoch allein für die SITCOM-Entwicklung relevant und werden daher auf dem Benutzerrechner nicht installiert. Das Testprojekt zeichnet sich durch zwei besondere Merkmale aus. Zum einen, kann es auf die inneren Objektmethoden des SITCOM-Extractors zugreifen ohne das *SemanticChangeInterface* zu benutzen. Zum anderen muss die Entwicklungsumgebung nicht gestartet werden, damit die Tests durchgeführt werden können.

Damit das Objektorientierungs-Prinzip der Kapselung eingehalten wird, darf der *Semantic-ChangeExtractor* nur die nötigsten Methoden für äußere Softwaremodule sichtbar machen. Da die Testmethoden jedoch auch Zugriff auf einige private Methoden des SemanticChange-Extractors benötigen, gibt es die .NET-Compiler Direktive *InternalsVisibleTo(KlassenName)*. Mit ihr können auch private Methoden für die Testklassen zugänglich gemacht werden, ohne dass gegen das Prinzip der Kapselung verstoßen wird.

Unter Zuhilfenahme dieser Direktive kann beim Testen auch erreicht werden, dass der *ChangeExtractor* ohne alle anderen Pakete gestartet wird. Für jeden Testfall simuliert das Testprojekt daraufhin Benutzereingaben und leitet sie an die entsprechenden Methoden des

Extractors weiter. Das entstandene SITCOM-Ergebnis wird abgefangen und mit dem Sollergebnis verglichen. Auf diese Weise muss für einen Testlauf keine Testinstanz der gesamten Visual Studio Entwicklungsumgebung gestartet werden. So benötigt ein Testlauf einen Bruchteil einer Sekunde statt 40-50 Sekunden auf einem handelsüblichen Computer.

5.3 LOKALE ARCHITEKTUR

Nachdem im vorangegangenen Kapitel die globale Paketarchitektur vorgestellt worden ist, wird sich in diesem Kapitel der Architektur der Hauptkomponente, dem SemanticChange-Extractor gewidmet. In ihm werden die Mechanismen zur Ausführung des SITCOM Baumvergleichs implementiert. Eine stark komprimierte Abstraktion des UML-Klassendiagramms wird in Abbildung 33 gezeigt.

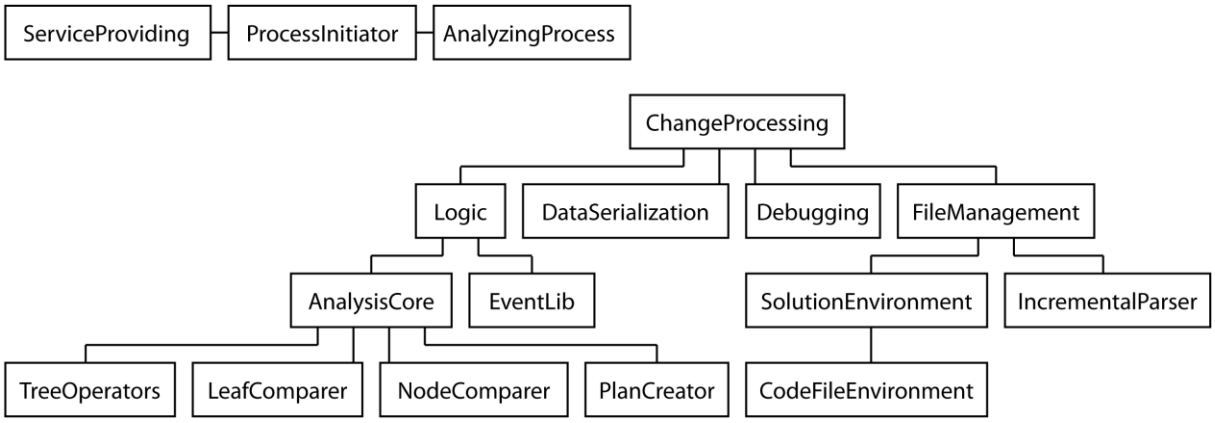


Abbildung 33 Die Architektur des SematicChangeExtractor-Pakets

5.3.1 Prinzip

Das Prinzip hinter der Architektur entspricht dem Prinzip des Singleton-Designpatterns [**68**]. Es wird ein global zugänglicher Referenzpunkt (das Singleton) gesetzt, auf den zugegriffen wird, um an gewünschte Funktionalität zu gelangen. In obiger Architektur entspricht dieser Referenzpunkt der *ChangeProcessing*-Klasse.

Ähnlich einer Mindmap, wird das Singleton als Wurzel eines Baums betrachtet, in dem die benötigte Funktionalität des Programms sinnvoll unterteilt wird. Die Wurzel beschreibt den gesamten Pool an Funktionen, welcher für die Erstellung eines SITCOM-Plans benötigt wird. Dieser Pool an Funktionen wird in die Abschnitte *Logic*, *DataSerialization Debugging* und *FileManagement* unterteilt. Diese vier Unterteilungen werden in noch kleinere funktionale Teilabschnitte aufgegliedert, usw.

