Kontrollflusssensible Quelltext-Umgestaltung mithilfe von Constraint-Programmierung

Marcus Mews

Berlin 2016

Kontrollflusssensible Quelltext-Umgestaltung mithilfe von Constraint-Programmierung

vorgelegt von
Dipl.-Ing.
Marcus Mews
geb. in Berlin

von der Fakultät IV – Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades

Doktor der Ingenieurswissenschaften
– Dr.-Ing. –

genehmigte Dissertation

Promotionsausschuss:

Vorsitzender:	Prof. Dr.-Ing. Stefan Tai
Gutachter:	Prof. Dr.-Ing. Stefan Jähnichen
Gutachter:	Prof. Dr. rer. nat. Bernd Fischer
Gutachter:	Prof. Dr. rer. nat. Peter Pepper

Tag der wissenschaftlichen Aussprache: 6. Juni 2016

Berlin 2016

Kurzfassung

Das Umgestalten von Quelltext ist ein essentieller Bestandteil beim Entwickeln, Erweitern und Warten von Programmen. Wird Quelltext umgestaltet, ändern sich lediglich seine Struktur oder Bezeichner, während seine Funktion unverändert bleibt. Das Umgestalten geht jedoch Hand in Hand mit dem Einführen von Funktionsänderungen. Denn häufig können Funktionsänderungen nur mit vorangegangener oder anschließender Umgestaltung durchgeführt werden, um Quelltexteigenschaften wie Lesbarkeit und Verständlichkeit nicht zu beeinträchtigen. Diese Eigenschaften sind im Kontext von langjährig eingesetzten Programmen mit häufigem Änderungs- und Anpassungsbedarf von großer Bedeutung.

Beim Umgestalten von Programmen nutzen Entwickler automatische Funktionen der Entwicklungsumgebungen und vertrauen darauf, dass diese Funktionen verlässlich und fehlerfrei arbeiten. Ebenfalls sind diese Funktionen dann optimal einsetzbar, wenn sie flexibel auch in unübersichtlichen oder schwierigen Situationen angewendet werden können. Leider treffen beide Eigenschaften auf etablierte Entwicklungsumgebungen wie Eclipse nur eingeschränkt zu. Sie arbeiten nicht immer fehlerfrei und verweigern unter bestimmten Umständen die Quelltextumgestaltung.

Ziel dieser Arbeit ist, einen Ansatz zum Durchführen kontrollflusssensibler Java-Umgestaltungen vorzustellen, damit diese fehlerfrei, flexibel und praxistauglich eingesetzt werden können. Hierzu werden fünf kontrollflusssensiblen Umgestaltungen vorgestellt, die von einer gemeinsamen Betrachtung profitieren, da die durch sie durchgeführten Änderungen in wechselseitiger Beziehung stehen. Die fünf Umgestaltungen Einfügen oder Extrahieren lokaler Variablen, Verschieben/ Vertauschen von Ausdrücken oder Anweisungen sowie das Verschieben von Argumenten werden in der Arbeit als Constraint-Problem interpretiert. Dazu wird das relevante Teilprogramm als Gleichungssystem abstrahiert und durch einen Constraint-Löser gelöst. Aus der Lösung wird eine Menge von Transformationen abgeleitet, die einerseits den Umgestaltungswunsch erfüllen und andererseits das Programmverhalten nicht ändern.

Die Arbeit zeigt, dass der untersuchte Ansatz Umgestaltungen sowohl verhaltenserhaltend, flexibel als auch schnell genug durchführen kann. Im Rahmen zweier Untersuchungen wird der Ansatz zunächst anhand von Beispielen mit drei etablierten Entwicklungsumgebungen verglichen. Dabei kann gezeigt werden, dass der Ansatz in jedem Fall korrekt und flexibel arbeitet. In einer zweiten Untersuchung wird ermittelt, wie sich der Ansatz im Vergleich zu Eclipse an beliebigen Beispielen aus einer Reihe von quelloffenen Projekten verhält. Die Korrektheit wird dabei mithilfe einer Test-Suite festgestellt. Die Ergebnisse zeigen, dass der Ansatz im Vergleich zu Eclipse das Programmverhalten wesentlich seltener ändert und auch in schwierigen Situationen Umgestaltungen berechnen kann. Darüber hinaus wird gezeigt, dass in 90% aller Fälle Umgestaltungen in weniger als fünf Sekunden ermittelt werden können.

Abstract

The refactoring of source code is an essential part in the development, maintenance and extension of software. When refactoring source code, only its structure and symbol names may change but the original behavior of the program must remain the same. Nevertheless, refactoring is often accompanied by functional changes that change the program behavior during the development. This is due to the necessary to improve source code properties like readability, comprehensibility and maintainability before, during or after applying functional changes to the code. These source code properties are of great importance in long-term used software.

When refactoring source code, developers use automatic built-in functions of integrated development environments and rely on their reliability and correctness. Also, the refactoring functions are optimal in terms of their benefit for the programmer, when they are flexible enough to give helpful result even in difficult and confusing source code situations. Unfortunately, both of these properties apply to established development environments like Eclipse only partially. They do not always work correctly and often refuse to return helpful results in difficult situations.

The goal of this thesis is to present an approach to improve control flow sensitive refactorings in terms of correctness, benefits for the programmer and practicability. The approach is presented using five refactoring instances that benefit from being addressed together, since the changes they cause have a mutual impact on each other. In this thesis, the five refactoring instances, inline or extract of variables, move or switch expressions or statements, or move method arguments are mapped to a constraint problem. For that purpose, the relevant source code is abstracted to a system of equations and variables, which is solved using a common constraint solver. By comparing the initial variable values to the values of the solution, a set of transformations is retrieved. These transformations fulfill the refactoring intent of the developer and also keep the program behavior unchanged.

This work shows that the examined approach is able to improve the benefits of the refactoring instances while keeping the program behavior unchanged in a tolerable computation time. The approach is evaluated in two manners. First, the results of the approach are compared to the refactoring results of three other established development environments using 15 handpicked examples. This evaluation shows that the approach computes helpful and correct results in every case. Second, the approach is massively tested and compared to Eclipse using several open source projects. The correctness is determined using the test suites that come along with the open source projects. The results show that the approach alters the program behavior only very rarely compared to Eclipse, and the approach also computes helpful refactoring instances in difficult situations. Moreover it is shown that it computes a refactoring solution in 90% of all cases in less than five seconds.

Danksagung

Mein herzlicher Dank geht an meinen Bruder Maximilian und meine Eltern Michael, Ines und Ellen sowie an meine Schwester Marlene und an meine Freunde. Sie haben mich in der Zeit meiner Promotion stets unterstützt und mir Kraft und Stärke gegeben. Besonders danke ich meinem Bruder, der mich inspiriert hat, meine Träume nicht aufzugeben.

Ich danke meinem Doktorvater Stefan Jähnichen für seine Unterstützung und für die Möglichkeit, in seinem Fachgebiet arbeiten zu dürfen. Ich möchte Herrn Friedrich Steimann und Jens von Pilgrim für ihre hilfreichen fachlichen Hinweise danken, die mir den richtigen Weg aufgezeigt haben. Herrn Alexander Raake danke ich für die gemeinsame Arbeit in der Lehre, die stets von gegenseitiger Wertschätzung geprägt war. Ebenfalls danke ich all meinen, auch ehemaligen Kollegen am Fachgebiet. Mein besonderer Dank gilt meinem langjährigen Bürokollegen, Zellinsassen, Forschungsroboter und gutem Freund Kristian Duske.

Mein Dank geht auch an Marek Kalbus. Die Musik und sein Unterricht haben mir immer geholfen, mancherlei Sorge mit Abstand und Gelassenheit zu begegnen. Darüberhinaus bin ich für die Unterrichtsstunden als unverzichtbare Abwechslung und für seinen Rat als Außenstehender überaus dankbar.

Außerdem möchte ich allen Studenten, die bei mir eine Abschlussarbeit geschrieben haben, darunter besonders Yannick Heiber und Andreas Mentz, danken. Durch ihre Arbeit in meinem Themenfeld bekam meine Arbeit oft neue Impulse und konnte merklich profitieren.

Ohne diese Menschen wäre diese Arbeit nie entstanden – Danke.

Inhaltsverzeichnis

1 Einführung **1**
- 1.1 Problembeschreibung . . . 3
- 1.2 Herausforderungen . . . 7
- 1.3 Lösungsansatz . . . 9
  - 1.3.1 Vorgehensweise . . . 10
  - 1.3.2 Erhalt der Semantik . . . 11
  - 1.3.3 Umsetzung des Wunsches . . . 12
  - 1.3.4 Freiheitsgrade der Lösungen . . . 12
- 1.4 Unterstützte Umgestaltungen . . . 13
  - 1.4.1 Ausdrücke extrahieren/einfügen . . . 13
  - 1.4.2 Operanden verschieben/neuordnen . . . 14
  - 1.4.3 Parameter/Argumente neuordnen . . . 15
  - 1.4.4 Anweisungen verschieben . . . 16
- 1.5 Beitrag der Arbeit . . . 16
- 1.6 Aufbau der Arbeit . . . 17

2 Thematischer Hintergrund **19**
- 2.1 Quelltextumgestaltung . . . 19
- 2.2 Verwandte Arbeiten . . . 23
  - 2.2.1 Griswolds Arbeiten . . . 24
  - 2.2.2 Schäfers Arbeiten . . . 24
  - 2.2.3 Tips Arbeiten . . . 25
  - 2.2.4 Steimanns Arbeiten . . . 25
- 2.3 Programmabhängigkeitsgraphen . . . 27
  - 2.3.1 PDG-Elemente . . . 27
  - 2.3.2 Sequenzkongruenz . . . 29
  - 2.3.3 Transformationen . . . 30
  - 2.3.4 Grenzen von PDGs . . . 35
- 2.4 Java Development Tools . . . 38
  - 2.4.1 Beispiel . . . 39
  - 2.4.2 Definitionen . . . 39

3 Quelltextabstraktion **43**
3.1 Allgemeine Unterschiede zu PDGs . . . 44
3.2 Begriffsdefinitionen . . . 45
3.3 Elemente und Attribute . . . 46
3.3.1 Graphische Notation . . . 48
3.3.2 Komposition . . . 48
3.3.3 Fragment . . . 50
3.3.4 Knoten . . . 50
3.3.5 Klongruppe . . . 51
3.3.6 Kante . . . 51
3.3.7 Symbol . . . 51
3.4 Textuelle Notation . . . 52
3.4.1 Navigation . . . 52
3.4.2 Indizierung . . . 53
3.4.3 Kurzprädikate . . . 53
3.4.4 Kanten . . . 54
3.4.5 Könnteelemente . . . 58
3.4.6 Positionen . . . 61
3.5 Eigenschaften . . . 61
3.6 Transformationen . . . 63

4 Abbildung von Java 7 **75**
4.1 Erweiterte Notation . . . 76
4.1.1 Kontrollflusskanten . . . 76
4.1.2 Abbildungsregeln . . . 76
4.2 Abbildung des Quelltexts . . . 78
4.2.1 Unterstützte JDT-Anweisungen . . . 78
4.2.2 Ausdrücke . . . 79
4.2.3 Einfache Anweisungen . . . 83
4.2.4 Kontrollanweisungen . . . 86
4.2.5 Fragmente . . . 93
4.2.6 Transformierbarkeit: Extrahieren/Einfügen . . . 97
4.3 Abhängigkeiten im Quelltext . . . 99
4.3.1 Programmeffekte . . . 100
4.3.2 Effekte von Blockabstraktionen . . . 102
4.3.3 Abhängigkeitskanten . . . 103
4.3.4 Beispielabhängigkeiten . . . 105
4.4 Klonerzeugung . . . 106
4.4.1 Propagierung von Kompositionen . . . 107
4.4.2 Klone innerhalb von partiellen Abstraktionen . . . 109
4.4.3 Abstraktion expandieren . . . 111

5 Lösungsfindung **115**
5.1 Constraint-Problem . . . 116
5.1.1 Variablen . . . 116

5.1.2 Notation . . . 116
5.1.3 Generatorregeln . . . 119
5.1.4 Gleichungen des Änderungswunsches . . . 124
5.1.5 Optimierungskriterium . . . 130
5.2 Suchstrategie . . . 130
5.2.1 Teilprobleme . . . 131
5.2.2 Blockteilproblem . . . 133
5.2.3 Restteilproblem . . . 138
5.3 Quelltexttransformation . . . 138
5.3.1 Lösung des Gleichungssystems interpretieren . . . 138
5.3.2 Transformationen anwenden . . . 140
5.3.3 Zusätzliche Quelltextsemantik . . . 142
5.3.4 Verbindlichkeit . . . 144

6 Evaluation 147
6.1 Methodik . . . 148
6.1.1 Forschungsfragen . . . 148
6.1.2 Korrektheit . . . 149
6.1.3 Übermäßige Einschränkungen . . . 151
6.1.4 Mehrwert durch Constraint-Programmierung . . . 152
6.1.5 Laufzeiten . . . 153
6.2 Gegenüberstellung verschiedener Werkzeuge . . . 153
6.2.1 Ausdrücke einfügen . . . 154
6.2.2 Ausdrücke extrahieren . . . 157
6.2.3 Operanden vertauschen . . . 159
6.2.4 Anweisung verschieben . . . 160
6.2.5 Parameter/Argumente vertauschen . . . 162
6.3 Studienergebnisse . . . 163
6.3.1 Teststrategie . . . 163
6.3.2 Untersuchte Projekte . . . 165
6.3.3 Umgestaltungsstellen . . . 166
6.3.4 Ergebnis und Vergleich . . . 169
6.3.5 Berechnungsdauer und Komplexität . . . 174
6.3.6 Ursachen für Fehler und inkorrekte Umgestaltungen . . . 177
6.4 Grenzen des Ansatzes . . . 178
6.4.1 Operationen ändern . . . 178
6.4.2 Komplexe Strukturänderungen . . . 179
6.4.3 Unsichtbare Abhängigkeiten ignorieren . . . 180
6.5 Schlussauswertung . . . 181

7 Zusammenfassung 185

8 Ausblick 187

Abbildungsverzeichnis **189**

Tabellenverzeichnis **191**

Literaturverzeichnis **193**

« Aber was der Wille erstrebt, erreicht er. »

— Donald Duck, 1956
aus *«Donald, der Herr über alle Geschöpfe»* [7]

Kapitel 1

Einführung

Laut Lehmans Gesetz der Software-Evolution wächst die Funktionalität eines Programms stetig im Verlauf der Zeit. Im Gegenzug sinkt jedoch auch die Quelltextqualität und dessen Komplexität steigt kontinuierlich an [66, 67]. Es ist absehbar, dass diese Entwicklung zu kaum erweiterbaren oder wenigstens wartbaren Programmen führt. Man fragt sich, warum in solch vertrakten Situationen das Programm nicht einfach neu entwickelt wird, das dann nicht nur alle neuen Anforderungen erfüllt, sondern auch neu strukturiert und nach dem neuesten Stand der Technik entworfen ist? Die diesem Gedanken zugrunde liegende Denkweise fasst Yourdon mit den *drei Mythen der Software-Entwicklung* wie folgt zusammen:

Mythos 1:

> *« We can always afford to scrap out the old system and replace it with new a system, as long as we can demonstrate to the users & management that the new system will be better. »*

Mythos 2:

> *« We can be absolutely, positively, totally confident that a new replacement would be much, much, much better than the old one. »*

Mythos 3:

> *« We can always figure out what the old system is doing and translate it into a new implementation. »*
>
> — Yourdon, 1994 [129, S. 239f]

Laut Yourdon haben Software-Unternehmen mit Rang und Namen diese Aussagen als Mythen erkannt und versuchen nicht, ihnen nachzugehen. Hieraus ergibt sich jedoch die Notwendigkeit, mit bestehenden Software-Systemen über lange Zeiträume zu arbeiten, sie weiter zu entwickeln und mit ihnen zu leben. Folglich sind Programme einem lange andauernden Prozess der Umstrukturierung und Neugestaltung unterworfen, der sich aus den laufend ändernden Anforderungen der jeweiligen Gegenwart ergibt. Denn schließlich hat es sich als unmöglich erwiesen, alle Nutzeranforderungen, gesetzlichen Rahmenbedingungen und künftigen Änderungen, usw. bereits in der Entwicklungsphase einer Software abzusehen und vorausschauend und fehlerfrei in das Programm zu integrieren.

Von Fowler wurde die *Metapher der zwei Hüte* geprägt, die besagt, dass ein Entwickler zwei unterschiedliche Arbeitsweisen – symbolisiert durch die Hüte – abwechselnd anwendet [35, S. 47]. Die erste Arbeitsweise konzentriert sich auf die Erweiterung des Programms, also auf die Implementierung neuer Funktionen. Die zweite besteht im Umgestalten des bestehenden Quelltexts, um dessen Struktur und Lesbarkeit zu verbessern und so die Arbeit in der implementierenden Phase zu vereinfachen. Nach Beck [13] sollte ein Programmierer dynamisch zwischen diesen beiden Phasen wechseln können, je nachdem, ob für die aktuelle Implementierung Umgestaltungen eine Verbesserung erbringen oder nicht. Diese Aufteilung in zwei Arbeitsweisen wurde bereits 1992 von Griswold beschrieben, der damit die Wartung von Software besser strukturieren wollte [48]. Um diese klare Trennung zu ermöglichen, ist es notwendig, in der Umgestaltungsphase auf jegliche Verhaltensänderung zu verzichten und sich alleine auf die Interna der Software zu konzentrieren. In der Praxis wurde von Murphy-Hill et al. allerdings gezeigt, dass diese klare Trennung noch nicht immer von den Entwicklern eingehalten wird [79]. Der Verzicht auf Verhaltensänderungen in der Umgestaltungsphase ermöglicht jedoch eine bessere Überprüfung der Korrektheit der Umgestaltungen. Wird die Trennung, wie von Fowler empfohlen, durch eine Test-Suite unterstützt, so gilt, dass sich niemals die Testergebnisse durch die zweite Phase ändern sollten. Sollte doch eine solche Änderung auftreten, so ist entweder der Test zu spezifisch und sollte korrigiert werden oder die durchgeführte Umgestaltung ist fehlerhaft. Durch dieses Vorgehen können Fehler frühzeitig erkannt und dadurch leichter lokalisiert und behoben werden.

Besonders durch die agilen Entwicklungsmethoden hat seit Veröffentlichung des *Agilen Manifests* [13] die Bedeutung von Umgestaltungen stark zugenommen [1]. Agile Entwicklungsmethoden zeichnen sich durch kurze Iterationszyklen aus, zu deren Ende immer eine auslieferbare Programmversion fertiggestellt wird. Hierdurch werden zwar einerseits Fortschritte schnell sichtbar, andererseits müssen dadurch häufig bereits implementierte Funktionen modifiziert oder an die nächsten Entwicklungsarbeiten angepasst werden. Die Möglichkeit, den Quelltext umzugestalten, wird daher gerade bei agiler Software-Entwicklung intensiv genutzt, um auch ohne aufwendige, der Implementierung vorangehende Planungsphasen hochqualitative Software zu schreiben. Ohne die Möglichkeit, strukturelle Anpassungen auch zu einem späteren Zeitpunkt noch umzusetzen, wäre eine agile Entwicklung kaum möglich. Robert C. Martin schreibt daher in seinem Standardwerk *Agile Software Development: Principles, Patterns, and Practices*:

> *« I strongly recommend that you always practice such refactoring for every module you write and for every you maintain. »*
>
> — Martin, 2003 [69, S. 42]

Die Konsequenz daraus, Software-Systeme über lange Zeit nutzen zu müssen, ist, dass bis zu 75% der Kosten bei der Software-Entwicklung erst nach der Auslieferung eines Programms entstehen [4]. Bereits 2002 betrugen die jährlichen, geschätzten, totalen Kosten von Programmfehlern allein in den USA ca. 60 Mrd. USD [116, Reduktionsdiagramm Abb. 5-4]. Übertragen auf die heutige Zeit und auf alle Nationen liegt diese Schätzung um ein Vielfaches höher. Nicht nur die für diese Summe ursächlichen Programmfehler müssen kontinuierlich aus den eingesetzten Software-Systemen entfernt werden. Ferner werden diese Systeme auch stetig neuen Anforderungen angepasst. Bei diesen Arbeiten werden jedoch 80% der Zeit nur zum Verstehen der Programme und lediglich 20% zum Editieren

verwendet [62, Tortendiagramm Abb. 3]. Daher liegt die durchschnittliche Produktivität eines professionellen Entwicklers bei nur 10 Zeilen Quelltext pro Tag [16]. Dies zeigt, wie wichtig gut strukturierte und leicht verständliche Programme sind, um ihre Wartung effizient und kostengünstig zu gestalten.

Hilfen, um Quelltext schneller umzugestalten, fanden daher schnell Eingang in die gängigen Entwicklerwerkzeuge von Programmierern. Bereits in ihrer ersten Version verfügte die Entwicklungsplattform *Eclipse* [117] im Jahr 2002 über einige simple Umgestaltungen wie dem extrahieren von Methoden und dem Umbenennen von Programmelementen [37]. In der Zwischenzeit wurden weitere und bessere Funktionen eingeführt, die Quelltextumgestaltungen vereinfachen. Diese Hilfen arbeiten semiautomatisch [71] und funktionieren auf Basis einer Interaktion mit dem Entwickler. Dabei wählt der Benutzer manuell das zu bearbeitende Element und die gewünschten Parameter aus und stößt anschließend die automatische Umgestaltung an. Diese Art von Unterstützung ist mittlerweile in allen etablierten Entwicklungsplattformen für die Programmiersprache *Java* [43] wie *Eclipse*, *IntelliJ IDEA* [54] und *NetBeans* [80] integriert, die zusammen einen Marktanteil von etwa 91% aller Plattformen ausmachen [126, Umfrage unter 2164 Entwicklern].

Obwohl diese Werkzeugunterstützung vorhanden ist, führen dennoch viele Entwickler Umgestaltungen lieber manuell durch [60, Abb. 1]. Als Grund geben sie dafür an, dass die angebotenen Umgestaltungen fehlerhaft seien [60, Abb. 2], [95, 110]. Üblicherweise basieren manuell durchgeführte Umgestaltungen auf den Ansätzen *kopieren/einfügen* und *ausschneiden/einfügen* [48]. Leider unterlaufen den Programmierern dabei häufig auch einfache syntaktische sowie semantische Fehler. Beispielsweise passen sie nicht alle Aufrufe einer Methode an, wenn sie die Implementierung der Methode modifiziert haben [48]. Hierdurch entstehen Inkonsistenzen im Programm, die möglicherweise fehlerhaft sind oder zukünftig fehleranfällig sein werden.

Zusammengefasst stellen die etablierten Entwicklungsumgebungen fehlerbehaftete Funktionen zur Umgestaltung des Quelltexts zur Verfügung, die daher von den Entwicklern nur mit Einschränkungen genutzt werden. Mit verbesserten Umgestaltungswerkzeugen ließe sich ihr Einsatz erhöhen und somit die Fehleranfälligkeit senken und die Programmiereffizienz steigern. Einige dieser bisher fehlerhaften Umgestaltungen zu verbessern und weiter zu entwickeln, ist daher Thema dieser Arbeit.

1.1 Problembeschreibung

Aktuelle Entwicklungswerkzeuge verfügen mittlerweile über eine Vielzahl von Umgestaltungsfunktionen. Laut Murphy [78] und Murphy-Hill [79] werden elf verschiedene Umgestaltungsfunktionen regelmäßig von Entwicklern genutzt. Sieben davon werden in Abbildung 1.1 dargestellt, die von mindestens fünfzehn Prozent der Entwickler eingesetzt werden. Selten genutzte Umgestaltungsfunktionen, die nicht in Abbildung 1.1 aufgeführt sind, werden nach Murphy nur noch von fünf oder weniger Prozent aller Entwickler genutzt. Hierzu gehört beispielsweise das Einführen zusätzlicher Methodenparameter [78].

Diese Arbeit fokussiert sich auf die Gruppe der kontrollflusssensiblen Umgestaltungen. Kontrollflusssensible Umgestaltungen können den Kontrollfluss innerhalb von Methoden ändern. Dazu zählen das Verschieben von Anweisungen und Ausdrücken, das Extrahieren und Einfügen lokaler Variablen sowie das Ändern von Methodensignaturen. Diese

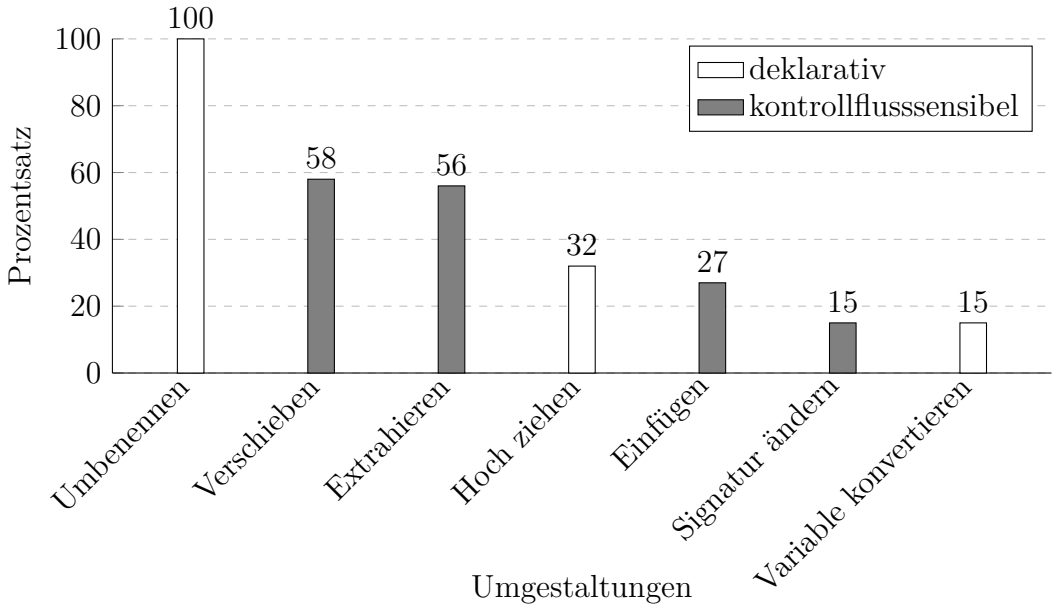


Abbildung 1.1: Anteil der Entwickler, die Umgestaltungshilfen nutzen (nach [78])

Umgestaltungen werden laut Murphy von 15-58% aller Entwickler werkzeugunterstützt durchgeführt. Jedoch arbeiten die etablierten Entwicklungsumgebungen Eclipse, IntelliJ IDEA und NetBeans bei all diesen Umgestaltungen fehlerhaft. Dies wird an den folgenden beiden Beispielen illustriert.

```
1  class A {
2    int x = 42;
3    int m() {
4      int y = x; // y einfügen
5      int x = 23;
6      return y;
7    }
8  }
```

Quelltext 1.1: Variable `y` einfügen (aus [95])

Das erste Beispiel wurde aus der Arbeit von Schäfer [95, S. 45, Quelltext 3.1] entnommen. Es zeigt ein Programm, auf das die Umgestaltung *Variable einfügen* bezogen auf die Variable `y` angewendet wird. Ziel dieser Umgestaltung ist, dass dort wo es verwendet wird, das Literal `y` durch `x` ersetzt wird. Außerdem soll die Deklaration und Initialisierung von `y` entfernt werden, sofern `y` nicht weiter verwendet wird. Im Beispiel 1.1 führt diese einfache Ersetzung jedoch zu einer Änderung der Programmsemantik. Wenn in Zeile 6 `y` ersetzt wird, sodass `return x;` entsteht, ändert sich die Rückgabe der Methode, da die Datenflussabhängigkeit missachtet wird. Dennoch führt Eclipse diese Änderung wie beschrieben und damit fehlerhaft durch.

Im Beispielprogramm 1.2 soll die Signatur der Methode `kappuru` verändert werden, indem ihre beiden Argumente miteinander vertauscht werden. Diese Umgestaltung wirkt sich nicht nur auf die hier aufgeführte Signatur aus, sondern auch auf den Methodenaufruf von `kappuru` in Zeile 7. Neben den Argumenten in ihrer Signatur müssen auch die Parameter ihrer Aufrufe vertauscht werden. In der abgebildeten Programmversion wird als Parameter

jeweils der gleiche Wert, wie 0 oder 1, übergeben. Werden beide Parameter geradewegs miteinander vertauscht, würde sich dies ändern. Der Grund dafür ist, dass sie untereinander eine Datenflussabhängigkeit besitzen, da der zweite Parameter `i++` den Wert der Variablen `i` modifiziert. Folglich führt das von den genannten, etablierten Entwicklungsumgebungen durchgeführte, einfache Vertauschen beider Parameter zu keiner korrekten Umgestaltung. In diesem Beispiel kann die Fehlerursache auf den Seiteneffekt eines Unterausdrucks zurückgeführt werden.

```
1  abstract class A {
2    abstract int kappuru(int yukochan, int makun);//Parameter vertauschen
3
4    void foo() {
5      int i = 0;
6      while (i<10)
7        kappuru(i, i++);
8    }
9  }
```

Quelltext 1.2: *Signatur ändern*: Parameter von `kappuru` vertauschen

Diese Arbeit unterstützt Java-Programme der Version 7 und orientiert sich an der Java-Sprachspezifikation [43]. Wenn auf bestimmte Grammatikregeln oder Zusammenhänge von Java 7 verwiesen werden soll, wird der zugehörige Paragraph in der Sprachspezifikation samt Java-Logo angegeben (z. B. `§1.0`). Als technische Grundlage werden die *Java Development Tools*, kurz JDT [34], genutzt, die sich in der Implementierung eng an die Java-Sprachspezifikation halten. Einzelne Sprachelemente werden daher als Verweis (mit Eclipse-Logo und URL) genutzt, um auf JDT-Elemente wie eine `InfixExpression` verweisen zu können.[1]

Programmverhalten

Die Beispiele zeigen, dass das Programmverhalten durch fehlerhafte Umgestaltungen verändert wird. Wenn sich beim Umgestalten das Verhalten des Programms ändert, müssen diese unerwünschten Änderungen in einem weiteren Arbeitsschritt wieder entfernt werden. Diese Änderungen werden dazu entweder vom Entwickler erkannt oder treten beim Durchführen einer Test-Suite als fehlgeschlagene Testfälle zutage. Werden die Änderungen vom Entwickler übersehen oder durch mangelnde Testabdeckung von der Test-Suite nicht gefunden, verbleiben sie unbeabsichtigt im Quelltext und können unbeabsichtigtes Programmverhalten verursachen. Dieses unbeabsichtigte Programmverhalten wird dann erst später, womöglich beim Kunden, erkannt und anschließend korrigiert. Martin Fowler definiert daher, dass Programmumgestaltungen das *«sichtbare Programmverhalten»* nicht ändern sollen [35]. Mit dem sichtbaren Programmverhalten ist dabei das für den Anwender erkennbare Ein-/Ausgabeverhalten gemeint. Üblicherweise zählen das Laufzeit- und Speicherverhalten nicht dazu. In dieser Arbeit wird unter dem sichtbaren Verhalten das Ein-/Ausgabeverhalten jeder Methode mit allen Programmvariablen verstanden, deren Verhalten sich durch eine Umgestaltung nicht ändern darf. Die Programmverhaltensänderungen werden mithilfe eines ausgiebigen Testverfahrens, basierend auf Test-Suite-Ergebnissen vor und nach Durchführung der Umgestaltung, identifiziert.

[1] Sofern die direkte Übersetzung vom Deutschen ins Englische existiert, werden im Sinne der Lesbarkeit auch die deutschen Begriffe (z. B. `Infixausdruck`) verwendet.

Programmabhängigkeiten

Das Programmverhalten entsteht u. a. durch Lese-/Schreibabhängigkeiten im Quelltext. Beispielsweise wird einer bestimmten Variablen zunächst ein Wert zugewiesen, der an einer späteren Stelle in der Methode wieder gelesen wird. In den beiden oben aufgeführten Beispielen ändert sich das Programmverhalten nur aus dem Grund, weil Datenflussabhängigkeiten missachtet wurden. Besonders häufig bestehen zwischen verschiedenen Ausdrücken wie Variablenverwendungen und Methodenaufrufen Datenflussabhängigkeiten. Werden solche Ausdrücke verschoben, extrahiert oder eingefügt, müssen ihre Abhängigkeiten beachtet werden. Andernfalls ändern sich wie in den Beispielen die Eingabewerte einer Operation und die Ausgabewerte sind nach der Umgestaltung nicht mehr dieselben, sodass sich das Programmverhalten ändert.

Abhängigkeiten entstehen auch durch Seiteneffekte (§15.1) von Ausdrücken oder Unterausdrücken. Besonders Unterausdrücke wie Methodenargumente, die untereinander Abhängigkeiten wegen Seiteneffekten besitzen, können bei künftigen Weiterentwicklungen schnell Programmfehler verursachen. Wenngleich die Verwendung Seiteneffekt behafteter Unterausdrücke gemeinhin als schlechter Programmierstil gilt und die Java-Sprachspezifikation von dieser Programmierweise abrät §15.7, kann sie dennoch häufig angetroffen werden. Dies gilt umso mehr, je mehr fortschrittliche Sprachkonstrukte von Programmierern verwendet werden. Implizite Methodenaufrufe in Java-Programmen werden an vielen Stellen genutzt. Die einfachsten sind davon `toString`-Methoden, die ohne expliziten Aufruf im Hintergrund ausgeführt werden, wenn Objekte z. B. auf der Konsole ausgegeben werden sollen. Ebenfalls werden Standardschnittstellen wie `Iterable` oder `Closeable` verwendet, um mächtige Hochsprachkonstrukte wie die `for-each`-Schleife oder den `try-with-resources`-Block bequem nutzen zu können. Die Methoden, die im Hintergrund ausgeführt werden, sind daher nicht mehr im unmittelbaren Bewusstsein des Entwicklers und unterliegen dadurch mehr und mehr der Gefahr, mitsamt ihren Seiteneffekten vergessen zu werden.

Redundante Implementierungen

Häufig müssen unterschiedliche Umgestaltungen ähnliche oder sogar gleiche Algorithmen bei der Durchführung von Umgestaltungen nutzen. Wie Thies [118] vorgestellt hat, werden in Werkzeugen wie Eclipse jedoch nicht immer bereits bestehende Implementierungen wiederverwendet. Stattdessen werden teilweise für jede Umgestaltung eigene Algorithmen verwendet, die Vor- und Nachbedingungen sowie die Durchführung der Umgestaltung implementieren. Das kann dazu führen, dass sich verschiedene Umgestaltungen, die gleiche Vorbedingungen besitzen, unterschiedlich verhalten. Beispielsweise trifft das auf die Umgestaltungen *Change Signature* und *Use Interface Where Possible* im Programmierwerkzeug IntelliJ IDEA zu. Während *Change Signature* alle notwendigen Vorbedingungen prüft, beachtet *Use Interface Where Possible* eine der Vorbedingungen nicht. Idealerweise sollten jedoch alle Umgestaltungen einen einheitlichen Satz von Regeln für Vor-/Nachbedingungen nutzen, sodass keine redundanten Implementierungen vorliegen und sich die Umgestaltungen konsistent und einheitlich verhalten.

Lösungsberechnung

Neben den in etablierten Entwicklungsumgebungen implementierten Ansätzen wurden auch in der Forschung neue Umgestaltungsansätze entwickelt [95, 120]. Diese arbeiten im Vergleich zu den Entwicklungsumgebungen immer fehlerfrei, indem sie alle relevanten Abhängigkeiten im Programm detektieren und respektieren. Allerdings ist zum Umgestalten des Programms nicht nur wichtig, dass alle Abhängigkeiten bewahrt werden. Ebenfalls ist wünschenswert, dass eine Lösung gefunden wird, die den Umgestaltungswunsch umsetzt. Denn die vielen Abhängigkeiten, die in einem Programm gefunden werden können, führen möglicherweise dazu, dass der Umgestaltungswunsch nicht direkt umgesetzt werden kann. Dies trifft bereits zu, wenn mindestens eine der Abhängigkeiten oder mehrere in Kombination eine Umgestaltung verhindern. In solchen Fällen ist es wünschenswert, wenn kleinere, zusätzliche Änderungen am Quelltext vorgenommen werden. Dadurch können die Abhängigkeiten so umstrukturiert werden, dass sie einerseits weiterhin bestehen bleiben, aber andererseits der Umgestaltungswunsch des Entwicklers umgesetzt werden kann. Dabei muss beachtet werden, dass solche Folgeänderungen ihrerseits zu weiteren Folgeänderungen führen können. Dies kann zu langen Ketten von Folgeänderungen führen [113, 121, 118], die in ihrer Gesamtheit das Programmverhalten nicht verändern dürfen.

1.2 Herausforderungen

Um die genannten kontrollflusssensiblen Umgestaltungen zu verbessern, stellt sich diese Arbeit den folgenden vier Herausforderungen.

Programmsemantik erhalten

In dieser Arbeit wird Programmsemantik von Java-Programmen als das Ein-/Ausgabeverhalten des Programms verstanden. Das Verhalten von Java-Programmen wurde bisher allerdings nicht komplett formal spezifiziert. In den Java-Spezifikationen wird die Semantik von Java-Sprachkonstrukten lediglich semiformal spezifiziert. Dennoch soll das in der Praxis wahrnehmbare Programmverhalten durch Umgestaltungen nicht verändert werden.

Die etablierten Entwicklungsumgebungen implementieren verschiedene Ansätze, um Umgestaltungen möglichst semantikerhaltend durchzuführen. Wie von verschiedenen Autoren gezeigt wurde [97, 120, 110, 118, 103], funktionieren diese Implementierungen jedoch nicht in allen Fällen einwandfrei. Das bedeutet, dass die Durchführung von Umgestaltungen das Ursprungsprogramm entweder verändert oder sogar unkompilierbar macht. Ein Grund dafür ist, dass Abhängigkeiten im Quelltext nicht erkannt werden. Zu diesen Abhängigkeiten zählen einerseits die bereits erwähnten Datenflussabhängigkeiten, aber auch Kontrollfluss- und strukturelle Abhängigkeiten, die durch Sprachkonstrukte entstehen. Beispielsweise können Anweisungen in Java-Programmen innerhalb von verschiedenen Blöcken stehen, wie einem `try`-Block. Die Umgestaltung muss u. a. beachten, dass eine Anweisung, die eine Ausnahme wirft, ihren `try`-Block nicht verlassen darf, da sich ansonsten das Terminierungsverhalten der Methode ändert (☕ §14.1).

Freiheitsgrade nutzen

Trotz der vielen Programmabhängigkeiten können einzelne Anweisungen oder Ausdrücke meist in ihrer Position modifiziert werden, ohne dass sich das Programmverhalten ändert. Ebenfalls können Ausdrücke häufig extrahiert und in lokalen Variablen gespeichert werden. Das liegt daran, dass die Menge aller Programmabhängigkeiten die Ausprägung des Programms nur selten vollständig bestimmt. Folglich verhält es sich mit den Abhängigkeiten und Programmausprägungen wie mit einem unterbestimmten Gleichungssystem. Die Abhängigkeiten bilden die Gleichungen und die verschiedenen Programmausprägungen werden in verschiedenen Variablen festgelegt. Genau wie zu unterbestimmten Gleichungssystemen mehrere Lösungen existieren, können Methoden eines Programms bei einer gleichbleibenden Abhängigkeitsmenge verschieden gestaltet sein. Diese Vielzahl der Ausprägungen eines Programms bei derselben Menge von Abhängigkeiten werden *Freiheitsgrade* genannt, die mögliche Veränderungen an einer Methode repräsentieren. In dieser Arbeit sollen die folgenden Freiheitsgrade genutzt werden, um Umgestaltungen durchführen zu können:

- Anweisungen sollen verschoben,
- Ausdrücke sollen extrahiert sowie einfügt und
- Operanden und Argumente sollen vertauscht werden können.

Diese Freiheitsgrade repräsentieren einerseits alle Umgestaltungen, die der vorgestellte Ansatz unterstützen soll. Andererseits sollen sie einzeln oder auch in beliebiger Kombination weitere Änderungen ermöglichen, um eine korrekte Umgestaltung durchführen zu können.

Angemessene Korrektheit

Die Anzahl der Programmabhängigkeiten schränkt die Freiheitsgrade ein. Je mehr Programmabhängigkeiten in einer Methode erkannt werden, desto weniger Freiheitsgrade existieren und die Anzahl der möglichen Programmausprägungen verringert sich. Unter Umständen verringert sie sich soweit, dass keine weitere Lösung als die bestehende Programmausprägung existiert und somit jede gewünschte Umgestaltung unmöglich wird. Folglich ist die Anzahl der Abhängigkeiten kritisch und muss so gering wie möglich, aber so hoch wie nötig sein.

Die Abhängigkeiten resultieren einerseits aus statischen Quelltexteigenschaften, die direkt aus dem Quelltext abgelesen werden können. Dazu zählt beispielsweise welche Anweisungen in welchen Blöcken stehen, wie Ausdrücke verschachtelt sind und welche intraprozeduralen Datenflüsse in der Methode vorkommen. Andererseits existieren meist auch Abhängigkeiten innerhalb einer Methode, die nicht offensichtlich sind und daher eingehende Analysen erfordern. Hierzu zählen interprozedurale Datenflüsse, die durch Aliase und Seiteneffekte von Ausdrücken und Methodenaufrufen entstehen. Die statischen Verfahren, um Aliase zu erkennen, sind unentscheidbar und liefern je nach Berechnungsaufwand unterschiedlich präzise Abschätzungen [65, 6, 107, 51, 74]. Auf Basis solcher Schätzverfahren können dann Abhängigkeiten im Quelltext festgelegt werden. Allerdings beeinträchtigt die Wahl des Schätzverfahrens die Anzahl der Abhängigkeiten erheblich. Falls beispielsweise sichergestellt werden soll, dass unter keinen Umständen eine Abhängigkeit übersehen werden soll, wird eine Überabschätzung eines sicheren Verfahrens genutzt,

das jedoch weit mehr Abhängigkeiten berechnet, als in der Realität existieren. Dies führt jedoch mit hoher Wahrscheinlichkeit dazu, dass die Freiheitsgrade beim Umgestalten stark eingeschränkt oder gar eliminiert werden. Demnach stehen die beiden Ziele *Sicherheit* und *Freiheitsgrade* zueinander in Konflikt, sodass zwischen beiden abgewogen werden muss, um einen sinnvollen Kompromiss zu erreichen.

Berechnungsdauer

Umgestaltungen werden von Entwicklern ständig während des Programmierens durchgeführt. Häufig werden dazu Werkzeuge genutzt, da sie schneller und fehlerfrei(-er) arbeiten. Daher ist neben der Zuverlässigkeit auch die Geschwindigkeit eine wichtige Eigenschaft von Umgestaltungshilfen. Würde die Berechnungsdauer von Umgestaltungen viele Sekunden oder gar Minuten benötigen, wären sie unpraktikabel für den täglichen Einsatz. Ausschlaggebend für die Berechnungsdauer von kontrollflusssensiblen Umgestaltungen ist die Größe der Methode mit ihren Anweisungen, Ausdrücken und Abhängigkeiten. Je mehr Anweisungen, Ausdrücke und Abhängigkeiten bestehen, desto größer wird das Umgestaltungsproblem und seine Lösungsdauer. Beim Lösen wird versucht, mithilfe der Freiheitsgrade eine Ausprägung des Programms zu identifizieren, die den Umgestaltungswunsch des Entwicklers umsetzt. Hierbei reicht es nicht aus, irgendeine Lösung zu finden. Damit das Programm durch die Umgestaltung nicht unverhältnismäßig stark modifiziert wird und dadurch womöglich kaum wiederzuerkennen ist, wird diejenige Lösung gesucht, die sich am wenigsten vom ursprünglichen Programm unterscheidet. Dazu müssen alle möglichen Lösungen erkannt und anschließend priorisiert werden.

1.3 Lösungsansatz

Ziel dieser Arbeit ist, Umgestaltungen zu ermitteln, die immer dann, wenn keine direkte, fehlerfreie Umgestaltung existiert, einen Alternativvorschlag berechnet, um eine Umgestaltung doch noch durchführen zu können. Dieser Alternativvorschlag besteht dann neben dem Umgestaltungswunsch immer auch aus weiteren zusätzlichen Änderungen. Die Menge dieser zusätzlichen Änderungen soll dabei so klein wie möglich gehalten werden, damit das resultierende Programm dem Ursprungsprogramm weiterhin möglichst ähnlich sieht.

Die Umsetzung des Ziels soll mithilfe eines Constraint-Lösungsverfahrens [122] geschehen. Der Lösungsansatz orientiert sich dabei an vorangegangen Arbeiten aus dem Umfeld von Schäfer und Steimann [121, 25, 37, 120, 118]. Dafür soll das Umgestaltungsproblem vom Quelltext in ein Gleichungssystem überführt werden, das von einem Constraint-Löser gelöst werden kann. Mithilfe verschiedener Bedingungsgleichungen soll eine möglichst semantikerhaltende, minimale Menge an Änderungen gefunden werden. Das gestellte Ziel soll erreicht sein, wenn gezeigt werden konnte, dass der Ansatz (1) praktikabel ist, d. h. leistungsfähig bzgl. seiner Berechnungsdauer, (2) die Programmsemantik möglichst nicht ändert und (3) Umgestaltungslösungen findet, wo andere Ansätze nur fehlerhafte oder keine Lösungen finden.

1.3.1 Vorgehensweise

Der Lösungsansatz ist in Abbildung 1.2 vereinfacht dargestellt. Ein ursprüngliches Programm P wird zunächst in eine Abstraktion A überführt, die im Folgenden auch Quelltextabstraktion genannt wird. Die Quelltextabstraktion abstrahiert das Programm und enthält alle Abhängigkeiten, die für die Umgestaltung und Korrektheit relevant sind. Anhand eines Regelsatzes wird auf Basis der Abstraktion ein Constraint-Problem CSP erstellt. Der Regelsatz beschreibt auf generische Weise, in welchen Situationen welche Constraint-Gleichungen erzeugt werden. Das so erstellte Constraint-Problem wird mithilfe eines Lösers gelöst. Falls eine Lösung CSP_{LSG} existiert, wird aus der Veränderung zur initialen Wertebelegung CSP_{INIT} des Constraint-Problems eine Menge von Transformationen δa ermittelt. Diese Transformationen werden anschließend auf die Programmtransformationen δp abgebildet. Sie erfüllen den Umgestaltungswunsch des Nutzers und bewahren gleichzeitig alle ursprünglich vorhandenen Programmabhängigkeiten. Die Programmtransformationen werden auf das Ursprungsprogramm angewendet, sodass das transformierte Programm P' entsteht. Da ein integrierter Ansatz genutzt wird, der für alle Umgestaltungen die gleiche Quelltextabstraktion und den gleichen Regelsatz verwendet, besteht nicht das Risiko einer redundanten Implementierung von Vorbedingungen oder Abhängigkeiten.

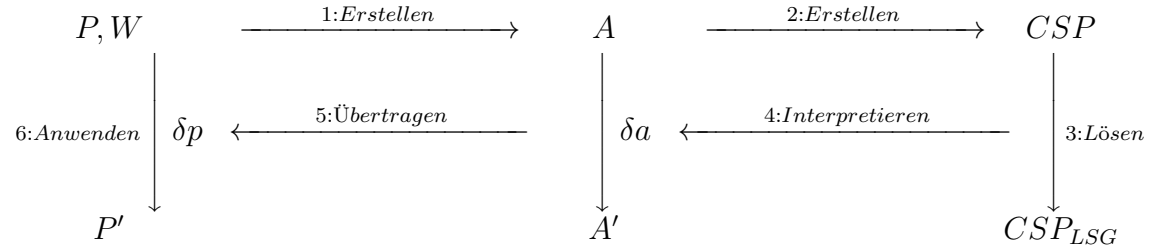


Abbildung 1.2: Aus Programm P und Umgestaltungswunsch W werden die Transformationen δp berechnet.

Der Umgestaltungswunsch W bezieht sich immer auf Elemente im Programm. Daher wird er ebenfalls zunächst auf Elemente der Abstraktion und schließlich im Constraint-Problem abgebildet. Der Wunsch erzwingt durch eine zusätzliche Bedingung im Gleichungssystem eine Änderung, sodass sich CSP_{INIT} und CSP_{LSG} unterscheiden müssen, sofern eine Lösung des Constraint-Problems existiert. Manche Umgestaltungswünsche müssen jedoch nicht immer vollständig im Quelltext umgesetzt werden. Ein Wunsch, dessen teilweise Umsetzung bereits sinnvoll ist, ist das Einfügen einer lokalen Variablen an all ihren Verwendungsorten. In solchen Situationen wird statt zusätzlichen Gleichungen das Optimierungskriterium angepasst, um auszudrücken, dass die Variable an möglichst vielen Orten eingesetzt werden soll. Durch diese Anpassung des Optimierungskriteriums wird jene berechnete Lösung gewählt, die dem Umgestaltungswunsch am nächsten kommt. Das Optimierungskriterium wird ebenfalls dazu genutzt, die Menge der berechneten Lösung nach der Anzahl der Änderungen zu priorisieren. Enthält eine Lösung sehr viele Änderungen im Vergleich zum Ursprungsprogramm, so wird diese schlechter bewertet. Dem Entwickler wird schließlich jene Lösung präsentiert, die die wenigsten Änderungen gegenüber des ursprünglichen Quelltexts besitzt. Falls auch der Umgestaltungswunsch durch das Optimierungskriterium ausgedrückt wird, müssen beide Optimierungsziele gegeneinander abgewägt werden.

1.3.2 Erhalt der Semantik

Unter der Programmsemantik wird in dieser Arbeit das Ein-/Ausgabeverhalten des Programms verstanden. Dieses Verständnis lehnt sich an frühere Arbeiten von Kuck, Ferrante und Horwitz [5, 64, 31, 53, 128] an, die sich mit der Äquivalenz von Programmen beschäftigt haben. Auf ihre Arbeiten gehen Programmabhängigkeitsgraphen (engl. PDG) zurück, die für die Programmsemantik relevante Aspekte repräsentieren. Anhand zweier verschiedener PDGs, die zu verschiedenen Programmen gehören, und einer bestimmten Menge von Transformationen kann festgestellt werden, ob die zwei zugehörigen Programme äquivalent sind. Hierzu wird der Begriff der *Sequenzkongruenz* eingeführt. Unterscheiden sich zwei PDGs ausschließlich durch eine Menge von sequenzkongruenten Transformationen, sind sie sequenzkongruent zueinander. Ebenfalls gilt, dass zwei sequenzkongruente PDGs zwei Programme repräsentieren, die äquivalent in Hinblick auf ihr Ein-/Ausgabeverhalten sind. Die Arbeiten von Griswold [45, 48] im Gebiet der Programmumgestaltungen basieren auf dem Konzept der Sequenzkongruenz. Die von ihm durchgeführten Programmumgestaltungen lassen sich auf sequenzkongruente PDG-Transformationen zurück führen, sodass seine Umgestaltungen keine Verhaltensänderungen am Programm bewirken.

In dieser Arbeit wird ein Programm auf eine Quelltextabstraktion A abgebildet, die an PDGs angelehnt ist. Die Abstraktion A ist eine Erweiterung von PDGs, die wesentlich mehr Informationen enthält, damit eine Rücktransformation der Abstraktion A' auf ein Programm P' immer möglich ist. Zu diesen Informationen gehören Hochsprachenkonstrukte, wie bspw. die Verschachtelung von Ausdrücken oder die Blockzugehörigkeit von Anweisungen. PDGs bilden Maschinen- oder Binärcode ab und können daher auf diese Informationen verzichten. Zwei Abstraktionen A und A' unterscheiden sich durch eine Menge von Transformationen δa, die die Umgestaltung repräsentieren. Für diese Transformationen δa wird ebenfalls die Sequenzkongruenz genutzt, mit deren Hilfe Transformationen von A beschrieben werden, die die Programmsemantik nicht verändern.

Durch die Transformationen δa der Abstraktion werden folglich keine semantisch relevanten Eigenschaften des ursprünglichen Programms verändert. Allerdings trifft dies nur unter der Annahme zu, dass die Abstraktion das Programm mit all seinen Abhängigkeiten korrekt abbildet. Wenn die Anweisungen und Ausdrücke unbemerkte Seiteneffekte besitzen und somit nicht alle Abhängigkeiten in der Abstraktion dargestellt werden, werden Änderungen ermöglicht, die das Programmverhalten ändern können. Zur Erkennung von Abhängigkeiten wird eine interprozedurale Datenflussanalyse [2] genutzt. Weitere Abhängigkeiten, die sich aus der Programmstruktur ergeben, werden direkt aus dem Quelltext abgelesen. Abhängigkeiten, die durch Aliase entstehen, werden in dieser Arbeit jedoch nicht bestimmt. Wie zuvor bereits erwähnt, schränkt eine zu hohe Anzahl an Abhängigkeiten die Freiheitsgrade des Ansatzes ein und zudem sind präzise Aliasanalysen sehr zeitintensiv. Daher wird in dieser Arbeit auf eine Aliasanalyse und die daraus resultierenden Abhängigkeiten verzichtet. Nichtsdestotrotz können sowohl die Aliasanalyse und ihre, aber auch weitere, Abhängigkeiten später nachgerüstet werden. Da in dieser Arbeit auf eine Aliasanalyse verzichtet wird, kann folglich die Korrektheit der Umgestaltungen nicht garantiert werden. Um zu zeigen, dass der Ansatz die Semantik erhält oder mindestens besser als andere Werkzeuge arbeitet, wird stattdessen – wie im Forschungsgebiet der Java-Umgestaltungen üblich [113, 119, 118] – eine ausgiebige Untersuchung an Beispielen durchgeführt.

1.3.3 Umsetzung des Wunsches

Der Ansatz berechnet auf Grundlage der Quelltextabstraktion A eine Menge an Transformationen, die das Programmverhalten unverändert lassen, sofern A alle Abhängigkeiten berücksichtigt. Dass die berechneten Transformationen den Umgestaltungswunsch des Entwicklers umsetzen, resultiert aus der Art, wie die Transformationen aus der Lösung des Constraint-Problems abgelesen werden. Sofern ein Wunsch direkt als eigene Bedingungsgleichung in das Constraint-Problem einfließt, bewirkt er eine erzwungene Änderung einer Constraint-Variablen in der Lösung. Da der Wunsch genau jene Gleichungsvariable auf den Wunschwert festlegt, die später verwendet wird, um die Transformation abzulesen, ist sichergestellt, dass der Umgestaltungswunsch des Entwicklers in der Lösung enthalten ist. Falls der Umgestaltungswunsch jedoch über Optimierungskriterium das Berechnungsziel festlegt, muss überprüft werden, ob die optimale Lösung eine Transformation enthält, die den Wunsch teilweise oder ganz umsetzt.

1.3.4 Freiheitsgrade der Lösungen

Der Ansatz bietet die Freiheitsgrade, dass (1) Operanden und Anweisungen in ihrer Reihenfolge verändert, sowie dass (2) Ausdrücke extrahiert und eingefügt werden können. Ferner besitzt er den Freiheitsgrad, dass Variablendeklarationen und Zuweisungen entfernt werden können. Im Nachgang werden dadurch notwendig oder überflüssig gewordene Sprachkonstrukte wie Deklarationen erstellt oder entfernt. Durch die Freiheitsgrade werden Umgestaltungen möglich, die sonst nicht oder nur unter Änderung der Semantik durchgeführt werden könnten. Folglich profitiert die Anwendbarkeit des Ansatzes mit der Größe der Freiheitsgrade. Je besser diese Freiheitsgrade eingesetzt werden können, desto wahrscheinlicher ist ein Umgestalten eines Programms erfolgreich.

Das Ändern der Reihenfolge von Operanden oder Anweisungen setzt der Ansatz über partiell geordnete Anweisungen bzw. Operanden um. Auf diese Weise kann eine totale Ordnung der Anweisungen bzw. Operanden gefunden werden, die sowohl den Abhängigkeiten im Quelltext genügt, als auch dem Optimierungskriterium oder neuen Abhängigkeiten, die durch den Umgestaltungswunsch hinzu gekommen sind. Da die Anweisungen bzw. Operanden als Graph implementiert sind, dessen Kanten die Abhängigkeiten repräsentieren, ist dieser Freiheitsgrad bereits optimal ausgeprägt.

```
void f() {
  int s, t;
  t = 1;        // Wunsch: t an beiden Stellen einfügen
  s = t + t;
}
```

Quelltext 1.3: Ausdruck an zwei Stellen einfügen

Um den zweiten Freiheitsgrad – das Einfügen/Extrahieren von Ausdrücken – umzusetzen, wird die Abstraktion in einem zusätzlichen Schritt umgeformt. Diese Umformung ist notwendig, damit ein Ausdruck, der nur einmal im Quelltext existiert, durch die Umgestaltung an mehreren Stellen im Quelltext eingefügt werden kann. Im Quelltextbeispiel 1.3 lautet der Umgestaltungswunsch, dass die lokale Variable `t` an zwei Stellen im Programm eingefügt werden soll. Daher muss das zugewiesene Literal `1` an diesen beiden Stellen eingefügt werden. Im ursprünglichen Quelltext und seiner Abstraktion existiert jedoch nur eine

Repräsentation des Literals `1`. Jedoch werden im Fall des Einfügens zwei Repräsentationen benötigt, da das Literal `1` an zwei unterschiedlichen Stellen im Quelltext eingefügt werden soll. Die Umformung erzeugt daher eine Kopie des Literals, damit eine Lösung berechnet werden kann, in der das Literal an beiden Stellen eingefügt werden kann. Die Umformung wirkt sich allerdings auf alle Ausdrücke aus, die durch eine Umgestaltung eingefügt werden könnten. Dadurch steigt die Anzahl der Elemente in der Abstraktion. Jedoch wirkt sich dieser Anstieg nur bedingt nachteilig auf das Finden einer Lösung aus. Obwohl die nicht benötigten Ausdrücke im Gleichungssystem vorhanden sind, kann eine optimale Lösung gefunden werden. Lediglich die Anzahl der Variablen und Gleichungen, und damit auch der Berechnungsaufwand, steigen.

Das Entfernen von Deklarationen und/oder Zuweisungen ermöglicht der Ansatz über seinen dritten Freiheitsgrad. Die Existenz von Deklarationen und Zuweisungen wird in der Quelltextabstraktion als Bool'sche Variable repräsentiert. Nur wenn eine lokale Variable an einer Stelle im betrachteten Programm durch eine Zuweisung definiert wird, existiert auch ihre Deklaration. Ebenfalls existiert eine Zuweisung nur dann, wenn sie durch keine *Einfüge*-Umgestaltung überflüssig wird.

1.4 Unterstützte Umgestaltungen

Fünf Umgestaltungen sollen vom Ansatz unterstützt werden. Die ersten beiden davon werden im nächsten und die anderen in den folgenden Abschnitten vorgestellt.

1.4.1 Ausdrücke extrahieren/einfügen

Ausdrücke treten in Programmen an verschiedenen Stellen auf: Sie können direkt in Anweisungen auftreten, wie bspw. bei der Zuweisung auf der rechten Seite oder als Bedingung einer Alternative. Ferner können Ausdrücke verschachtelt sein und Unterausdrücke enthalten. In all diesen Fällen kann ein Ausdruck durch eine zusätzliche lokale Variable ersetzt werden. Diese lokale Variable wird dann zuvor deklariert und mit dem ersetzten Ausdruck initialisiert, sodass sich das Ein-/Ausgabeverhalten des Programms nicht ändert. Diese Art der Quelltextumgestaltung wird je nach Entwicklungswerkzeug unterschiedlich benannt. Bspw. verwendet die Plattform Eclipse die Bezeichnung „Extrahiere lokale Variable“ (engl. extract local variable). In dieser Arbeit wird diese Umgestaltung als *Ausdruck extrahieren* bezeichnet. Beim Extrahieren von Ausdrücken im engeren Sinn wird lediglich der gewählte Ausdruck durch eine Variable ersetzt. Andere Ausdrücke, die semantisch identisch sind, bleiben unangetastet und werden nicht durch die neue Variable ersetzt, obwohl dies möglich wäre. Wird eine zusätzliche Klonerkennung (engl. clone detection) [63, 12, 93, 94] durchgeführt, könnten semantische Klone gefunden und ebenfalls durch die lokale Variable ersetzt werden. Klonerkennung wird im Rahmen dieser Arbeit nicht betrachtet. Der Umgestaltungswunsch des Nutzers besteht hier aus der Angabe eines Ausdrucks mit dem Hinweis, dass dieser extrahiert werden soll. Werden Klone erkannt, so kann der Umgestaltungswunsch auf jeden erkannten Klon erweitert werden.

Die komplementäre Umgestaltung zum Extrahieren ist das Einfügen von Ausdrücken. In gängigen Entwicklungsumgebungen wie Eclipse wird diese Umgestaltung „Einfügen“ (engl. inline) genannt. Ihre Bezeichnung in dieser Arbeit lautet etwas ausführlicher *Ausdruck*

einfügen. Ausgangspunkt dieser Umgestaltung ist eine Variablenzuweisung z, die aus einer Variablen v auf der linken Seite sowie einem zugewiesenen Ausdruck a auf der rechten Seite besteht. Beim *Ausdruck einfügen* werden alle lesenden Variablenverwendungen von v durch den Ausdruck a ersetzt. Dies gilt jedoch nur für solche Variablenverwendungen, die eine Schreib-/Leseabhängigkeit zur Zuweisung z besitzen: Folgt später im Quelltext eine erneute Zuweisung zu v, werden die dann folgenden Variablenverwendungen nicht durch die Umgestaltung modifiziert. Der Umgestaltungswunsch des Nutzers besteht hier aus der Angabe des Ausdrucks mit dem Wunsch, dass dieser an allen möglichen Stellen eingefügt werden soll.

```
void f() {
  int c;
  c = 1+2;
  lin(c);
}
```

Quelltext 1.4: Extrahierter Ausdruck `1+2`

```
void f() {
  int c;

  lin(1+2);
}
```

Quelltext 1.5: Eingefügter Ausdruck `1+2`

Im Beispiel 1.4 (links) und 1.5 (rechts) wird von rechts nach links ein Ausdruck extrahiert. Umgekehrt wird der Additionsausdruck `1+2` im Methodenaufruf `lin` da eingefügt, wo das Argument `c` steht. Im rechten Beispiel ist daher die Addition ein Argument des Funktionsaufrufs von `lin` und somit ein Unterausdruck. Dieser kann durch die Umgestaltung *Ausdruck extrahieren* herausgelöst werden, sodass der Quelltext im linken Beispiel entsteht. Die Umgestaltung *Ausdruck einfügen* transformiert dementsprechend das linke Beispiel in das Beispiel auf der rechten Seite.

1.4.2 Operanden verschieben/neuordnen

Etliche Operationen, die in Ausdrücken verwendet werden können, verfügen über zwei oder mehr Operanden. Bspw. betrifft das die meisten mathematischen Operationen wie Addition oder Division und die logischen Operationen wie die Verundung. Sind die Operationen kommutativ, so können ihre Operanden beliebig miteinander vertauscht werden. Dieses Vertauschen der Operanden wird lediglich durch Abhängigkeiten eingeschränkt, die zwischen den einzelnen Operanden existieren. Bspw. können Seiteneffekte auftreten, wenn ein Operand eine Methode aufruft. Solche Seiteneffekte können nachfolgende Operanden beeinflussen, sodass ein Vertauschen der Operanden zu einer Änderung der Programmsemantik führt. Bestehende Entwicklungsumgebungen bieten keine Unterstützung für das Umgestalten von Operanden. Grund dafür ist sicherlich, dass die Arbeitserleichterung durch Werkzeugunterstützung minimal ist, wenn das Werkzeug die Änderung lediglich textuell durchführt. Da im Rahmen dieser Arbeit jedoch Semantik erhaltende Umgestaltungen entwickelt werden, können Nutzer profitieren, wenn die Entwicklungsumgebung beim Fehlervermeiden hilft.

Beim Umgestalten von Operanden im Quelltext wird zwischen zwei Arten der Umgestaltung unterschieden: das *Neuordnen von Operanden* und das *Verschieben eines Operanden*. Beim Neuordnen von Operanden wird für alle Operanden einer Operation angegeben, welche neue Position sie bekommen sollen. Das bedeutet, dass der Nutzer explizit in seinem Umgestaltungswunsch angeben muss, an welche Stelle jeder Operand platziert werden soll.

Demgegenüber muss der Nutzer beim Verschieben eines Operanden lediglich angeben, welcher Operand in welche Richtung verschoben werden soll. Über die anderen

Operanden macht der Nutzer keine Angaben. Je nachdem, welche Abhängigkeiten zwischen den einzelnen Operanden existieren, kann so eine Lösung gefunden werden, die den Umgestaltungswunsch umsetzt.

```
int i = 1 + 2 + 3;
```

Quelltext 1.6: Ursprüngliche Addition

```
int i = 3 + 1 + 2;
```

Quelltext 1.7: Neu geordnete Operanden

Im Beispiel 1.6 (links) und 1.7 (rechts) werden die Operanden eines Ausdrucks neu geordnet.

1.4.3 Parameter/Argumente neuordnen

Werden beim Ändern einer Methodensignatur die Parameter neu geordnet, müssen verschiedene Änderungen im Quelltext durchgeführt werden. Einerseits müssen alle anderen Methodensignaturen von überschreibenden Methoden angepasst werden. Andererseits müssen ebenfalls alle Methodenaufrufe angepasst werden, die an die geänderten Signaturen binden. Das Ändern von Signaturen bzgl. der Neuordnung von Parametern wird in zwei Arbeitsschritte aufgeteilt. Zunächst muss überprüft werden, ob die neue Methodensignatur keinen Konflikt mit bestehenden Methoden hervorruft. Hierfür werden bestehende Ansätze genutzt, die mit Vor- und Nachbedingungen zuverlässig arbeiten und in Eclipse integriert sind [35, 121, 61]. Wenn im zweiten Arbeitsschritt die notwendigen Änderungen der Methodenaufrufe betrachtet werden, kann der in dieser Arbeit vorgestellte Ansatz genutzt werden. Er integriert sich dadurch als einzelner Arbeitsschritt in die Quelltextumgestaltung *Methodensignatur ändern* (engl. change method signature). Ähnlich wie beim Neuordnen von Operanden müssen beim Neuordnen der Argumente die Abhängigkeiten zwischen einzelnen Argumenten beachtet werden. Andernfalls kann eine Änderung der Methodensignatur dazu führen, dass das Programmverhalten geändert wird. Der Umgestaltungswunsch des Nutzers betrifft hier die Methodensignatur unter Angabe der Änderungen. Sollen Änderungen an der Methodensignatur durchgeführt werden, die die Reihenfolge der Parameter nicht beeinflusst, wie bspw. Typänderungen, kommt der in dieser Arbeit vorgestellte Ansatz nicht zum Einsatz.

```
class C {
  void rome(int a, int b) {
    // ...
  }

  void exists(int i1, int i2) {
    rome(i1, i2);
  }
}
```

Quelltext 1.8: Ursprüngliche Methode

```
class C {
  void rome(int b, int a) {
    // ...
  }

  void exists(int i1, int i2) {
    rome(i2, i1);
  }
}
```

Quelltext 1.9: Vertauschte Parameter

Im Beispiel 1.8 (links) und 1.9 (rechts) werden die Parameter der Methode `rome` neu geordnet. Durch das Vertauschen der Parameter `a` und `b` werden auch die Argumente der zugehörigen Methodenaufrufe entsprechend der neuen Signatur neu geordnet. Da dem Entwickler beim Neuordnen von Methodenargumenten meist nicht alle Aufrufe präsent sind, könnte er fehlerhaft durchgeführte Umgestaltungen schnell übersehen.

1.4.4 Anweisungen verschieben

Entwicklungsumgebungen wie Eclipse unterstützen das Verschieben einzelner Textzeilen im Quelltext im Sinne eines reinen Texteditors. Mithilfe einer Tastenkombination wird die Zeile, in der sich die Schreibmarke befindet, nach oben oder unten verschoben. Hierbei wird jedoch nicht auf zusammenhängende Sprachkonstrukte geachtet: Wenn sich bspw. Anweisungen über mehrere Zeilen erstrecken, wird nur die ausgewählte Zeile verschoben, sodass Anweisungen auseinander gerissen werden können. Ebenfalls werden Abhängigkeiten zwischen einzelnen Anweisungen vernachlässigt: Sind aufeinander folgende Anweisungen bspw. durch eine Datenflussabhängigkeit miteinander verbunden, geht diese aufgrund eines textbasierten Verschiebens von Zeilen verloren und die Programmsemantik ändert sich folglich. Im Rahmen dieser Arbeit wird eine Quelltextumgestaltung eingeführt, die ähnlich wie beim Verschieben von Operanden das verhaltensbewahrende Verschieben von Anweisungen erlaubt. Zur Durchführung dieser Umgestaltung gibt der Nutzer die zu verschiebende Anweisung und die gewünschte Richtung an.

```
i = 1; // abwärts schieben
i++;
foo();
```

Quelltext 1.10: Ursprüngliche Reihenfolge der Anweisungen

```
foo();
i = 1;
i++;
```

Quelltext 1.11: Neue Reihenfolge der Anweisungen

Im Beispiel 1.10 (links) und 1.11 (rechts) werden die Anweisungen eines Blocks neu geordnet. Die Methode `foo` hat im Beispiel keine Lese-/Schreibabhängigkeit zur Variablen `i`. Da die Datenflussabhängigkeit der ersten beiden Anweisungen beachtet wird, wird der Methodenaufruf ganz nach oben verschoben. Eclipse hätte lediglich die ersten beiden Anweisungen vertauscht und damit ihren Datenfluss aufgehoben.

1.5 Beitrag der Arbeit

Im Folgenden sind die wesentlichen Beiträge dieser Arbeit in Kürze zusammengefasst:

- Die *Quelltextabstraktion* (Kapitel 3) bildet einerseits alle notwendigen Quelltextabhängigkeiten ab und lässt andererseits möglichst viele Freiheitsgrade zum Finden einer Umgestaltungslösung offen. Die Umgestaltungslösung besteht aus eine Menge von sequenzkongruenten Transformationen (Abschnitt 3.6), die die durch die Abstraktion abgebildete Programmsemantik nicht verändern.

- Die *Abbildungsvorschrift* (Kapitel 4) eines Java-7-Quelltexts auf eine Quelltextabstraktion beschreibt, wie aus einzelnen Programmelementen eine Quelltextabstraktion erzeugt wird. Durch die Abbildungsvorschrift kann der Ansatz auf bestehende Programme angewendet werden.

- Ein *Regelsatz* (Abschnitt 5.1) spezifiziert, in welchen Situationen welche Constraint-Gleichungen erzeugt werden. Er übernimmt somit die Abbildung der Quelltextabstraktion auf ein Constraint-Problem, in dem letztendlich Abhängigkeiten und Freiheitsgrade implementiert sind.

- Das Constraint-Problem wird mithilfe einer angepassten *Lösungsstrategie* (Abschnitt 5.2) gelöst. Sie berücksichtigt die Struktur der Constraint-Probleme, die von Quelltextabstraktion und Regelsatz erzeugt werden. Durch die Lösungsstrategie wird ein Lösen der Umgestaltungsprobleme in angemessener Zeit möglich und somit ein Einsatz erst praktikabel.

- Mithilfe der prototypischen Implementierung *CodeShaker* (Abschnitt 6) wird der Ansatz ausgewertet. CodeShaker ist als Eclipse-Plugin implementiert und kann wie etablierte Werkzeuge auf gewohnte Art eingesetzt werden. Ferner bietet es mehrere zusätzliche Ansichten und Konfigurationsmöglichkeiten, um beispielsweise die Quelltextabstraktion einer Methode oder Statistiken anzuzeigen.

- Eine *Evaluation* (Kapitel 4) mit ausgewählten Beispielen zeigt, wie sich der Ansatz im Vergleich zu etablierten Werkzeugen verhält. Da ein formaler Korrektheitsbeweis einer formalen Spezifikation von Java 7 bedürfte, die nicht existiert, wird stattdessen auf eine massive Testreihe von quelloffenen Projekten zurückgegriffen. Anhand von 133768 CodeShaker-Testfällen wird evaluiert, inwieweit der Ansatz korrekte Ergebnisse auch im Vergleich zur Eclipse IDE berechnet. Ebenfalls wird ausgewertet, wie hoch die Berechnungsdauern sind und bis zu welcher Methodengröße der Ansatz Umgestaltungen in akzeptabler Zeit berechnen kann (Abschnitt 6.3).

1.6 Aufbau der Arbeit

Im folgenden zweiten Kapitel wird auf die Grundlagen, Programmabhängigkeitsgraphen (PDGs) und verwandte Arbeiten eingegangen. Außerdem werden die *Java Development Tools* (JDT) kurz vorgestellt, die die Grundlage der technischen Umsetzung dieser Arbeit bilden. Anschließend wird im Kapitel 3 die der Arbeit zugrunde liegende Quelltextabstraktion, eine zugehörige Notation und die Unterschiede zwischen der Abstraktion und PDGs vorgestellt. Ferner werden die sequenzkongruenten Transformationen der Abstraktion eingeführt. Wie Java-Programme auf die zuvor vorgestellte Abstraktion abgebildet werden, wird im vierten Kapitel erklärt. In Kapitel 5 wird beschrieben, wie aus der Abstraktion ein Constraint-Problem erzeugt und dieses mithilfe einer optimierten Suchstrategie und einem Constraint-Löser in praxistauglicher Zeit gelöst wird. Außerdem wird im fünften Kapitel gezeigt, welche Transformationen aus diesen Constraint-Lösungen abgelesen und wie diese auf den ursprünglichen Quelltext angewendet werden. Schließlich werden die Evaluationsergebnisse im sechsten Kapitel vorgestellt. Die Arbeit endet mit einer Zusammenfassung und einem Ausblick.

Kapitel 2

Thematischer Hintergrund

Das folgende Unterkapitel erläutert die allgemeinen Hintergründe des Themengebiets der Programmumgestaltungen. Dabei wird auf ihre Entstehung und Entwicklung sowie auf relevante Zusammenhänge zu anderen Themengebieten eingegangen. Anschließend werden Programmabhängigkeitsgraphen vorgestellt, an die sich die in dieser Arbeit verwendete Quelltextabstraktion anlehnt. Als wichtigste technische Grundlage der Arbeit werden die Java Development Tools erklärt, auf die die Umsetzung und Evaluation der Arbeit aufbaut. Schließlich werden die verwandten Arbeiten und ihre Ähnlichkeiten und Unterschiede zu dieser Arbeit präsentiert.

2.1 Quelltextumgestaltung

Wie Philipps und Rumpe in ihrer Arbeit über die Ursprünge von Programmumgestaltungen einleiten, gab es in der Vergangenheit kaum Erfindungen, die aus einem *«Big Bang»* hervor gingen [88]. Stattdessen evolvieren neue Ideen aus voran gegangenen und werden von vielen Arbeitsgruppen und Individuen sowie verwandten und fremden Domänen und Techniken beeinflusst, bis sie sich etabliert haben und ihnen schließlich ein passender Name gegeben wird.

Den Quelltextumgestaltungen voraus gegangen sind u. a. verschiedene Arbeiten im Feld der Programmtransformationen, die beim Kompilieren, Generieren, Synthetisieren, etc. von Programmen eingesetzt werden [10, 2, 127, 87]. Compiler nutzen beispielsweise vollständig automatisch ablaufende Programmtransformationen, um eine Hochsprache in eine Maschinensprache umzuwandeln und sie in Hinblick auf Speicherverbrauch und/oder Prozessorauslastung zu optimieren [2, 127]. Der von Bauer, Partsch et al. entwickelte Ansatz der Transformationsprogrammierung zielt hingegen darauf ab, halbautomatisch, komplette Programme auf Basis von formalen Spezifikationen zu generieren, die dadurch per Konstruktion korrekt sind [86, 87, 17]. Allerdings können die genannten Ansätze klar von Quelltextumgestaltungen unterschieden werden, da letztere weder vollautomatisch ablaufen noch das Ziel haben, ein fertiges Programm anhand einer Spezifikation zu generieren [9, 11, 101]. Nichtsdestotrotz nutzen Quelltextumgestaltungen viele Techniken, die zuvor im Übersetzerbau, der Transformationsprogrammierung u. a. entwickelt wurden.

Als eigenes Forschungsfeld in der Informatik hat sich das Thema der Quelltextumgestaltung von Programmen erst später entwickelt. Lange war die Terminologie unklar und wurde von vielen Beteiligten uneinheitlich verwendet. Chikofsky [20] etablierte 1990

eine einheitliche Terminologie und definierte den Begriff *Umstrukturierung* für klassische Programmiersprachen als

> *« To restructure a software is to change its internal structure without affecting its external behavior. »*
>
> — Chikofsky, 1990 [20]

Diese Definition bezieht sich allerdings nur auf klassische Programmiersprachen außerhalb des objektorientierten Kontexts. Kurze Zeit später, 1992, übertrug daher Opdyke [84] das Mittel der Umstrukturierungen auf objektorientierte Programmiersprachen (kurz: OO-Sprachen) und nannte es *Umgestaltung*, im Englischen *Refactoring*. Opdyke führte eine erste, vollständige Umsetzung des Umgestaltungsgedankens von OO-Sprachen durch und nutzte dafür klar definierte Vorbedingungen für jede Umgestaltung. Außerdem übertrug er den Begriff der Verhaltensgleichheit auf OO-Programme.

> *« Refactoring is basically the object-oriented variant of restructuring. »*
>
> — Mens und Tourwé, 2004 [71]

1999 kategorisierte und spezifizierte Fowler alle bis dahin bekannten und etablierten Umgestaltungen im Standardwerk für Quelltextumgestaltungen *Improving the Design of Existing Code* [35]. Dabei unterschied er zwischen insgesamt sieben Anwendungszwecken, für die Umgestaltungen eingesetzt werden können. Ziel war ein Katalog aller Umgestaltungen, der Entwicklern konkrete Hinweise geben kann, um nachteilhafte Programmstrukturen effektiv eliminieren zu können. Aus seiner Arbeit auf dem Gebiet der Quelltextumgestaltungen entsprang die wohl bekannteste Definition von Umgestaltungen:

> *« Refactoring (noun): a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior. »*
>
> — Fowler, 1999 [35, S. 46]

Das Ziel von Quelltextumgestaltungen ist die Verbesserung der Quelltextqualität. Sowohl während der Entwicklung, als auch während der Wartung von Programmen und besonders im Rahmen von agilen Entwicklungsprozessen sind Quelltextumgestaltungen wichtige und alltägliche Arbeitsabläufe. Eingesetzt werden Umgestaltungen um die Lesbarkeit und Wartbarkeit von Programmen zu verbessern, sowie ihre Fehleranfälligkeit zu reduzieren [60]. Der Begriff Quelltextumgestaltung wird ferner sowohl im allgemeinen als auch im speziellen Kontext verwendet. Allgemein bezeichnet er unspezifische Umgestaltungen am Programmquelltext, die vor allem auf eine Umstrukturierung abzielen, ohne das Verhalten des Programms maßgeblich zu verändern. Diese allgemeinen Änderungen umfassen Umstrukturierungen jedweder Art und können dazu führen, dass während der Phase der Programmumgestaltung das Programm in weiten Teilen funktionsuntüchtig ist. Je nachdem, wie weitreichend die durchzuführenden Quelltextumgestaltungen sind, kann sich diese Phase über mehrere Tage hinziehen. Allgemeine Umgestaltungen können nur sehr eingeschränkt von Entwicklungswerkzeugen unterstützt werden, da die durchgeführten Änderungen zu abstrakt oder weitreichend sind.