5.3.2 Prozessinitiierung

Der Initialisierungvorgang des *SemanticChangeExtractors* geschieht durch die *ServiceProvi-ding*-Komponente. Sie implementiert alle Vorgaben, die erfüllt werden müssen, damit der *ChangeExtractor* als VSPackage genutzt werden kann. Weiterhin verknüpft sie die Schnittstellen des ProcessInitiators passend mit denen des *SemanticChangeInterface*. Der ProcessInitiator ähnelt einer "Schaltzentrale". Je nach eingehendem Event wird ein entsprechender Prozess gestartet, z.B. wird ein neuer Syntaxbaum geladen, wenn eine neue

Datei mit dem Editor betrachtet wird, oder ein Syntaxbaum neu erstellt, wenn eine Datei neu erstellt wird.
Eine wichtige Komponente, welche vom ProcessInitiator angesprochen wird, ist der *AnalyzingProcess*. Er führt alle Schritte aus, die für eine Planbildung erforderlich sind, und greift dabei auf die Funktionalitäten zu, die durch das Singleton angeboten werden. Ein Ausschnitt aus der AnalyzingProcess-Klasse wird in Abbildung 34 gezeigt.

```
Edit editCopy = edit.ToDeleteEdit();

// Step 1: Find least common parent node.
singleton.SolutionEnvironment.getFileByName(filename).IncrementalParser.retrieveParents(
    out parentNodeBefore, out parentNodeAfter);

// Step 1*: Insertions are seen as inverted deletions.
swap(ref parentNodeBefore, ref parentNodeAfter);

// Step 2: Handle leaf modification.
List<Leaf> X;
leafResult = singleton.AnalysisCore.LeafComparer.compareNodes(parentNodeBefore, parentNodeAfter,
    editCopy, out X);

// Step 3: Handle node modification.
List<Node> XEnhancement;
nodeResult = singleton.AnalysisCore.NodeComparer.compareNodes(parentNodeBefore, parentNodeAfter,
    leafResult, editCopy, out XEnhancement);

// Step 4: Combine leaf and node results.
plan = singleton.AnalysisCore.LeafNodeCombiner.BuildPlan(leafResult, nodeResult, X, XEnhancement);

// Step 4*: Reverse the plan.
plan = plan.Inverse();
```

Abbildung 34 Ausschnitt aus der Klasse AnalyzingProcess

Im Codeausschnitt sind alle Schritte abgebildet, die bei der Verarbeitung einer Insert-Text-editierung ausgeführt werden. Der Codeausschnitt ist die programmatische Umsetzung des SITCOM-Verfahrens, siehe Kapitel 3. Die im Quellcode mit *Step 1* und *Step 1** kommentierten Zeilen entsprechen der Ermittlung des kgV. Hinter den Quellcode-Abschnitt *Step 2*
verbirgt sich die Ermittlung der Ausgangssituation und die Anwendung der Blattregeln. *Step 3* wendet daraufhin die Knotenoperationen an. In *Step 4* werden die gefeuerten Regeln zu einem Gesamtplan zusammengefasst. Abschließend wird in *Step 4** der Plan invertiert, da es sich um die Verarbeitung einer Insert-Texteditierung handelt.

5.3.3 Zentrale Klassen

Aus dem Singleton in Abbildung 34 gehen eine Fülle von unterschiedlichen Klassen hervor. Auf die wichtigsten unter ihnen, namentlich Logic, DataSerialization und FileManagement wird im Folgenden eingegangen.

(a) Logic
Bei der *Logic*-Komponente handelt es sich um eine Komposition von Klassen, welche Funktionen anbieten, die zum Vergleich von Syntaxbäumen benötigt werden. Im *LeafComparer* sind die Regeln zur Detektion von Blattveränderungen implementiert. Das

Gegenstück dazu ist der *NodeComparer*, welcher gleiches auf Knotenebene durchführt. Der *PlanCreator* nutzt beide Regelwerke, und wendet sie auf eine Welt an, um einen Plan zu erhalten. Bevor dir Regeln jedoch angewandt werden können, müssen Operatoren, wie *Move Leaf* oder *Split Node*, definiert werden. Dies geschieht im Paket *TreeOperators*. Weitere Funktionalitäten, wie etwa die Nachfolgerermittlung von Knoten, sind direkt in der *AnalysisCore*-Klasse integriert, oder der Übersichtlichkeit wegen gekürzt.
Zusätzlich befindet sich in der *AnalysisCore*-Klasse eine Verknüpfung zu einer nicht aufgeführten *HigherDimActions*-Klasse. Hier ist die Extraktion semantischer höherwertiger Operationen vorgesehen. Beispielsweise könnten hier Kopier- oder Verschiebe-Vorgänge zwischen zwei Klassen erkannt werden.

(b) FileManagement
Da ein Software Projekt in der Regel aus mehr als nur einer Codedatei besteht, müssen mehrere Syntaxbäume parallel gehalten werden. Diese Aufgabe wird durch die *FileManagement*-Komponente erfüllt. Über den *IncrementalParser* wird ein Syntaxbaum geladen, modifiziert oder erstellt. Die Abspeicherung und Kontrolle der Syntaxbäume geschieht über die *SolutionEnvironment* und *CodeFileEnvironment* Klassen. Wird beispielsweise die Quellcode-Datei "MyClass.cs" gelöscht, so wird *CodeFileEnvironment.removeFile("MyClass.cs")* aufgerufen und der entsprechende Syntaxbaum aus dem Speicher genommen.

(c) DataSerialization
Zuvor wurde vereinfacht angenommen, dass für jede Quellcode-Datei genau ein Syntaxbaum existiert. Dies ist jedoch nur bedingt richtig. Die Architektur wurde derart verwirklicht, dass die Grundlagen für die Erkennung von höherwertigen Events bestehen. Wird z.B. zu einem Zeitpunkt eine Textfläche innerhalb einer Codedatei gelöscht, und zu einem späteren Zeitpunkt in derselben Datei wieder eingefügt, so besteht die Möglichkeit, dass ein *semantisch höherwertiges Verschieben* stattgefunden hat.
Damit ein solches Verschieben detektiert werden kann, muss der Zustand eines Syntaxbaums zu unterschiedlichen Zeitpunkten bekannt sein. Um den Zustand zu rekonstruieren, muss daher stets ein Syntaxbaum und mehrere Operationen zwischengespeichert werden. Wenn die Entwicklungsumgebung am Abend geschlossen und am nächst morgen wieder geöffnet wird, so soll auch auf die Syntaxbaumzustände des Vortags zugegriffen werden können.
Zu ausgewählten Zeitpunkten, wie etwa das Speichern oder Schließen einer Datei, werden daher der Syntaxbaum und eine Kette von zugehörigen Operationen auf der Festplatte angelegt, gelöscht, gespeichert oder geladen. Derartige Aktionen werden durch die *DataSerialization*-Komponente realisiert.

5.4 PROTOTYP

Nachdem der SITCOM-Algorithmus und dessen Architektur beschrieben worden sind, wird in diesem Abschnitt der implementierte Prototyp vorgestellt. Dieses Kapitel dient dazu, dass Look-and-Feel der Endanwendung zu vermitteln.
Hierzu wird in Abbildung 35 ein Screenshot vom fertigen Programm gezeigt. Es ist zu erkennen, dass der *SemanticChangeDependancyGraph* in einem Toolfenster platziert ist, welches direkt in die Entwicklungsumgebung eingebunden werden kann, siehe Abbildung 35

rechts. Für jede Eingabe im Quellcode-Editor (Abbildung, links) wird zeitgleich der Graph, der die erkannten semantischen Aktionen beschreibt, eingeblendet (Abbildung, rechts). Wenn die Maus über einen Knoten des angezeigten Graphs fährt, werden genauere Informationen eingeblendet. Der Graph ist dadurch entstanden, dass im Codebeispiel die Zeichenfolge *"i ="* eingefügt worden ist, in der Abbildung mit einem grünen Rahmen kenntlich gemacht.

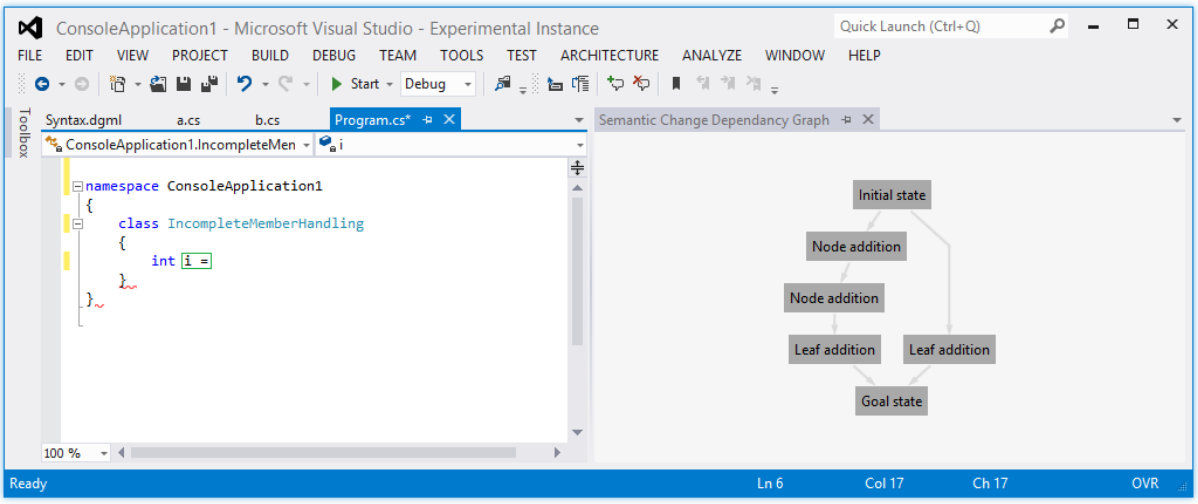


Abbildung 35 Der SemanticChangeDependancyGraph

Der *SemanticChangeDependancyGraph* empfängt die von SITCOM erkannten Veränderungen und stellt diese graphisch dar. Neben diesem reinen Visualisierungstool sollen jedoch auch weitere Tools angebunden werden können, welche von diesen semantischen Veränderungen profitieren. Wie diese Anbindung geschieht wird im Codebeispiel in Abbildung 36 gezeigt.

```
private void root_Loaded(object sender, RoutedEventArgs e)
{
    IChangeEventPropagation service = Package.GetGlobalService(typeof(SChangeEventPropagation)) as IChangeEventPropagation;
    EventRaiser changeEventRaiser = (EventRaiser)service.getEventRaiser();
    changeEventRaiser.SyntaxTreeChange += changeEventRaiser_SyntaxTreeChange;
}

void changeEventRaiser_SyntaxTreeChange(object sender, OperationGraphNode e)
{
    container.Graph = generateGraph(e);
}
```