Von der Quelltextumgestaltung im allgemeinen Kontext können jene im speziellen Kontext abgegrenzt werden [27]. Diese spezifischen Umgestaltungen sind klar definiert und ändern den Quelltext innerhalb enger Grenzen. Es existieren spezifische Instanzen von Umgestaltungen wie das Umbenennen von Bezeichnern wie Variablen oder Methoden, oder das Verschieben solcher Quelltextelemente an andere Programmstellen. Andere gebräuchliche Umgestaltungen sind das Extrahieren, Einfügen oder Konvertieren von Variablen, oder das Ändern von Methodensignaturen [78, Balkendiagramm Abb. 6]. Spezifische Umgestaltungen können zwar manuell durchgeführt werden, jedoch bieten etablierte Programmierwerkzeuge entsprechende Funktionalitäten, um spezifische Umgestaltungen auf Knopfdruck automatisch durchführen zu können. Die dafür angebotenen Funktionen werden bereits sehr häufig von Entwicklern eingesetzt [60, 78]. Bei dieser automatischen Durchführung gibt der Entwickler an, an welcher Stelle eine Umgestaltung durchgeführt werden soll und – falls erforderlich – zusätzliche notwendige Informationen. Während allgemeine Umgestaltungen zu überschaubaren Verhaltensänderungen des Programms führen können, die in der Praxis auch toleriert werden [60], wird von spezifischen Umgestaltungen erwartet, dass durch sie das Programmverhalten nicht modifiziert wird, besonders dann, wenn sie von einem Werkzeug durchgeführt werden.

Unverändertes Programmverhalten ist eine wesentliche Eigenschaft von spezifischen Umgestaltungen und wird auch unveränderte Semantik genannt. Gemeint ist damit das beobachtbare Verhalten eines Programms bezüglich seines Ein-/Ausgabeverhaltens. Nicht gemeint sind Speicher- und Prozessorverhalten, die durch Umgestaltungen verändert werden können. Griswold [45] stellte in seiner Arbeit Umgestaltungen für die Programmiersprache *Scheme* vor und beschäftigte sich dabei auch mit der Frage, wie er zeigt, dass seine Umgestaltungen semantikerhaltend sind. Dafür griff er auf Programmabhängigkeitsgraphen (PDGs) zurück, die über die Eigenschaft der *Sequenzkongruenz* verfügen. Wie Horwitz [53] und Yang [128] gezeigt haben, gilt für zwei sequenzkongruente PDGs, dass sie immer äquivalente Programme repräsentieren. Da Griswolds Umgestaltungen auf PDGs abgebildet werden konnten und dort nur sequenzkongruente PDGs erzeugten, waren ebenso seine Programmumgestaltungen semantikerhaltend. In späteren Arbeiten über Quelltextumgestaltungen wurden andere Wege wie formale Beweise oder Argumentationen verwendet, um den Erhalt der Semantik zu zeigen [120, 97, 110]. Zusätzlich wurden auch Ansätze von Soares et al. erforscht, die bestehende Umgestaltungen verifizieren können [103, 102]. Soares' Arbeit basiert auf einem Testansatz, bei dem die Umgestaltungen mithilfe von generierten Tests ausgiebig untersucht und ausgewertet werden.

Wenn Quelltextumgestaltungen werkzeugunterstützt durchgeführt werden, müssen sie in semi- und vollautomatische Ansätze unterschieden werden. Semiautomatische Ansätze arbeiten mithilfe von Nutzereingaben, während vollautomatische Ansätze autonom Umgestaltungen vornehmen. In ihrer sehr umfangreichen Überblicksarbeit zum Thema Quellumgestaltungen identifizieren Mens und Tourwé [71] drei Phasen, die mit dem Umgestaltungsprozess verbunden sind:

1. Feststellen, ob ein Programm umgestaltet werden sollte.

2. Spezifische Umgestaltungen und ihre Anwendungsstellen im Programm identifizieren.

3. Die identifizierten Umgestaltungen anwenden.

Vollautomatische Werkzeuge [28, 77, 76] führen alle genannten Schritte selbstständig durch. Das heißt, dass sie nur einmal gestartet werden müssen und das umzugestaltende Programm solange modifizieren, bis aus Sicht des Werkzeugs kein Anlass mehr zu weiteren Umgestaltungen besteht. Der Vorteil dieser Ansätze ist, dass die Qualität der Quelltexte bezogen auf bestimmte Metriken auf diese Weise vollautomatisch angeglichen und verbessert werden kann [50, 82, 83]. Diese Ansätze kommen als Assistent während des Programmierens oder als Quelltextverbesserungswerkzeug zum Einsatz. Allerdings leiden vollautomatische Werkzeuge unter Nachteilen wie der Einführung bedeutungsloser Variablennamen, die durch fehlende Eingaben des Nutzers entstehen können [19]. Ebenfalls können spezifische Quelltextkonstruktionen, die Eigenheiten der Anwendungsdomäne modellieren, nicht beachtet werden. Dadurch kann unter Umständen ein Quelltext durch vollautomatische Umgestaltungen auch unverständlicher werden.

Es existieren verschiedene Ansätze um festzustellen, ob ein Programm umgestaltet werden soll. Einer der bekanntesten beruht auf sogenannten *Bad-Smells* (a. d. Englischen: *bedenkliche Programmteile*) [35, 39, 130, 27] im Quelltext. Sie bezeichnen eine Stelle im Quelltext, die als unelegant gilt und den Konventionen sauberen Programmierens widerspricht. Ein bekanntes Beispiel sind duplizierte Quelltextsequenzen. Diese sorgen für erhöhten Wartungsaufwand im Fall von Programmerweiterungen und erhöhen die Gefahr, beim Bereinigen von Programmierfehlern Fehlerstellen zu übersehen oder zu vergessen. Werkzeuge wie jCOSMO [29] können eingesetzt werden, um bedenkliche Programmteile zu finden und so dem Entwickler zu helfen, Anwendungsstellen für Umgestaltungen zu finden. Dabei ist auch wichtig, welche Umgestaltungen in welcher Reihenfolge zur Behebung dieser Programmteile eingesetzt werden [68].

Auch die von Gamma [38] beschriebenen Entwurfsmuster können dazu genutzt werden, sinnvolle Programmstellen zu identifizieren, an denen Umgestaltungen durchgeführt werden sollten. Entwurfsmuster sind wiederkehrende Strukturen die dazu dienen, bestimmte Anforderungen in Programmen umzusetzen. Durch ihren Einsatz wird das Entwickeln, Strukturieren, Verstehen und damit auch das Warten von Programmen vereinfacht. Ebenfalls beschreibt Gamma Antimuster, deren Einsatz zu negativen Effekten führt und daher vermieden werden sollte. Sie eignen sich deswegen auch als hilfreiche Indikatoren um festzustellen, wo Quelltextumgestaltungen sinnvoll eingesetzt werden könnten. Meyer [72, S. 40-63] beschreibt Indizien zur Quelltextqualität von objektorientierten Software in je fünf Kriterien, Regeln und Prinzipien. Diese Indizien sind teilweise von einander disjunkt und dienen besonders dazu, den Quelltext modular zu strukturieren.

Um festzustellen, ob und wo ein Programm umgestaltet werden muss, werden ferner auf Heuristiken basierende Suchverfahren eingesetzt. Bei der Suche sind eine Vielzahl von Metriken ausschlaggebend, da sich die Metriken auch widersprechen können und durch mehrere Metriken mehr Verbesserungsstellen im Programm gefunden werden können. Jensen et al. [56] zielen anhand von Metriken darauf ab, nicht nur einzelne Umgestaltungen durchzuführen. Stattdessen sollen ihre Metriken Programmstellen identifizieren, an denen gleich ganze Entwurfsmuster im Programm eingeführt werden können. Um diese Entwurfsmuster in den Quelltext einzuführen, werden schließlich gleich eine Reihe von Umgestaltungen durchgeführt.

Schließlich ist ein weiterer Einsatzzweck, Umgestaltungen einzusetzen, eine vollautomatische, strukturelle Transformation eines Programms. Moghadams et al. [75] nutzen UML-Modelle, um die bestehende Programmstruktur zu interpretieren. Mithilfe eines

Soll-UML-Modells, das bestimmte Anforderungen des Entwicklers besser umsetzt, werden zunächst Unterschiede zwischen beiden Modellen festgestellt. In einem nächsten Schritt werden notwendige Quelltextumgestaltungen identifiziert, die das Programm in die Soll-UML-Struktur überführen. Moghadams et al. können auf diese Weise die anvisierten Soll-UML-Modelle zu 90% umsetzen.

Die Vielzahl von Umgestaltungen werden in dieser Arbeit in zwei Gruppen unterteilt: deklarative und kontrollflusssensible Umgestaltungen. *Deklarative Umgestaltungen* ändern deklarative Programmeigenschaften wie die Bezeichner von Variablen oder Methoden. Ebenfalls können sie deklarierte Typen konvertieren, die Deklarations- oder Implementierungsorte von Variablen oder Methoden verschieben oder Klassen- oder Schnittstellenhierarchien ändern. Davon abgegrenzt werden hier *Kontrollflusssensible Umgestaltungen* modifizieren im Gegensatz dazu den Kontrollfluss eines Programms. Hierzu gehören Umgestaltungen, die zum Verschieben von Anweisungen oder Ausdrücken führen. Das sind insbesondere Umgestaltungen, die Methoden extrahieren oder einfügen, die lokale Variablen einführen oder eliminieren oder Anweisungen bzw. Ausdrücke direkt vertauschen. Ebenfalls kann das Ändern von Methodensignaturen kontrollflusssensibel sein, wenn zwei oder mehr Argumente der Methode miteinander vertauscht werden. Grund dafür ist, dass mit der Änderung der Signatur einer Methode eine Anpassung all ihrer Aufrufe einhergeht, die zum Vertauschen von Ausdrücken führen kann. Von den 68 von Fowler [35] herausgearbeiteten Umgestaltungen können insgesamt 15 als ausschließlich deklarativ eingeordnet werden. Die restlichen 53 Umgestaltungen sind vollständig (bspw. das Extrahieren von Ausdrücken) oder teilweise kontrollflusssensibel (bspw. das Ändern einer Methodensignatur).

2.2 Verwandte Arbeiten

Für diese Arbeit sind vor allem die Arbeiten der Autoren Griswold, Steimann und Schäfer interessant, da sie thematisch eng mit dieser Arbeit verwandt sind. Der hier vorgestellte Ansatz wird von diesen Arbeiten anhand der folgenden drei Kriterien abgegrenzt:

Üblicherweise unterstützen die vorgestellten Arbeiten nicht alle erdenklichen Umgestaltungen. Stattdessen fokussieren sie sich auf spezielle Umgestaltungen, wie das Umbenennen oder Einfügen/Extrahieren von Variablen. Um eine hierfür klare Unterscheidung durchführen zu können, (1) werden die Umgestaltungen in deklarative und kontrollflusssensible Umgestaltungen unterteilt. In die Gruppe der deklarativen Umgestaltungen fallen Umgestaltungen, die Bezeichner im Quelltext wie bspw. Variablennamen ändern. In die Gruppe der kontrollflusssensiblen Umgestaltungen gehören diejenigen Umgestaltungen, die den Kontrollfluss einer Methode ändern, wie bspw. das Einfügen/Extrahieren von Variablen.

Neben dieser Unterscheidung wird diese Arbeit (2) bzgl. ihrer Herangehensweise von den Arbeiten von Griswold, Steimann und Schäfer abgegrenzt. Während Steimann ebenfalls einen Gleichungssystem basierten Ansatz nutzt, verwenden Griswold und Schäfer einen Muster basierten Ansatz zum Ermitteln der Umgestaltungen.

Schließlich (3) werden die vorgestellten Arbeiten auch in Hinsicht auf ihren Verifikationsansatz (argumentativ, formell und/oder testbasiert) und ihre Evaluation (Anzahl möglicher Einsatzorte im Quelltext und Geschwindigkeit) vorgestellt. In Anlehnung an die Arbeiten von Griswold führen wir unsere Umgestaltungen auf bekannte Transformationen zurück, die semantikerhaltend sind.

2.2.1 Griswolds Arbeiten

Griswold [45] nutzt Programmabhängigkeitsgraphen (PDGs) [31] und stellt semantikerhaltende PDG-Transformationen vor. Er nutzt die Programmiersprache *Scheme* und zeigt an ihr exemplarisch die Anwendbarkeit seiner Umgestaltungen. In Griswolds Ansatz werden nur dann Programme umgestaltet, wenn die zugehörigen PDGs ebenfalls entsprechend semantikerhaltend transformiert werden können. Um zu gewährleisten, dass die Semantik erhalten bleibt, bleiben bei Griswolds Transformationen alle PDG-Operationen unverändert und es werden keine Abhängigkeiten entfernt. Ferner werden PDG-Subgraphen nur dann ersetzt, wenn die Ersetzung sequenzkongruent [5] ist. Griswold spaltet seine Umgestaltungen in diese kleinen, atomaren Transformationen, die sowohl auf PDGs als auch auf die Programmiersprache Scheme angewendet werden können. Er stellt insgesamt sechs solcher atomaren Transformationen vor, u. a. die Transitivität und Distribution oder die Substitution. Die ersten beiden sind notwendig für das Einfügen von Variablen und die letztere wird beim Umbenennen von Variablen verwendet.

In späteren Arbeiten erweitert Griswold seine technische Unterstützung für *Scheme*-Programme, um wiederkehrende und zeitraubende Arbeiten beim Umgestalten zu automatisieren [47], [46]. Sein Software-Werkzeug ist damit eins der ersten Umgestaltungswerkzeuge, wie sie heute als Teil etablierter IDEs Gang und Gäbe sind.

2.2.2 Schäfers Arbeiten

Schäfer et al. [97] betrachten sowohl deklarative als auch kontrollflusssensible Umgestaltungen, wie das Umbenennen und Einfügen von Variablen. Die semantische Korrektheit der Umgestaltungen basiert auf dem Erhalt von Abhängigkeiten, Sprachvereinfachungen und sog. Mikro-Refactorings. Für deklarative Umgestaltungen wie dem Umbenennen von Variablen stellt der Ansatz eine neue Herangehensweise vor. Diese nutzt einen vereinheitlichten Ansatz, um einerseits die Bindungen von Bezeichnern und Bewahrung von Bindungen zu ermöglichen und andererseits Mehrfachvergabe von Bezeichnern zu verhindern. Zusätzlich nutzen sie Kontroll- und Datenflussabhängigkeiten, um korrekte Umgestaltungen zu ermöglichen, die den Quelltext umgestalten und/oder Quelltextelemente verschieben können. Sowohl diese Bedingungen als auch die Kontroll- und Datenflussabhängigkeiten werden von Schäfer et al. allgemein als Abhängigkeiten bezeichnet. Dafür wurde ein einheitliches Konzept entwickelt, das in ihren verschiedenen Umgestaltungen wiederverwendet wird. Ferner arbeitet der Ansatz auf Sprachvereinfachungen: Semantischer Zucker wird in einfachere Sprachkonstrukte überführt und Sprachbeschränkungen und -vereinfachungen werden eingeführt. Die Umgestaltungen selbst bestehen aus Mikro-Refactorings, die von verschiedenen Umgestaltungen wiederverwendet werden.

In [96] nutzen Schäfer et al. Attributgrammatiken, um einen wichtigen Teil des Umgestaltungsansatzes zu verifizieren. Dabei konzentrieren sie sich auf das Ermitteln von Bindungen zwischen Bezeichnern im Programm. Dieser Teil des Ansatzes wird im Theoremprüfer Coq formalisiert und bewiesen.

In [98] untersuchen Schäfer et al. den Einfluss von nebenläufigen Programmen auf mögliche Semantikänderungen durch Umgestaltungen. Sie erweitern einige Java-Umgestaltungen derart, dass diese bei ihrer Durchführung die Nebenläufigkeit mehrerer Threads beachten und somit weiterhin semantikerhaltend sind. Schäfer et al. betrachteten u. a. die Umge-

staltungen *Einfügen einer lokalen Variablen* und *Pull-Up-Member*. Ihre Arbeit haben sie mittels eines schriftlichen Beweises verifiziert.

In [100] wird eine Umgestaltung eingeführt, um Java-Sprachkonstrukte nebenläufiger Programme schneller ersetzen zu können. Die Arbeit fokussiert sich darauf, Semaphoren, die durch die Java-Schlüsselwörter `synchronize`/`synchronized` gekennzeichnet werden, durch Wiedereintrittssperren (engl. reentrant locks) (siehe [85]) zu ersetzen. Schäfer et al. evaluieren den Ansatz, indem sie an ausgewählten Beispielen zeigen, wieviele Synchronisationskonstrukte sich maximal ersetzen ließen. Sie zeigen allerdings nicht, dass die so umstrukturierten Programme weiterhin das gleiche Ein-/Ausgabeverhalten aufzeigen.

Im Gegensatz zu Schäfers Ansatz konzentriert sich diese Arbeit ausschließlich auf kontrollflusssensible Umgestaltungen. Der größte Unterschied ist jedoch, dass diese Arbeit ein Gleichungssystem zum Ermitteln von Lösungen von Umgestaltungen nutzt. Wie Schäfer ermittelt der hier vorgestellte Ansatz notwendige Bindungen zum Erhalt der Programmsemantik. Die Bindungen werden in ein Gleichungssystem überführt und auf diese Weise bewahrt. Die Evaluation wird an existierenden Programmen durchgeführt und zeigt durch vorhandene Test-Suiten, dass die Umgestaltungen keine Verhaltensänderungen herbei geführt haben.

2.2.3 Tips Arbeiten

Tip et al. [121], [120] fokussieren sich auf deklarative Umgestaltungen wie das Extrahieren einer Schnittstelle. Ihr Ansatz beruht auf der Identifikation und dem Erhalt von Typabhängigkeiten, die vor und nach der Durchführung einer Umgestaltung im Programm beibehalten werden müssen. Nur wenn das umgestaltete Programm alle Abhängigkeiten des Ursprungsprogramms einhält, ist die Umgestaltung des Programms semantikerhaltend. Wie Schäfers Ansatz bestimmen Tip et al. das umgestaltete Programm prozedural und finden für einige Umgestaltungswünsche keine semantikerhaltende Lösung.

Ebenso wie Tip et al. basiert der Ansatz dieser Arbeit auf Programmabhängigkeiten, die sowohl vor als auch nach der Umgestaltung gültig sein müssen. Allerdings nutzt diese Arbeit im Gegensatz für die Suche nach semantikerhaltenden Umgestaltungen einen Constraint-Löser. Hierdurch erhält der Ansatz ein größeres Maß an Flexibilität und kann in mehr Situationen eine Umgestaltung semantikerhaltend durchführen.

2.2.4 Steimanns Arbeiten

Steimann et al. [110] verwenden im Gegensatz zu den bereits genannten Arbeiten Constraint-Löser, um aus Umgestaltungswünschen das umgestaltete Programm zu bestimmen. Ihr Ansatz konzentriert sich dabei auf das Erhalten der Bindungen deklarativer Programmeigenschaften wie Attribute oder Methoden. Daher unterstützen sie hauptsächlich Umgestaltungen zum Ändern deklarativer Programmeigenschaften wie das Umbenennen oder verschieben von Variablen oder Methoden. Steimann et al. haben ihre Fortschritte in mehreren Arbeiten vorgestellt. Zunächst stellten sie in [108] einen UML-Ansatz vor, der UML-Modelle auf Gleichungssysteme (engl. Constraint Systems) überführt und löst. In [110] übertrugen Steimann et al. ihren Ansatz auf Programme, die in Eiffel geschrieben sind. Darin wurden drei Umgestaltungen vorgestellt: das Umbenennen von Programmelementen wie Variablen, Typnamen und das Ändern von Zugreifbarkeiten (§6.6). Steimann et al.

führten ihre Evaluation auf Beispielprogrammen mit insgesamt ca. 167kLOC [3] durch. In durchschnittlich weniger als 1 Sekunde könnte ihr Ansatz je nach Refactoring und Beispielprogramm Lösungen zu jeder Umgestaltung berechnen. Die Verifikation ihres Ansatzes und ob die Umgestaltungen semantikerhaltend sind bzw. ob sie die Semantik besser als Vergleichswerkzeuge erhalten, zeigen Steimann et al. argumentativ.

In [99] wird Steimanns Ansatz mit der Arbeit von Schäfer verknüpft, sodass der Ansatz auf Java übertragen werden kann. Die Abhängigkeiten im Quelltext werden von Schäfers Arbeit (mithilfe von JastAddJ [44]) erkannt und dann in Steimanns Ansatz genutzt um das Gleichungssystem anzureichern. Die Evaluation wird anhand von Schäfers bestehender Evaluation durchgeführt und die Umgestaltungen werden mit bestehenden Test-Suiten von Eclipse verglichen und übertreffen diese.

In [112] stellt Steimann eine Erweiterung seines Ansatzes vor um adhoc-Umgestaltungen adaptiv durchzuführen. Der Nutzer führt dabei beliebig gewählte Änderungen am Quelltext durch und legt dabei fest, dass diese Änderungen keine Semantikänderungen hervorrufen sollen. Das Werkzeug erkennt die vorgenommenen Änderungen und schlägt – sofern möglich – weitere Änderungen vor, um die Semantik zu erhalten. In seiner Evaluation zeigt Steimann durch Simulation von Nutzerwünschen, dass der Ansatz technisch praktikabel ist.

In [111] erweitert Steimann seine Arbeit um einen wichtigen Aspekt bzgl. der Bildung des Gleichungssystems. Bisher wurden die Ursprungsprogramme direkt in ein Gleichungssystem übertragen, das anschließend gelöst wurde. In seiner Ergänzung werden Gleichungssysteme erzeugt, die nicht nur das Ursprungsprogramm repräsentieren, sondern auch Modifikationen vom Ursprungsprogramm, die mögliche Lösungen repräsentieren. Durch das größere Gleichungssystem können auch Umgestaltungen berechnet werden, die die Struktur des Gleichungssystems ändern würden. Es wird gezeigt, dass der zusätzliche Berechnungsaufwand in akzeptablen Grenzen liegt und der erweiterte Ansatz somit praktikabel bleibt.

Schließlich wird der Ansatz von v. Pilgrim in [89] auf ein Co-Refactoring übertragen. Dabei können verschiedene und semantisch miteinander verknüpfte Programme und/oder Modelle gleichzeitig umstrukturiert werden. Durch ihre Verknüpfung werden Bedingungen und Abhängigkeiten von beiden Instanzen berücksichtigt, sodass weder das eine noch das andere Modell/Programm fehlerhaft umstrukturiert wird. Der Co-Refactoring-Ansatz wird anhand von frei zugänglichen Programmen, die zumindest teilweise durch Modelle abstrahiert sind, technisch evaluiert.

In der Dissertation von Andreas Thies wird Steimanns Arbeit ausgebaut und für Java 6 eingesetzt. Thies beschäftigt sich ebenfalls mit den deklarativen Umgestaltungen wie dem Umbenennen oder Verschieben von Klassenattributen in der Klassenhierarchie. Die Evaluation wird u. a. an ca. 100'000 Beispielen durchgeführt und anhand von Metriken ausgewertet. Die Dissertation von Thies, die Arbeiten von Steimann, v. Pilgrim et al. und die vorliegende Arbeit haben sehr ähnliche Fragestellungen und verfahren nach einem ähnlichen Lösungsansatz. Die Vorarbeiten übertragen umzustrukturierende Programme in Gleichungssysteme, aus deren Lösungen semantikerhaltende Transformationen abgelesen werden können. Daher orientiert sich die vorliegende Arbeit an der Methodik und Nomenklatur, sofern dies möglich ist. Im Gegensatz zu Thies konzentriert sich diese Arbeit auf *kontrollflusssensible* Umgestaltungen für Java 7. Obwohl beide Arbeiten den Ansatz verfolgen, Constraint-Programmierung zum Finden von Umgestaltungen zu

nutzen, unterscheiden sich die Lösungswege deutlich. Der grundlegende Unterschied beider Arbeiten ist, dass deklarative Umgestaltungen keine Reihenfolgen berechnen müssen. Dem gegenüber müssen kontrollflusssensible Umgestaltungen hauptsächlich die Positionen von Anweisungen und Operanden – also Reihenfolgen – berechnen. Dabei sind bspw. Datenflussabhängigkeiten entscheidend für die Durchführung von Umstrukturierungen.

2.3 Programmabhängigkeitsgraphen

Programmabhängigkeitsgraphen (PDG, engl. Program Dependency Graph) wurden von Ferrante et al. [31] vorgestellt und gingen aus Kontroll- und Datenflussgraphen hervor. Die verschiedenen Varianten von Programmabhängigkeitsgraphen haben gemeinsame Wurzeln in der Arbeit von Kuck et al. [64]. PDGs repräsentieren den Maschinencode als Multigraph, wobei PDG-Knoten sogenannte Compiler-Triplets [2] abstrahieren. In Multigraphen können dieselben Knoten durch mehrere Kanten miteinander verbunden werden. PDG-Kanten ordnen Quell- und Zielknoten transitiv und können in Kontroll- und Datenflusskanten unterteilt werden. Üblicherweise stellen Abhängigkeitsgraphen nur einen Ausschnitt eines Programms dar. Da die Graphen schnell sehr groß und unübersichtlich werden, sind sie auch in dieser Arbeit auf die gerade wichtigen Sachverhalte reduziert.

> **Programmabhängigkeitsgraph (PDG):** *Ein PDG eines Programms besteht aus einer Menge N von Knoten sowie einer Menge E von Kanten. Knoten repräsentieren Programmoperationen und Kanten repräsentieren Programmabhängigkeiten zwischen zwei Knoten.*

Die in der Arbeit genutzte Quelltextabstraktion baut auf PDGs und der Eigenschaft der Sequenzkongruenz auf. Sie benutzt die gleichen Programmabhängigkeiten und Transformationen und erweitert beide. Daher werden im Folgenden zunächst die wichtigsten Elemente von PDGs wie Knoten und Kanten eingeführt. Anschließend wird die Sequenzkongruenz erklärt und schließlich werden die PDG-Transformationen und Einschränkungen dargelegt. PDGs werden hier jedoch nicht zur formalen Verifikation des Ansatzes genutzt. Stattdessen dienen sie dazu, die Programmabhängigkeiten und Transformationen der Quelltextabstraktion zu erklären. Die folgenden Definitionen der PDGs und seiner Transformationen richten sich nach der Arbeit von Griswold, der PDGs ebenfalls als informelle Basis für seine Quelltexttransformationen genutzt hat [45].

2.3.1 PDG-Elemente

Zu jedem Programm existiert ein PDG, der als eine Menge von Knoten N und Kanten E spezifiziert wird. Eine Kante $e = FD(u, v)$ besteht aus den Start- und Zielknoten u und v und verfügt außerdem über einen Kantentyp, in diesem Fall FD.

> **PGD-Kante:** *Eine PDG-Kante e vom Knoten u zu v und vom Typ T wird wie folgt notiert.*
>
> $$e = T(u, v)$$

Die Kantentypen können in die beiden Kategorien Kontroll- und Datenflussabhängigkeiten unterteilt werden. Zusätzlich können Datenflusskanten mit dem Attribut *loop-carried* gekennzeichnet werden, wenn sie durch einen Schleifendurchlauf im Quelltext verursacht werden.

Knoten

Knoten repräsentieren einfache Operationen und können ein- und/oder ausgehende Kanten besitzen. Sie können als Compiler-Triplets verstanden werden, die aus Operation, Argumenten und Resultat bestehen [2]. Compiler-Triplets werden auch als 3-Adresscode bezeichnet [2], da sie nur Operationen mit maximal zwei Operanden unterstützen. Die Operation $Op(n)$ eines Knotens n ist dann der Operationsaufruf oder eine Sprunganweisung.

Kontrollflussabhängigkeiten

Kontrollflussabhängigkeiten entstammen Kontrollflussgraphen (engl. CFG, control flow graph) [2], [31]. Sie repräsentieren Kontrollflussabhängigkeiten, die durch bedingte (`if`-Anweisung) oder iterative (`while`-Anweisung) Quelltextblöcke entstehen. Sie werden aus Programmsprüngen erstellt, indem alle Sprünge und Sprungziele durch Kontrollflusskanten ersetzt werden. Kontrollflusskanten $e = CD(u, v)$ werden mit CD beschriftet und bei bedingten Anweisungen mit `true` oder `false` gekennzeichnet. Existiert bspw. eine Kontrollflusskante $e = CD(u, v)_{true}$ von einer `if`-Bedingung zu einer Anweisung in ihrem Dannblock, wird die zugehörige Kontrollflusskante mit einem *true* gekennzeichnet. Mit *true* beschriftete Kontrollflusskanten werden auch von jedem `while`-Bedingungsknoten zu jedem Knoten seines Schleifenrumpfs erzeugt.

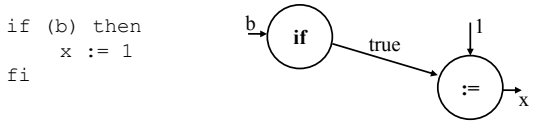


Abbildung 2.1: PDG-Beispiel: Kontrollflusskante

Kontrollflusskanten ordnen die Knoten einer Methode, sodass Knoten mit gleichen Bedingungen (z. B. eine `if`-Anweisung) Kanten mit gleichen Prädikaten besitzen. Allerdings geht dadurch die Information über Blockgrenzen im PDG verloren. Denn von zwei Knoten mit identischen Kontrollflussprädikaten eines PDGs kann nicht mehr ermittelt werden, ob ihre Quelltextanweisungen in zwei verschiedenen oder in der selben `if`-Anweisung gestanden haben, sofern die `if`-Bedingungen identisch sind.

Datenflussabhängigkeiten

Datenflusskanten repräsentieren Datenabhängigkeiten jeweils zwischen zwei Knoten im PDG. Datenabhängigkeiten entstehen durch Lese-/Schreibvorgänge der zugehörigen Knoten. Diese Datenzugriffe entstehen direkt über Variablen, Methodenrückgaben oder Rückgaben von Unterausdrücken in verschachtelten Ausdrücken. Seiteneffekte und Daten-Aliasing können hier nicht auftreten, da die von den Knoten repräsentierten Compiler-Triplets seiteneffektfrei sind und in der Maschinensprache kein Aliasing auftritt.

Datenflusskanten $e = FD(u, v)$ werden wie Kontrollflusskanten notiert und mit dem zugehörigen Kantentyp der Datenflussabhängigkeit ausgezeichnet. Zu den Kantentypen zählen: `AD` (engl. anti dependency), `DO` (engl. data order dependency), `OD` (engl. output dependency) und `FD` (engl. flow dependency). FD-Kanten markieren den Datenfluss von der Definition hin zur Verwendung einer Variablen. Komplementär zu FD-Kanten starten AD-Kanten von einer Variablenverwendung und enden bei ihrer nächsten Neudefinition. DO-Kanten sortieren aufeinander folgende Variablendefinitionen. Über bedingte Kontrollflüsse (bspw. durch `if`-Anweisungen) können unterschiedliche Variablendefinitionen einen Verwendungsort erreichen. DO-Kanten stellen sicher, dass sich Variablendefinitionen in der richtigen Reihenfolge überschreiben. OD-Kanten sortieren wie DO-Kanten aufeinanderfolgende Variablendefinitionen. Im Unterschied zu ihnen sortieren sie jedoch unbedingte Definitionen. Das bedeutet, dass in jedem Fall die der letzten Definition vorangehenden Definitionen überschrieben werden. [53]

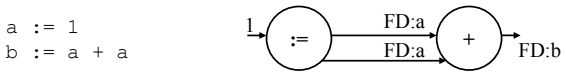


Abbildung 2.2: PDG-Beispiel: Addition

Sofern vorhanden, werden Datenflusskanten mit den zugehörigen Variablensymbolen beschriftet. Der Datenfluss der Kante $e = FD(u, v)_s$ entsteht z. B. durch die Variable `s`, die von u geschrieben und von v gelesen wird. Falls Daten ohne Variable, sondern als direkte Rückgabe eines Ausdrucks verwendet werden, wird die Datenflusskante nicht mit einem Variablensymbol beschriftet. Entsteht eine Datenflussabhängigkeit durch einen iterativen Kontrollfluss, ausgelöst durch eine Schleife, werden diese Kanten mit *loop-carried* markiert. Solche schleifengebundene Abhängigkeiten sind die einzigen Abhängigkeiten, die zwei Knoten entgegen der Richtung der restlichen Kanten sortieren.

2.3.2 Sequenzkongruenz

Die *Sequenzkongruenz* geht auf Horwitz [53] und Yang [128] zurück und beschreibt, wann zwei PDGs äquivalentes Programmverhalten repräsentieren. Unter dem Programmverhalten wird das Ein-/Ausgabeverhalten und Terminierungsverhalten verstanden. Laufzeit- und Speicherverhalten werden außer Acht gelassen.

Sind die PDGs zweier Programme sequenzkongruent, so gelten ihre Programme laut Yang [128] als semantisch äquivalent. Yang [128] nutzt dafür sogenannte PRGS, eine Variante von PDGs mit Anweisungen in SSA-Form [22]. Yang nutzen zum Erkennen der Sequenzkongruenz zweier PDGs einen 3-schrittigen Algorithmus. Nach Yang [128] werden die Daten- und Sequenzkongruenz wie folgt definiert.

> **Datenkongruenz:** *Zwei PDG-Knoten sind datenkongruent wenn gilt, dass ihre Knotenoperatoren und die Daten ihrer eingehenden Datenflusskanten gleich sind.*

> **Sequenzkongruenz (PDG-Knoten):** *Zwei PDG-Knoten sind sequenzkongruent wenn gilt, dass sie datenkongruent und die Prädikate ihrer eingehenden Kontrollflusskanten gleich sind.*

Sequenzkongruenz (PDG-Transformationen): *Eine PDG-Transformation ist sequenzkongruent wenn gilt, dass ihre Anwendung auf einen PDG immer einen dazu sequenzkongruenten PDG' erzeugt.*

Auch PDG-Transformationen können als sequenzkongruent bezeichnet werden, wenn ihre Anwendung auf einen PDG immer einen dazu sequenzkongruenten PDG' erzeugt. Solche sequenzkongruenten Transformationen bewahren also alle Abhängigkeiten sowie Kontrollflussprädikate und ändern die Knotenoperationen nicht [128]. Sie werden im nächsten Kapitel beschrieben.

2.3.3 Transformationen

Sequenzkongruente PDG-Transformationen werden genutzt, um Quelltexte semantikerhaltend umzugestalten. Die hier vorgestellten Transformationen wurden von Griswold [45] verwendet und sind meist reversibel. Im Folgenden werden vier seiner Transformationen am Beispiel vorgestellt. Seine Transformationen *Indirektion* und *Substitution* werden hier nicht vorgestellt, da sie für die Arbeit nicht relevant sind. Die PDG-Transformationen, mit denen eine *Abhängigkeit*, eine *Kantenbeschriftung* oder ein *leerer Knoten hinzugefügt* werden kann, wurden von Griswold nicht, implizit oder nur teilweise vorgestellt. Sie werden jedoch in dieser Arbeit benötigt und daher ebenfalls im Folgenden eingeführt. Um die Transformationen zu beschreiben, wird hier wie auch von Griswold die von Podgurski und Clarke [90] verwendete Notation genutzt.

PDG-Pfad 1: *Sind im PDG v_i Knoten und sind e_j Kanten von v_j nach v_{j+1} mit $0 \leq i \leq n+1$ und $0 \leq j \leq n$, dann wird der Pfad von v_0 über e_j nach v_{n+1} wie folgt notiert. Einzelne Knoten können ebenfalls Pfade sein.*

$$v_0 e_0 v_1 \ldots e_n v_{n+1}$$

PDG-Pfad 2: *Sind im PDG u und v Knoten und ist W eine Pfadmenge, und sind u bzw. v stets Vorgänger bzw. Nachfolger der Pfade in W, dann werden die Pfade von u über W nach v wie folgt notiert.*

$$uWv$$

Abhängigkeit hinzufügen/entfernen

Wenn zwischen den Knoten u und v eines PDGs noch keine Abhängigkeitskante existiert, können sie durch zusätzliche Abhängigkeitskanten strenger geordnet werden. Ist der PDG bereits total geordnet, ist jede zusätzliche Abhängigkeitskante redundant, aber nicht verboten. PDG-Kanten dürfen nur dann zum Graphen hinzugefügt werden, wenn sie keine Schleifen erzeugen. Aus einem PDG dürfen ausschließlich zusätzlich hinzugefügte Kanten wieder entfernt werden.

Abhängigkeit hinzufügen/entfernen: *Eine Kontrollflusskante e_{true}, deren Prädikat immer wahr ist, kann zwei Knoten u und v desselben Kontrollflussknotens p_0, die zueinander keine transitive Abhängigkeit $\overset{\rightsquigarrow}{e}$ besitzen, hinzugefügt oder entfernt werden.*

$$p_0\{u, v\} \Leftrightarrow p_0\{u e_{true} v\}, \quad \nexists u \overset{\rightsquigarrow}{e} v \wedge \nexists v \overset{\rightsquigarrow}{e} u$$

Leeren Knoten hinzufügen/entfernen

Ein leerer Knoten v im PDG entspricht der Compiler-Anweisung `nop` im Quelltext und hat folglich keine Auswirkungen auf das Ein-/Ausgabeverhalten. `nop`-Operationen werden ausschließlich über Kontrollabhängigkeitskanten in einen PDG integriert. Leere Knoten dürfen nur dann aus einem PDG entfernt werden, wenn sie mit keiner Kante verbunden sind.

> **Leeren Knoten hinzufügen/entfernen:** *Die Menge der Nachfolger eines Kontrollflussknotens p_0 kann nach der folgenden Bedingung um einen Knoten v erweitert oder reduziert werden.*
>
> $$p_0\{\} \Leftrightarrow p_0\{v\}, \;\; Op(v) = nop \wedge \nexists ev \wedge \nexists ve$$

Kantenbeschriftung hinzufügen/entfernen

Wird eine Kantenbeschriftung im PDG hinzugefügt, ist dies gleichbedeutend mit dem Hinzufügen einer Variablendeklaration und einer Zuweisung/Initialisierung im Quelltext. Zusätzlich wird diese Variable dann anstelle des Ausdrucks im Quelltext verwendet. Das Entfernen einer Kantenbeschriftung wirkt jeweils umgekehrt. Die neue Variable wird dabei mit dem Ausdruck auf der rechten Seite der Datenflusskante initialisiert. Das Hinzufügen einer Kantenbeschriftung ist immer dann möglich, wenn der neue Variablenname keinen Konflikt erzeugt. Das Einführen von Kantenbeschriftungen wird von Griswold als semantikerhaltend angesehen; jedoch führt er keine eigenständige PDG-Transformationen dafür ein [45, S. 73].

> **Kantenbeschriftung hinzufügen/entfernen:** *Ein Symbol a kann einer Kante e hinzugefügt oder entfernt werden.*
>
> $$e \Leftrightarrow e_a$$

Abbildung 2.3: Eine Kantenbeschriftung wird hinzugefügt/entfernt.