Abbildung 36 Empfangen von Veränderungsoperationen

Es wird davon ausgegangen, dass das SITCOM-Tool in einem Fremdprojekt eingebunden werden soll. Der erste Schritt, der dafür notwendig ist, ist das Kopieren der SITCOM-Pakete an geeignete Pfade und das Referenzieren auf diese. Dies geschieht durch einen einfachen Doppelklick auf eine Installer-Datei. Daraufhin muss auf die referenzierten Pakete zugegriffen werden, wie dies geschehen kann, wird im obigen Codebeispiel gezeigt.
In der ersten Zeile des Codebeispiels wird anhand des .NET-Paketmanagers eine Referenz auf das benötigte SITCOM-Paket gesetzt und in der Variable `service` abgespeichert. Im nächsten Schritt wird auf den Teil des `services` zugegriffen, welcher die Veränderungsereignisse ausgibt. Dieser Teil wird in der Variablen `changeEventRaiser`

festgehalten. Dem `changeEventRaiser` können dann, wie in der Ereignisgesteuerten Programmierung üblich, Methodenzeiger übergeben werden. Im Codebeispiel zeigt der Methodenzeiger auf die Methode `changeEventRaiser_SyntaxTreeChanged(...)`. Immer wenn eine Syntaxbaum-Veränderung erkannt wird, wird diese Methode aufgerufen und ein `OperationGraphNode`-Objekt übergeben. Dieses übergebene Objekt wird im Codebeispiel mit `e` benannt. Der `OperationGraphNode` repräsentiert den Quellknoten des Graphs und enthält Verlinkungen zu allen Nachfolger-Knoten. Jeder Knoten enthält alle Informationen zu einer semantischen Veränderung.
Es wurde gezeigt, dass mit einem Doppelklick auf eine Installer-Datei und wenigen Zeilen Quellcode die SITCOM-Veränderungen komfortabel empfangen werden können. So können Drittprogramme unkompliziert von den SITCOM-Ereignissen profitieren. Einige Ideen für die Anwendungzwecke solcher Drittprogramme können im Motivationskapitel gefunden werden.

6 Ausblick

In Abschnitt 4.2.4 wurde gezeigt, dass das SITCOM-Verfahren korrekt und optimal wäre, wenn für jede mögliche Texteditierung auf einem Syntaxbaum passende Regeln gefunden werden würden. Da nur eine Teilmenge der insgesamt möglichen Regeln gefunden und implementiert worden ist, ist der derzeitig implementierte Prototyp demnach zwar korrekt, jedoch nicht immer optimal.
In Kapitel 4.2.3(c) wurde gezeigt, dass eine obere Schranke von ≈1,6 Millionen Situationen existiert, die daraufhin geprüft werden müssen, ob sie eine eigene Regel benötigen. Da die Überprüfung einer derart hohen Anzahl an unterschiedlichen Situation nicht sinnvoll ist, müssen Strategien gefunden werden, die Anzahl der Konstellationen zu reduzieren. Der nächste Schritt zur Verbesserung des SITCOM-Verfahrens ist die Ermittlung einer solchen Strategie. Mögliche Ansätze für eine solche Strategie ist die Klassifikation und das Vereinigen von unterschiedlichen Situationen oder das Auffinden und Löschen von nicht möglichen Situationen.
Neben der Verbesserung des SITCOM-Verfahrens ist auch seine Erweiterung möglich. Eine mögliche, semantisch-höherwertige Interpretationsmöglichkeit besteht in der Einführung eines *Advanced Move Node*-Operators. Er beschreibt das Verschieben eines Quellcode Abschnitts von einer Position an eine andere. Eine Verschiebung kann dabei als Folge aus den Baumoperatoren *Delete Node* (Ausschneiden) und ein *Insert Node* (wieder Einfügen) angesehen werden.
Nicht jede Folge aus *Delete Node und Insert Node* ist jedoch ein *Advanced Move Node*. Wird z.B. ein einzelner Buchstabe gelöscht und im späteren Verlauf der zufällig gleiche Buchstabe an einer anderen Position im Quellcode eingefügt, so ist die Wahrscheinlichkeit dafür, dass der Benutzer eine Verschiebung vorgesehen hat, sehr gering. Um zu entscheiden, ob es sich bei einer Aktionsfolge um ein Verschieben gehandelt hat, können zusammenhängende Faktoren überprüft werden. Mögliche helfende Faktoren sind Textgröße, zeitliche Nähe und Existenz im Zwischenspeicher.
Eine nochmals semantisch-höherwertige Interpretationsmöglichkeit, verglichen mit dem *Advanced Move Node*-Operator, ist ein *Transitive Move Node*-Operator. Die Namensgebung deutet dabei an, dass ein zusammengesetztes Verschieben stattfindet, welches aus mehreren Einzel-Verschiebungen zusammengesetzt ist. Wird ein Codeabschnitt von einer Stelle a an eine Stelle b verschoben, und anschließend von der Stelle b an eine Stelle c verschoben, so kann zusammengefasst werden, dass eine Verschiebung von a nach c stattgefunden hat.
Die Definitionen von weiteren, zusammenfassenden Operatoren können in Arbeiten von Robbes [**54**] [**69**] gefunden werden. Er schlägt z.B. vor, dass alle Änderungen, die in einer Methode stattfinden, in einem *Modify method*-Operator zusammengefasst werden können, oder dass das Erstellen von *AccessModifier*-Token, *ClassName*-Token und *ParameterList*-Node mit einem *Create Class*-Operator beschrieben werden kann.
Die vorgestellten höherwertigen Operationen sind der nächste Schritt, um die in der Motivation vorgestellten semantischen Tools zu verwirklichen. So können die *Modify method*- oder die *Create Class*-Operatoren z.B. dafür genutzt werden, Einflüsse auf ein UML-Klassendiagramm zu verfolgen oder für eine intelligente, automatisierte Zusammenfassung.

7 Fazit

Ein anerkanntes Mittel, um Aussagen über die Evolution von Software zu treffen, ist der Syntaxbaumvergleich. Bisherige Ansätze, die dieses Mittel nutzen, weisen Defizite auf. So können bisherige Baumvergleiche beispielsweise häufig nicht mit fehlerhaftem Quellcode oder sehr umfangreichem Quellcode umgehen. Eine Aufzählung der Defizite und Herausforderungen kann in Hassan und Holt [**34**] gefunden werden. Aus allen dort beschriebenen Anforderungen ist die in Abbildung 37 aufgeführte Vergleichsmatrix entstanden.
In dieser Masterarbeit wurde SITCOM, ein regelbasiertes Verfahren zur Identifikation von Baumoperationen, vorgestellt. Bei jeder Texteingabe des Programmierers zeichnet das SITCOM-Tool zeitgleich die zugehörigen Baumoperationen auf. So entsteht eine detailliertere Historie an Operationen, mit der genauere Aussagen über die Evolution eines Softwareartefakts getätigt werden kann, als es bei vergleichbaren Forschungsarbeiten bisher der Fall ist.
Stellt man SITCOM bereits vorhandenen Verfahren zur Änderungsextraktion gegenüber (siehe Abbildung 37) so ist zu erkennen, dass SITCOM noch weitere Stärken besitzt, die es von anderen Arbeiten hervorhebt. Dies sind insbesondere die Sprachunabhängigkeit und der Umgang mit fehlerhaften Syntaxbäumen.

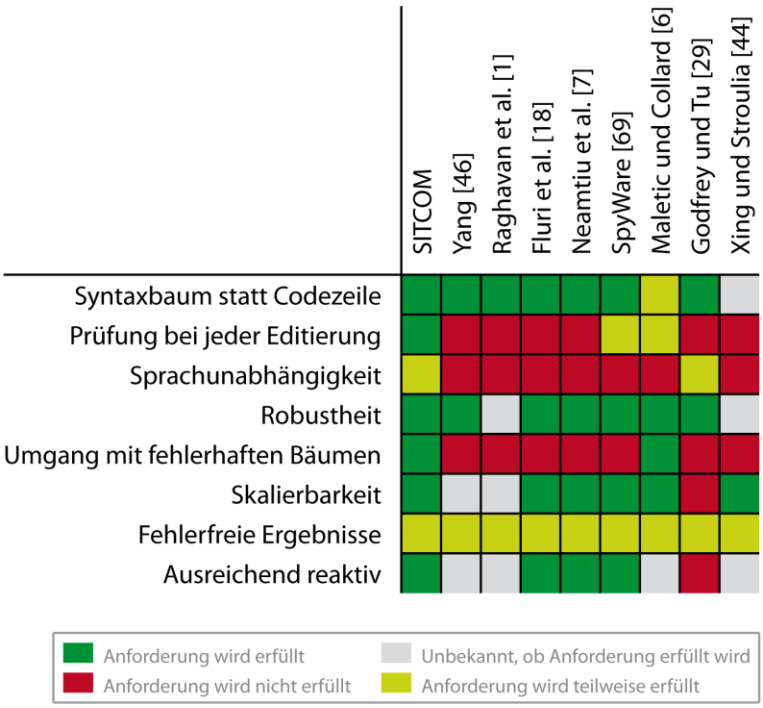


Abbildung 37 Verfahren zur Änderungsextraktion im Vergleich

In Kapitel 4.2 wurde die Korrektheit von SITCOM untersucht. Es wurde gezeigt, dass alle Veränderungen, die Syntax*token* betreffen, korrekt erkannt werden. Weiterhin wurde gezeigt, dass für alle Syntax*knoten* eine korrekte aber nicht zwingend optimale Interpretation gefunden wird. Die mangelnde Optimalität ist mit dem hohen Bedarf an Regeln zu begründen, der nicht

gedeckt werden konnte. Mögliche Lösungsansätze, um diesem Problem beizukommen, sind in Kapitel 6 gegeben.