Das Beispiel in Abbildung 2.3 zeigt, wie durch Einsetzen einer Variablen ein geschachtelter Ausdruck entsteht. Der definierende Ausdruck von `a` wird durch Verzicht auf die Variablenbeschriftung der FD-Kante als Unterausdruck der zweiten Addition eingefügt. Wird die Transformation in umgekehrter Richtung durchgeführt, kann der Unterausdruck `1 + 2` aus dem geschachtelten Ausdruck herausgelöst werden. Stattdessen wird dann die Variable `a` verwendet.

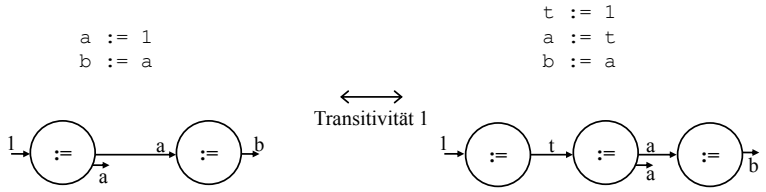


Abbildung 2.4: Transitivität: Fall 1

Transitivität

Die Transformation *Transitivität* [45] (engl. transitivity) fügt eine zusätzliche Zuweisung samt Variable t in den Quelltext ein oder löscht diese. Sie kann in zwei Fälle unterschieden werden, je nachdem wo die neue Variable eingeführt wird.

> **Transitivität 1:** *In allen Pfaden $ue_{ai}v_i$ mit dem Knoten u und der Variablen t kann die Zuweisung w vor der Kante e_{ai} eingefügt werden.*
>
> $$ue_{ai}v_i \Leftrightarrow ue_twe_{ai}v_i$$
>
> **Transitivität 2:** *In allen Pfaden $ue_{ai}v_i$ mit dem Knoten v_i und der Variablen t kann die Zuweisung w nach der Kante e_a eingefügt werden, sodass die Kante e_{ti} zu den Knoten v_i entstehen.*
>
> $$ue_av_i \Leftrightarrow ue_awe_{ti}v_i, \quad \nexists u'e'_av_i$$

Im ersten Fall (siehe Abb. 2.4) existiert eine Zuweisung $e_a = FD(u, v_i)$ auf der Variablen `a`, durch die der Wert `1` in der Variablen `b` der Wert `1` zugewiesen wird. Durch eine neue Zuweisung wird die Zwischenvariable `t` eingeführt. Im PDG werden durch die Transformation eine FD-Kante $e_t = FD(u, w)$ und der Zuweisungsknoten w eingefügt. Die bestehenden FD-Kanten ändern sich zu $e_{ai} = FD(w, v_i)$.

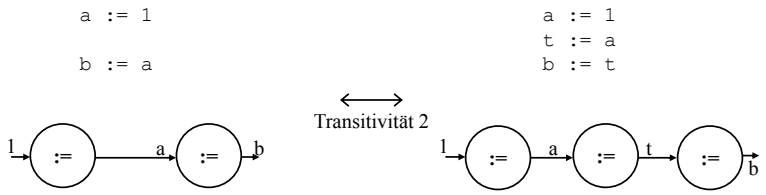


Abbildung 2.5: Transitivität: Fall 2

Im zweiten Fall wird die Zwischenvariable nicht vor `a` sondern vor `b` eingefügt. In Abbildung 2.5 wird die Zuweisung $e_a = FD(u, v_i)$ auf $e_{ai} = FD(u, w)$ geändert. Außerdem werden der Knoten w und die ihm folgenden Kanten $e_{ti} = FD(w, v_i)$ eingefügt. Der zweite Fall ist jedoch nur dann erlaubt, wenn alle v_i ausschließlich durch u definiert wurden und kein weiteres u' existiert, dass über eine Kante $e'_a = FD(u', v_i)$ einen der v_i Knoten definiert.

In Abbildung 2.6 wird eine Situation dargestellt, in der die Variable `a` an zwei verschiedenen Stellen u und u' definiert wird. Daher existieren auch zwei Abhängigkeitskanten

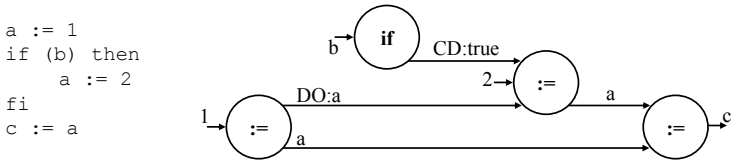


Abbildung 2.6: Transitivität bei mehreren Definitionsorten nicht möglich

zum Definitionsknoten von c. Folglich kann der zweite Fall der Transitivität hier nicht angewendet werden.

Datendistribution

Die Transformation *Datendistribution* [45] (engl. data distributivity) erzeugt Kopien von bestehenden Knoten und Kanten. Datenabhängigkeitskanten (FD) transportieren Variablenwerte von ihrer Definition zur Verwendung. Werden Variablen an mehreren Stellen verwendet, existieren auch mehrere FD-Kanten. Jede FD-Kante kann entweder von ihrem originalen Startknoten oder von einer sequenzkongruenten Kopie des Startknotens starten. Bei der Distribution wird eine sequenzkongruente Kopie eines PDG-Knotens erzeugt, die eine der ausgehenden FD-Kanten des Originalknotens erhält. Der kopierte Knoten erhält außerdem Kopien aller nicht-FD-Kanten des Originalknotens. Die Distributionstransformation kann solange durchgeführt werden, wie ein Knoten durch mehrere FD-Kanten mit Folgeknoten verbunden ist. Die Distribution kann rückgängig gemacht werden, wenn die Quellknoten zweier FD-Kanten sequenzkongruent sind.

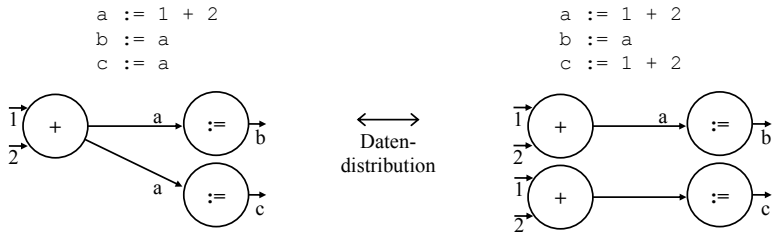


Abbildung 2.7: Distribution von Datenoperationen

> **Datendistribution:** *Sind alle Datenflusspfade durch den Knoten u und dem zu u sequenzkongruenten Knoten u′ gegeben, kann der Vorgänger von v durch die Kante e ersetzt werden.*
>
> $$ue_a v \Leftrightarrow u'ev, \;\; sequenzkongruent(u, u') \wedge \nexists we'_a v$$

Bei der Distribution in Abbildung 2.7 wird der Additionsknoten kopiert. Zusätzlich wird eine seiner ausgehenden FD-Kanten so modifiziert, sodass sie von der sequenzkongruenten Kopie des Additionsknotens aus startet.

Kontrolldistribution

Bei der *Kontrolldistribution*[45] (engl. control distributivity) werden PDG-Knoten entlang von Kontrollflusskanten bewegt. Die Transformation kann in die folgenden drei Fälle

untergliedert werden. Im ersten Fall wird ein PDG-Knoten außerhalb eines `if`-Blocks in beide Zweige dieses Blocks distributiert. Im zweiten Fall wird ein PDG-Knoten in nur denjenigen Zweig eines `if`-Blocks verschoben, der den PDG-Knoten verwendet. Und im letzten Fall wird ein PDG-Knoten in eine `while`-Schleife verschoben.

Im ersten Fall müssen die `if`-Blöcke über beide alternative Zweige (`then` und `else`) verfügen. Die verschobenen PDG-Knoten werden jedoch möglicherweise nicht in beiden `if`-Zweigen verwendet. Indem ein Knoten in beide `if`-Zweige distributiert wird, werden keine folgenden Datenabhängigkeiten verletzt.

Kontrolldistribution 1: *Sind die Datenflusspfade x_iu und der Kontrollflussknoten p_1 als Nachfolger des Kontrollflussknotens p_0 gegeben und sind die Datenflusspfade x_iu und x'_iu' datenkongruent zueinander, kann p_1 ein Vorgänger von x_iu werden.*

$$p_0\{p_1, xu_i\} \Leftrightarrow p_0p_1\{e_{true}xu_i, e_{false}x'u_i\}, \;\; datenkongruent(x, x')$$

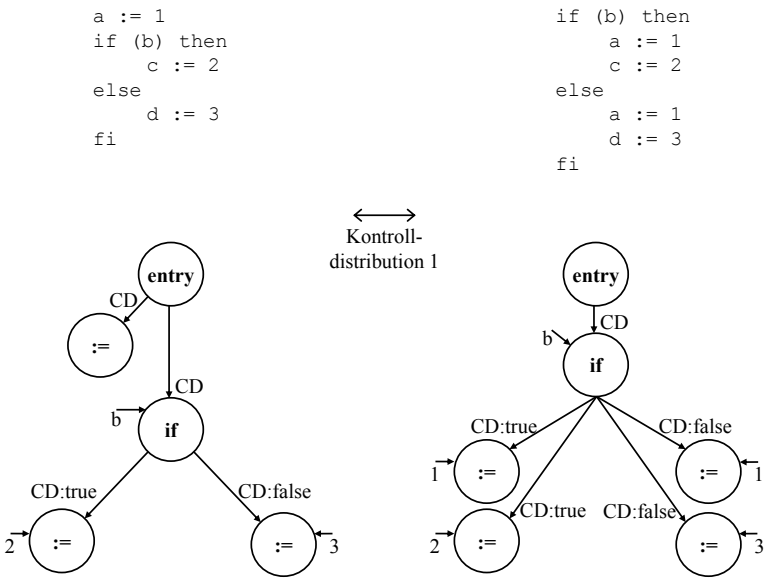


Abbildung 2.8: Distribution von Kontrollprädikaten

Das Beispiel in Abbildung 2.8 zeigt die Distribution eines Knotens in beide Zweige der `if`-Anweisung. Mit `entry` wird der Eintrittsknoten in die Methode gekennzeichnet. Indem der Knoten und seine Kopie in beide `if`-Zweige distributiert werden, wird immer einer der beiden datenkongruenten Knoten ausgeführt. Dadurch bleiben alle nachfolgenden Datenabhängigkeiten bspw. von Leseoperationen (im Beispiel nicht dargestellt) auf der Variablen `a` erhalten.

Kontrolldistribution 2: *Sind die Datenflusspfade x_iu und der Kontrollflussknoten p_1 als Nachfolger des Kontrollflussknotens p_0 gegeben kann x ein Nachfolger von p_1 werden.*

$$p_0\{x, p_1e\}u_i \Rightarrow p_0p_1\{ex, e\}u_i$$

Im zweiten Fall der Kontrolldistribution wird ein PDG-Knoten in nur einen Zweig des `if`-Blocks verschoben, und es werden daher keine Kopien von Knoten oder Kanten erzeugt. Voraussetzung für die Transformation ist, dass der verschobene Knoten ausschließlich in diesem `if`-Zweig verwendet wird. Diese Transformation kann die Nichtterminierungs- und Fehlereigenschaften eines Programms verbessern [45], da der verschobene PDG-Knoten nur noch bei entsprechender `if`-Bedingung ausgeführt wird. Um die Nichtterminierungseigenschaften eines Programms im Zuge von Transformationen nicht zu verschlechtern, ist diese Transformation nicht umgekehrt durchführbar.

Kontrolldistribution 3: *Sind die Datenflusspfade $x_i u$ und der Schleifenknoten p_1^* als Nachfolger des Kontrollflussknotens p_0 gegeben, kann x ein Nachfolger von p_1^* werden.*

$$p_0\{xe_s, p_1^* e_{true}\}u_i \Rightarrow p_0 p_1^*\{e_{true} x e_s, e_{true}\}u_i,$$

$$\nexists w e'_{LC(p_1^*):s} u_a \wedge \nexists x e_{AD:s} x \wedge \nexists p_1^* p_i^* v e_{AD:var} x$$

Der dritte Fall der Kontrolldistribution verschiebt PDG-Knoten in Programmschleifen. Da Programmschleifen möglicherweise mehrmals zur Laufzeit ausgeführt werden, existieren mehrere Bedingungen um Knoten in Schleifen zu verschieben. Erstens dürfen PDG-Knoten, die Variablen definieren, nicht in Schleifen verschoben werden, wenn innerhalb der Schleife eine Neudefinition derselben Variablen existiert. Die Kante $e'_{LC(p_1^*):s} u_a$ beschreibt eine loop-carried Kante auf dem Symbol s und dem Schleifenkopf p_1^*. Würde x in die Schleife verschoben, ginge e' durch die Neudefinition verloren. Zweitens dürfen PDG-Knoten keine AD-Kanten auf sich selbst besitzen. Solche Konstellationen repräsentieren eine Leseoperation gefolgt von Neudefinition derselben Variablen. Der Pfad $p_1^* p_i^* v e_{AD:var} x$ beschreibt eine AD-Kante aus der Schleife hin zu x. Falls ein solcher Pfad existiert, weist er auf eine Variablendefinition von *var* in der Schleife hin. Durch ein Verschieben von x in die Schleife würde dadurch eine zusätzliche FD-Kante (loop-carried), d. h. ein zusätzlicher Datenfluss, erzeugt werden.

2.3.4 Grenzen von PDGs

PDGs wurden ursprünglich eingeführt, um Compiler-Optimierungen semantikerhaltend durchführen zu können [30, 31, 125, 64]. Da bei Compiler-Optimierungen keine Rücktransformationen zu Quelltext von Hochsprachen erforderlich ist, können PDGs an vielen Stellen vereinfacht werden. Diese Vereinfachungen verhindern jedoch, aus modifizierten PDGs wieder Quelltext zu generieren. Die Ursache dafür ist, dass Konstrukte von Hochsprachen Einschränkungen für den Maschinencode bedeuten können. Werden diese Einschränkungen beim Modifizieren der PDGs bzw. des Maschinencodes nicht beachtet, ist die Rücktransformation nicht möglich.

Im Beispiel in Abbildung 2.9 wird der Postinkrementausdruck als Unterausdruck verwendet, sodass ein verschachtelter Ausdruck entsteht. Der Rückgabewert des Postinkrements wird ohne Variable an den übergeordneten Ausdruck übergeben. Daher besitzt die zugehörige Kante in Abbildung 2.9 kein Variablensymbol. Die als Unterausdruck hinzugefügte Postinkrementoperation verursacht einen Lese- und Schreibseiteneffekt auf der Variablen `a`. Die eigentlich kommutativen Operanden der Additionsoperation dürfen daher

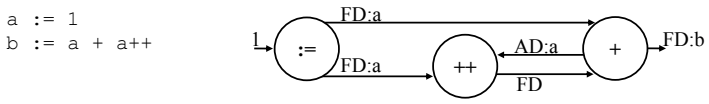


Abbildung 2.9: Seiteneffekte und Verschachtelung können zu Problemen führen.

nicht mehr vertauscht werden, da sich sonst das Ergebnis ändern würde. Di AD-Kante entsteht dadurch, dass Variable `a` als erster Operand der Addition gelesen und anschließend von der Postinkrementoperation modifiziert wird. Folglich ergeben die Abhängigkeiten im zweiten Beispiel eine Schleife und erzeugen so einen PDG, der Knoten mit gegenseitigen Abhängigkeiten enthält.

Im Folgenden werden fünf Eigenschaften von PDGs erläutert, die eine Rücktransformation zum Quelltext behindern.

Reihenfolge von Operanden

PDGs beschreiben lediglich Abhängigkeiten zwischen Knoten. Die durch Datenabhängigkeitskanten beschriebenen Datenzugriffe sind daher im PDG ungeordnet. Das Beispiel in Abbildung 2.2 zeigt einen Additionsknoten, der von zwei Operanden abhängt. Die PDG-Darstellung gibt jedoch keine Auskunft über die Reihenfolge der Operandenzugriffe.

Im Kontext von verschachtelten Ausdrücken und Seiteneffekten in Hochsprachen ist die Reihenfolge der Zugriffe jedoch entscheidend für die Programmsemantik. Im Beispiel in Abbildung 2.9 wird die Abhängigkeit der Programmsemantik von der Ausführungsreihenfolge[1] der Operanden deutlich. Durch Seiteneffekte können Operanden derselben Anweisung untereinander Abhängigkeiten besitzen und daher nicht vertauscht werden. Informationen über die Lesereihenfolge der Abhängigkeiten einer Operation fehlen in PDGs.

Verschachtelung von Ausdrücken

In PDGs können Ausdrücke nicht geschachtelt werden. Grund dafür ist, dass PDGs Maschinencode (*Compiler-Triplets*) abstrahieren, der Verschachtelung von Ausdrücken nicht erlaubt. Die Verschachtelung von Ausdrücken ist jedoch in Hochsprachen gängig und erlaubt eine kompakte Schreibweise von komplexen, zusammengesetzten Anweisungen. Sie führt aber auch zu einer Gruppierung von Ausdrücken. D. h. dass die Ausdrücke einer verschachtelten Anweisung nicht mit den Ausdrücken einer anderen verschachtelten Anweisung vermischt werden können, ohne beide Verschachtelungen aufzuheben. PDGs enthalten jedoch keine Abhängigkeiten, um diese Verschachtelungen zu modellieren und damit eine Vermischung zu verhindern.

Seiteneffekte

Compiler-Triplets modifizieren ausschließlich mit ihrem Rückgabewert einen Speicherplatz im Rechner. Bei ihrer Auswertung entstehen keine Seiteneffekte außerhalb dieses

[1] Die Auswertungsreihenfolge der Terminale ist nicht in allen Sprachen spezifiziert (bspw. C/C++ [18, 106]). In dieser Arbeit wird jedoch Java 7 als Zielsprache genutzt, in der die Auswertungsreihenfolge von Terminalen wie Literalen, Variablen, Methoden oder Prä-/Postoperationen klar von links nach rechts definiert wurde [43]. Die Operatorpräzedenz dagegen ist in beiden Sprachen spezifiziert.

Speicherplatzes. In Hochsprachen können Ausdrücke jedoch Seiteneffekte besitzen, wie sie im Beispiel 2.9 auftreten. Daher sind PDGs nur bedingt dafür geeignet, Hochsprachenprogramme zu modellieren. Bspw. verhindern Seiteneffekte die Anwendung der Datendistributionstransformation. Denn diese darf nicht durchgeführt werden, wenn der zu duplizierende Knoten Seiteneffekte bewirken kann. Andernfalls würde der Seiteneffekt nach der Transformation wiederholt auftreten.

Quelltextblöcke

Ferrante et al. [31] stellen in ihrer Arbeit Blöcke nur im Kontext von Alternativen und Schleifen vor. In Abb. 2.10 wird bspw. eine `if`-Anweisung auf ein Prädikat abgebildet. Dabei wird die `if`-Anweisung als einzelner Knoten aufgefasst. Er besitzt zu allen Anweisungen, die in seinem Dann- bzw. Sonstblock enthalten sind, eine Kontrollabhängigkeitskante. Die Kontrollabhängigkeitskante wird mit `true` bzw. `false` beschriftet, je nachdem, in welchem Block die Anweisung steht. Anweisungen in Unterblöcken werden transitiv spezifiziert: Jede Anweisung besitzt lediglich eine Kontrollflusskante zu ihrer nächst höheren Blockanweisung. Weitere Sprachkonstrukte wie `try`- oder `synchronized`-Anweisungen werden von Ferrante et al. nicht genannt. Sie könnten allerdings analog zu Alternativen und Schleifen auf PDGs abgebildet werden.

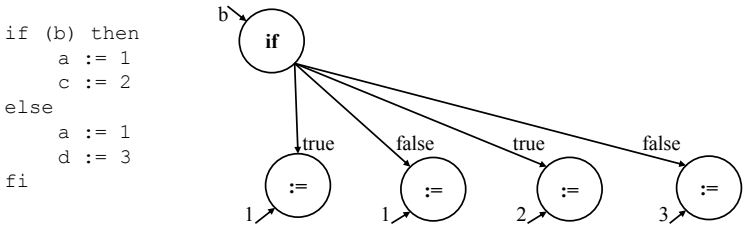


Abbildung 2.10: Wechselnde Knoten der `if`-Blöcke können nicht direkt auf den Quelltext abgebildet werden.

Wird die partielle Ordnung eines PDGs auf eine totale Ordnung abgebildet, existieren oft mehrere Möglichkeiten, die Knoten zu sortieren. Bei dieser Abbildung auf eine totale Ordnung stellen PDGs eine wichtige Eigenschaft nicht sicher: Anweisungen desselben Blocks müssen direkte Nachbarn sein. Die Anweisungen der Dann- und Sonstblöcke einer `if`-Anweisung in Abb. 2.10 besitzen untereinander keine Abhängigkeiten. Ihre Abhängigkeiten besitzen sie zu vorangehenden Anweisungen und zur `if`-Anweisung selbst. Folglich erlaubt die partielle Ordnung des PDGs eine totale Ordnung, bei der die Anweisungen beider Blöcke abwechselnd aufgeführt werden. Diese abwechselnde, totale Ordnung kann aber nicht direkt auf den Quelltext zurück abgebildet werden ober erzwingt eine Aufspaltung der ursprünglichen `if`-Anweisung.

Warren [125] stellt in seiner Arbeit hierarchische PDGs vor, um PDGs in Teilgraphen zu zerlegen. Dabei werden für jede Schleife Teilgraphen erstellt, die Informationen über Datenabhängigkeiten zusammenfassen. Durch diese Informationen können auch Teilgraphen mit Abhängigkeiten zu anderen Anweisungen und/oder Teilgraphen werden. Warren nutzt Teilgraphen jedoch nicht für andere Sprachkonstrukte wie bspw. Alternativen.

Geltungsbereiche von Variablen

Geltungsbereiche lokaler Variablen werden in Sprachen wie Java durch eine Deklaration und die zugehörige Blockgrenze definiert. Dabei beginnt der Geltungsbereich bei der Deklaration und endet an der Blockgrenze des Blocks, in dem sich die Deklaration befindet. Der Geltungsbereich der deklarierten Variablen erstreckt sich dabei auch auf alle Unterblöcke, die sich zwischen seinem Beginn und seinem Ende befinden.

Geltungsbereiche besitzen dabei u. a. die folgenden Einschränkungen: (1) Die Nutzung einer Variablen kann nur innerhalb ihres Geltungsbereichs erfolgen. (2) Innerhalb des Geltungsbereichs einer lokalen Variablen darf keine weitere lokale Variable mit demselben Namen deklariert werden. (3) Wird eine lokale Variable benutzt bzw. eingeführt, verdeckt sie innerhalb ihres Geltungsbereichs ggf. bereits existierende Variablen der Klasse oder Superklasse, sofern die Variablennamen identisch sind. Wird der Quelltext derart verändert, dass bspw. die Deklarationen oder Verwendungen von Variablen verschoben werden, müssen die genannten Einschränkungen beachtet werden, um die Programmsemantik nicht unbeabsichtigt zu verändern.

PDGs nach Ferrante et al. [31] bilden Geltungsbereiche von lokalen Variablen jedoch nicht bzw. nur teilweise ab. PDGs erzeugen eine Abhängigkeit von der Deklaration zur ersten Verwendung (Definition) einer Variablen. Folglich können keine Variablenverwendungen vor Deklarationen verschoben werden; denn die Verwendungen selbst sind ebenfalls durch Datenflussabhängigkeiten miteinander verbunden. Allerdings könnte eine Deklaration in einen Unterblock verschoben werden, sodass die darauf folgenden Verwendungen fehlerhaft werden können.

Beim Einführen einer neuen Variablen müssen ebenfalls die Einschränkungen bereits existierender Geltungsbereiche beachtet werden. Denn durch neue Variablen dürfen keine bereits existierenden Variablen verdeckt oder Konflikte mit ihnen erzeugt werden. Griswold [45] löst dieses Problem, indem er sogenannte Konturen einführt. Eine Kontur definiert einen Geltungsbereich, indem sie einen Teilgraphen des PDG enthält. Dabei entsteht eine verschachtelte Struktur, bei der Konturen weitere Konturen enthalten können.

2.4 Java Development Tools

Als Grundlage für die Abbildungsregeln von Java-Elementen auf die Quelltextabstraktion wird die Implementierung des abstrakten Syntaxbaums der *Java Development Tools*, kurz JDT [34], genutzt. Sie ermöglichen einen einfachen und technisch ausgereiften Zugriff auf den abstrakten Syntaxbaum eines Java-Programms und bieten außerdem Analysefunktionen, um bspw. Bindungsinformationen von Quelltextsymbolen zu erhalten.

Einzelne Sprachelemente werden als Verweis (mit Eclipse-Logo und URL) genutzt, um auf JDT-Elemente wie eine `InfixExpression` verweisen zu können. Sofern die direkte Übersetzung vom Deutschen ins Englische existiert, werden im Sinne der Lesbarkeit auch die deutschen Begriffe (z. B. `Infixausdruck`) verwendet. An einigen, wenigen Stellen weicht die JDT-Implementierung etwas von der Java-Sprachdefinition [43] ab. Hierzu zählen bspw. `Infixausdrücke`, die mehr als zwei Operanden besitzen. Die vorliegende Arbeit orientiert sich dann an der Implementierung von JDT. In den meisten Fällen sind die Implementierungen der Sprachelemente aber analog zu den Backus-Naur-Formen der Java-Syntax aus der Sprachdefinition.

Der gesamte AST von JDT soll hier nicht erklärt werden, da er zu umfangreich ist. Da diese Arbeit aber viel auf die einzelnen Elemente des JDT-ASTs eingeht, sollen einige von ihnen und ihre prinzipielle Struktur anhand eines Beispiels vorgestellt werden. Außerdem werden Definitionen eingeführt, die bei der Beschreibung in den Folgekapiteln genutzt werden, um einzelne Elemente des ASTs zu benennen.

2.4.1 Beispiel

Das folgende Beispiel soll eine Vorstellung vermitteln, wie ein abstrakter Syntaxbaum (AST) eines gegebenen Quelltexts aussieht. Der Quelltext 2.1 ist im UML-Objektdiagramm in Abbildung 2.11 auszugsweise so als AST angegeben, wie er von JDT generiert wird. Der Quelltext stellt eine komplette Java-Quelltextdatei dar, die von JDT geparst wird.

```
class MyClass {
  int l = 0;
  void myMethod() {
    int a = 0;
    if (true)
      a++;
    a = l + in(a);
  }
  int in(int i) {
    return 42;
  }
}
```

Quelltext 2.1: Beispielquelltext zum Erzeugen eines abstrakten Syntaxbaums

Das oberste JDT-Element jeder Java-Quelltextdatei ist eine Instanz der Klasse `CompilationUnit`. Sie enthält die deklarierten Typen, die wiederum einzelne Methoden enthalten können. Die Attribute der Instanzen zeigen bspw. den jeweiligen Namen oder Typen an. Im Objektdiagramm wird nur die Methode `myMethod` dargestellt. Auf die Darstellung der Methode `in` und des Felds `l` wird aus Platzgründen verzichtet. In diesem Beispiel existiert nur ein `Block`, der den Rumpf der Methode repräsentiert. Er besitzt insgesamt drei Anweisungen, die relevant für die Erzeugung einer Quelltextabstraktion sind. Allerdings können auch Anweisungen wie eine Schleife einen Block besitzen, der dann ebenfalls wieder Anweisungen enthalten kann. Das Objektdiagramm zeigt, dass die `Deklarationsanweisung` aus `Deklarationsfragmenten` besteht, die einen Initialisierungsausdruck besitzt. Ferner wird ersichtlich, dass die Alternative (`If`-Anweisung) keinen Dannblock besitzt und der Sonstblock aus einer `Ausdrucksanweisung` besteht. Ein Ausdrucksanweisung enthält lediglich einen `Ausdruck`, der allerdings Unterausdrücke besitzen kann. Im Objektdiagramm existieren zwei Ausdrucksanweisungen mit einer Postinkrementoperation und einer Zuweisung als Ausdruck. Die Ausdrücke verweisen schließlich auf Unterausdrücke der Klasse `SimpleName`, die Terminale im AST sind und Symbole bezeichnen, die auf Variablen verweisen können.

2.4.2 Definitionen

JDT ist eine umfangreiche PlugIn-Sammlung von Eclipse, die aus vielen Klassen und Funktionen besteht, um Quelltexte zu parsen und bereits einfache Analysen darauf durchführen

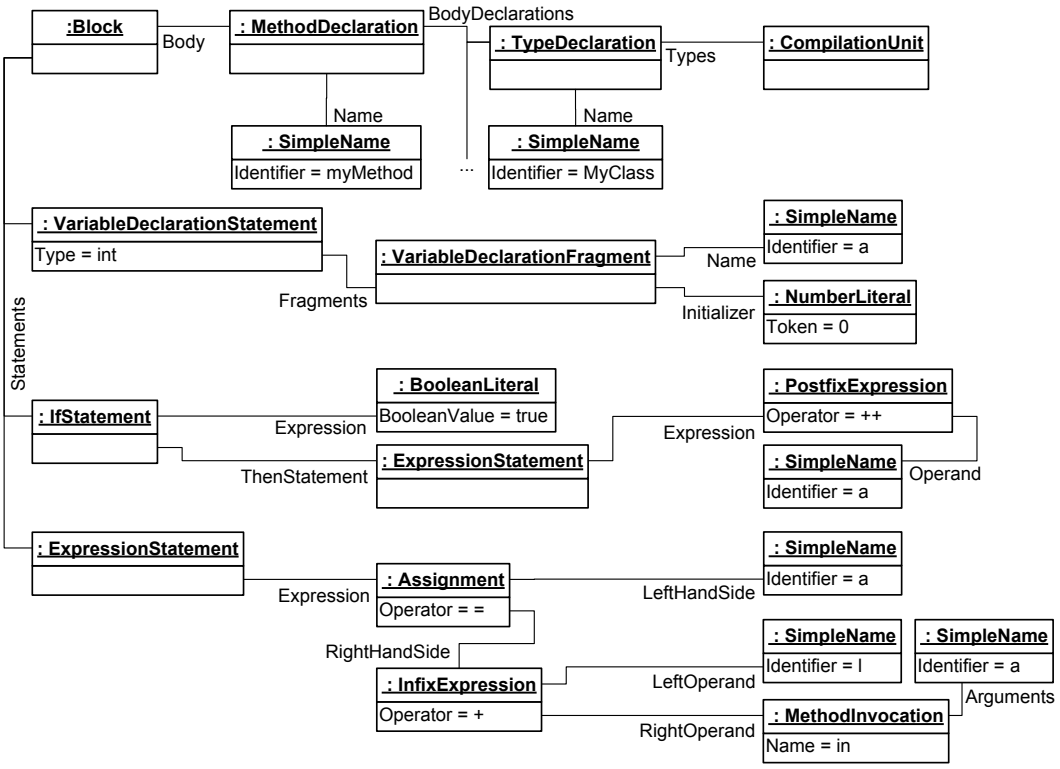


Abbildung 2.11: UML-Objektdiagramm des gekürzten JDT-AST zum Quelltext 2.1

zu können. Durch die Analysen stellt JDT einen abstrakten Syntaxgraphen bereit, der Bindungsinformationen über die verwendeten Symbole im Quelltext besitzt. Diese werden in dieser Arbeit genutzt, um Datenabhängigkeiten zu erkennen.

Die Definitionen in den folgenden Abschnitten führen eine einheitliche Terminologie ein.

Elemente von JDT

Wenn in den Folgekapiteln Aussagen über Elemente aus dem AST von JDT getroffen werden, wird der Term *JDT-Element* verwendet. JDT-Elemente sind dabei solche Instanzen von Klassen aus JDT, die Teil des ASTs sind und durch parsen des Quelltexts erzeugt wurden. Dies können bspw. Anweisungen, Ausdrücke, Typen, etc. sein.

> **JDT-Element:** *Ist e eine Instanz einer Unterklasse der JDT-Klasse $ASTNode$, wird e JDT-Element genannt.*

Quelltextblöcke kommen an verschiedenen Stellen im Quelltext vor. Zunächst ist der Rumpf einer Methode ein Quelltextblock. Ferner können auch Alternativen, Schleifen oder andere Anweisungen im Quelltext Blöcke besitzen. Der Aufbau des abstrakten Syntaxbaums bestimmt dabei die Hierarchie der Quelltextblöcke.

> **Block:** *Alle JDT-Anweisungen, die ein Unterelement der JDT-Klasse Block besitzen, werden als Quelltextblock oder Block bezeichnet.*

> **Elternblock:** *Ist b_p ein Quelltextblock und b ein weiterer Quelltextblock mit der Eigenschaft, dass b_p den Block b als Unterelement besitzt, dann wird b_p als Elternblock von b bezeichnet.*
>
> *Sind b_p1, b_p2 und b ebenfalls Quelltextblöcke mit den Eigenschaften, dass b_p1 ein Elternblock von b_p2 und b_p2 ein Elternblock von b ist, dann wird auch b_p1 als Elternblock von b bezeichnet.*

Ist b_p ein Elternblock von b wird umgekehrt b als Unterblock von b_p bezeichnet. Wird ein Block b_1 als *direkter* Eltern- bzw. Unterblock eines anderen Blocks b_2 bezeichnet, befindet sich kein weiterer Eltern- bzw. Unterblock zwischen b_1 und b_2.

> **Abgeschlossener Quelltextblock:** *Ein Quelltextblock b, der die Unterblöcke B_U besitzt, aber keinen im AST übergeordneten Quelltextblock, wird abgeschlossener Quelltextblock genannt. Die Menge aller Blöcke eines abgeschlossenen Quelltextblocks enthält ihn selbst sowie all seine transitiven Unterblöcke.*

Reihenfolge von Quelltextelementen

Häufiger ist die Reihenfolge von JDT-Elementen im Quelltext ausschlaggebend für die Reihenfolge von Abstraktionselementen, deren Abhängigkeiten oder Kontrollfluss. Daher wird die injektive Funktion $Pos : Elements \rightarrow \mathbb{N}$ eingeführt, die eine Menge von Abstraktionselementen auf eine natürliche Zahl entsprechend ihrer Position im Quelltext abbildet.

In den Abbildungsregeln von Kapitel 4 wird die Abbildung $SuccPairs$ verwendet. Sie bildet eine Menge E von Abstraktionselementen auf eine Menge von 2-Tupeln der Menge E ab und nutzt dafür eine injektive Sortierfunktion Pos. Jedes 2-Tupel (a, b) hat die Eigenschaft, dass das Element a direkt vor dem Element b liegt und sich kein anderes Element in M zwischen beiden befinden kann.

$$
\begin{aligned}
&SuccPairs_{Pos} : \mathcal{P}(E) \rightarrow \mathcal{P}(E \times E) \\
&SuccPairs_{Pos}(E) \triangleq \{(a,b) | \{a,b\} \subseteq E \wedge Pos(a) < Pos(b) \wedge \\
&\qquad\qquad \nexists c \in E : Pos(a) < Pos(c) < Pos(b)\}
\end{aligned}
$$

Sprunganweisungen

In Java bewirken einige Anweisungen Kontrollflusssprünge. Hierzu zählen die JDTKlassen `Continue`-, `Break`-, `Return`- und `ThrowStatement`, die im Folgenden als Sprunganweisungen bezeichnet werden.

> **Sprunganweisung:** *JDT-Anweisungen vom Typ Continue-, Break-, Return- oder ThrowStatement werden Sprunganweisungen genannt.*

Die `Assert`-Anweisung (`§14.10`) wird in dieser Arbeit nicht als Sprunganweisung interpretiert, da sie laut Sprachdefinition dazu genutzt werden sollen, Programmfehler zu finden. Der durch sie ausgelöste Java-Error dient dazu, die Programmausführung zu beenden (`§11.2`), da er auf einen fehlerhaften Programmzustand hinweist, der durch eine Fehlerbehandlung nicht behoben werden kann.

Symbole und Variablen

Programmvariablen existieren im Quelltext zunächst als JDT-Elemente der Klasse `Name`. Mit dem Begriff *Symbol* wird in dieser Arbeit ein JDT-Element der Klasse `Name` bezeichnet, das sich im AST eines abgeschlossenen Quelltextblocks befindet. Um festzustellen, ob zwei solche Symbole im Quelltext dieselbe *Variable* meinen, wird mithilfe von JDT die Variablenbindung (`IVariableBinding`) festgestellt.

> **Symbol:** *Ist s ein JDT-Element der Klasse Name, zu dem mithilfe von JDT eine Instanz vom Typ IVariableBinding ermittelt werden kann, wird s als Symbol bezeichnet.*

> **Variable:** *Ist s ein Symbol und v seine zugehörige Instanz der Klasse IVariableBinding, wird v als Variable bezeichnet. Zwei verschiedene Symbole können identische Variablen besitzen.*

Zusammenfassung

In diesen Kapitel wurde auf die Grundlagen dieser Arbeit eingegangen. Dazu zählt zunächst der allgemeine Hintergrund zum Thema Quelltextumgestaltungen und das Umfeld, in das sich diese Arbeit eingliedert. Zu den wichtigen Grundlagen dieser Arbeit gehören die Programmabhängigkeitsgraphen (PDGs), an die die später genutzte Abstraktion angelehnt ist. Da der vorgestellte Ansatz mithilfe der Java Development Tools (JDT) umgesetzt und evaluiert wird, wurde eine kurze Einführung in dieses Thema gegeben. Die verwandten Arbeiten fokussierten sich entweder auf deklarative Umgestaltungen oder nutzten zur Lösungsfindung keine Constraint-Programmierung. Diese Arbeit unterscheidet sich von bisherigen Ansätzen dadurch, dass sie kontrollflusssensible Umgestaltungen mithilfe von Constraint-Programmierung löst und dadurch verbessert.

Kapitel 3

Quelltextabstraktion

In diesem Kapitel wird die Quelltextabstraktion vorgestellt, auf die ein Java-Quelltext abgebildet werden kann. Die Quelltextabstraktion ist eine umfangreiche Erweiterung von Programmabhängigkeitsgraphen (PDGs), deren Elemente Klassen sind, die verschiedene Attribute besitzen. Sie wird im Folgenden als *Quelltextabstraktion* oder kurz als *Abstraktion* bezeichnet.

Die Quelltextabstraktion ist ein Zwischenergebnis des Ansatzes, ein Constraint-Problem zu erzeugen, um damit anschließend Umgestaltungen zu berechnen. Hierfür bildet sie die für die Programmsemantik relevanten Zusammenhänge wie Programmabhängigkeiten ab. Sie unterscheidet dafür bereits die Zusammenhänge danach, ob aus bestimmten Informationen der Abstraktion später Constraint-Variablen werden oder nicht. Andere Informationen, die für das Constraint-Problem nicht zwingend erforderlich sind, werden in der Abstraktion nicht repräsentiert.

Allerdings können bei der Abbildung von Quelltext auf die Abstraktion (Kapitel 4) nicht alle Programmabhängigkeiten direkt aus dem Quelltext abgelesen und in die Abstraktion eingefügt werden. Hierzu gehören bspw. Datenabhängigkeiten zwischen Variablendefinitionen und -verwendungen. Die Quelltextabstraktion besitzt folglich zunächst nicht alle Abhängigkeiten. Die noch fehlenden Abhängigkeiten werden in einem zusätzlichen Analyseschritt erzeugt (Unterkapitel 4.3). Dabei werden dann jedoch Informationen aus dem Quelltext benötigt, die nicht Teil der Abstraktion sind. Die Zusatzinformationen betreffen den Kontrollfluss und die Programmeffekte (z. B. Lesen/Schreiben einer Variablen) bezogen auf die Abstraktion. Diese Informationen werden später temporär oder über Abbildungsfunktionen in die Abstraktion übernommen. Damit jedoch die Datenflussanalyse effizient durchgeführt werden kann, wurde die Abstraktion so konstruiert, dass sie auch mit Kontrollflüssen und Programmeffekten umgehen kann. Hierzu gehört vor allem, dass alle Anweisungen und Ausdrücke so auf die Abstraktion abgebildet werden, dass sie eindeutige Ein- und Austrittsknoten für den Kontrollfluss besitzen.

Die weiteren Ausprägungen der Abstraktion werden in diesem Kapitel erläutert. Zunächst werden die allgemeinen Unterschiede zu PDGs und Definitionen für die Abstraktion erläutert und eingeführt. Anschließend werden die Elemente und Attribute der Abstraktion vorgestellt. Die folgende textuelle Notation wird in den Kapiteln 4 und 5 genutzt. Schließlich werden die Eigenschaften der Abstraktion und die Transformationen der Abstraktion präsentiert. Die Abbildung eines Java-Quelltexts auf die hier vorgestellte Quelltextabstraktion wird im Kapitel 4 eingeführt.

3.1 Allgemeine Unterschiede zu PDGs

Programmabhängigkeitsgraphen sind aus mehreren Gründen für die Erzeugung eines Constraint-Problems unzureichend. Die Quelltextabstraktion dient daher als Zwischenschritt, um ein Constraint-Problem zu erzeugen, das schließlich zum Finden einer Umgestaltung genutzt werden kann. Die notwendigen Erweiterungen betreffen vor allem den Umgang mit Blöcken, verschachtelten Ausdrücken und der Reihenfolge von Operanden. Anhand der folgenden Beispiele werden die wesentlichen Unterschiede zwischen der Quelltextabstraktion und PDGs erläutert.

Blöcke

Vier der fünf in dieser Arbeit vorgestellten Umgestaltungen modifizieren nur den Quelltext innerhalb einer einzigen Methode. Dabei wird die Berechnung der Umgestaltung blockweise durchgeführt. Das bedeutet, dass für jeden `Block` eine eigene Abstraktion erstellt und das Umgestaltungsproblem als Teilproblem separat für diesen gelöst wird. Auf diese Art wird verhindert, dass einzelne Anweisungen ihre Blöcke verlassen oder in untergeordnete Blöcke eindringen.

Bei der Transformation *Ausdruck einfügen* ist jedoch erwünscht, dass einzelne Ausdrücke in untergeordnete Blöcke verschoben werden. Um dies zu ermöglichen, werden jene Ausdrücke, die eingefügt werden sollen, den Java-Elementen eines Unterblocks des Teilproblems zusätzlich als Kopien hinzugefügt. Wird eine Einfügen-Transformation erstellt, werden diese Kopien auch dazu verwendet. Falls keine Einfügen-Transformation erstellt wird, werden die Kopien nicht weiter beachtet.

Reihenfolge von Operanden

Die Operanden von Quelltextoperationen werden in PDGs auf Knoten abgebildet. PDGs werden jedoch für Quelltexte verwendet, deren Operationen keine Seiteneffekte besitzen oder verschachtelt werden können. Daher entfällt in PDGs auch die Notwendigkeit, die Operanden einer Operation zu ordnen. Da im Hochsprachenquelltext Operanden jedoch Seiteneffekte besitzen und verschachtelt sein können, ist das Erkennen und Festlegen ihrer Reihenfolge entsprechend ihrer Auswertungsreihenfolge (`§15.7`) notwendig. Wird die Reihenfolge von Operanden beim Umgestalten nicht im Gleichungssystem beachtet, könnten beim Lösen des Gleichungssystems Transformationen berechnet werden, die nicht mehr auf den Quelltext anwendbar sind oder dessen Semantik verändern.

Daher verfügen sowohl die Abstraktion als auch das Gleichungssystem über eine Repräsentation der Operandenposition. Die Operandenposition repräsentiert dabei die Lesereihenfolge der Operanden einer Operation. Um ein Vertauschen der Operanden zu ermöglichen bzw. zu unterbinden, können die Operandenpositionen kommutativ bzw. nicht-kommutativ modelliert werden.

Verschachtelung von Ausdrücken

Die Verschachtelung von `Ausdrücken` im Quelltext erzeugt zusätzliche Abhängigkeiten, die in PDGs nicht dargestellt werden. Die Abhängigkeiten entstehen dadurch, dass verschachtelte Quelltextausdrücke in ihrer Ausführungsreihenfolge direkt aufeinander folgen

und nie durch Ausdrücke anderer Anweisungen unterbrochen werden können. Diese Abhängigkeit wird von PDGs nicht abgebildet, da in PDGs Knoten nicht total geordnet werden.

In der Abstraktion und folglich auch im Gleichungssystem werden spezielle Kanten genutzt, um Ausdrücke eng aneinander zu binden. Diese Kanten werden IS-Kanten (engl. immediate successor edge, dt. unmittelbare Nachfolgerkante) genannt. Auf diese Weise entsteht zwischen verschachtelten Ausdrücken derselben Anweisung eine totale Ordnung. Diese kann nur durch Extrahieren eines der beteiligten Ausdrücke gelockert werden. Die Anweisungen bleiben untereinander weiterhin nur partiell durch ihre Abhängigkeiten geordnet.

Geltungsbereiche von Variablen

Geltungsbereiche werden in PDGs nicht modelliert. In der Quelltextabstraktion werden daher Blöcke genutzt, um die folgenden drei Bedingungen für Geltungsbereiche lokaler Variablen einzuhalten: (1) Variablen werden erst nach ihrer Deklaration verwendet, indem zusätzliche Abhängigkeitskanten in der Abstraktion eingesetzt werden. Diese starten bei der Deklaration und enden bei der/den ersten Verwendungen (Definitionen). (2) Variablen werden nur innerhalb des Blocks verwendet, in dem sie deklariert worden sind. Die Verwendung in Unterblöcken ist ebenfalls erlaubt. Diese Bedingung wird automatisch eingehalten, da das Umgestaltungsproblem anhand der Blöcke in Teilprobleme untergliedert wird. Hierdurch kann die Verwendung einer Variablen ihren Block nicht verlassen. (3) Beim Einführen von neuen Variablen dürfen keine Namenskonflikte mit bereits existierenden Variablen entstehen. Damit diese Bedingung eingehalten wird, wird eine Analyse der Variablen im Geltungsbereich (§6.3) durchgeführt. Dabei wird geprüft, ob der gewünschte Variablenname konfliktfrei genutzt werden kann. Alternativ wird ein neuer, konfliktfreier Name gefunden.

3.2 Begriffsdefinitionen

Eine Quelltextabstraktion repräsentiert abgeschlossene Quelltextblöcke wie bspw. Methodenrümpfe. Innerhalb von Methodenrümpfen können weitere Quelltextblöcke existieren, bspw. durch Schleifen oder Alternativen. Diese können beliebig verschachtelt sein, sodass ein einziger Methodenrumpf insgesamt sehr viele Unterblöcke besitzen kann. Für den Methodenrumpf und jeden Unterblock innerhalb der Methode wird jeweils eine *partielle Quelltextabstraktion* erstellt. Gemeinsam bilden alle partiellen Abstraktionen die komplette Quelltextabstraktion eines Umgestaltungsproblems. Im folgenden wird mit Hinblick auf das JDT-Element `Block` der Begriff Quelltextblock mit *Block* abgekürzt, sofern der Zusammenhang aus dem Kontext ersichtlich wird. Ferner wird bereits die Begriffe *Komposition* und *Effekt* verwendet, die in den Unterkapiteln 3.3 und 4.3 erklärt werden. Kompositionen repräsentieren Anweisungen und Ausdrücke des Quelltexts, während Effekte die Auswirkungen von Anweisungen und Ausdrücken repräsentieren. Zu diesen Auswirkungen gehören bspw. das Lesen sowie Schreiben von Variablen, das Verursachen von Programmsprüngen sowie das Deklarieren von Variablen.

Partielle Quelltextabstraktion: *Ist P ein abgeschlossener Quelltextblock,*

ist B_P die Menge all seiner Blöcke, ist $b \in B_P$ und ist E_b die Menge aller Quelltextelemente von b, dann wird die Quelltextabstraktion a_b von b, die mindestens die Anweisungen und Ausdrücke aus E_b repräsentiert, partielle Quelltextabstraktion genannt.

Quelltextabstraktion: *Ist P ein abgeschlossener Quelltextblock, der die Menge aller Blöcke B_P besitzt, dann wird die Gesamtheit aller partiellen Quelltextabstraktionen der Blöcke aus B_P Quelltextabstraktion genannt.*

Blockabstraktion: *Ist P ein abgeschlossener Quelltextblock mit der Menge B_P seiner Blöcke und sind $b_1, b_2 \in B_P$ mit der Eigenschaft, dass b_2 ein Unterblock von b_1 ist, ist ferner c_2 eine Komposition in b_1, die den Effekt von b_2 und all seiner Unterblöcke repräsentiert, dann wird c_2 als Blockabstraktion von b_2 bezeichnet.*

Die Quelltextabstraktion bildet alle Anweisungen und Ausdrücke aller Blöcke eines abgeschlossenen Blocks ab. Für jede partielle Quelltextabstraktion werden aus dem zugehörigen Quelltext Abstraktionselemente erzeugt, die wichtige Eigenschaften speichern. Wird für einen Unterblock ein Abstraktionselement erzeugt, wird dieses Abstraktionselement *Blockabstraktion* genannt. Es repräsentiert die transitiven Lese- und Schreibeffekte des Unterblocks, ohne seine Anweisungen und Ausdrücke explizit zu modellieren. Im folgenden Unterkapitel werden die Elementtypen und ihre Attribute vorgestellt.

Freiheitsgrad: *Ein Freiheitsgrad ist eine Eigenschaft der Quelltextabstraktion, mithilfe von Constraint-Variablen und -gleichungen gleichzeitig mehrere, sequenzkongruente Programmvarianten modellieren zu können. Die Menge aller Freiheitsgrade der Quelltextabstraktion ermöglicht die Berechnung von sequenzkongruenten Transformationen, um eine Umgestaltung durchzuführen.*

Die in dieser Arbeit genutzte Abstraktion besitzt mehrere Transformationen, die im letzten Unterkapitel dieses Kapitels vorgestellt werden. Einige dieser Transformationen werden als Vor- oder Nachbereitungsschritt durchgeführt, indem sie auf die Abstraktion und folglich den Quelltext direkt angewendet werden. Dabei ändert sich die Abstraktion strukturell, da bspw. Abstraktionselemente wie Kanten hinzugefügt oder entfernt werden. Andere Transformationen ändern die Abstraktion nicht strukturell sondern lediglich Variablenwerte von bestehenden Abstraktionselementen. Die Änderungsmöglichkeiten dieser Transformationen werden *Freiheitsgrade* der Abstraktion genannt. Bspw. werden die Positionswerte von Anweisungen lediglich als Variablenwerte gespeichert. Folglich ist die Reihenfolge von Anweisungen im Quelltext ein Freiheitsgrad in der Abstraktion. Die anderen Freiheitsgrade sind das Einfügen/Extrahieren von Ausdrücken, das Umsortieren von Operanden sowie die Eigenschaft, ob eine Variablendeklaration existiert oder nicht. Sie werden jeweils durch Variablenwerte in der Abstraktion gespeichert. Somit repräsentiert ein und dieselbe strukturelle Quelltextabstraktion mehrere Varianten eines Programms.

3.3 Elemente und Attribute

Die Funktion der Quelltextabstraktion ist, die notwendigen Informationen zu modellieren, um ein Constraint-Problem erzeugen zu können. Die Informationen der Abstraktion werden

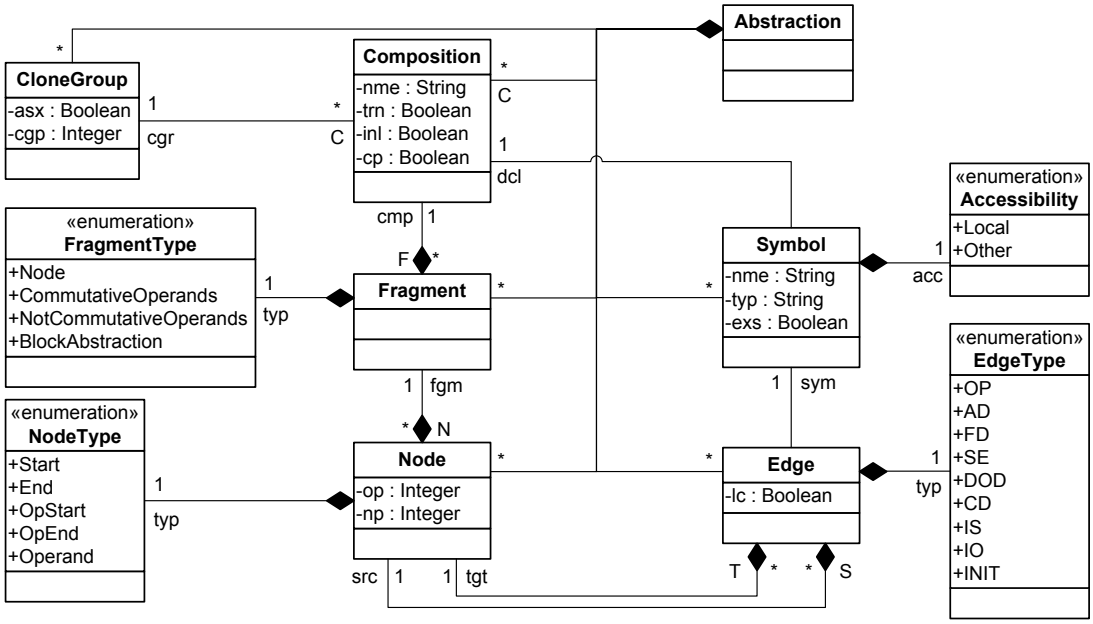


Abbildung 3.1: Quelltextabstraktion als UML-Klassendiagramm

dazu auf Elemente (Klassen im UML-Diagramm) und Attribute (Attribute der Klassen) abgebildet. Die Attribute können jedoch darüberhinaus in *fixe* und *variable* Attribute unterschieden werden. Beide Attributarten modellieren Informationen aus dem zugrunde liegenden Quelltext, wobei die fixen Attribute zum Erzeugen von Constraint-Gleichungen genutzt werden. Dagegen werden aus den variablen Attributen die Constraint-Variablen erzeugt, die in den Constraint-Gleichungen verwendet werden. Unterkapitel 5.1 präsentiert die Erzeugung eines Constraint-Problems aus der Quelltextabstraktion.

Variable Attribute werden auch CSP-Variablen genannt. CSP-Variablen können im Constraint-Problem einen beliebigen Wert aus ihrer Wertemenge annehmen. Bspw. kann die CSP-Variable, die die Position eines Knotens speichert, unterschiedliche Werte besitzen, da die zugehörige Anweisung ebenfalls potentiell an verschiedenen Positionen im Quelltext stehen kann. Durch die CSP-Variablen werden in der Quelltextabstraktion die Freiheitsgrade umgesetzt.

Aus der später berechneten Wertebelegung der CSP-Variablen wird am Schluss der Berechnung eine konkrete Ausprägung der Abstraktion und eine Menge an Transformationen abgeleitet. Damit diese Ausprägung die Syntax der Sprache und die Semantik des Programms nicht verletzt, werden Constraint-Gleichungen genutzt, die die variablen Attribute zueinander in Relation setzen. Hierdurch werden die Wertebereiche der Attribute soweit eingeschränkt, dass kein Programm entsteht, das entweder nicht kompilierbare oder zum Ursprungsprogramm semantisch verschiedene Programme erzeugt.

Im Gegensatz zu CSP-Variablen, deren Werte variabel sind, besitzen fixe Variablen ganz konkrete Werte. Bspw. hat eine Kante einen Quell- und Zielknoten, die nicht variabel sind. Dementsprechend können die Quell- und Zielknoten von Kanten der Abstraktion nicht durch Freiheitsgrade modifiziert werden.

Die Constraint-Gleichungen der Attribute, ihre Wertänderungen und die zugehörigen Umgestaltungen werden in Unterkapitel 5.1 erläutert. Zunächst wird die Quelltextabstraktion spezifiziert, wobei die einzelnen Klassen und Attribute genannt und kurz einzeln,

nacheinander präsentiert werden. Die genauen Zusammenhänge und Bedeutungen der Klassen und Attribute werden nach und nach in den nächsten beiden Kapiteln erklärt. In Abbildung 3.1 ist die Quelltextabstraktion als UML-Klassendiagramm dargestellt.

3.3.1 Graphische Notation

Um ausgewählte Abstraktionsbeispiele graphisch darstellen zu können, wird eine graphische Notation für die folgenden Elemente der Abstraktion eingeführt: Kompositionen, Fragmente, Knoten und Kanten. Jede graphische Darstellung zeigt immer nur eine partielle Abstraktion. In den meisten Fällen ist das ausreichend, da nur ein Quelltextblock in den Beispielen betrachtet wird. Andernfalls werden die Darstellungen der partiellen Abstraktionen einzeln aufgeführt.

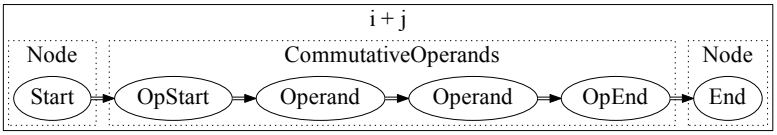


Abbildung 3.2: Abstraktionsbeispiel für die Addition `i + j`

Abbildung 3.2 zeigt die Abstraktion eines Additionsausdrucks. Der äußere Kasten mit dem durchgezogenen Rahmen stellt die Komposition dar. Die inneren Kästen mit dem gestrichelten Rahmen repräsentieren die Fragmente und die Ovale repräsentieren die Knoten. Die Kanten werden durch die Pfeile dargestellt. Die Komposition wird mit dem zugehörigen Java-Element beschriftet und die Fragmente sowie Knoten mit ihren Typen. Die Typen der Fragmente, Knoten und Kanten sind die jeweiligen, im Diagramm gezeigten Aufzählungstypen. Sofern es sinnvoll ist, werden Zusatzinformationen den Fragmentbeschriftungen in Klammern hinzugefügt. Das können z. B. Informationen über die Zugehörigkeit eines Fragments zu einem Java-Unterelement sein. Kanten werden mit dem zugehörigen Symbol und/oder dem Kantentyp beschriftet. Bei Kontrollflussabhängigkeitskanten (Typ CD) wird auf die Beschriftung meist verzichtet. Kanten vom Typ IS sind besondere Kanten, die die so verbundenen Knoten unmittelbar aufeinander folgend anordnen. Sie sind notwendig, damit ausgeschlossen werden kann, dass sich ein dritter Knoten zwischen zwei IS-Kantenknoten befindet. Dadurch kann eine Vermischung von Knoten verschiedener Quelltextanweisungen verhindert werden. IS-Kanten werden mit Doppellinien gezeichnet, alle anderen mit einfachen Linien. Auf die genaue Ausprägung der Abstraktion in Abhängigkeit des jeweiligen Java-Elements wird in Kapitel 4 eingegangen.

3.3.2 Komposition

Kompositionen repräsentieren `Anweisungen` und `Ausdrücke` des Quelltexts. Sowohl Kontrollstrukturen (wie Schleifen oder Alternativen) als auch einfache Anweisungen (wie Deklarationen oder Zuweisungen) werden durch Kompositionen repräsentiert. Außerdem repräsentieren Kompositionen Ausdrücke wie bspw. Additionen. Die Ausdrücke im Quelltext können auch verschachtelt sein, wobei dann jeder Ausdruck durch eine eigene Komposition repräsentiert wird. Diese Verschachtelung wird auf Ebene der Abstraktion jedoch aufgelöst

und durch Abhängigkeiten zwischen den Knoten verschiedener Kompositionen modelliert. Kompositionen ähneln den so genannten Basic-Blocks aus dem Übersetzerbau [2], da Kompositionen ebenfalls immer genau einen Eintritts- sowie Austrittsknoten besitzen. Allerdings werden von Basic-Blocks Sprachkonstrukte auf Maschinencodeebene abstrahiert, während Kompositionen Quelltextkonstrukte abstrahieren. Die Ein- und Austrittsknoten einer Komposition sind vom Typ *Start* bzw. *End* und sind in eigenen Fragmenten geschachtelt. Diese Ein- und Austrittsknoten besitzen jeweils ein eigenes Fragment, damit die Zuordnungen von Kompositionen zu Fragmenten und von Fragmenten zu Knoten möglichst einheitlich gestaltet werden kann. Zusätzliche Fragmente von Kompositionen repräsentieren Eigenschaften wie Operanden von Quelltextelementen.

```
abstract int foo();
void nesting(int i) {
  i = foo() + 1;
}
```

Quelltext 3.1: Quelltext mit verschachteltem Additionsausdruck

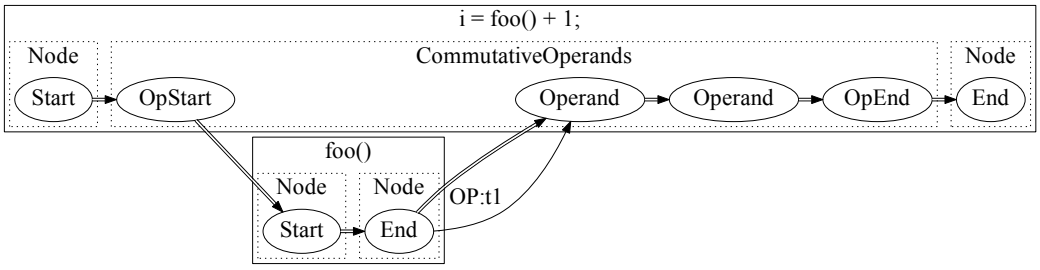


Abbildung 3.3: Verschachtelung in der Abstraktion

Abbildung 3.3 zeigt die Quelltextabstraktion des Programms 3.1. Das Programm besitzt eine Verschachtelung, da der Additionsausdruck den Methodenaufruf einbettet. In der Abstraktion wird die Verschachtelung aufgelöst, indem für jeden Ausdruck eine Komposition erzeugt wird. Außerdem wird für die Leseoperation eines Operanden – hier die Addition – ein Operandenknoten verwendet. Über eine OP-Abhängigkeitskante vom eingebetteten Ausdruck zum zugehörigen Operanden wird die Verschachtelung und Abhängigkeit dargestellt. Die Knoten einer Komposition inklusive der Knoten der eingebetteten Kompositionen sind über einen Pfad aus IS-Kanten miteinander verbunden.

Kompositionen verfügen über verschiedene Attribute, wobei die letzten beiden CSP-Variablen sind. Die ersten drei Attribute sind fix. Das Attribut *nme* gibt Auskunft über den Bezeichner des repräsentierten Ausdrucks. Für die Einfügen-Umgestaltung müssen in der Abstraktion Kopien erzeugt werden, da es passieren kann, dass ein Ausdruck an mehreren Stellen im Quelltext eingefügt werden soll. Daher besitzen Kompositionen das Attribut *cgr*, das auf eine Klongruppe verweist. Mit der Klongruppe speichert die Abstraktion, welche Kompositionen Kopien zueinander sind. Da eine Einfügen-Umgestaltung nicht immer möglich ist, existiert das Attribut *trn*, das angibt, ob der repräsentierte Ausdruck aus seinem übergeordneten Ausdruck prinzipiell eingefügt werden kann. Die CSP-Variable *inl* speichert, ob der repräsentierte Ausdruck im Quelltext eingefügt ist oder einzeln als

eigene Anweisung steht. Die CSP-Variable *cp* gibt an, ob eine Komposition im Zuge einer Einfügen-Umgestaltung kopiert werden soll.

3.3.3 Fragment

Fragmente repräsentieren spezifische Teile eines Quelltextelements. Hierzu gehören die Operanden von Ausdrücken und die Unterelemente von Anweisungen wie bspw. die Bedingung einer Alternative. Fragmente werden mit dem Attribut *typ* in verschiedene Typen unterschieden. Sie enthalten je nach Fragmenttyp einen oder mehrere Knoten. Fragmente vom Typ *Node* besitzen genau einen Knoten und repräsentieren meist den Start oder das Ende einer Komposition. Ferner kann ein *Node*-Fragment auch einen Knoten mit dem Typen *Decl* oder *Aux* besitzen. Solche Fragmente werden genutzt, um den Effekt einer Deklaration oder eines zusätzlichen Quelltextelements in der Abstraktion abzubilden.

Andere Fragmente repräsentieren bspw. Leseeffekte von eingebetteten Ausdrücken einer Anweisung. Sie werden unter dem Begriff *Operandenfragment* zusammengefasst und sind vom Typ *CommutativeOperands*, *NotCommutativeOperands* oder *BlockAbstraction*. Operandenfragmente besitzen Operandenknoten, die die Leseeffekte abbilden. Je mehr Operanden ein Ausdruck wie bspw. eine Addition im Quelltext besitzt, desto mehr Operandenknoten besitzt das zugehörige Operandenfragment. Jedes Operandenfragment beginnt mit einem *OpStart*-Knoten und endet mit einem *OpEnd*-Knoten. Diese beiden Knoten bilden den Ein- und Austrittsknoten des Operandenfragments. Sie sind wichtig, da die Operandenknoten zwischen ihnen möglicherweise im Zuge einer Umgestaltung vertauscht werden. Um die Modellierung der Abhängigkeiten zu vereinfachen, werden die Operandenfragmente anhand ihrer Ein- und Austrittsknoten zu Kompositionen aneinander gekettet. Die Abhängigkeiten, die aufgrund von Datenflüssen und anderen Effekten entstehen, beziehen sich wiederum direkt auf die vertauschbaren Operandenknoten. Fragmente vom Typ *BlockAbstraction* repräsentieren Blöcke im Quelltext, die weitere Anweisungen enthalten können. Blockabstraktionen modellieren jedoch nur die Effekte der Variablen und Ausdrücke und nicht die Quelltextelemente innerhalb des Blocks. Durch Blockabstraktionen werden Programmabhängigkeiten des Blocks zu anderen Quelltextelementen außerhalb des Blocks berücksichtigt. Eine Blockabstraktion kann beliebig viele Knoten vom Typ *Operand* enthalten.

Fragmente modellieren keinen Freiheitsgrad, der für eine der hier betrachteten Umgestaltungen relevant ist. Daher besitzen sie keine CSP-Variablen. Mit dem Attribut *cmp* verweist ein Fragment auf seine zugehörige Komposition.

3.3.4 Knoten

Knoten repräsentieren spezifische Teile eines Fragments. Sie symbolisieren einzelne Schritte des Kontrollflusses innerhalb eines Programms. Jeder Knoten kann Programmeffekte besitzen, wie bspw. das Lesen oder Schreiben einer Variablen oder das Auslösen eines Programmsprungs. Auf Grundlage dieser Knoteneffekte werden später Abhängigkeiten identifiziert und Kanten zwischen Knoten erstellt. Mit dem Attribut fgm verweisen Knoten auf ihr zugehöriges Fragment. Außerdem geben sie mit der CSP-Variablen *np* ihre eindeutige Ausführungsposition an, entsprechend der Reihenfolge, nach der die Knoten der Programmelemente im Programm ausgewertet werden. Der Typ eines Knotens wird

im Attribut *typ* gespeichert. Die Knoten eines Operandenfragments haben einen der Typen *OpStart*, *Operand* oder *OpEnd*. Die CSP-Variable *op* eines *Operand*-Knotens wird verwendet, um zu spezifizieren, an welcher Stelle ein Operand gelesen wird. Bei Knoten anderer Typen ist der Wert vom *op*-Attribut undefiniert. Die anderen Knotentypen *Aux*, *Start* und *End* werden nur für einzelne Knoten genutzt. Ein *Aux*-Knoten wird für Sonderfälle von Java-Anweisungen genutzt und die *Start*- und *End*-Knoten bilden immer Anfang und Ende einer Komposition. Letztere dienen als Eintritt- und Austrittsknoten einer Komposition dazu, den Kontrollfluss zwischen Kompositionen sowie Abhängigkeiten einheitlich zu modellieren.

3.3.5 Klongruppe

In der Quelltextabstraktion können von derselben Anweisung bzw. demselben Ausdruck Klone erzeugt werden. Dies geschieht, wenn im Zuge einer Einfügen-Umgestaltung, ein Quelltextausdruck kopiert werden muss, da er an mehreren Quelltextstellen eingefügt werden soll. In der Abstraktion werden für jede Kopie eines Ausdrucks Kopien von Kompositionen erzeugt. Diese Kopien von Kompositionen werden Klone genannt. Die Information, welche Kompositionen eine Klonbeziehung zueinander besitzen, wird mithilfe von Klongruppen in der Abstraktion modelliert. Wenn zwei Kompositionen eine Klonbeziehung zueinander besitzen, verweisen sie auf dieselbe Klongruppe. Klongruppen geben über ihre Attribute an, ob das Original des zu kopierenden Ausdrucks existieren soll und an welcher Position es sich befindet. Hierfür wird die CSP-Variable *asx* genutzt, die angibt, ob das Original existiert oder gelöscht werden kann. Ein Löschen wird dann notwendig, wenn der einzufügende Ausdruck aufgrund der Einfügen-Umgestaltung an allen Stellen eingefügt werden kann. Die CSP-Variable *cgp* gibt die Position des Originals an, sofern es nach der Umgestaltung noch existiert.

3.3.6 Kante

Kanten repräsentieren Abhängigkeiten zwischen zwei Knoten. Sie verbinden zwei Knoten vom Quellknoten zum Zielknoten mittels der Attribute *src* und *tgt*. Wenn das Attribut *lc* (engl. loop-carried, siehe Abschnitt 2.3.1) falsch ist, gilt, dass der Zielknoten vom Quellknoten abhängt und somit nach dem Quellknoten ausgeführt werden muss. Umgekehrt hängt der Quellknoten vom Zielknoten ab, wenn das Attribut wahr ist. Solche umgekehrten Abhängigkeiten entstehen, wenn der Kontrollfluss aufgrund einer Schleife im Quelltext zurück springt. Der Typ der Abhängigkeit wird im Attribut *typ* gespeichert. Manche Abhängigkeiten, die von den Kanten repräsentiert werden, beziehen sich auf Symbole im Quelltext. Beispielsweise beziehen sich Datenflussabhängigkeiten oft auf spezifische Symbole. Sofern verfügbar, verweisen Kanten daher auch auf das spezifische Symbol (*sym*), das in der Abhängigkeit eine Rolle spielt.

3.3.7 Symbol

Symbole der Abstraktion repräsentieren Symbole aus dem Quelltext. Da in der Abstraktion kein Unterschied zwischen Symbol und Variable gemacht wird, existiert hier nur das Element *Symbol*, das den Namen einer Variablen repräsentiert. Dieser Variablenname

wird durch das Attribut *nme* gespeichert. Dass Attribut *acc* gibt den Geltungsbereich der Variablen an und kann den Wert *Local* oder *Other* besitzen. Der Wert *Local* bedeutet dabei, dass die Variable innerhalb eines Quelltextblocks durch eine Anweisung deklariert wurde. Repräsentiert das Symbol eine Quelltextvariable, die bspw. als Argument oder Instanzvariable deklariert wurde, ist der Wert *Other*. Das Attribut wird im Constraint-Problem dazu genutzt, um entscheiden zu können, ob eine Variable aus einem Block entfernt werden darf. Sofern verfügbar, referenzieren Symbole jene Komposition, die die Deklaration des Symbols enthält (*dcl*). Ferner geben lokale Symbole mithilfe des Attributs *exs* an, ob sie noch verwendet werden.

3.4 Textuelle Notation

Die textuelle Notation wird dazu genutzt, die Abbildung des Quelltexts auf die Abstraktion zu beschreiben (Kapitel 4). Sie wird ebenfalls dazu verwendet, um die Abbildungsregeln zu definieren, die aus einer Abstraktion ein Constraint-Problem erzeugen (Kapitel 5). Im Folgenden wird von einer gegebenen Abstraktion ausgegangen, deren Kompositions-, Fragment-, Knoten-, Kanten-, Klongruppen- und Symbolmengen mit *Compositions*, *Fragments*, *Nodes*, *Edges*, *CloneGroups* und *Symbols* bezeichnet werden. Die Menge der partiellen Abstraktionen der gegebenen Abstraktion wird mit *PartialAbstractions* bezeichnet.

3.4.1 Navigation

Verweist ein Element e über eines seiner Attribute auf ein anderes Element e_a der Abstraktion, kann mithilfe der Punktschreibweise und der Attributbezeichnung ähnlich einer Dereferenzierung auf den Attributwert zugegriffen werden. Demnach verweist der Ausdruck *f.cmp* direkt zur Komposition des Fragments f. Ebenfalls kann über Verkettung zu Unterelementen eines Elements navigiert werden: *f.cmp.cgr* verweist auf die Klongruppe der Komposition *f.cmp*. Beschreibt ein Attribut keine Referenz sondern eine Variable, kann durch die Punktschreibweise auf den Wert dieser Variablen zugegriffen werden. Um bspw. auf den Wert des Attributs *trn* einer Komposition *c* zuzugreifen, genügt die Schreibweise *c.trn*.

> **Attributnavigation:** *Ist e ein Element mit dem Attribut a, kann mit e.a auf den Wert des Attributs zugegriffen werden. Ein Attributwert kann auch eine Referenz auf ein anderes Element der Abstraktion sein.*

Die Rückrichtung dieser Navigation führt von einem Element zu eine Menge von Elementen. Diese Mengen werden in den Definitionen der folgenden Kapitel benötigt. Für die Elemente der Quelltextabstraktion werden die folgenden Rückwärtsnavigationen eingeführt. Eine Rückwärtsnavigation besteht dabei immer aus einem Abstraktionselement, gefolgt von einem Punkt und einem Bezeichner. Wie die zuletzt aufgeführte Rückwärtsnavigation zeigt, ist auch eine verkettete Rückwärtsnavigation möglich. Im nun folgenden Text wird nur dann zwischen Wert-, Element- und Rückwärtsnavigation unterschieden, wenn dies erforderlich ist.

Rückwärtsnavigation: *Besitzt ein Element e über ein Attribut eine Referenz auf ein anderes Element e_p, wird die Navigation von e_p auf all seine Elemente e Rückwärtsnavigation genannt. Für die einzelnen Attribute der Abstraktion wird die Rückwärtsnavigation wie folgt definiert.*

$$
\begin{aligned}
c.F &\triangleq \{f \in Fragments | f.cmp = c\}, c \in Compositions \\
f.N &\triangleq \{n \in Nodes | n.fgm = f\}, f \in Fragments \\
g.C &\triangleq \{c \in Compositions | c.cgr = g\}, g \in CloneGroups \\
n.S &\triangleq \{e \in Edges | e.src = n\}, n \in Nodes \\
n.T &\triangleq \{e \in Edges | e.tgt = n\}, n \in Nodes \\
c.F.N &\triangleq \bigcup_{f \in c.F} (f.N), c \in Compositions \\
a.C &\triangleq \text{alle Kompositionen der partiellen Abstraktion } a \\
A.C &\triangleq \bigcup_{a \in A} (a.C), a \in PartialAbstractions
\end{aligned}
$$

3.4.2 Indizierung

Indizes werden für drei verschiedene Zwecke eingesetzt. Erstens werden tiefgestellte Indizes verwendet, um sonst gleiche Bezeichner voneinander zu unterscheiden (z. B. n_i, n_j). Zweitens werden hochgestellte Indizes inklusive Trennstrich genutzt, um Prädikate anzugeben. Dies wird im folgenden Abschnitt über Kurzprädikate erläutert. Außerdem wird drittens der tiefgestellte Index *INIT* verwendet, um auf den initialen Wert, der vor der Umgestaltung galt, zu verweisen. Außerdem wird der tiefgestellte Index *SOLVED* verwendet, um auf den Variablenwert der Lösung zu verweisen. Bspw. kann mit dem Ausdruck $c.inl_{INIT}$ auf den ursprünglichen Wert von *inl* der Komposition c zugegriffen werden um zu ermitteln, ob c ein Unterausdruck war.

3.4.3 Kurzprädikate

Eine Menge S kann durch eine verkürzte Prädikatschreibweise gefiltert werden. Das annotierte Prädikat P bezieht sich dabei auf alle Elemente $s \in S$. Ist $P(s)$ gültig, so ist s in der resultierenden Menge enthalten.

Kurzprädikat: *Ist S eine Menge und P ein Prädikat über S, dann können Kurzprädikate wie folgt definiert werden:*

$$S|^P \triangleq \{s \in S | P(s)\}$$

Um Mengen zu spezifizieren, können auch Attribute der Elemente einer Menge ausgewertet werden. Dazu wird das entsprechende Attribut, beginnend mit einem Punkt, als Bedingung notiert. Soll bspw. jedes Element einer Menge von Kompositionen das Attribut *inl* mit dem Wert wahr besitzen, kann dies durch folgende Schreibweise ausgedrückt werden.

Attributprädikat: *Ist C eine Menge und att ein Attribut von jedem Element $c \in C$ und ist x der Wert von att, dann können Attributprädikate wie folgt definiert werden.*

Ist das Attribut vom Typ Boolean, dann kann auf die Angabe von wahr bzw. falsch verzichtet werden. Um auszudrücken, dass ein bool'sches Prädikat falsch sein soll, kann das logische Nicht verwendet werden.

$$C|^{.att=x} \triangleq \{c \in C | c.att = x\}$$
$$C|^{.inl} \triangleq \{c \in C | c.inl = wahr\}$$

Häufig werden Mengen nach Typ gefiltert, um bspw. alle Startknoten zu erhalten. Hierfür kann ein Typprädikat als verkürzte Schreibweise zum Filtern nach Typen verwendet werden. Die Typen beginnen dabei immer mit einem großen Buchstaben.

Typprädikat: *Ist S eine Menge mit der Eigenschaft, dass jedes Element in S das Attribut typ besitzt und ist T ein Typ, dann können Typprädikate für Mengen wie folgt definiert werden:*

$$S|^{T} \triangleq S|^{.typ=T}$$

In Anlehnung an Attribut- und Typprädikate für Mengen können auch einzelne Elemente mit hochgestellten Prädikaten versehen werden. Der senkrechte Bedingungsstrich ist dabei optional. Bspw. kann so ein Startknoten n^{Start} einer Gruppe direkt aus einer Menge gewählt werden.

Elementprädikat: *Ist el eine Element, das das Attribut typ besitzt, dann können Typprädikate für Elemente wie folgt definiert werden.*

$$el|^{T} \triangleq el.typ = T$$

3.4.4 Kanten

Kanten stellen Abhängigkeiten dar, die aus dem Quelltext abgelesen und in die Abstraktion überführt wurden. Wird eine Kante mindestens mit Quelle, Ziel und Typ angegeben, ist sie eindeutig und kann als eigener Bezeichner verwendet werden. Für die Notation der Kanten wird statt einer Tupelnotation eine Schreibweise genutzt, die ein Kantensymbol ($\rightarrow$) verwendet und verschiedene Kantenelemente spezifiziert. Diese Notation kann auf eine Tupelnotation abgebildet werden. Im Abschnitt Kantenselektor wird genauer erläutert, wie einzelne Kanten genauer spezifiziert werden. Wird statt des Pfeilsymbols ein Pfeilsymbol mit Doppelspitze ($\twoheadrightarrow$) genutzt, bezeichnet dies immer eine Menge von Kanten. Diese kann ebenfalls, wie im Abschnitt Kantenmengen erläutert, präziser eingeschränkt werden. Andernfalls dient die Kantennotation als Filter, um eine oder mehrere Kanten mit bestimmten Eigenschaften zu beschreiben.