Das SITCOM-Tool kann auf unterschiedliche Weisen genutzt werden. Eine Möglichkeit ist dessen Nutzung als Datenquelle für eigene Tools. Dabei ist der Hauptvorteil gegenüber alternativen Veränderungsextraktoren der, dass SITCOM auch die Veränderungsextraktion während der Codeeingabe erlaubt, siehe Abbildung 37. In Kapitel 1.1 ist eine Liste möglicher Tools gegeben.

Neben der Toolentwicklung kann das SITCOM-Tool auch direkt verwendet werden. Die Ergebnisse, die das Tool findet, werden in einer externen Datei abgespeichert. In ihr ist eine Historie an Baumveränderungen abgespeichert, die, verglichen mit anderen Datenquellen, detaillierter und aussagekräftiger ist. Solche Änderungshistorien sind eine wichtige Ressource zur Forschung über die Natur von Codeveränderungen, siehe Kapitel 1.1 .

Ein weiterer Beitrag, neben dem Tool selbst, ist die Aufdeckung von zwei Aspekten, die bisher in der Forschung zur Software-Evolution noch keine Beachtung gefunden haben. Beim ersten Aspekt handelt es sich um die Beachtung von Ordnungsrelationen zwischen Baumoperatoren, siehe Kapitel 3.4.3 und 3.6 . Der zweite Aspekt zeigt die Tatsache, dass Konstellationen existieren, in denen die Anwendung jeglicher Baumoperatoren dazu führt, dass gegen die Grammatik, anhand welcher der Baum aufgebaut ist, verstoßen wird, siehe Kapitel 4.1.1.

Abschließend lässt sich zusammenfassen, dass mit SITCOM ein Verfahren entwickelt worden ist, bei welchem es als bisher einzigem gelingt, die Veränderungen von Quelltext bei jeder Editierung zu verfolgen. Dabei ist es egal, ob der Syntaxbaum vollständig ist oder nicht. Dies eröffnet eine Fülle an neuen Möglichkeiten, sowohl für die Entwicklung von darauf aufbauenden Tools als auch für die Erforschung im Bereich der Software-Evolution.

Abbildungsverzeichnis[1]

[1] Alle Abbildungen wurden mit Adobe Illustrator erstellt oder von der Originalquelle kopiert um ein einheitliches Aussehen zu gewährleisten. Die farbigen Syntaxbäume sind mit dem Roslyn-Tool [**64**] erstellt worden.

Literaturverzeichnis

[1] S. Raghavan, R. Rohana, D. Leon, A. Podgurski, and V. Augustine, "Dex: A Semantic-Graph Differencing Tool for Studying Changes," Case Western Reserve University, 2004.

[2] D. Jackson and D. A. Ladd, "Semantic Diff: A Tool for Summarizing the Effects of Modifications," Carnegie Mellon University, 1994.

[3] S. Horwitz, "Identifying the Semantic and Textual Differences," in *Proceedings of the ACM SIGNPLAN'90*, New York, 1990, pp. 234-245.

[4] J. Krinke, "Identifying Similar Code with Program Dependence Graphs," in *Eigth Working Conference on Reverse Engineering*, Stuttgart, Germany, 2001.

[5] G. Antoniol, B. Caprile, A. Potrich, and P. Tonella, "Design-Code Traceability for Object Oriented Systems," in *Annals of Software Engineering 9* , Trier, 2000, pp. 35-58.

[6] J. I. Maletic and M. L. Collard, "Supporting Source Code Difference Analysis," in *20th IEEE International Conference on Software Maintenance*, Chicago, USA, 2004, pp. 210-219.

[7] I. Neamtiu, J. S. Foster, and M. Hicks, "Understanding Source Code Evolution Using Abstract Syntax Tree Matching," in *MSR'05*, Missouri, USA, 2005, pp. 1-5.

[8] D. Harel and B. Rumpe, "Modeling Languages: Syntax, Semantics and All That Stuff," Aug. 2000.

[9] D. B. Roberts, "Practical Analysis for Refactoring," University of Illinois Doktorarbeit, 1999.

[10] E. Merlo, K. Kontogiannis, and J. F. Girard, "Structural and Behavioral Code Representation for Program Understanding," Centre de Recherche Informatique de Montréal, 1992.

[11] S. Haefliger, G. von Krogh, and S. Spaeth, "Code Reuse in Open Source Software," *Management Science*, pp. 180-193, Jan. 2008.

[12] R. Prieto-Diaz, "Status report: software reusability," *Software, IEEE*, vol. 10, no. 3, pp. 61-66, 1993.

[13] P. Weißgerber and S. Diehl, "Identifying Refactorings from Source-Code Changes," *Proceedings of 21st IEEE/ACM International Conference on Automated Software Engineering*, 2006.

[14] H. Krahn and B. Rumpe, "Towards Enabling Architectural Refactorings through Source Code Annotations," *Proc. der Modellierung*, 2006.

[15] M. Hammad, M. L. Collard, and J. I. Maletic, "Automatically Identifying Changes that Impact Code-to-Design Traceability," *ICPC'09 Submission*, 2009.

[16] A. Delater and B. Paech, "Traceability between System Model, Project Model and Source Code".

[17] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller, "Mining Version Histories to

Guide Software Changes," *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, vol. 31, pp. 429-445, Jun. 2005.

[18] B. Fluri, M. Würsch, M. Pinzger, and H. C. Gall, "Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction," *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, vol. 33, pp. 725-744, Nov. 2007.

[19] H. Kagdi, "Improving Change Prediction with Fine-Grained Source Code Mining," *ASE' 07*, Nov. 2007.

[20] A. T. T. Ying, G. C. Murphy, M. C. Chu-Carroll, and R. Ng, "Predicting Source Code Changes by Mining Change History," *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, vol. 30, no. 9, pp. 574-586, Sep. 2004.

[21] B. Westfechel, "Structure-Oriented Merging of Revisions of Software Documents," *Proc. Third Int'l conf. Software Configuration Management*, 1991.

[22] G. L. Thione and D. E. Perry, "Parallel Changes: Detecting Semantic Interferences," *Computer Software and Applications Conference, 2005. COMPSAC 2005. 29th Annual International* , vol. 2, no. 1, pp. 47-56, Jul. 2005.

[23] D. Binkley, S. Horwitz, and T. Reps, "Program Integration for Languages with Procedure Calls," *ACM Transactions on Software Engineering and Methodology*, vol. 4, no. 1, pp. 3-35, 1995.

[24] CollabNet. (2014, Apr.) Apache SubVersion. [Online]. http://subversion.apache.org/

[25] The Open Group. (2014, Apr.) The Open Group Base Specifications Issue 6. [Online]. http://pubs.opengroup.org/onlinepubs/009695399/utilities/diff.html

[26] The WinMerge Development Team. (2014, Apr.) WinMerge. [Online]. http://winmerge.org/

[27] D. Binkley, "Using semantic differencing to reduce the cost of regression testing," *International Conference on Software Maintenance*, pp. 41-52, 1992.

[28] F. I. Vokolos and P. G. Frankl, "Pythia: a regression test selection tool based on textual differencing," *3rd International Conference on Reliability, Quality, and Safety of Software Intensive Systems*, p. 1997.

[29] Q. Tu and M. W. Godfrey, "An integrated approach for studying architectural evolution," in *10th International Workshop on Program Comprehension (IWPC'02)*, IEEE Computer Society Press, 2002, pp. 127-136.

[30] T. Zimmermann, S. Diehl, and A. Zeller, "How History Justifies System Architecture (or not)," in *IEEE International Workshop on Principles of Software Evolution (IWPSE03)*, Helsinki, Finland, 2003.

[31] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers*, 2nd ed., P. I. Edition, Ed. 2007.

[32] J. Hein, T. Jiang, L. Wang, and K. Zhang, "On the complexity of comparing evolutionary trees," *Discrete Applied Mathematics*, pp. 153-169, 1996.

[33] M. Farach and M. Thorup, "Fast Comparison of Evolutionary Trees," Oxford University, Oxford, England, DIMACS Technical Report, 1993.

[34] A. E. Hassan and R. C. Holt, "Studying the Evolution of Software Systems Using

Evolutionary Code Extractors," University of Waterloo, 2004.

[35] F. E. Allen, "Control Flow Analysis," *SIGPLAN Notices*, Jul. 1970.

[36] C. Kecher, *UML 2.0 – Das umfassende Handbuch*. Galileo Computing, 2006.

[37] S. Klusener and R. Lämmel, "Deriving tolerant grammars from a base-line grammar," in *IEEE International Conference Software Maintenance*, Amsterdam, 2003.

[38] M. N. Armstrong and C. Trudeau, "Evaluating architectural extractors," in *Working Conference on Reverse Engineering (WCRE98)*, Honolulu, 1998, pp. 30-39.

[39] G. C. Murphy, D. Notkin, W. G. Griswold, and E. S. Lan, "An empirical study of static call graph extractors," in *ACM Transactions on Software*, New York, USA, 1998, pp. 158-191.

[40] C. Kaner and W. P. Bond, "Software Engineering Metrics: What Do They Measure and How Do We Know?," *METRICS 2004*, pp. 1-12, 2004.

[41] K. Kontogiannis, "Evaluation experiments on the detection," in *Working Conference on Reverse Engineering (WCRE'97)*, Amsterdam, Niederlande, 1997.

[42] H. Gall, K. Hajek, and M. Jazayeri, "Detection of logical coupling based on product release history," in *IEEE International Conference on Software Maintenance (ICSM98)*, Washington D.C., 1998.

[43] H. Gall, M. Jazayeri, and J. Krajewski, "CVS Release History Data for Detecting Logical Couplings," in *IEEE International Workshop on Principles of Software Evolution (IWPSE03)*, Helsinki, Finland, 2003.

[44] Z. Xing and E. Stroulia, "Analyzing the Evolutionary History of the Logical Design of Object-Oriented Software," *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, vol. 31, pp. 850-868, Oct. 2005.