Kanten von/zu/zwischen Kantenpunkten

Kanten verbinden immer zwei Knoten. Um die Notation abzukürzen, sind jedoch auch Schreibweisen erlaubt, bei denen Fragmente oder Kompositionen links oder rechts des Pfeilsymbols stehen. Falls Fragmente oder Kompositionen zur Spezifikation von Kanten verwendet werden, muss definiert sein, welche konkreten Knoten des Abstraktionselements dann für die Kantenspezifikation gemeint sind. Zunächst werden die möglichen Quell- und Zielelemente einer Kante sprachlich vereinheitlicht.

Kantenpunkt, Quellpunkt, Zielpunkt: *Ist e ein Knoten, ein Fragment oder eine Komposition und ist e als Quell- oder Zielelement einer Kante angegeben, dann kann e Kantenpunkt, bzw. Quell- oder Zielpunkt genannt werden.*

Kantendefinition mit Kantenpunkten: *Sind e_s und e_e Kantenpunkte und werden zur Definition einer Kante genutzt, dann werden e_s und e_e nach der Abbildungsvorschrift E2n auf Knoten abgebildet, sofern sie keine Knoten sind.*

Tabelle 3.1 definiert die Abbildungsvorschrift $E2n : Elementklasse \times Typ \times Position \rightarrow Knoten$, die vorgibt, wie Kantenpunkte an einer bestimmten Position der Kantendefinition auf Knoten abgebildet werden. Bspw. verbindet eine Kante $f \rightarrow n$ einen Knoten des Fragments $f \in Fragments|^{CommutativeOperands}$ mit dem Knoten n (Zeile drei in Tabelle 3.1). Da f ein Fragment vom Typ $CommutativeOperands$ und nicht vom Typ $Node$ ist, gilt: $f \rightarrow n \triangleq n_f \rightarrow n, n_f = n|^{OpStart} \in f.N$.

Elementklasse	Typ	Position	Knoten von e
Fragment	$Node$	Quellknoten	$n \in e.N$
Fragment	$Node$	Zielknoten	$n \in e.N$
Fragment	$\neg Node$	Quellknoten	$n\|^{OpStart} \in e.N$
Fragment	$\neg Node$	Zielknoten	$n\|^{OpEnd} \in e.N$
Composition	$*$	Quellknoten	$n\|^{Start} \in e.F.N$
Composition	$*$	Zielknoten	$n\|^{End} \in e.F.N$

Tabelle 3.1: Abbildung $E2n$ eines Elements e auf seinen Knoten

Kantenmengen

Kantenmengen werden mit dem Doppelpfeilsymbol ($\twoheadrightarrow$) notiert und immer dann verwendet, wenn eine Menge von Kanten dargestellt werden soll. Der folgende Kantenmengenselektor bietet die Möglichkeit, die Kantenmenge genauer zu spezifizieren.

Kantenmengenselektor: *Die Menge aller Kanten mit den Eigenschaften, dass ihr Quellpunkt Element der Menge N_s, ihr Zielpunkt Element der Menge N_t, ihr Symbol Element der Menge S, ihr Typ Element der Menge T ist und sie das Kurzprädikat P erfüllt, wird wie folgt notiert. In der Kurzschreibweise können statt der Mengen einzelne Elemente sowie ein Attributprädikat angegeben werden.*

$$N_s \overset{S}{\underset{T}{\twoheadrightarrow}}|^P N_t \triangleq \{e \in Edges | e.src \in N_s \wedge e.tgt \in N_t \wedge e.typ \in T \wedge e.sym \in S \wedge P(e)\}$$

$$n_s \overset{s}{\underset{t}{\twoheadrightarrow}}|^P n_t = \{n_s\} \overset{\{s\}}{\underset{\{t\}}{\twoheadrightarrow}}|^P \{n_t\}$$

Einfache Kantenmengen

Die Kantenmenge $n_s \twoheadrightarrow n_t$ spezifiziert alle Kanten, die den Kantenpunkt n_s Richtung Kantenpunkt n_t verbinden: $n_s \twoheadrightarrow n_t \stackrel{\wedge}{=} \{e \in Edges | e.tgt = n_s \wedge e.src = n_t\}$.

Vor- und Nachbereiche

Ist ein Knoten bekannt, können durch die Vor-/Nachbereichsschreibweise alle ein-/ausgehenden Kanten ermittelt werden. Die Vor-/Nachbereichsschreibweise orientiert sich an der Notation für Kanten, wobei das Pfeilsymbol auf die resultierende Kantenmenge hinweist. Vor- bzw. Nachbereich eines Kantenpunkts werden notiert, indem der Quell- bzw. Zielpunkt unspezifiziert bleibt. Die Menge der resultierenden Kanten kann weiter eingeschränkt werden, indem der gewünschte Kantentyp angegeben wird. Der Vorbereicht wird wie folgt notiert: $\twoheadrightarrow n \stackrel{\wedge}{=} \{e \in Edges | e.tgt = n\}$. Der Nachbereich wird dazu analog definiert, indem statt des Zielpunkt der Quellpunkts angegeben wird.

Eingeschränkte Vor- und Nachbereiche

Die Kantenmengen von Vor- bzw. Nachbereich können auf eine Menge von Quell- bzw. Zielpunkten beschränkt werden. Ist bspw. die Menge aller Kanten im Vorbereich eines Kantenpunkts n gesucht, deren Quellpunkt in der Menge N liegt, kann durch die folgende Notation der Vorbereich eingeschränkt werden. Gleiches gilt für den Nachbereich. Ist E die Menge aller Kanten und N die Menge aller Kantenpunkte, werden alle Kanten, deren Zielpunkte der Kantenpunkt n ist, wie folgt notiert: $N \twoheadrightarrow n \stackrel{\wedge}{=} \{e \in Edges | e.tgt = n \wedge e.src \in N\}$. Analog dazu können auch der Nachbereich, das Symbol und der Kantentyp als Menge angegeben werden.

Symbolkanten

Die Menge aller Kanten eines Symbols kann mit der folgenden Notation beschrieben werden. Wie bei der Vor-/Nachbereichsschreibweise können die Kanten durch zusätzliche Angabe eines Kantentyps weiter eingeschränkt werden: $\stackrel{s}{\twoheadrightarrow} \stackrel{\wedge}{=} \{e \in Edges | e.sym = s\}$.

Typkanten

Die Menge aller Kanten bestimmter Kantentypen kann mit der folgenden Notation beschrieben werden. Wie bei der Vor-/Nachbereichsschreibweise können die Kanten durch zusätzliche Angabe eines Kantentyps weiter eingeschränkt werden: $\underset{t}{\twoheadrightarrow} \stackrel{\wedge}{=} \{e \in Edges | e.typ = t\}$ Durch Angabe mehrerer Kanteneigenschaften können Kantenmengen spezifiziert werden, die bspw. alle Kanten mit Typ Op und mit einem Zielpunkt n_t beschreiben: $\underset{Op}{\twoheadrightarrow} n_t$.

Kantenselektor

Kanten werden mit dem Pfeilsymbol ($\rightarrow$) notiert, das beim Quellpunkt der Kante startet und an ihrem Zielpunkt endet. Soll die Kante zusätzlich typisiert werden, so kann die Typinformation unterhalb der Kante notiert werden. Das Symbol einer Kante wird oberhalb des Pfeilsymbols notiert. Weitere Prädikate werden per Prädikatsstrich spezifiziert. Der

Kantenselektor beschreibt dann eine beliebige Kante aus der angegeben Kantenmenge. Ist die Menge leer, so ist der Kantenselektor nicht definiert.

> **Kantenselektor:** *Eine Kante mit den Eigenschaften, dass ihr Quellpunkt Element der Menge N_s, ihr Zielpunkt Element der Menge N_t, ihr Symbol Element der Menge S, ihr Typ Element der Menge T ist und die Kante das Kurzprädikat P erfüllt, wird wie folgt notiert. In der Kurzschreibweise können statt der Mengen einzelne Elemente sowie ein Attributprädikat angegeben werden.*

$$N_s \underset{T}{\overset{S}{\rightarrow}} |^P N_t \;\in\; N_s \underset{T}{\overset{S}{\twoheadrightarrow}} |^P N_t$$
$$n_s \underset{t}{\overset{s}{\rightarrow}} |^P n_t = \{n_s\} \underset{\{t\}}{\overset{\{s\}}{\rightarrow}} |^P \{n_t\}$$

Beispielsweise bezeichnet $n_s \underset{Op}{\rightarrow} n_t$ eine OP-Kante, die bei n_s startet und auf n_t zielt. Die Kante $n_s \rightarrow |^{.lc} n_t$ spezifiziert hingegen eine Kante mit dem wahren Attribut lc. Hingewiesen sei darauf, dass die Schreibweise $n_s \rightarrow n_t$ nur eine mögliche Kante aus der Menge aller Kanten spezifiziert, die von n_s nach n_t verlaufen.

Kantenpfade

Sollen zwei Kanten spezifiziert werden, die durch einen gemeinsamen Kantenpunkt miteinander verbunden sind, kann eine Kurzschreibweise verwendet werden. In der Kurzschreibweise werden dafür die beteiligten Kantenpunkte direkt hintereinander geschrieben, jeweils getrennt durch das Pfeilsymbol. Die Pfade können beliebig lang sein. Aufeinander folgende Kanten spezifizieren genau i Kanten, wenn $i + 1$ Knoten verbunden werden. In den folgenden Definitionen wird als Abkürzung der Menge *Edges* nur E notiert.

> **Kantenpfad 1:** *Sind n_1, n_2 und n_3 Kantenpunkte, werden zwei Kanten, die diese drei Elemente der Reihe nach verbinden, wie folgt definiert. Kantenpfade können beliebig lang sein.*

$$n_1 \underset{T}{\rightarrow} n_2 \underset{T}{\rightarrow} n_3 \stackrel{\wedge}{=} \{n_1 \underset{T}{\rightarrow} n_2, n_2 \underset{T}{\rightarrow} n_3\}$$

Die Kurzschreibweise für zwei aufeinander folgende Kanten kann auch für Kantenmengen verwendet werden.

> **Paralleler Kantenpfad:** *Sind n_1 und n_3 Kantenpunkte aus der Menge N, wird eine Kantenmenge, die alle Kanten von n_1 zu N sowie von N zu n_3 verbindet, wie folgt notiert.*

$$n_1 \underset{T}{\twoheadrightarrow} N \twoheadrightarrow n_3 \stackrel{\wedge}{=} \{n_1 \underset{T}{\rightarrow} n \in E | n \in N\} \cup \{n \underset{T}{\rightarrow} n_3 \in E | n \in N\}$$

Kantenpunkte einer geordneten Menge werden mit dem Symbol Zickzackkante ($\rightsquigarrow$) sukzessive aneinander gekettet. Dazu muss bekannt sein, welche Reihenfolge die Kantenpunkte besitzen. Hierfür wird eine injektive Funktion genutzt, die bspw. anhand des gegeben Quelltext für jedes Element die zugehörige Quelltextposition liefert.

Kantenpfad 2: *Ist N eine Menge von Kantenpunkten, $Pos : N \to \mathbb{N}$ eine injektive Abbildung eines Kantenpunkts $n \in N$ auf eine natürliche Zahl und T ein Katentyp, wird die Kantenmenge, die die Kantenpunkte von N der Reihe nach verbindet, Kantenpfad über N und Pos genannt und wie folgt notiert.*

$$\overset{N,Pos}{\underset{T}{\rightsquigarrow}} \hat{=} \{n_a \underset{T}{\to} n_b \in E | n_a, n_b \in N \wedge Pos(n_a) < Pos(n_b) \\ \wedge \nexists n_c \in N : Pos(n_a) < Pos(n_c) < Pos(n_b)\}$$

Wird vor oder nach dem Kantenpfad ein Kantenpunkt notiert, wird wie bei den aufeinander folgenden Kanten eine weitere Vorgänger- oder Nachfolgerkante spezifiziert. Nachfolgend wird die Definition des Kantenpfads inklusive Vorgängerkante gegeben, die analog für den Kantenpfad inklusive Nachfolgerkante und auch inklusive Vorgänger- sowie Nachfolgerkante gilt.

Kantenpfad 2 plus Vorgänger- und/oder Nachfolgerkante: *Sind n_s und n_e Kantenpunkte und ist $\overset{N,Pos}{\underset{T}{\rightsquigarrow}}$ ein Kantenpfad über N und Pos mit dem Kantentyp T, sind ferner $n_s \underset{T}{\to} n_0$ die Vorgänger- und $n_n \underset{T}{\to} n_e$ die Nachfolgerkanten des Kantenpfads, werden die Kantenmengen, die eine Vorgänger- und/oder Nachfolgerkante enthalten, wie folgt notiert.*

$$n_s \overset{N,Pos}{\underset{T}{\rightsquigarrow}} \hat{=} \overset{N,Pos}{\underset{T}{\rightsquigarrow}} \cup \{n_s \underset{T}{\to} n_0 \in E | n_0 \in N \wedge \forall n_i \in N : Pos(n_0) \leq Pos(n_i)\},$$

$$\overset{N,Pos}{\underset{T}{\rightsquigarrow}} n_e \hat{=} \overset{N,Pos}{\underset{T}{\rightsquigarrow}} \cup \{n_n \underset{T}{\to} n_e \in E | n_n \in N \wedge \forall n_i \in N : Pos(n_i) \leq Pos(n_n)\},$$

$$n_s \overset{N,Pos}{\underset{T}{\rightsquigarrow}} n_e \hat{=} n_s \overset{N,Pos}{\underset{T}{\rightsquigarrow}} \cup \overset{N,Pos}{\underset{T}{\rightsquigarrow}} n_e$$

3.4.5 Könnteelemente

In Kapitel 4 wird die hier vorgestellte Notation in Abbildungsregeln genutzt, um Java auf Abstraktionselemente abzubilden. Die Abbildungsregeln beziehen sich dabei teilweise auf Abstraktionselemente, die nicht immer im Quelltext existieren, wie bspw. der Sonstblock einer `If`-Anweisung. In solchen Fällen müssten dann Fallunterscheidungen in den Abbildungsregeln gemacht werden. Im Fall einer `If`-Anweisung ist solch eine Fallunterscheidung noch überschaubar, aber andere Sprachkonstrukte wie die `For`-Schleife oder die `Try`-Anweisung bestehen aus wesentlich mehr Quelltextelementen, die optional sein können. Im Fall der `For`-Schleife existieren bspw. acht verschiedene Fälle, da die Initialisierung, die Bedingung, die Aktualisierung und der Schleifenrumpf optional sind. Damit auf die Angabe jeder einzelnen Fallunterscheidung verzichtet werden kann, werden Könnteelemente eingeführt, die je nach Quelltextsituation existieren könnten. Dies erleichtert die Notation der Spezifikationen, da so mithilfe einer einzigen Abbildungsregel alle Fälle definiert werden können. Ob ein optionales Quelltextelement als Könnteelement interpretiert wird, ist eine Designentscheidung der Abbildungsregeln. Welche Elemente der Abstraktion Könnteelemente sein können und wie sie in den Spezifikationen eingesetzt werden können, wird im Folgenden erläutert.

Nullelement

Zur Definition der Könnteelemente wird ein Nullelement benötigt. Hierfür wird das Nullelement eingeführt, dass Teil jeder Menge der Abstraktion ist. Für das Nullelement gilt, dass es die Kardinalität einer Menge nicht beeinflusst.

> **Nullelement:** *Das Nullelement $\perp$ ist Element jeder Menge der Abstraktion. Das Nullelement erhöht die Kardinalität einer Menge nicht. Enthält eine Menge lediglich das Nullelement, wird sie als leer ($\emptyset$) betrachtet. Wird auf ein Element e einer Menge E, die das Nullelement enthält, mit der Schreibweise $e \in E$ oder $\{e\} \subseteq E$ verwiesen, ist e nie das Nullelement.*

Könnteelemente

Mit den Könnteelementen können Elemente wie Knoten spezifiziert werden, die nicht notwendiger Weise existieren. Je nachdem, welche Klasse das Element hat, wird es Könnteknoten, Könntekante oder Könntefragment genannt. Ein Element e existiert dann, wenn es im zugehörigen Quelltext existiert. Für die Definition von Könnteelementen wird vorausgesetzt, dass ein Nullelement $\perp$ in jeder Menge der Abstraktion existiert. Die Definition von Kantenpunkten wird analog auf *Könntepunkte* übertragen, mit der Eigenschaft, dass Könntepunkte optionale Quelltextelemente repräsentieren.

> **Könnteelement:** *Besitzt ein Quelltextelement p ein Unterelement p_u, sind e und e_u ihre zugehörigen Abstraktionselemente, ist ferner p_u laut Sprachdefinition optional, kann e_u als Könnteelement dargestellt werden. Könnteelemente werden mit hochgestelltem Fragezeichen indiziert. Elemente, die mit Könnteelementen spezifiziert wurden, sind ebenfalls Könnteelemente.*
>
> *Ein Element der Abstraktion, das ein optionales Quelltextelement p repräsentiert und als Könnteelement $e^?$ dargestellt wird, verhält sich wie folgt.*

$$e^? \triangleq \begin{cases} e, & falls\ p\ im\ Quelltext\ existiert \\ \perp & sonst \end{cases}$$

> **Könntepunkt:** *Ist e ein Kantenpunkt, kann $e^?$ als Könntepunkt bezeichnet werden.*

Könnteselektor

Könntepunkte werden in der Abstraktion verwendet, um bspw. Kanten zu erzeugen. Für die allgemeine Spezifikation ist dabei irrelevant, ob sie in einem speziellen Quelltextbeispiel tatsächlich existieren oder nicht. Wenn ein Könntepunkt existiert, kann er Teil einer Kantenspezifikation sein. Existiert er nicht, muss eventuell ein anderer Könntepunkt an seine Stelle treten. Um solche Fallunterscheidungen abzukürzen, wird die folgende Schreibweise eingeführt.

Könnteselektor: *Ist $e_1^?$ ein Könntepunkt und ist e_2 ein Kantenpunkt der Abstraktion und repräsentiert $e_1^?$ ferner das Quelltextelement p, wird durch die folgende Funktion einer der beiden selektiert.*

$$[e_1^?||e_2] \stackrel{\wedge}{=} \begin{cases} e_1, & falls\ p\ im\ Quelltext\ existiert \\ e_2 & sonst \end{cases}$$

Diese Schreibweise kann mit beliebig vielen Könntepunkten fortgeführt werden. Dabei wird sie immer von links nach rechts ausgewertet. D. h. dass der Ausdruck zu genau jenem Element auswertet, das als erstes in der Liste existiert. Eine Liste von zwei Könntepunkten und einem Kantenpunkt ergibt sich daher wie folgt:

$$[n_1^?||n_2^?||n_s] \stackrel{\wedge}{=} [[n_1^?||n_2^?]||n_s].$$

Könntekante

Könntekanten werden so notiert, dass sie zwei Kantenpunkte miteinander verbinden, wobei mindestens einer der beiden ein Könntepunkt ist. Da unklar ist, ob der Könntepunkt existiert, ist ebenso unklar, ob die Könntekante selbst existiert. Wenn z. B. n_s als Quell- und $n_e^?$ als Zielpunkt einer Kante definiert sind, wird diese Könntekante wie folgt notiert: $n_s \to n_e^?$.

Könntekante: *Ist mindestens einer der beiden Kantenknoten ein Könntepunkt, wird die Kante als Könntekante bezeichnet.*

Eine Könntekante, deren Zielpunkt $n_e^?$ ein Könntepunkt ist und die das Quelltextelement p repräsentiert, verhält sich wie folgt:

$$n_s \to n_e^? \stackrel{\wedge}{=} \begin{cases} n_s \to n_e, & falls\ p\ im\ Quelltext\ existiert \\ \bot & sonst \end{cases}$$

Eine Könntekante, deren Quellpunkt $n_s^?$ ein Könntepunkt ist und das Quelltextelement p repräsentiert, verhält sich wie folgt:

$$n_s^? \to n_e \stackrel{\wedge}{=} \begin{cases} n_s \to n_e, & falls\ p\ im\ Quelltext\ existiert \\ \bot & sonst \end{cases}$$

Eine Könntekante, deren Quell- und Zielpunkt Könntepunkte sind, verhält sich wie folgt:

$$n_s^? \to n_e^? \stackrel{\wedge}{=} \begin{cases} n_s \to n_e, & falls\ n_s^? \to n_e \neq \bot \wedge n_s \to n_e^? \neq \bot \\ \bot & sonst \end{cases}$$

Könntepfad

Pfade verbinden beliebig viele Kantenpunkte miteinander und erzeugen so Kanten vom ersten über jeden folgenden bis zum letzten Kantenpunkt des Pfads. Ist einer der beteiligten Kantenpunkte des Pfads ein Könntepunkt, entsteht ein Könntepfad. Da unklar ist, ob das optionale Quelltextelement des Könntepunkts existiert, ist folglich auch unklar, welche Kanten der Könntepfad tatsächlich beschreibt. Daher werden Könntepfade in normale Pfade überführt.

Könntepfad: *Ist k ein Pfad und mindestens einer der beteiligten Kantenpunkte ein Könntepunkt, wird k als Könntepfad bezeichnet.*

Teilpfadansicht: *Ist k ein (Könnte-)kantenpfad mit mindestens drei Knoten, der die Knoten n_1, n_2 und n_3 enthält, dann kann k wie folgt dargestellt werden, um den Teilpfad anzuzeigen. Die Vorgänger und Nachfolger des Teilpfads können leer sein.*

$$k = \ldots \to n_1 \to n_2 \to n_3 \to \ldots$$

Reduktionsschritt: *Ist k ein Könntepfad, der den Teilpfad aus den Knoten n_1, $n_2^?$ und n_3 enthält und repräsentiert der Könntepunkt $n_2^?$ das Quelltextelement p, kann k über den folgenden Schritt um einen Könntepunkt reduziert werden.*

$$\ldots \to n_1 \to n_2^? \to n_3 \to \ldots \stackrel{\wedge}{=} \begin{cases} \ldots \to n_1 \to n_2 \to n_3 \to \ldots, & falls\ p\ existiert \\ \ldots \to n_1 \to n_3 \to \ldots & sonst \end{cases}$$

Mithilfe des Reduktionsschritts können Könntepfade auf Pfade zurückgeführt werden. Auf einen Könntepfad k wird dieser Reduktionsschritt für jedes n_2 eines Teilpfads $\ldots \to n_1 \to n_2^? \to n_3 \to \ldots$ von k angewendet. Das Resultat ist ein Pfad ohne Könntepunkte oder ein Pfad, dessen Anfangs- und/oder Endpunkte Könntepunkte sein können. Beide Resultate werden durch die Pfaddefinition auf einzelne Kanten bzw. Könntekanten überführt. Schließlich kann auch als Resultat statt eines Pfads eine einzige Kante bzw. Könntekante entstehen, die ebenfalls bereits definiert wurde.

3.4.6 Positionen

Die einzelnen Knoten werden durch ihre numerische Position im Attribut *np* total geordnet. Die Werte zweier *np*-Variablen sind daher für zwei verschiedene Knoten immer verschieden. Ein Knoten n_q *liegt vor* einem Knoten n_e, wenn der Positionswert des Knotens n_q kleiner als der von n_e ist. Ein Knoten n_q *liegt direkt vor* n_e, wenn der Positionswert von n_q genau um 1 kleiner ist als der von n_e.

3.5 Eigenschaften

Zur Durchführung der Umgestaltungen werden Transformationen benötigt, die das Programmverhalten nicht ändern. Daher werden die PDG-Eigenschaften der Daten- und Sequenzkongruenz werden nun auf die Abstraktion übertragen. Sind die Abstraktionen zweier Programme sequenzkongruent, so gelten ihre Programme als semantisch äquivalent. Sequenzkongruente Transformationen werden genutzt, um Ursprungsprogramme semantikerhaltend umzuformen.

Datenkongruenz: *Zwei Kompositionen sind datenkongruent wenn gilt, dass ihre Operatoren und die Daten ihrer eingehenden Datenflusskanten gleich sind.*

Ferner gilt, dass die Kompositionen keine Instanz und kein Feld erzeugen dürfen.

Sequenzkongruenz (Kompositionen): *Zwei Kompositionen sind sequenzkongruent wenn gilt, dass sie datenkongruent sind und sie innerhalb derselben partiellen Abstraktion stehen.*

Sequenzkongruenz (Transformationen): *Eine Transformation einer Abstraktion ist sequenzkongruent wenn gilt, dass ihre Anwendung auf eine Abstraktion immer eine dazu sequenzkongruente Abstraktion erzeugt.*

Semantik: *Die Semantik eines abgeschlossenen Quelltextblocks ist die Menge all seiner Abhängigkeiten und Operationen. Zwei abgeschlossene Quelltextblöcke sind semantisch äquivalent, wenn ihre Abhängigkeiten und Operationen gleich sind oder sie sequenzkongruent ineinander umgeformt werden können.*

Die Abhängigkeitskanten repräsentieren auch einen Datenfluss, der sich parallel zum Kontrollfluss verhält. Dementsprechend startet eine Kante e meist an einem Quellknoten, der in der Ausführungsreihenfolge vor dem Quellknoten von e liegt. Die Ausnahme bilden loop-carried Kanten, die genau in umgekehrter Richtung verlaufen. Für die Menge $E|^{\neg lc}$ aller nicht-loop-carried Kanten einer Abstraktion gilt, dass der durch sie erzeugte Teilgraph keine Schleifen enthält. Diese Invariante gilt für jede Abstraktion und ist insbesondere auch in Hinblick auf die im folgenden Unterkapitel 3.6 wichtig, da keine Transformation dazu führen darf, dass Schleifen entstehen.

Schleifenfreiheit: *Die nicht-loop-carried Abhängigkeitskanten einer partiellen Abstraktion erzeugen keine Schleifen.*

Ferner besitzen die Knoten der Abstraktion Positionswerte, die eine totale Ordnung der Knoten erzeugen. Diese Ordnung unterliegt jedoch der Ordnung, die durch die Abhängigkeitskanten entsteht. Das bedeutet, dass die Positionswerte die Knoten nur so anordnen dürfen, dass der Quellknoten immer vor dem Zielknoten der Kanten liegt.

Positionskongruenz: *Sind n_1 und n_2 zwei Knoten einer Abstraktion und existiert eine sie verbindende Kante, die keine loop-carried Kante ist, dann beschreibt die folgende Gleichung die Positionskongruenz zwischen Abhängigkeitskanten und Positionswerten.*

$$n_1 \rightarrow |^{\neg.lc}\, n_2 \Rightarrow n_1.np < n_2.np$$

Die Kompositionen bestehen aus Fragmenten, die wiederum aus Knoten bestehen. Jede Komposition besitzt genau einen Start- und einen Endknoten, die Start bzw. Ende des Kontrollflusses einer Komposition markieren. Um zu verhindern, dass sich zwischen zwei Knoten ein dritter Knoten befindet, werden IS-Kanten (engl.: immediate successor, dt.: unmittelbarer Nachfolger) genutzt. Sie verhindern, dass sich die Knoten verschiedener Kompositionen in ihrer Reihenfolge vermischen. Die IS-Kanten der Knoten einer Komposition bilden einen Pfad, der ab Startknoten der Komposition beginnt und an ihrem Endknoten endet.

IS-Kantenpfad: *Die Knoten N jeder Komposition sind beginnend beim Startknoten n_s und endend beim Endknoten n_e über einen IS-Kantenpfad über N miteinander verbunden.*

3.6 Transformationen

Die Umgestaltungen werden in Transformationen aufgespaltet, um ein Ursprungsprogramm bzw. seine Abstraktion ändern zu können. Einige davon verhalten sich analog zu den von PDGs bekannten Transformationen. Andere existieren für PDGs nicht, da PDGs für Maschinensprachen entwickelt wurden. Bspw. unterstützen sie daher keine Deklarationen, die beim Anwenden von Umgestaltungen jedoch u. U. hinzugefügt oder entfernt werden müssen. Die hier vorgestellten Transformationen sind sequenzkongruent und ändern daher die Programmsemantik nicht. Sie sind ferner umkehrbar und können daher in beiden Richtungen angewendet werden.

Die Transformationen werden im vorgestellten Ansatz als Vor- oder Nachbereitungsschritt oder während der Lösungsfindung angewendet. Die während der Lösungsfindung eingesetzten Transformationen werden Freiheitsgrade der Abstraktion genannt. Die Anwendung dieser Transformationen spiegelt sich in einer Änderung der Variablenwerte von Abstraktionselementen wieder. Bspw. bestimmt die Variable *inl* einer Komposition, ob eine Komposition eingefügt oder extrahiert werden soll. Die anderen Transformationen, die als Vor- oder Nachbereitungsschritt verwendet werden, modifizieren die Abstraktion so, dass die intendierten Änderungen vorgenommen werden können. Bspw. werden Klone erzeugt, damit ein Ausdruck an mehreren Stellen im Quelltext eingefügt werden kann. Die genauen Anwendungen der Transformationen werden in den beiden folgenden Kapiteln erläutert.

Neben den in den folgenden Abschnitten erwähnten Bedingungen, die zu jeder Transformation angegeben sind, muss außerdem stets die Schleifenfreiheit erhalten bleiben. Das bedeutet, dass keine Transformation dazu führen darf, dass eine Schleife von Abhängigkeitskanten entsteht. Das Resultat einer solchen Schleife wären Knoten, die gegenseitig von einander abhängig wären. Folglich würde eine Abhängigkeitsschleife eine partielle Ordnung der Knoten anhand der Abhängigkeiten verhindern.

Komposition einfügen/extrahieren

Das Einfügen bzw. Extrahieren von Kompositionen in der Abstraktion verhält sich ähnlich zur PDG-Transformation, die Kantenbeschriftungen hinzufügt oder entfernt. Wie bei PDGs bewirkt diese Transformation auf Quelltextebene, dass eine Zuweisung erstellt bzw. entfernt wird. Diese Transformation wird genutzt, wenn durch eine Umgestaltung Ausdrücke eingefügt oder extrahiert werden sollen.

Komposition einfügen/extrahieren: *Sind c_1 und c_2 Kompositionen und ist n_2 ein Knoten von c_2 und existiert eine OP-Kante von c_1 zu n_2, dann kann c_1 in c_2 eingefügt bzw. aus c_2 extrahiert werden. Durch das Einfügen von c_1 wird der IS-Kantenpfad von c_1 ein Teilpfad des IS-Kantenpfads von c_2. Der Wert von c_1.inl ist genau dann wahr, wenn c_1 eingefügt ist. Die Transformation ist umkehrbar.*

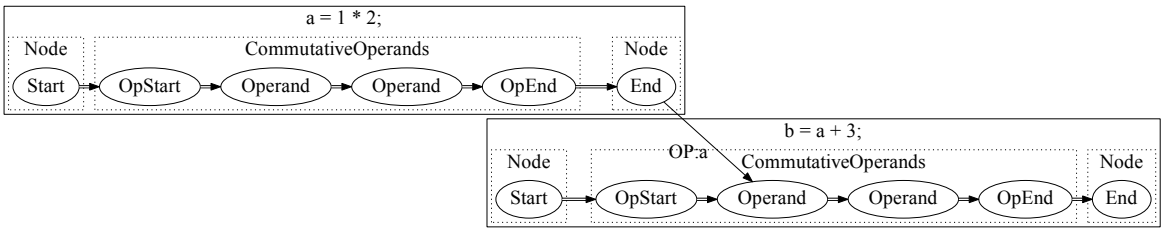
```
int a, b;
void foo() {
  a = 1 * 2; // einfügen
  b = a + 3;
}
```

Quelltext 3.2: Ausdruck extrahiert

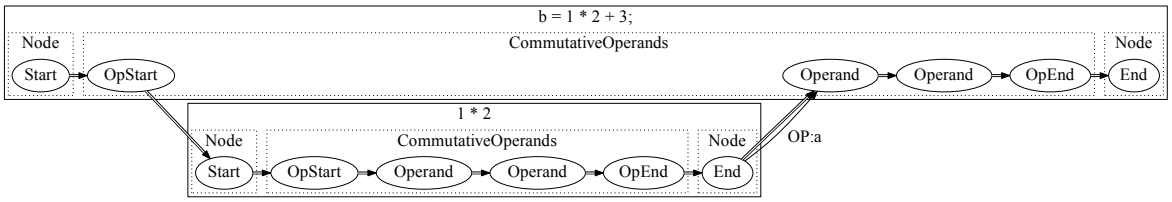
```
int a, b;
void foo() {

  b = 1 * 2 + 3;
}
```

Quelltext 3.3: Ausdruck eingefügt



(a) Abstraktion des Quelltexts 3.2



(b) Transformierte Abstraktion

Abbildung 3.4: Komposition einfügen/extrahieren

Das sich auf Quelltext 3.2 beziehende Beispiel in Abbildung 3.4 zeigt die Änderung, die das Einfügen bzw. Extrahieren einer Komposition bewirkt. Falls die Komposition eingefügt ist, wird der Multiplikationsausdruck als Unterausdruck betrachtet und ebenfalls an seiner Verwendungsstelle eingefügt (siehe Quelltext 3.3). Der IS-Kantenpfad p der Multiplikation wird dadurch in den IS-Kantenpfad der Addition eingebettet. Ist die Komposition extrahiert, existieren die IS-Kanten nicht, die p einbetten. Auf Ebene des Quelltexts entsteht dann eine Zuweisung, die den Wert der Multiplikation in der beschrifteten Variablen speichert (siehe Quelltext 3.2). Ihr Wert wird in der Addition anschließend gelesen. Soll ein Ausdruck an mehreren Stellen eingefügt werden, wird zusätzlich die Transformation *Datendistribution* gebraucht, die im Folgenden erklärt wird.

Transitivität

Die Transformation Transitivität verhält sich analog zur PDG-Transitivität, Fall 1. Auf den Quelltext wirkt sich die Transitivität so ähnlich aus, wie das Hinzufügen oder Entfernen einer Kantenbeschriftung. Denn durch diese Transformation können Zwischenvariablen eingefügt oder entfernt werden.

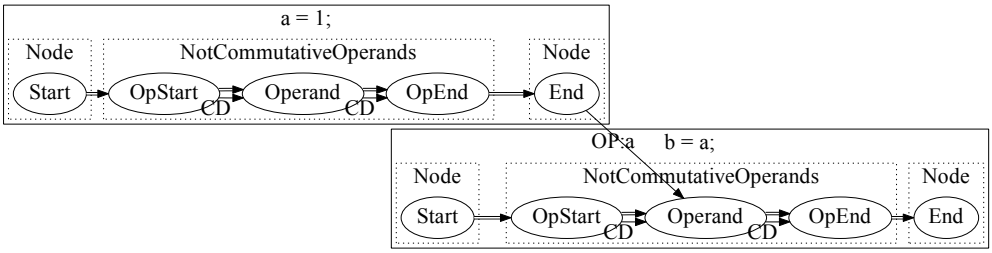
```
int a, b, t;
void foo() {
  a = 1;

  b = a;
}
```

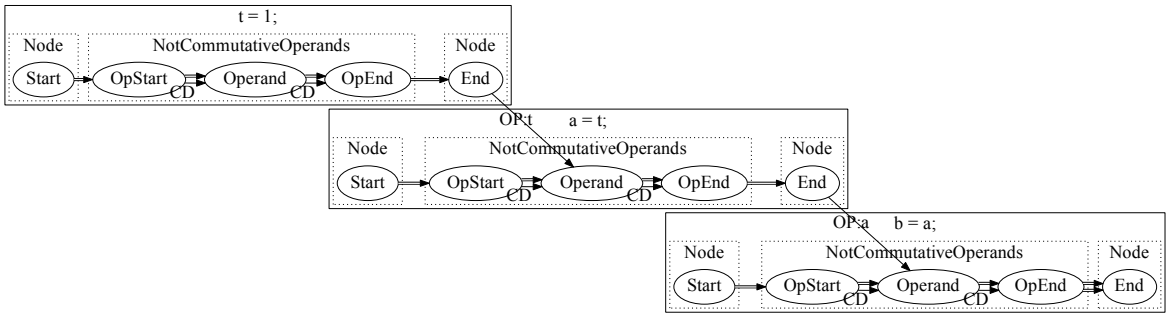
Quelltext 3.4: Ursprungsquelltext

```
int a, b, t;
void foo() {
  t = 1;
  a = t;
  b = a;
}
```

Quelltext 3.5: Transformierter Quelltext



(a) Abstraktion des Quelltexts 3.4



(b) Transformierte Abstraktion

Abbildung 3.5: Komposition einfügen/extrahieren

Transitivität: *Sind c_1 und c_2 Kompositionen und ist n_2 ein Knoten von c_2 und existiert eine OP-Kante von c_1 zu n_2, dann kann wie folgt eine neue Komposition c' eingefügt werden, wobei c' eine Zuweisung und n' den Operandenknoten von c' repräsentiert, der die Leseoperation der rechten Seite von c' darstellt.*

$$c_1 \xrightarrow[OP]{a} n_2 \Leftrightarrow c_1 \xrightarrow[OP]{t} n', c' \xrightarrow[OP]{a} n_2$$

Das Beispiel in Abbildung 3.5 veranschaulicht die Transitivität. Als notwendige Bedingung dieser Transformation muss eine Kante des Typs OP existieren. Existiert sie, kann eine neue Komposition c' erstellt werden, deren Operation eine Zuweisung ist. Diese Zuweisung weist den Wert des Ausdrucks der neu eingeführten Variablen `t` zu. Ihr Wert wird direkt im Anschluss der ursprünglichen Variablen `a` zugewiesen, die weiterhin im Programm verwendet wird.

Die Transitivität wird im vorgestellten Ansatz ebenfalls genutzt, um das Einfügen/Extrahieren von Ausdrücken zu ermöglichen. Der zweite Fall der PDG-Transitivität wird in der Abstraktion nicht benötigt und daher nicht auf die Abstraktion überführt.

Datendistribution

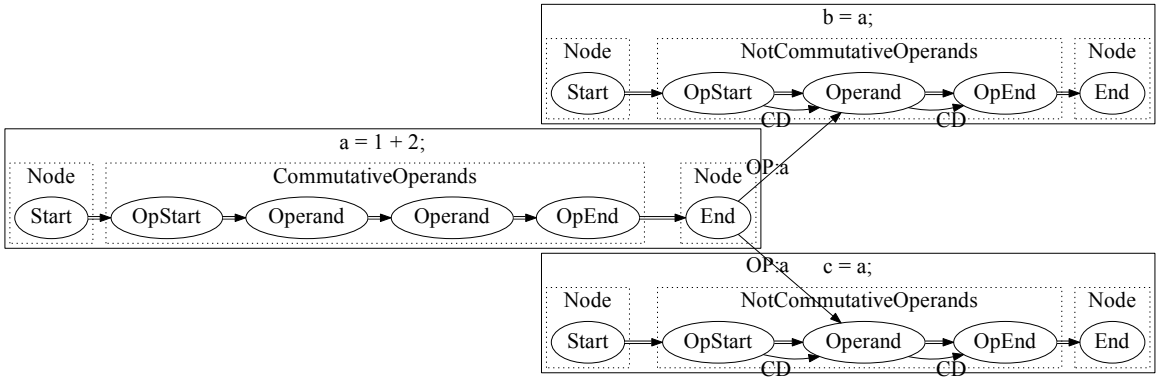
Die letzte Transformation der Abstraktion, die sich an PDG-Transformationen anlehnt, ist die Datendistribution. Genau wie bei PDGs können auch Kompositionen kopiert werden. Dieses Kopieren wird in dieser Arbeit auch Klonen genannt. Das Kopieren ist notwendig, um Umgestaltungen zu ermöglichen, bei denen ein Ausdruck an mehreren Stellen eingefügt werden soll, da in solchen Situationen der einzufügende Ausdruck kopiert werden muss. Beim Erstellen einer Kopie wird eine neue, sequenzkongruente Komposition erstellt. Diese befindet sich meist in der gleichen partiellen Abstraktion, repräsentiert die gleiche Operation und ist folglich auch bzgl. ihrer Fragmente und Knoten gleich. Allerdings können sich zwei sequenzkongruente Kompositionen in Hinblick auf ihre ausgehenden Datenflusskanten unterscheiden. Die Kanten der Typen OP, FD und SE repräsentieren Datenflüsse, die entstehen, wenn am Quellknoten ein Wert geschrieben wird, der am Zielknoten gelesen wird. Wird eine Kopie einer Komposition erzeugt, besitzt sie mindestens einen dieser Datenflüsse zu einem der Zielknoten. Daher können sich die Kopien bzgl. ihrer OP-, FD- und SE-Kanten unterscheiden. In der folgenden Definition werden OP-, FD- und SE-Kanten abgekürzt als DA-Kanten (Datenabhängigkeit) bezeichnet. Außerdem wird das Prädikat *Zielgleich* verwendet, das sich auf zwei Kantenmengen K_1 und K_2 bezieht. Es ist wahr, wenn für jede Kante k_a in K_1 bzw. K_2 eine Kante k_b in K_2 bzw. K_1 existiert, für die gilt, dass sich k_a und k_b mit Ausnahme ihrer Startknoten nicht unterscheiden.

> **Datendistribution:** *Sind c_1, c_1' und c_1^* sequenzkongruente Kompositionen, dann kann c_1 in c_1' und c_1^* durch die Datendistribution transformiert werden, wenn die folgenden Eigenschaften gelten. Die Transformation ist umkehrbar.*
>
> 1. $Zielgleich(c_1.F.N \underset{\neg DA}{\twoheadrightarrow}, c_1'.F.N \underset{\neg DA}{\twoheadrightarrow})$
> 2. $Zielgleich(c_1.F.N \underset{\neg DA}{\twoheadrightarrow}, c_1^*.F.N \underset{\neg DA}{\twoheadrightarrow})$
> 3. $c_1 \underset{DA}{\twoheadrightarrow} = c_1' \underset{DA}{\twoheadrightarrow} \cup c_1^* \underset{DA}{\twoheadrightarrow}$
> 4. $|c_1 \underset{DA}{\twoheadrightarrow}| > 0 \ \wedge |c_1' \underset{DA}{\twoheadrightarrow}| > 0 \ \wedge |c_1^* \underset{DA}{\twoheadrightarrow}| > 0$

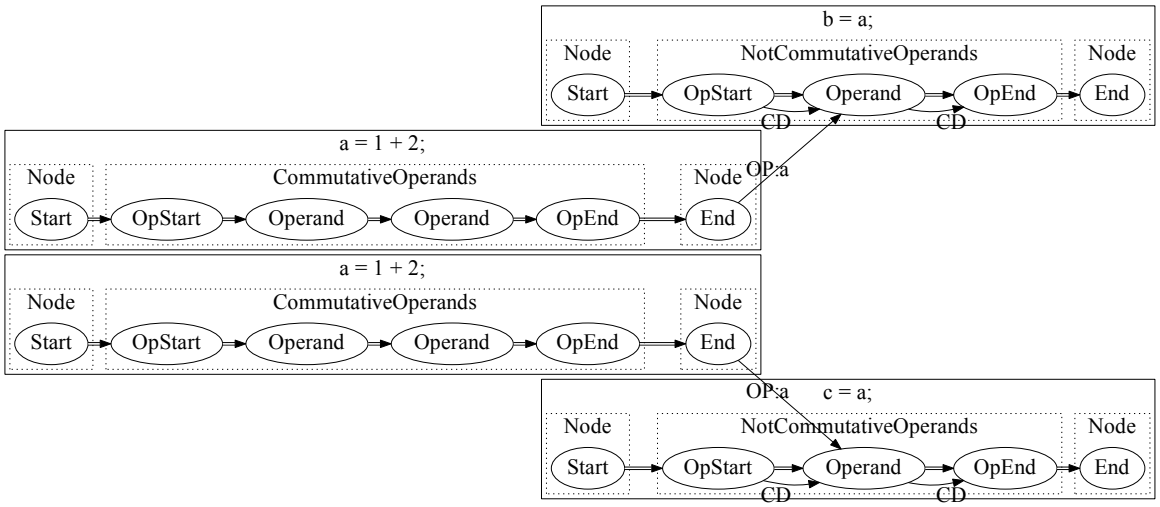
```
int a,b,c;
void foo() {
  a = 1 + 2;
  b = a;
  c = a;
}
```

Quelltext 3.6: Beispiel für Datendistribution

Die Bedingungen in der Definition der Datendistribution sind, dass die Mengen aller Kanten, die von einem der Knoten von c_1, c_1' oder c_1^* aus starten, zielgleich sind (Bedingungen 1 und 2). Außerdem muss gelten, dass die Menge der DA-Kanten, die von den durch die Distribution erzeugten Kompositionen c_1' und c_1^* aus starten, gleich ist zu der Menge von DA-Kanten, die von c_1 aus starten (Bedingung 3). Schließlich müssen sowohl die erzeugten Kompositionen c_1' und c_1^* als auch die ursprüngliche Komposition c_1 mindestens eine DA-Kante besitzen (Bedingung 4);



(a) Datendistribution nicht durchgeführt



(b) Transformierte Abstraktion

Abbildung 3.6: Datendistribution

In Abbildung 3.6 wird ein Beispiel für die Datendistribution gezeigt. Anlass dieser Transformation ist, dass eine Variable an zwei Stellen im Quelltext verwendet wird. Die Forderung, dass für jede dieser Stellen eine Komposition existieren soll, bewirkt die Erstellung einer Kopie der Komposition. Daher dient die Datendistribution als Hilfstransformation, um Ausdrücke im Quelltext an mehreren Stellen einfügen zu können. Die durch sie erzeugten Kopien sind zunächst nicht im Quelltext sichtbar und wirken sich nur auf die Abstraktion aus. Wird außerdem die Transformation *Komposition einfügen* angewendet, werden die Kopien sichtbar, da sie im Quelltext eingefügt werden. Wie das Abstraktionsbeispiel zeigt, sind die ursprüngliche und die kopierte Komposition mit Ausnahme ihrer Kanten gleich. Jede der beiden Kompositionen hat eine andere OP-Kante. Die Vereinigungsmenge ihrer OP-, FD- und SE-Kanten jedoch entspricht der Kantenmenge dieser Typen der ursprünglichen Komposition.

Da die Datenkongruenz nicht gilt, wenn der zu kopierende Ausdruck eine Instanziierung oder einen Feldzugriff repräsentiert, kann die Datendistribution in solchen Fällen nicht

angewendet werden. Andernfalls würden Kopien einer Instanziierung erstellt werden können, deren Referenzen im Quelltext auf verschiedene Instanzen zeigen würden.

Operanden umsortieren

Die Leseoperation von Operanden im Quelltext wird durch Operandenknoten in Operandenfragmenten repräsentiert. Zu den Operandenfragmenten gehören die kommutativen Fragmente der Typen *CommutativeOperands* und *BlockAbstraction*. Bei ihnen ist ein Umsortieren der Operandenknoten mithilfe dieser Transformation möglich.

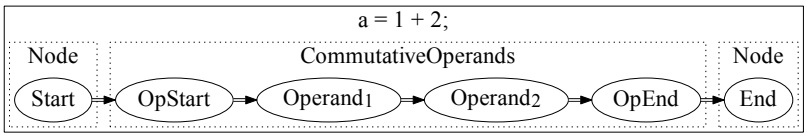
> **Operanden umsortieren:** *Ist co ein Fragment vom Typ CommutativeOperands oder BlockAbstraction und ist N die Menge aller Knoten von co, wobei n_1 der OpStart- und n_i der OpEnd-Knoten von co sind, dann können die Knoten in N beliebig durch einen IS-Kantenpfad p verbunden sein, solange n_1 der Start und n_i das Ende von p sind. Die Transformation ist umkehrbar.*

```
int a;
void foo() {
  a = 1 + 2;
}
```

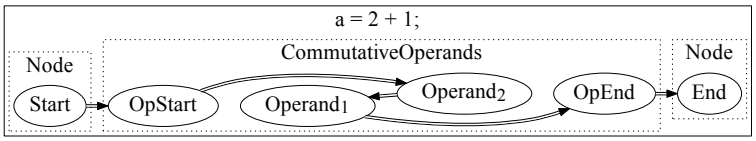
Quelltext 3.7: Ursprungsquelltext

```
int a;
void foo() {
  a = 2 + 1;
}
```

Quelltext 3.8: Vertauschte Operanden



(a) Abstraktion des Quelltexts 3.7



(b) Transformierte Abstraktion

Abbildung 3.7: Datendistribution

Beim Umsortieren von Operanden werden die Operandenknoten eines Fragments neu angeordnet. Das Beispiel in Abbildung 3.7 zeigt einen Operandentausch für zwei Operanden einer Addition. Zu jeder Zeit sind die Knoten des betroffenen Fragments über einen IS-Kantenpfad miteinander verbunden. Die Reihenfolge der Operandenknoten wurde im Beispiel vertauscht. Sie kann aber prinzipiell beliebig gewählt werden. Wie bei allen Transformationen gilt für das Umsortieren von Operanden, dass die Schleifenfreiheit mit anderen, bestehenden Abhängigkeiten erhalten bleiben muss.

Anweisungen umsortieren

Das Umsortieren von Anweisungen im Quelltext wird in die Abstraktion übertragen, indem die Positionswerte der Knoten, Kompositionen und ggf. auch der Klongruppen verändert werden. Die partielle Ordnung der Knoten durch die Abhängigkeitskanten schränkt jedoch die Positionswerte der Knoten und der sich daraus ergebenen Positionswerte der Kompositionen und Klongruppen ein. Existiert bspw. eine Abhängigkeit zwischen zwei Anweisungen im Quelltext, wird diese auf eine Abhängigkeitskante zwischen zwei Knoten der Kompositionen abgebildet. Folglich können diese beiden Knoten und damit auch ihre Kompositionen in ihrer Reihenfolge nicht miteinander vertauscht werden.

```
1   int a, b;
2   void foo() {
3     a = 1;
4     a++;
5     b++;
6   }
```

Quelltext 3.9: Anweisungen umsortieren

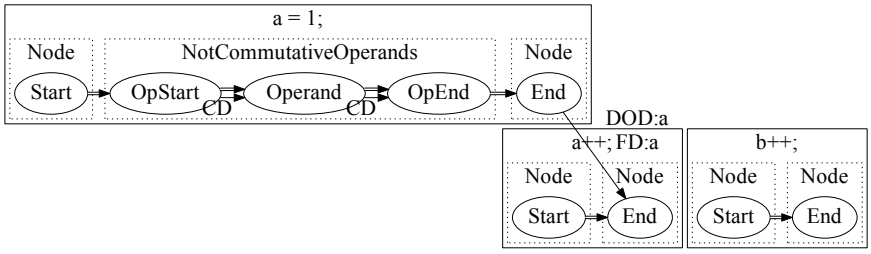


Abbildung 3.8: Anweisungen umsortieren

Das Beispiel in Abbildung 3.8 zeigt die Kompositionen von drei Anweisungen aus Quelltext 3.9. Während die ersten beiden Anweisungen (Zeilen 3 und 4) eine Schreib-/Leseabhängigkeit besitzen und daher nicht vertauscht werden können, besteht keine Abhängigkeit zwischen den letzten beiden Anweisungen (Zeilen 4 und 5). Daher können die Kompositionen der ersten beiden Anweisungen nicht vertauscht werden, während die Kompositionen der unteren beiden Anweisungen vertauscht werden können. Das Umsortieren der Anweisungen wirkt sich nur auf die Positionswerte der Knoten und Kompositionen aus. Demnach spiegelt die graphische Darstellung der Abstraktion die Änderungen der Anweisungsreihenfolge nicht wieder. Die Transformation *Anweisung umsortieren* ergibt sich aus der Abstraktionseigenschaft der Positionskongruenz.

Deklaration umwandeln

Im Quelltext können Variablen auf verschiedene Arten deklariert und initialisiert werden. Bspw. können die Deklaration und Initialisierung gemeinsam in einer (z. B. `int i=1;`) oder getrennt in zwei Anweisungen (z. B. `int i; i=1;`) durchgeführt werden. Ferner kann eine Deklaration beliebig viele Variablen deklarieren. Die Abstraktion kann diese verschiedenen Arten abbilden. Jedoch wird der Freiheitsgrad für die Umgestaltung *Ausdruck einfügen*

nur dann unterstützt, wenn die Deklaration und Initialisierung getrennt durchgeführt werden. Außerdem ist das Löschen einer Variablendeklaration nur dann möglich, wenn die die Deklaration lediglich eine Variable deklariert. Daher ist eine Umwandlung anderer Deklarationsarten notwendig, um das Einfügen von Ausdrücken bzw. Löschen von Variablendeklarationen in den vorgestellten Ansatz zu integrieren. Die beiden folgenden Transformationen werden deswegen als Vorbereitungsschritt durchgeführt, wenn die Umgestaltung *Ausdruck einfügen* auf eine Variable einer Deklaration angewendet wird. Der erste Fall der Umwandlung einer Deklaration spaltet die Initialisierung einer Deklaration ab und erzeugt eine zusätzliche Zuweisung. Im zweiten Fall der Umwandlung wird eine Deklaration, die mehrere Variablen deklariert, in einzelne Deklarationsanweisungen aufgespaltet.

Deklaration umwandeln 1: *Ist c eine Komposition, die eine Variablendeklarationsanweisung repräsentiert und die genau eine Variable mitsamt Initialisierung deklariert, dann kann c so in die Kompositionen c_d und c_i so zerlegt werden, dass c_d die Deklaration repräsentiert und c_i eine Zuweisung, deren rechte Seite die Initialisierung von c ist. Die durch die Zerlegung entstehende Kante $n_d \underset{INIT}{\rightarrow} n_w$ startet vom Deklarationsknoten in c_d und endet beim Endknoten von c_i. Die Transformation ist umkehrbar.*

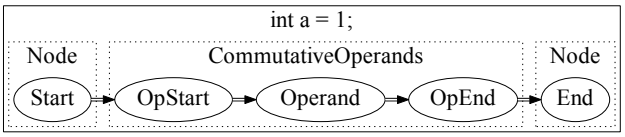
```
void foo() {
  int a = 1;

}
```

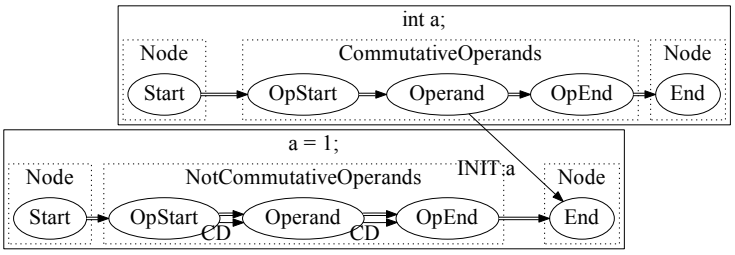
Quelltext 3.10: Ursprungsquelltext

```
void foo() {
  int a;
  a = 1;
}
```

Quelltext 3.11: Deklaration entfernt



(a) Abstraktion des Quelltexts 3.10



(b) Transformierte Abstraktion

Abbildung 3.9: Deklaration umwandeln 1

Das Beispiel in Abbildung 3.9 zeigt eine Deklarationsanweisung, die eine Variable deklariert und initialisiert. Die Transformation bewirkt, dass die Deklarationsanweisung die Initialisierung verliert. Im Gegenzug wird eine Zuweisung erstellt, deren rechte Seite der

Initialisierungsausdruck ist. Außerdem wird eine Abhängigkeitskante in der Abstraktion von der Deklaration zur Initialisierung erzeugt. Diese Kante bewirkt, dass sich die Komposition der Deklaration immer vor der Komposition der Initialisierung befindet.

Deklaration umwandeln 2: *Ist c eine Komposition, die eine Variablendeklarationsanweisung repräsentiert und die n Variablen deklariert, dann kann c in die Kompositionen c_1 bis c_n so zerlegt werden, dass jedes $c_i, i = 1..n$ eine eigene Variablendeklarationsanweisung repräsentiert.*

$$c \Leftrightarrow c_1, \ldots, c_i$$

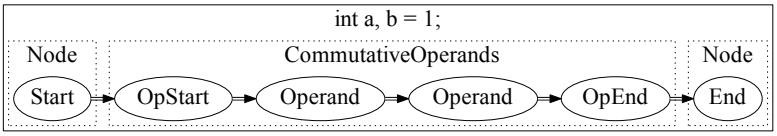
```
void foo() {
  int a, b = 1;

}
```

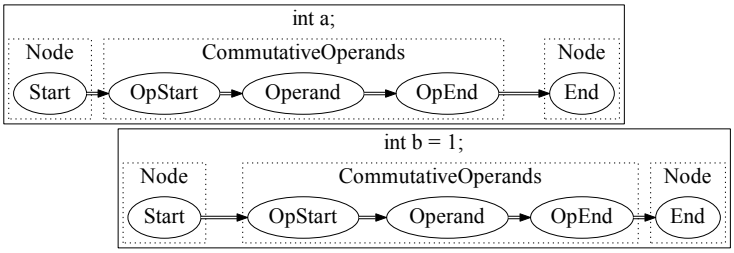
Quelltext 3.12: Ursprungsquelltext

```
void foo() {
  int a;
  int b = 1;
}
```

Quelltext 3.13: Deklaration entfernt



(a) Abstraktion des Quelltexts 3.12



(b) Transformierte Abstraktion

Abbildung 3.10: Deklaration umwandeln 2

Das Beispiel in Abbildung 3.10 zeigt eine Deklarationsanweisung, die zwei Variablen deklariert. Ob die einzelnen Variablen ebenfalls initialisiert werden, ist für die Transformation irrelevant. Die Transformation bewirkt, dass für jede deklarierte Variable eine eigene Deklarationsanweisung existiert. Eine etwaige Initialisierung der Variablen wird in die neue Anweisung übernommen. Das Ergebnis dieser Transformation kann daher durch die Transformation *Deklaration umwandeln 1* weiter transformiert werden.

Deklaration hinzufügen/entfernen

Deklarationen einer lokalen Variablen können dann im Quelltext entfernt werden, wenn die Variable an keiner Stelle gelesen wird. Ein solcher Fall tritt ein, wenn die Variable zuvor als Zwischenvariable verwendet wurde, der ihr zugewiesene Ausdruck aber mittlerweile

aufgrund der *Ausdruck einfügen*-Umgestaltung an der ehemaligen Lesestelle der Variablen steht. Wird ein Variable auf diese Weise obsolet, kann ihre Deklaration im Quelltext entfernt werden und folglich auch die sie repräsentierende Komposition in der Abstraktion. Umgekehrt muss eine Variablendeklaration eingeführt werden, wenn ein ehemaliger Unterausdruck herausgelöst und einer neuen Zwischenvariablen zugewiesen wird. Dies tritt ein, wenn die Umgestaltung *Ausdruck extrahieren* durchgeführt wird.

Deklaration entfernen/hinzufügen: *Ist c_d eine Komposition, die eine Variablendeklarationsanweisung der Variablen v ohne Initialisierung repräsentiert und existieren keine Kanten in der Menge $\overset{v}{\twoheadrightarrow}$ vom Typ FD oder SE, dann existiert c_d genau dann, wenn die Variable $n.fgm.cmp.inl$ wahr ist für jeden Startknoten n einer Kante in $\overset{v}{\twoheadrightarrow}$. Die Transformation ist umkehrbar.*

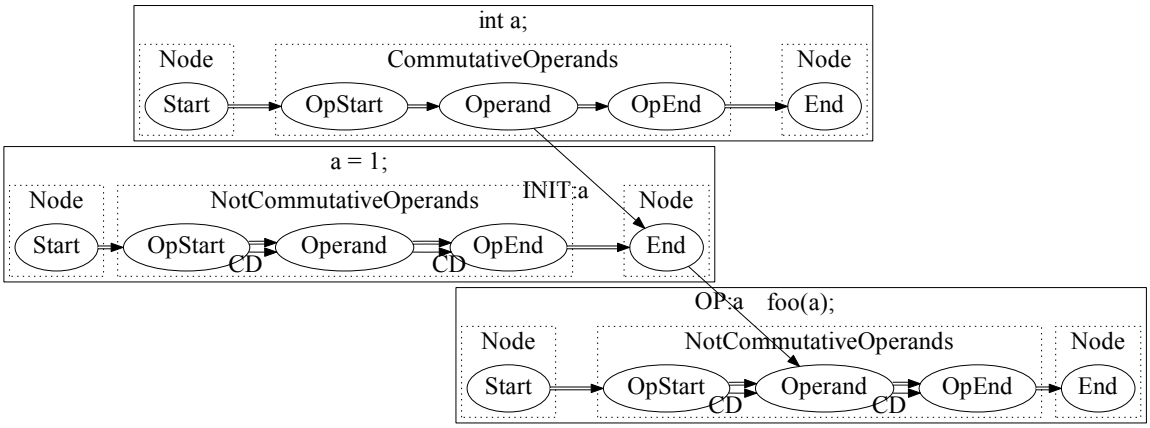
```
void foo(int x) {
  int a;
  a = 1;
  foo(a);
}
```

Quelltext 3.14: Ursprungsquelltext

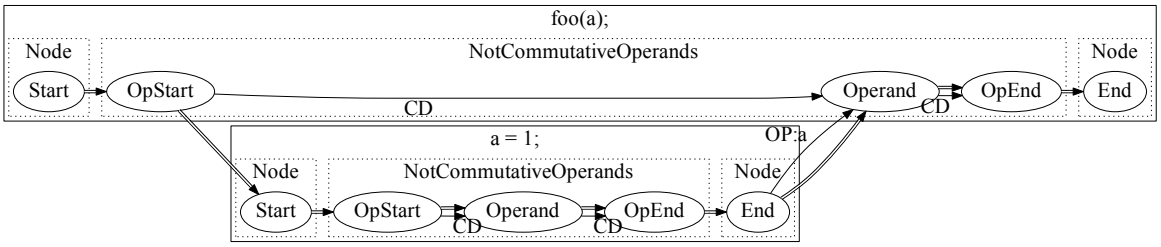
```
void foo(int x) {


  foo(1);
}
```

Quelltext 3.15: Deklaration entfernt



(a) Abstraktion des Quelltexts 3.14



(b) Transformierte Abstraktion

Abbildung 3.11: Deklaration hinzufügen/entfernen

Das Beispiel in Abbildung 3.11 zeigt die Abstraktion eines Programms (Quelltext 3.14), das aus drei Anweisungen besteht. Da durch die Umgestaltung *Ausdruck einfügen* das

Programm so modifiziert wird, dass die Variable `i` weder geschrieben noch gelesen wird, wird `i` obsolet. Folglich kann die Deklaration dieser Variablen aus dem Programm entfernt werden (Quelltext 3.15). Umgekehrt wird eine Deklaration von i dem Programm hinzugefügt, wenn `i` als Zwischenvariable eingeführt wird. In diesem Fall ist außerdem wichtig, dass eine Abhängigkeitskante in der Abstraktion von der Deklaration zur Initialisierung erzeugt wird. Diese Kante bewirkt, dass sich die Komposition der Deklaration immer vor der Komposition der Initialisierung befindet.

Zusammenfassung

In diesem Kapitel wurde die Quelltextabstraktion vorgestellt, die die Grundlage des Ansatzes dieser Arbeit bildet. Mithilfe der Abstraktionselemente werden Quelltextelemente und relevante Zusammenhänge dargestellt. Die eingeführte Notation wird verwendet, um Elemente und ihre Eigenschaften zu beschreiben. Hervorzuheben sind Könnteelemente, die das Beschreiben von optionalen Quelltextkonstrukten stark vereinfachen. Die von den PDGs bekannte Eigenschaft der Sequenzkongruenz wurde für die Abstraktion und ihre Transformationen übernommen. Sie erklärt den Erhalt der Semantik der genutzten Transformationen. Die Transformationen werden an verschiedenen Stellen im Ansatz angewendet und dienen hauptsächlich dazu, die Freiheitsgrade umzusetzen und somit die Überführung in ein Constraint-Problem zu ermöglichen.

Kapitel 4

Abbildung von Java 7

Für die im vorigen Kapitel eingeführte Quelltextabstraktion wird nun vorgestellt, wie aus Java-7-Programmen eine Quelltextabstraktion erzeugt wird. Hierfür wird in mehreren Schritten vorgegangen.

$$P, W \xrightarrow[Abbildungsregeln]{} A_0 \xrightarrow[Wunschanpassung]{} A_1 \xrightarrow[Abhängigeitserkennung]{} A_2 \xrightarrow[Klonerzeugung]{} A_3$$

Abbildung 4.1: Von Programm P und Umgestaltungswunsch W zur Abstraktion A_3.

Zunächst wird die im letzten Kapitel eingeführte Notation um Kontrollflusskanten erweitert, die für dieses Kapitel wichtig sind. Außerdem werden Abbildungsregeln eingeführt, die eine Abbildungsvorschrift von Java-Quelltextelementen auf die Abstraktion beschreiben. Darauf aufbauend werden die Quelltextelemente auf Abstraktionselemente abgebildet (Unterkapitel 4.2). Diese Abbildungen ergeben die initiale Abstraktion A_0 eines Quelltexts. Die anschließende *Wunschanpassung* modifiziert die Abstraktion leicht, um den Umgestaltungswunsch umsetzen zu können und erzeugt A_1. Bspw. ist die Umgestaltung *Ausdruck einfügen* im Constraint-Problem nur für Zuweisungen definiert und nicht für Deklarationen mit Initialisierungen. Die Wunschanpassung würde eine Deklaration mit Initialisierung auf eine Deklaration und eine Zuweisung transformieren (siehe Transformation *Deklaration umwandeln 1*, Abschnitt 3.6). Sie wird im Abschnitt 5.1.4 im Zusammenhang mit der Umgestaltung *Ausdruck einfügen/extrahieren* präsentiert. Danach erzeugt eine *Abhängigkeitserkennung* (Unterkapitel 4.3) die Abstraktion A_2. Die Kontrollflussinformationen werden hier genutzt, um Abhängigkeitskanten zu erstellen. Die für diese Abhängigkeitserkennung ebenfalls benötigten Informationen über Programmeffekte, wie das Lesen oder Schreiben von Variablen, werden aus dem Quelltext abgelesen. Im letzten Schritt wird die Klonerzeugung durchgeführt, sodass schließlich die Abstraktion A_3 entsteht. Die Klonerzeugung ist notwendig, um die Umgestaltung *Ausdruck einfügen* im Constraint-Problem zu ermöglichen.

Wurde die Abstraktion mitsamt aller Abhängigkeiten erzeugt, sind die Kontrollflusskanten und Effektinformationen für das weitere Vorgehen nicht mehr relevant. Aus der Abstraktion A_3 wird anschließend ein Constraint-Problem erstellt, das das Umgestaltungsproblem lösen soll (Kapitel 5). Da die Evaluation nicht auf Basis der Java-Sprachdefinition, sondern auf Basis von JDT durchgeführt wird, wird die Abbildungsvorschrift ebenfalls auf Basis von JDT angegeben. Ferner richten sich die hier vorgestellten Abbildungsregeln nach dem Stand der Implementierung und sind somit durch die Evaluation getestet.

4.1 Erweiterte Notation

Im Folgenden werden Kontrollflusskanten und Abbildungsregeln eingeführt, die in diesem Kapitel benötigt werden, um eine Quelltextabstraktion inklusive der Programmabhängigkeiten zu erzeugen.

4.1.1 Kontrollflusskanten

Kontrollflusskanten werden innerhalb dieses Kapitels als eine zusätzliche Kantenart verwendet. Hierfür wird die Kantenmenge $ControlFlow$ eingeführt. Kontrollflusskanten dienen dazu, den Kontrollfluss innerhalb und zwischen JDT-Elementen zu spezifizieren und, darauf aufbauend, Datenflussanalysen durchführen zu können. Sie unterscheiden sich von den Kanten der Abstraktion nur dadurch, dass sie immer Elemente der Kantenmenge $ControlFlow$ sind, wenn sie existieren. Die Kantennotation entspricht der bisher eingeführten Notation. Eine Kontrollflusskante ist bereits durch Angabe von Quelle und Ziel eindeutig definiert und kann so als Bezeichner verwendet werden. Da die Angabe eines Kantentyps überflüssig ist, wird die Kante mit gestricheltem Symbol dargestellt. In den Abbildungen wird auf die Darstellung von Kontrollflusskanten meist verzichtet. Falls sie doch abgebildet werden, werden sie durch eine gestrichelte Linie gekennzeichnet.

4.1.2 Abbildungsregeln

Die Erzeugung der Abstraktion aus abgeschlossenen Quelltextblöcken wird mithilfe von Abbildungsregeln einzeln für jedes JDT-Element beschrieben. Gemeinsam ergeben sie eine Abbildungsvorschrift von einem abgeschlossenen Java-Quelltextblock zu einer Abstraktion. Diese Regeln nutzen die in Kapitel 3.4 eingeführte Notation. Die in dieser Arbeit genutzten Abbildungsregeln bestehen aus den drei Teilen *Allquantor*, *Existenzquantor* und *Prädikatteil*.

> **Abbildungsregel:** *Eine Abbildungsregel besteht aus einem Allquantor über eine Menge von JDT-Elementen, einem Existenzquantor über Elemente der Quelltextabstraktion und Prädikaten, die die Abstraktionselemente genauer bestimmen.*

Mit dem Allquantor wird zuerst definiert, für welche JDT-Elemente die Regel anzuwenden ist. Die JDT-Elemente werden entsprechend ihres JDT-Klassennamens, aber ohne Eclipse-Logo, bezeichnet. Die Menge der JDT-Elemente bezieht sich dabei auf alle JDT-Elemente, die innerhalb des `Blocks` der betrachteten partiellen Abstraktion existieren. Mit dem Existenzquantor wird spezifiziert, auf welche Elemente der Abstraktion diese JDT-Elemente abgebildet werden. Werden Könnteelemente in diesem Teil spezifiziert, bedeutet dies, dass diese Elemente genau dann in der Abstraktion erzeugt werden, wenn die Bedingung dafür, dass sie existieren, erfüllt ist. Schließlich können im Prädikatteil nach dem Doppelpunkt durch Konjunktion zusätzliche Eigenschaften festgelegt werden. Kanten werden hierbei immer mit Quell- und Zielknoten spezifiziert. Ist eine Kante eine Kontrollflusskante, wird auf die Angabe des Typs verzichtet, da sie auch ohne Typangabe eindeutig spezifiziert ist.

$$
\begin{aligned}
&\forall e \in JDTElements \\
&\exists \\
&c \in Compositions, \{n_s, n_e\} \subseteq Nodes, \{f_s, f_e\} \subseteq Fragments|^{Node} \\
&: \\
&n_s|^{Start} \wedge n_e|^{End} \wedge f_s.N = \{n_s\} \wedge f_e.N = \{n_e\} \wedge \\
&c.F = \{f_s, f_e\} \wedge \\
&n_s \underset{IS}{\rightarrow} n_e \subseteq Edges|^{IS} \wedge \\
&n_s \dashrightarrow n_e \subseteq ControlFlow
\end{aligned}
\tag{4.1}
$$

Die Beispielabbildungsregel 4.1 ist typisch für alle Abbildungsregeln. Sie wird wie folgt gelesen:

> *Sei e ein Element aus JDTElements, dann gilt dass ein c in der Menge Compositions existiert, die Menge aus n_s und n_e eine Teilmenge von Nodes ist und die Menge aus f_s und f_e eine Teilmenge von Fragments der Typen Node ist und diese Elemente die folgenden Eigenschaften haben:*
>
> - *n_s ist ein Startknoten und das einzige Element der Knoten von f_s.*
> - *n_e ist ein Endknoten und das einzige Element der Knoten von f_e.*
> - *Die Fragmente von c sind f_s und f_e.*
> - *Die IS-Kante von n_s nach n_e existiert in der Menge $Edges|^{IS}$.*
> - *Die Kontrollflusskante von n_s nach n_e existiert in der Menge $ControlFlow$.*

Die Abbildungsregel erzeugt für jedes JDT-Element eine Komposition und zwei Knoten sowie zwei Fragmente vom Typ Knoten. Für diese Elemente soll ferner gelten, dass der Knoten n_s dem Fragment f_s sowie der Knoten n_e dem Fragment f_e zugeordnet wird. Außerdem werden beide Fragmente der Fragmentmenge $c.F$ der Komposition c zugewiesen. Schließlich werden IS- und Kontrollflusskanten zwischen den beiden Knoten definiert.

Sofern keine näheren Angaben zu den Attributen der erzeugten Abstraktionselemente gemacht werden, wird davon ausgegangen, dass die Werte der variablen Attribute dem zugrundeliegenden Quelltext entsprechend gesetzt werden. Dies betrifft die Knoten-, Klongruppen-, Kompositions- und Operandenpositionen, den *inl*-Wert und die Existenzwerte der Attribute *asx* und *exs*. Die Werte der fixen Attribute werden standardmäßig wie folgt gesetzt. Der Name einer Komposition oder eines Symbols (*nme*) ergibt sich aus der zugehörigen Zeichenkette im Quelltext. Der Wert des Kompositionsattributs *trn* wird im Abschnitt 4.2.6 definiert. Der *lc*-Wert einer Kante ist $falsch$ und der Typ *typ* eines Symbols ist der Name des Variablentyps.

In den Abbildungsregeln werden häufig Elementmengen genutzt. Die Mengen *Compositions*, *Fragments*, *Nodes*, *Edges*, *Symbols* und *CloneGroups* beziehen sich dabei immer auf die Abstraktion des betrachteten abgeschlossenen Quelltextblocks. Diese Mengen werden oft auf bestimmte Typen eingeschränkt. Im Beispiel betrifft das die Fragmentmenge, die auf Fragmente vom Typ Knoten eingeschränkt wird. Zur besseren

Lesbarkeit werden in den folgenden Abbildungsregeln die Typen von Fragmenten wie folgt abgekürzt: *Node* mit *N*, *CommutativeOperands* mit *CO*, *NotCommutativeOperands* mit *NCO* und *BlockAbstraction* mit *BA*.

Die Abbildungsregeln beschränken sich auf die groben Zusammenhänge bei der Abbildung von JDT- auf Abstraktionselemente. Details, wie bspw. die exakte Abbildung von Labels im Quelltext oder den präzisen Angaben von Reihenfolgen von Operanden, Deklarationsfragmenten oder `Catch`-Blöcken, werden nur skizziert, da sie in ihrer Gesamtheit den Rahmen dieser Arbeit sprengen würden.

4.2 Abbildung des Quelltexts

In dieser Arbeit werden `Methodendeklarationen` und `Initialisierungsblöcke` als abgeschlossene Blöcke betrachtet. Unterblöcke sind alle `Blöcke`, die als Teil von Kontrollflussanweisungen wie `Alternativen` oder Java-Schleifen auftreten.

In den folgenden Abschnitten wird zunächst erläutert, wie aus den JDT-Elementen eines Quelltextblocks die Abstraktionselemente erzeugt werden. Dabei werden die Abbildungen von JDT-Elementen auf Kompositionen der Abstraktion vorgestellt. In den Abbildungen werden häufig CD-Kanten in Fragmenten vom Typ *NotCommutativeOperands* gezeigt, die in diesem Unterkapitel ohne Typnamen in den Abbildungen dargestellt werden. Kontrollflusskanten werden nicht in den Abbildungen dargestellt, da sie nur für die Datenflussanalyse benötigt werden und sonst kein Teil der Abstraktion sind. Im Anschluss werden erst Ausdrücke, dann einfache Anweisungen und zuletzt Kontrollanweisungen betrachtet.

Danach wird erklärt, welche Fragmente aus JDT-Elementen entstehen. Nachdem Anweisungen und Ausdrücke auf Abstraktionselemente abgebildet wurden, werden die Abstraktionselemente genauer erklärt. Dazu wird zunächst auf Operandenfragmente eingegangen, die Teil von vielen Kompositionen sind. Im Abschnitt 4.2.5 wird präsentiert, wie Blockabstraktionen genutzt werden, um mit Unterblöcken eines abgeschlossenen Quelltextblocks umzugehen. Anschließend wird der Wert des Kompositionsattributs *trn* definiert. Zum Abschluss dieses Unterkapitels wird präsentiert, wie, wann und warum Klone in der Abstraktion erstellt werden. Klone sowie das Attribut *trn* ermöglichen bzw. verhindern das Anwenden der Einfügen-Umgestaltung.

4.2.1 Unterstützte JDT-Anweisungen

Der Ansatz unterstützt den überwiegenden Teil der JDT-Elemente. Zu den nicht unterstützten JDT-Elemente gehören das `Type"-Declaration"-Statement` und das `Empty"-State"-ment`, da sie in der Praxis kaum genutzt werden.[1] `Blöcke` werden hingegen unterstützt, allerdings nur, wenn sie Teil einer `Alternative`, einer Java-Schleife, eines `Syn"-chro"-nized"-State"-ments`, eines `Try"-State"-ments` oder einer `Catch"-Clause` sind. Einzeln stehende `Blockanweisungen` werden nicht unterstützt.[2] Allerdings ließen sie sich leicht nachrüsten, da sie sehr ähnlich zu `Synchronized`-Anweisungen sind. Die `Ausdrucksanweisung` selbst wird nicht in der Quelltextabstraktion modelliert.

[1] In den untersuchten Projekten der Evaluation (Kapitel 6) traten diese beiden Anweisungen nie auf.

[2] Blockanweisungen werden sehr selten genutzt. In der Evaluation (Kapitel 6) konnte nur an acht Stellen ein `Block` gefunden werden (alle im Projekt Math).

Stattdessen wird der in ihr enthaltene Ausdruck in der Abstraktion abgebildet. Ebenfalls wird das LabeledStatement nicht explizit in der Abstraktion repräsentiert. Die in ihm eingeschlossene Anweisung ist stattdessen ein Teil der Abstraktion und besitzt das Label. Tabelle 4.1 zeigt für alle JDT-Anweisungen eine Übersicht.

JDT-Anweisung	Unterstützt	JDT-Anweisung	Unterstützt
AssertStatement	✓	ReturnStatement	✓
Block	✓[a]	SuperConstructorInvocation	✓
BreakStatement	✓	SwitchCase	✓
ConstructorInvocation	✓	SwitchStatement	✓
ContinueStatement	✓	SynchronizedStatement	✓
DoStatement	✓	ThrowStatement	✓
EmptyStatement	-	TryStatement	✓
ExpressionStatement	✓[b]	TypeDeclarationStatement	-
ForStatement	✓	VariableDeclarationStatement	✓
IfStatement	✓	WhileStatement	✓
LabeledStatement	✓[c]		

Tabelle 4.1: Unterstützte JDT-Anweisungen

[a] Einzeln stehende Blöcke werden nicht unterstützt.
[b] Wird auf Expressions abgebildet.
[c] Das LabeledStatement LS verweist immer auf ein Statement S. Hier wird jedes LS auf ein S abgebildet und das Label im Folgenden als Eigenschaft von S betrachtet.

Jede Anweisung wird beim Erzeugen einer zugehörigen Abstraktion in eine Komposition und mehrere Fragmente sowie Knoten und Kanten überführt. Welche Abstraktionselemente das genau sind, wird ab Abschnitt 4.2.3 erläutert. Einige der Anweisungen besitzen auch Ausdrücke als Unterelemente. Wie diese in Abstraktion überführt werden, wird im folgenden Unterkapitel beschrieben.