[45] P. Bille, "A Survey on Tree Edit Distance and Related Problems," *Theoretical Computer Science*, vol. 337, no. 1-3, pp. 217-239, Jun. 2005.

[46] W. Yang, "Identifying Syntactic Differences Between Two Programs," *SOFTWARE-PRACTICE AND EXPERIENCE*, vol. 21, no. 2, pp. 739-755, Jul. 1991.

[47] E. Myers, "An O(ND) Difference Algorithm and Its Variations," *Algorithmica*, vol. 1, no. 2, pp. 251-266, 1986.

[48] D. S. Hirschberg, "Algorithms for the Longest Common Subsequence Problem," *Journal of the Association for Computing Machingery*, vol. 24, no. 4, pp. 664-675, Oct. 1977.

[49] Software Architecture Group. (2014, Apr.) CPPX. [Online]. http://www.swag.uwaterloo.ca/cppx/

[50] G. W. Adamson and J. Boreham, "The Use of an Association Measure Based on Character Structure to Identify Semantically Related Pairs of Words and Document Titles," *Information Storage and Retrieval*, vol. 10, pp. 253-260, Aug. 1974.

[51] J. I. Maletic, M. L. Collard, and M. Andrian, "Source Code Files as Structured Documents," *Proceedings of the 10th IEEE International Workshop on Program Comprehension (IWPC'02)*, p. 4, 2002.

[52] C. M. Hoffmann and M. J. O'Donnell, "Pattern Matching in Trees," *Journal of the ACM*, vol. 29, no. 1, pp. 68-95, Jan. 1982.

[53] S. M. Selkow, "The tree-to-tree editing problem," in *Information processing letters*, Knoxville, TN, USA, 1977, pp. 184-187.

[54] R. Robbes, "Of Change and Software," Faculity of Informatics Doktorarbeit, December 2008.

[55] R. Brooks, "A robust layered control system for a mobile robot," in *IEEE Journal of Robotics and Automation*, 1986, pp. 14-23.

[56] R. Brooks, "Asynchronous distributed control system for a mobile robot," in *SPIE Conference on Mobile Robots*, 1986, pp. 77-84.

[57] D. Weld, "An introduction to least commitment planning," *AI magazine*, 1994.

[58] S. Russell and P. Norvig, "Artificial Intelligence: A Modern Approach," 2011.

[59] R. Fikes and N. Nilsson, "STRIPS: a new approach to the application of theorem proving to problem solving," *Artificial Intelligence*, vol. 2, pp. 189-208, 1971.

[60] The Eclipse Foundation. (2014, Apr.) eclipse. [Online]. https://www.eclipse.org

[61] Oracle Corporation. (2014, Apr.) NetBeans IDE. [Online]. https://netbeans.org/

[62] Microsoft. (2014, Apr.) Visual Studio. [Online]. http://www.visualstudio.com/

[63] Apple Inc. (2014, Apr.) Xcode. [Online]. https://developer.apple.com/xcode/

[64] Microsoft. (2014, Apr.) .NET Compiler Platform ("Roslyn"). [Online]. http://msdn.microsoft.com/en-us/vstudio/roslyn.aspx

[65] Microsoft. (2014, Apr.) Microsoft Developer Network. [Online]. http://msdn.microsoft.com/en-us/library/bb166424.aspx

[66] CodePlex. (2014, Apr.) Graph#. [Online]. http://graphsharp.codeplex.com/

[67] M. Eiglsperger, M. Siebenhaller, and M. Kaufmann, "An Efficient Implementation of Sugiyama's Algorithm for Layered Graph Drawing," *JOURNAL OF GRAPH ALGORITHMS AND APPLICATIONS*, 2005.

[68] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Entwurfsmuster. Elemente wiederverwendbarer objektorientierter Software*. München: Addison-Wesley, 2004.

[69] R. Robbes, "Mining a Change-Based Software Repository," *Fourth International Workshop on Mining Software Repositories (MSR'07)*, 2007.

[70] M. M. Lehman, J. F. Ramil, P. D. Wernick, D. E. Perry, and W. M. Turski, "Metrics and laws of software evolution the nineties view," in *Fourth International Software Metrics Symposium (Metrics97)*, Albuquerque, NM, 1997.

[71] M. W. Godfrey and Q. Tu, "Evolution in open source software: A case study," in *IEEE International Conference on Software Maintenance (ICSM 2000)*, San Jose, California, 2000, pp. 131-142.

[72] D. Draheim and L. Pekacki, "Process-Centric Analytical Processing of Version Control Data," in *IEEE International Workshop on Principles of Software Evolution (IWPSE03)*, Helsinki, Finland, 2003.

[73] T. L. Graves and A. Mockus, "Inferring Change Effort from Configuration Management Databases," *Software Metrics Symposium, 1998. Metrics 1998. Proceedings. Fifth International* , pp. 267-273, Nov. 1998.

Anhang

Anhang 1. CD-Inhalt

- Digitale Version der vorliegenden Arbeit
- Projektordner mit den Quellcode-Dateien zum SITCOM-Tool
- SITCOM-Installerdatei

Eidesstattliche Erklärung

Hiermit erkläre ich, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.
Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Ulm, den 01.05.2014

Sebastian Mechelke