4.2.2 Ausdrücke

Um Ausdrücke auf ihre Abstraktion abzubilden, werden lediglich zwei Abstraktionsregeln benötigt. Das liegt daran, dass die Ausdrücke alle sehr ähnlich sind und sich ihre Unterschiede in zusätzliche Abbildungsfunktionen ausgliedern lassen. Die Abbildung erzeugt für jeden Ausdruck eine Komposition. Der überwiegende Teil aller Ausdrücke besteht aus einem Operator und zusätzlich einem oder mehreren Operanden. Die Operanden sind entweder Terminale wie Literale oder sie sind Unterausdrücke. In beiden Fällen werden sie auf ein Fragment der Komposition abgebildet. Wie viele und welche Operanden dabei entstehen, wird später in Absatz 4.2.5 erklärt. Zunächst wird präsentiert, welche Kompositionen, Fragmente und Knoten sowie Kanten von einem Ausdruck erzeugt werden. Die Abstraktion vereinheitlicht die Operationen, die im Quelltext nach Pre-, Post- oder Infixnotation unterschieden werden. Dadurch können all diese Operationen mithilfe von nur einer Abbildungsregel auf die Abstraktion abgebildet werden. Eine Ausnahme davon bilden Konditionalausdrücke, die aus einem Bedingungs-, einem Dann- und einem Sonstausdruck bestehen. Die folgende Abbildungsregel gibt an, wie Konditionalausdrücke

in eine Abstraktion überführt werden.

$$
\begin{aligned}
&\forall e \in ConditionExpressions \\
&\exists \\
&c \in Compositions, \\
&\{n_{OS}, n_C, n_T, n_E, n_{OE}, n_s, n_e\} \subseteq Nodes, \\
&\{f_s, f_e\} \subseteq Fragments|^{N}, \\
&cond \in Fragments|^{NCO} \\
&: \\
&n_s|^{Start} \wedge n_e|^{End} \wedge f_s.N = \{n_s\} \wedge f_e.N = \{n_e\} \wedge \\
&c.F = \{f_s, cond, f_e\} \wedge \\
&n_{OS}^{OpStart} \wedge n_C|^{.op=1} \wedge n_T|^{.op=2} \wedge n_E|^{.op=3} \wedge n_{OE}^{OpEnd} \wedge \\
&cond.N = \{n_{OS}, n_C, n_T, n_E, n_{OE}\} \wedge \\
&n_s \underset{IS}{\rightarrow} cond \underset{IS}{\rightarrow} n_e \subseteq Edges|^{IS} \wedge \\
&n_s \dashrightarrow cond \dashrightarrow n_e \subseteq ControlFlow \wedge \\
&n_{OS} \dashrightarrow n_C \dashrightarrow n_T \dashrightarrow n_{OE} \subseteq ControlFlow \wedge \\
&n_C \dashrightarrow n_E \dashrightarrow n_{OE} \subseteq ControlFlow
\end{aligned}
\tag{4.2}
$$

Zunächst wird in der Abbildungsregel 4.2 spezifiziert, dass für jede `Con"-di"-tion"-Ex"-pres"-sion` eine Komposition c erzeugt wird. Mit ihr werden außerdem die Knoten n_s und n_e der Fragmente f_s und f_e und die Knoten $n_{OS}^{OpStart}$, n_C, n_T, n_E, n_{OE}^{OpEnd} des *cond*-Fragments erstellt. Die graphische Darstellung in Abbildung 4.2f zeigt die Komposition des Konditionalausdrucks mit seinen Fragmenten, Knoten und Kanten. Das *cond*-Fragment dient dazu, die drei Teile des Konditionalausdrucks (Bedingungs-, Dann- und Sonstausdruck) zu beherbergen. Es hat den Typ *NotCommutativeOperands* (*NCO*), da seine Operanden nicht vertauscht werden dürfen, ohne die Bedingungsoperation zu ändern. Der Bedingungsteil der Abbildungsregel spezifiziert die Zuordnung der Knoten zu den Fragmenten, sowie der Fragmente zur Komposition c. Desweiteren werden die IS-Kanten und der Kontrollfluss zwischen den Fragmenten definiert. Da der Kontrollfluss innerhalb des Operandenfragments verzweigt, existieren hier auch zwei Abhängigkeitspfade. Am Bedingungsknoten n_C verzweigt der Kontrollfluss und bewirkt daher eine Kante von n_C zum Dannknoten n_T und eine weitere zum Sonstknoten n_E.

In JDT können Ausdrücke als Anweisungen genutzt werden. Dafür stellt JDT die `Ausdrucksanweisungen` zur Verfügung, die den relevanten Ausdruck besitzen. In der Quelltextabstraktion werden `Ausdrucksanweisungen` nicht separat, sondern genau wie `Ausdrücke` behandelt. Da nicht alle JDT-Ausdrücke in der Abstraktion auf Kompositionen abgebildet werden, wird das Prädikat *Represented* verwendet, das für jeden Ausdruck angibt, ob er abgebildet wird oder nicht. Wird ein Quelltextelement wie bspw. ein Literal nicht als eigene Kompositionen abgebildet, wertet *Represented* zu falsch aus. *Represented* wird durch Tabelle 4.3 in Unterkapitel 4.2.5 zu Erklärung der Fragmente und ihres Aufbaus definiert.

Die nächste Abbildungsregel 4.3 definiert die Abstraktion für `Ausdrücke`. Damit sind alle abgeleiteten Klassen der Klasse `Expression` gemeint, auch solche, die als

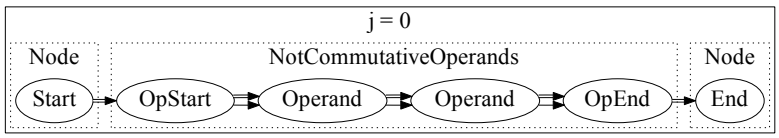
Unterelement einer `Ausdrucksanweisung` im AST vorkommen. Annotationen müssen in dieser Arbeit nicht beachtet werden, da sie den Kontrollfluss nicht beeinflussen und auch keine Programmabhängigkeiten erzeugen und werden daher ignoriert (siehe Prädikat *Represented* in Abschnitt 4.2.5). Wie bei der vorherigen Abbildungsregel werden hier zunächst auch Start- und Endknoten sowie -fragmente definiert und im Prädikatteil einander zugeordnet. Außerdem wird das Fragment *expr* definiert, dessen Typ entweder *CommutativeOperands* oder *NotCommutativeOperands* ist. Hierfür wird die Funktion *IsCommutative* genutzt, die einen Ausdruck auf einen Wahrheitswert abbildet, je nachdem, ob der zugehörige Ausdruck kommutativ ist oder nicht. Ihre Abbildungsvorschrift kann in Tabelle 4.2 eingesehen werden. `VariableDeclarationExpression`s sind immer kommutativ und Infixausdrücke sind es nur dann, wenn ihre Operation kommutativ ist.

$$
\begin{aligned}
&\forall e \in \{ee \in Expressions | Represented(ee)\} \\
&\exists \\
&c \in Compositions, \{n_s, n_e\} \subseteq Nodes, \{f_s, f_e\} \subseteq Fragments|^N, \\
&expr \in Fragments \\
&: \\
&n_s|^{Start} \wedge n_e|^{End} \wedge f_s.N = \{n_s\} \wedge f_e.N = \{n_e\} \wedge \\
&c.F = \{f_s, expr, f_e\} \wedge \\
&IsCommutative(e) \Rightarrow expr.typ = CO \wedge \\
&\neg IsCommutative(e) \Rightarrow expr.typ = NCO \wedge \\
&n_s \underset{IS}{\rightarrow} expr \underset{IS}{\rightarrow} n_e \subseteq Edges|^{IS} \wedge \\
&n_s \dashrightarrow expr \dashrightarrow n_e \subseteq ControlFlow
\end{aligned}
\tag{4.3}
$$

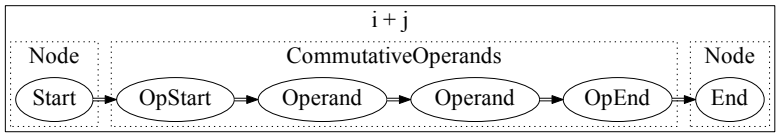
JDT-Infixoperation	Bedingung	*IsCommutative*
VariableDeclarationExpression	-	✓
InfixExpression	Operation = Addition (+)	✓
InfixExpression	Operation = Multiplikation (∗)	✓
InfixExpression	Operation = Gleich (==)	✓
InfixExpression	Operation = Ungleich (!=)	✓
InfixExpression	Operation = XOR (^)	✓
InfixExpression	Operation = Bitweises Und (&)	✓
InfixExpression	Operation = Bitweises Oder (|)	✓
InfixExpression	sonst	✗
sonst	-	✗

Tabelle 4.2: Kommutativitätsfunktion *IsCommutative* von JDT-Ausdrücken

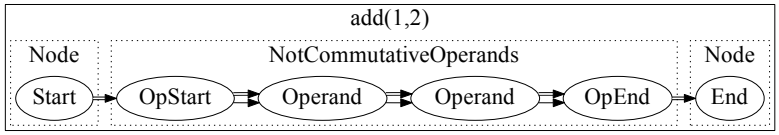
Die Kommutativität der Ausdrücke ist wichtig, um während der Umgestaltung entscheiden zu können, ob zwei Operanden vertauscht werden dürfen oder nicht. Die Kommutativität richtet sich dabei nach der gängigen Definition. Die Operationen *logisches Und* und *logisches Oder* sind nicht kommutativ, da aus Gründen der Optimierung ihre Ausführung



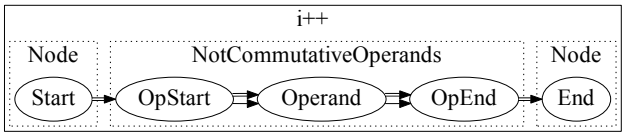
(a) Abstraktion einer Zuweisungsanweisung



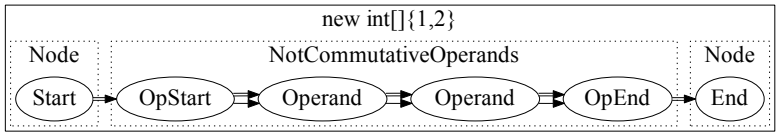
(b) Abstraktion eines Additionsausdrucks



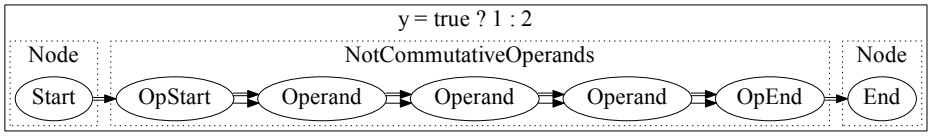
(c) Abstraktion eines Methodenaufrufs



(d) Abstraktion eines Postinkrementausdrucks



(e) Abstraktion einer Feldinitialisierung



(f) Abstraktion eines Konditionalausdrucks

Abbildung 4.2: Abstraktionen von Java-Ausdrücken

frühzeitig abgebrochen wird, falls bereits ein Ergebnis feststeht. Ein Vertauschen der Operanden dieser Operationen kann daher die Programmsemantik verändern.

Die in Abbildung 4.2 gezeigten Abstraktionen zeigen exemplarisch, welche Abstraktionen für spezielle JDT-Programmelemente entstehen. Zu sehen ist, dass jede Abstraktion immer mit *Start-* bzw. *End*-Knoten beginnt bzw. endet und das Operandenfragment je nach JDT-Klasse des `Ausdrucks` kommutativ ist oder nicht. Interessant sind hierbei die Operandenfragmente, die entweder vom Typ *CommutativeOperands* oder *NotCommutativeOperands* sind. Die Operanden der kommutativen Operandenfragmente haben zueinander keine Kanten, da die Operanden umsortiert werden dürfen. Im Gegensatz dazu besitzen die nicht kommutativen Operandenfragment Kontrollflussabhängigkeitskanten (CD-Kanten) zwischen den Knoten, die ein Umsortieren der Operanden verhindern. Bspw. können die Argumente eines Methodenaufrufs nicht miteinander vertauscht (siehe Abb. 4.2c). Ebenfalls können die Wenn-, Dann- und Sonstausdrücke eines Bedingungsausdrucks nicht vertauscht werden (siehe Abb. 4.2f). Hierbei darf die CD-Kante nicht mit dem tatsächlichen Kontrollfluss verwechselt werden, der sich hier von den CD-Kanten unterscheidet. Von der JDT-Klasse ist auch die Anzahl der Operandknoten der Operandenfragmente abhängig. Sie richtet sich nach der Zahl der Unterelemente und wird in Abschnitt 4.2.5 genauer erläutert. Beispielsweise hat ein Infixausdruck (Abb. 4.2b) immer mindestens zwei Operandknoten. Die IS-Kanten in den Abstraktionen sind durch die Doppellinien dargestellt und zeigen, wie die Knoten und damit die Fragmente aneinander gekoppelt werden.

4.2.3 Einfache Anweisungen

In diesem Unterkapitel werden Abbildungsvorschriften der JDT-Anweisungen vorgestellt, die Kontrollflüsse ohne Verzweigung oder Unterblöcke besitzen. Diese Anweisungen bestehen lediglich aus ein oder zwei Unterausdrücken, die möglicherweise kommutativ sind.

Zusicherung

Die `Zusicherung` besteht aus einem oder zwei Ausdrücken. Der erste Ausdruck repräsentiert die Zusicherungsbedingung und der zweite, optionale Ausdruck die Fehlernachricht. Sofern die Fehlernachricht im JDT-Element existiert, wird das Könnteelement $msg^?$ erzeugt. Beide Ausdrücke sind jeweils in nicht-kommutative Fragmente eingebettet, da sie im Quelltext nicht vertauscht werden können.

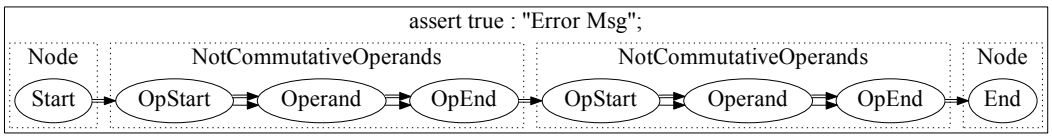


Abbildung 4.3: Abstraktion einer Zusicherung

$$
\begin{aligned}
&\forall as \in AssertStatements \\
&\exists \\
&c \in Compositions, \{n_s, n_e\} \subseteq Nodes, \{f_s, f_e\} \subseteq Fragments|^{N}, \\
&\{expr, msg^{?}\} \subseteq Fragments|^{NCO} \\
&: \\
&n_s|^{Start} \wedge n_e|^{End} \wedge f_s.N = \{n_s\} \wedge f_e.N = \{n_e\} \wedge \\
&c.F = \{f_s, expr, msg^{?}, f_e\} \wedge \\
&n_s \underset{IS}{\rightarrow} expr \underset{IS}{\rightarrow} msg^{?} \underset{IS}{\rightarrow} n_e \subseteq Edges|^{IS} \wedge \\
&n_s \dashrightarrow expr \dashrightarrow msg^{?} \dashrightarrow n_e \subseteq ControlFlow
\end{aligned}
\tag{4.4}
$$

Sprunganweisungen

Die Abbildungsregel 4.5 bildet diese Sprunganweisungen auf Elemente der Abstraktion ab. Sofern die Sprunganweisung über einen Rückgabeausdruck oder die zu werfende Ausnahme verfügt, besitzt auch die Abstraktion ein zugehöriges Fragment. In der Abbildungsregel wird dieser Unterausdruck mit dem Könnteelement $ret^{?}$ bezeichnet. Sprungeffekte werden später ausgewertet, um die zugehörigen Kontrollflusskanten zu erstellen.

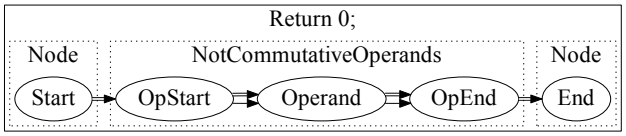


Abbildung 4.4: Abstraktion einer Rückgabeanweisung

$$
\begin{aligned}
&\forall js \in Sprunganweisungen \\
&\exists \\
&c \in Compositions, \{n_s, n_e\} \subseteq Nodes, \{f_s, f_e\} \subseteq Fragments|^{N}, \\
&ret^{?} \in Fragments|^{NCO} \\
&: \\
&n_s|^{Start} \wedge n_e|^{End} \wedge f_s.N = \{n_s\} \wedge f_e.N = \{n_e\} \wedge \\
&c.F = \{f_s, ret^{?}, f_e\} \wedge \\
&n_s \underset{IS}{\rightarrow} ret^{?} \underset{IS}{\rightarrow} n_e \subseteq Edges|^{IS} \wedge \\
&n_s \dashrightarrow ret^{?} \dashrightarrow n_e \subseteq ControlFlow
\end{aligned}
\tag{4.5}
$$

Deklaration (Anweisung)

`Deklarationsanweisungen` werden wie `Deklarationsausdrücke` behandelt. Bei beiden gilt, dass `Deklarationsfragmente` einzelne Operanden in einer kommutativen

Gruppe sind. Dadurch ist später das Umsortieren der Deklarationen innerhalb der Anweisung möglich.

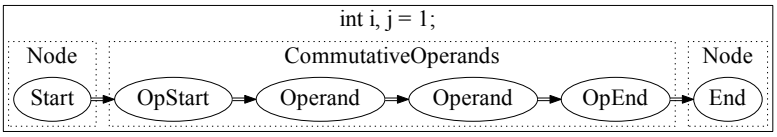


Abbildung 4.5: Abstraktion einer Deklaration

$$
\begin{aligned}
&\forall ds \in DeclarationStatements \\
&\exists \\
&c \in Compositions, \{n_s, n_e\} \subseteq Nodes, \{f_s, f_e\} \subseteq Fragments|^{N}, \\
&dcls \in Fragments|^{CO} \\
&: \\
&n_s|^{Start} \wedge n_e|^{End} \wedge f_s.N = \{n_s\} \wedge f_e.N = \{n_e\} \wedge \\
&c.F = \{f_s, dcls, f_e\} \wedge \\
&n_s \underset{IS}{\rightarrow} dcls \underset{IS}{\rightarrow} n_e \subseteq Edges|^{IS} \wedge \\
&n_s \dashrightarrow dcls \dashrightarrow n_e \subseteq ControlFlow
\end{aligned}
\tag{4.6}
$$

Konstruktoraufrufe

`Konstruktoraufrufe` und `Superkonstruktoraufrufe` werden wie Methodenaufrufe behandelt. Für alle Argumente wird eine nicht-kommutative Gruppe erstellt, die durch IS- und Kontrollflusskanten mit dem Start- und Endknoten verbunden wird. Besitzt ein Konstruktor Argumente, wird das Könntefragment $args^{?}$ erzeugt. Es besitzt für jedes Argument einen Operandknoten.

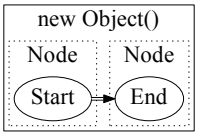


Abbildung 4.6: Abstraktion einer Deklaration

$$\begin{aligned}
&\forall ds \in ConstructorInvocation \cup SuperConstructorInvocation \\
&\qquad \exists \\
&c \in Compositions, \{n_s, n_e\} \subseteq Nodes, \{f_s, f_e\} \subseteq Fragments|^{N}, \\
&\qquad args^{?} \in Fragments|^{NCO} \\
&\qquad : \\
&n_s|^{Start} \wedge n_e|^{End} \wedge f_s.N = \{n_s\} \wedge f_e.N = \{n_e\} \wedge \\
&\qquad c.F = \{f_s, args^{?}, f_e\} \wedge \\
&\qquad n_s \underset{IS}{\rightarrow} args^{?} \underset{IS}{\rightarrow} n_e \subseteq Edges|^{IS} \wedge \\
&\qquad n_s \dashrightarrow args^{?} \dashrightarrow n_e \subseteq ControlFlow
\end{aligned} \tag{4.7}$$

4.2.4 Kontrollanweisungen

Im Folgenden werden die Abbildungsvorschriften der JDT-Kontrollanweisungen auf die Abstraktion beschrieben. Im Gegensatz zu einfachen Anweisungen besitzen die Kontrollanweisungen Unterblöcke und meist Kontrollflussverzweigungen. Die Unterblöcke werden auf Blockabstraktionen abgebildet, die die Lese-/Schreibeffekte der Anweisungen und Ausdrücke im Unterblock abstrahieren.

Alternative

`Alternativen` bestehen aus einem Ausdruck, der die Bedingung repräsentiert, aus einem Dannblock, der ausgeführt wird, wenn die Bedingung zutrifft und möglicherweise aus einem Sonstblock der ansonsten ausgeführt wird. Der Sonstblock wird als das Könntefragment $else^{?}$ in der Abbildungsregel beschrieben und existiert nur dann, wenn eine Sonstanweisung bzw. ein Sonstblock im Quelltext existiert. Die Abstraktion erzeugt auch dann ein Block für den Dann- bzw. Sonstteil, wenn der Dann- bzw. Sonstteil statt aus einem `Block` aus einer einzelnen Anweisung besteht. Der Kontrollfluss verzweigt sich hinter der Bedingung und vereinigt sich wieder zum Endknoten der Komposition. Der durch die IS-Kanten entstehende Pfad der einzelnen Abstraktionselemente besitzt dagegen eine lineare Reihenfolge ohne Verzweigungen.

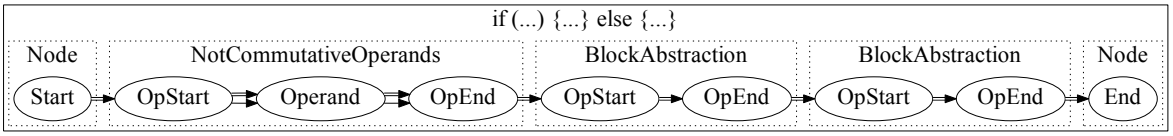


Abbildung 4.7: Abstraktion einer Alternative

$$
\begin{aligned}
&\forall if \in IfStatements \\
&\exists \\
&c \in Compositions, \{n_s, n_e\} \subseteq Nodes, \{f_s, f_e\} \subseteq Fragments|^{N}, \\
&cnd \in Fragments|^{NCO}, \{b_{if}, b^{?}_{else}\} \subseteq Fragments|^{BA} \\
&: \\
&n_s|^{Start} \wedge n_e|^{End} \wedge f_s.N = \{n_s\} \wedge f_e.N = \{n_e\} \wedge \\
&c.F = \{f_s, cnd, b_{if}, b^{?}_{else}, f_e\} \wedge \\
&n_s \underset{IS}{\rightarrow} cnd \underset{IS}{\rightarrow} b_{if} \underset{IS}{\rightarrow} b^{?}_{else} \underset{IS}{\rightarrow} n_e \subseteq Edges|^{IS} \wedge \\
&n_s \dashrightarrow cnd \dashrightarrow b_{if} \dashrightarrow n_e \cup cnd \dashrightarrow b^{?}_{else} \dashrightarrow n_e \subseteq ControlFlow
\end{aligned}
\tag{4.8}
$$

Schleife: `For`

Etwas komplizierter sind `For`-Schleifen. Sie können aus den vier Elementen Initialisierung, Bedingung, Aktualisierung und Schleifenrumpf bestehen. Die Elemente Initialisierung, Aktualisierung und Schleifenrumpf werden als Blöcke aufgefasst und daher von Blockabstraktionen abstrahiert. Allerdings lässt die Sprachspezifikation von Java ein Weglassen alle Unterelemente einer Schleife zu (`§14.14.1`). Daher werden die vier genannten Elemente als Könnteelemente dargestellt und nur dann erzeugt, wenn sie im Quelltext existieren. Die Schleifenbedingung wird allerdings von einem Operandenfragment abgebildet. Die Elemente werden durch IS-Kanten entsprechend eines einmaligen Schleifendurchlaufs angeordnet. Die Kontrollflüsse zwischen den Elementen werden je nach ihrer Existenz festgelegt. Mithilfe der Könnteelemente können die Kanten jedoch sehr kompakt definiert werden. Im Beispiel in Abbildung 4.8 wird eine `For`-Schleife dargestellt, die über keinen Schleifenrumpf verfügt. Daher besitzt ihre Abstraktion auch insgesamt nur zwei Blockabstraktionsfragmente.

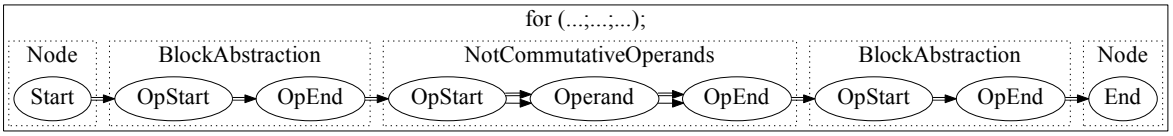


Abbildung 4.8: Abstraktion einer `For`-Schleife

$$
\begin{aligned}
&\forall fs \in ForStatements \\
&\exists \\
&c \in Compositions, \{n_s, n_e\} \subseteq Nodes, \{f_s, f_e\} \subseteq Fragments|^{N}, \\
&cnd^{?} \in Fragments|^{NCO}, \{decl^{?}, b^{?}, upds^{?}\} \subseteq Fragments|^{BA} \\
&: \\
&n_s|^{Start} \wedge n_e|^{End} \wedge f_s.N = \{n_s\} \wedge f_e.N = \{n_e\} \wedge \\
&c.F = \{f_s, decl^{?}, cnd^{?}, upds^{?}, b^{?}, f_e\} \wedge \\
&n_s \underset{IS}{\rightarrow} decl^{?} \underset{IS}{\rightarrow} cnd^{?} \underset{IS}{\rightarrow} upds^{?} \underset{IS}{\rightarrow} b^{?} \underset{IS}{\rightarrow} n_e \subseteq Edges|^{IS} \wedge \\
&n_s \dashrightarrow decl^{?} \dashrightarrow cnd^{?} \dashrightarrow b^{?} \dashrightarrow upds^{?} \cup cnd^{?} \dashrightarrow n_e \subseteq ControlFlow \wedge \\
&[upds^{?}||b^{?}] \dashrightarrow |^{.lc}[cnd^{?}||b^{?}] \subseteq ControlFlow
\end{aligned}
\tag{4.9}
$$

Schleife: `For Each`

Die `For-Each`-Schleife wird ähnlich abstrahiert wie die gewöhnliche `For`-Schleife. Genau wie bei der `For`-Schleife ist der Schleifenrumpf laut Sprachspezifikation optional (§14.14.2). Folglich wird das zugehörige Könntefragment $b^{?}$ der Abstraktion nur dann erzeugt, wenn er existiert. Allerdings verfügt `For-Each`-Schleife über zusätzliche, implizite Elemente, die nicht im Quelltext stehen. Dies sind die Methodenaufrufe der `Iterable`-Schnittstelle `iterator()`, `hasNext()` und `next()`. In Abbildung 4.9 sind sie in Klammern der Knotenfragmente angegeben. Da diese Aufrufe vom Entwickler implementierte Methoden aufrufen können, die möglicherweise Seiteneffekte haben, können durch sie Abhängigkeiten beim Schleifendurchlauf entstehen. Folglich werden sie explizit in der Abstraktion mithilfe von Hilfsknoten modelliert.

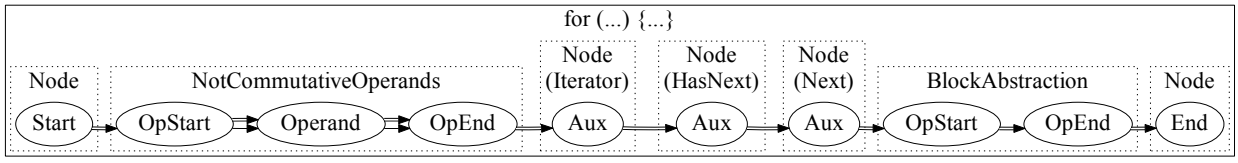


Abbildung 4.9: Abstraktion einer `For-Each`-Schleife

$$
\begin{aligned}
&\forall fes \in ForEachStatements \\
&\exists \\
&c \in Compositions, \{n_s, n_e\} \subseteq Nodes, \{f_s, it, hnx, nxt, f_e\} \subseteq Fragments|^{N}, \\
&\{col\} \subseteq Fragments|^{NCO}, b^{?} \in Fragments|^{BA} \\
&: \\
&n_s|^{Start} \wedge n_e|^{End} \wedge f_s.N = \{n_s\} \wedge f_e.N = \{n_e\} \wedge \\
&c.F = \{f_s, it, hnx, nxt, b^{?}, f_e\} \wedge \\
&n_s \underset{IS}{\rightarrow} col \underset{IS}{\rightarrow} it \underset{IS}{\rightarrow} hnx \underset{IS}{\rightarrow} nxt \underset{IS}{\rightarrow} b^{?} \underset{IS}{\rightarrow} n_e \subseteq Edges|^{IS} \wedge \\
&n_s \dashrightarrow col \dashrightarrow it \dashrightarrow hnx \dashrightarrow nxt \dashrightarrow b^{?} \cup hnx \dashrightarrow n_e \subseteq ControlFlow \wedge \\
&[b^{?}||nxt] \dashrightarrow |^{.lc}hnx \subseteq ControlFlow
\end{aligned}
\tag{4.10}
$$

Schleife: `While`

Aus `While`-Schleifen wird eine Komposition erzeugt, die aus vier Fragmenten besteht. Neben zwei Knotenfragmenten für Start- und Endknoten sind das ein Fragment für die Schleifenbedingung und eins für die Blockabstraktion. Die Schleifenbedingung besitzt einen Operanden, der im einfachsten Fall bspw. das Literal `true` sein kann. Die Blockabstraktion $b^{?}$ wird dann erzeugt, wenn die Schleife einen Schleifenrumpf besitzt (§14.12). $b^{?}$ hat für alle im Block gelesenen Variablen einen Operanden. Die Abstraktion in Abbildung 4.10 zeigt eine Blockabstraktion ohne Operanden.

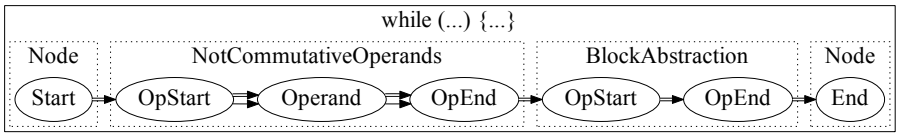


Abbildung 4.10: Abstraktion einer `While`-Schleife

$$
\begin{aligned}
&\forall ws \in WhileStatements \\
&\exists \\
&c \in Compositions, \{n_s, n_e\} \subseteq Nodes, \{f_s, f_e\} \subseteq Fragments|^{N}, \\
&cnd \in Fragments|^{NCO}, b^{?} \in Fragments|^{BA} \\
&: \\
&n_s|^{Start} \wedge n_e|^{End} \wedge f_s.N = \{n_s\} \wedge f_e.N = \{n_e\} \wedge \\
&c.F = \{f_s, cnd, b^{?}, f_e\} \wedge \\
&n_s \underset{IS}{\rightarrow} cnd \underset{IS}{\rightarrow} b^{?} \underset{IS}{\rightarrow} n_e \subseteq Edges|^{IS} \wedge \\
&n_s \dashrightarrow cnd \dashrightarrow b^{?} \cup cnd \dashrightarrow n_e \subseteq ControlFlow \wedge \\
&[b^{?}||cnd] \dashrightarrow |^{.lc}cnd \subseteq ControlFlow
\end{aligned}
\tag{4.11}
$$

Schleife: `Do While`

Schleifen vom Typ `Do-While` sind in der Abstraktion fast identisch zu `While`-Schleifen aufgebaut. Beide unterscheiden sich hauptsächlich in der Reihenfolge des Operandenfragments der Bedingung und der Blockabstraktion des Schleifenrumpfs. Wie bei der `While`-Schleife wird die Blockabstraktion b genau dann erstellt, wenn die Schleife einen Schleifenrumpf besitzt (§14.13). Außerdem verlaufen die Kontrollflüsse in beiden Schleifenarten geringfügig unterschiedlich: Während die Schleifenkante des Kontrollflusses in der `While`-Schleife nach der Blockabstraktion startet, startet sie bei `Do-While`-Schleifen nach dem Operandenfragment.

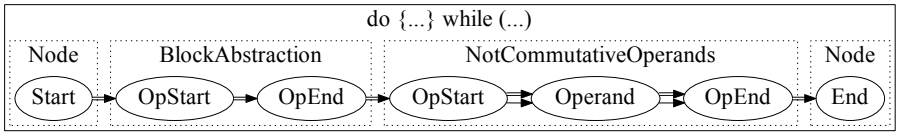


Abbildung 4.11: Abstraktion einer `Do-While`-Schleife

$$
\begin{aligned}
&\forall dws \in DoWhileStatements \\
&\exists \\
&c \in Compositions, \{n_s, n_e\} \subseteq Nodes, \{f_s, f_e\} \subseteq Fragments|^{N}, \\
&b \in Fragments|^{BA}, cnd \in Fragments|^{NCO} \\
&: \\
&n_s|^{Start} \wedge n_e|^{End} \wedge f_s.N = \{n_s\} \wedge f_e.N = \{n_e\} \wedge \\
&c.F = \{f_s, b, cnd, f_e\} \wedge \\
&n_s \underset{IS}{\rightarrow} b \underset{IS}{\rightarrow} cnd \underset{IS}{\rightarrow} n_e \subseteq Edges|^{IS} \wedge \\
&n_s \dashrightarrow b \dashrightarrow cnd \dashrightarrow n_e \cup cnd \dashrightarrow |^{lc} b \subseteq ControlFlow
\end{aligned}
\tag{4.12}
$$

`Try`-Anweisung

`Try`-Anweisungen besitzen in der Abstraktion sowohl für den `Try`-Block als auch für jeden `Catch`-Block eine Blockabstraktion. Der Ressourcenblock (eingeführt in Java 7) wird in der hier beschriebenen Variante nicht modelliert. Er könnte jedoch mithilfe impliziter Knoten ähnlich wie bei der `For-Each`-Anweisung später nachgerüstet werden. Wichtig bei ihrer Modellierung ist, das die Reihenfolgen der impliziten Öffnen- und Schließenanweisungen kongruent bleiben können.

Eine `Try`-Anweisung kann beliebig viele `Catch`-Blöcke besitzen (§14.20). In der Abbildungsregel 4.13 wird die Funktion *CatchBlocks* genutzt, um eine Menge von Blockabstraktionen zu erzeugen, die die `Catch`-Blöcke der `Try`-Anweisung ts repräsentieren. Diese Menge der Blockabstraktionen für `Catch`-Blöcke ist CT. Die Kontrollflüsse werden vom Startknoten über den `Try`- und den möglicherweise existierenden `Finally`-Block, hin zum Endknoten, modelliert. Ebenfalls existieren Kontrollflüsse vom `Try`-Block zu jedem `Catch`-Block und von dort weiter zum Endknoten bzw. zum `Finally`-Block $fnl^{?}$, sofern dieser

existiert. Als sinnvolle Überabschätzung wird davon ausgegangen, dass jeder der `Catch`-Blöcke des Quelltexts auch durch eine im `Try`-Block ausgelöste Ausnahme erreichbar ist, da etablierte Entwicklungsumgebungen Warnungen anzeigen, falls `Catch`-Blöcke unerreichbar sind.

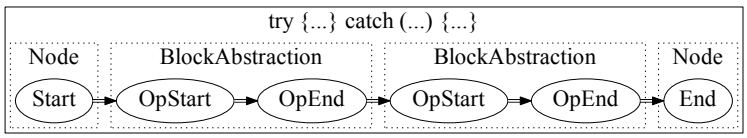


Abbildung 4.12: Abstraktion einer `Try`-Anweisung

$$
\begin{aligned}
&\forall ts \in TryStatements \\
&\exists \\
&c \in Compositions, \{n_s, n_e\} \subseteq Nodes, \{f_s, f_e\} \subseteq Fragments|^{N}, \\
&\{try, fnl^{?}\} \subseteq Fragments|^{BA}, \\
&CT = CatchBlocks(ts) \wedge CT \subseteq Fragments|^{BA} \\
&: \\
&n_s|^{Start} \wedge n_e|^{End} \wedge f_s.N = \{n_s\} \wedge f_e.N = \{n_e\} \wedge \\
&c.F = \{f_s, try, fnl^{?}, f_e\} \cup CT \wedge \\
&n_s \underset{IS}{\rightarrow} try \overset{CT,Pos}{\underset{IS}{\rightsquigarrow}} fnl^{?} \underset{IS}{\rightarrow} n_e \subseteq Edges|^{IS} \wedge \\
&n_s \dashrightarrow try \dashrightarrow fnl^{?} \dashrightarrow n_e \subseteq ControlFlow \wedge \\
&try \dashrightarrow\!\!\!\!\!\rightarrow CT \dashrightarrow\!\!\!\!\!\rightarrow [fnl^{?}||n_e] \subseteq ControlFlow
\end{aligned}
\tag{4.13}
$$

`Switch`-Anweisung

Genau wie `Try`-Anweisungen können `Switch`-Anweisungen beliebig viele Blöcke besitzen (§14.11). Die Funktion $CaseBlocks$ gibt für eine `Switch`-Anweisung sws alle Blockabstraktionen an, die `Case`-Blöcke (inklusive des `Default`-Blocks) von sws repräsentieren. Die injektive Funktion $Pos : Fragment \rightarrow \mathbb{N}$ bildet jede Blockabstraktion einer `Case`-Anweisung c auf seine Position entsprechend der Quelltextposition von c ab. Ob eine `Switch`-Anweisung sws über einen `Default`-Block verfügt, kann über das Prädikat $HasDefault$ ermittelt werden. Wird eine `Case`-Anweisung cb durch eine `Break`-Anweisung beendet, wertet das Prädikat $HasBreak$ zu *wahr* und andernfalls zu *falsch* aus.

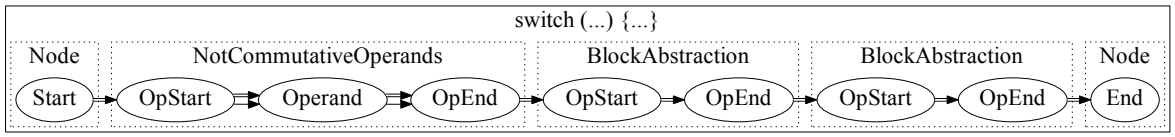


Abbildung 4.13: Abstraktion einer `Switch`-Anweisung

$$
\begin{aligned}
&\forall sws \in SwitchStatements \\
&\exists \\
&c \in Compositions, \{n_s, n_e\} \subseteq Nodes, \{f_s, f_e\} \subseteq Fragments|^{N}, \\
&swc \in Fragments|^{NCO}, \\
&CB = CaseBlocks(sws) \wedge CB \subseteq Fragments|^{BA} \\
&: \\
&n_s|^{Start} \wedge n_e|^{End} \wedge f_s.N = \{n_s\} \wedge f_e.N = \{n_e\} \wedge \\
&c.F = \{f_s, swc, f_e\} \cup CB \wedge \\
&n_s \underset{IS}{\rightarrow} swc \overset{CB,Pos}{\underset{IS}{\rightsquigarrow}} n_e \subseteq Edges|^{IS} \wedge \\
&n_s \dashrightarrow swc \dashrightarrow\!\!\!\!\!\rightarrow CB \subseteq ControlFlow \wedge \\
&\neg HasDefault(sws) \Rightarrow swc \dashrightarrow n_e \subseteq ControlFlow \wedge \\
&\forall (cb_a, cb_b) \in SuccPairs_{Pos}(CB) : \neg HasBreak(cb_a) \Rightarrow cb_a \dashrightarrow cb_b \in ControlFlow
\end{aligned}
\tag{4.14}
$$

`Sychronized`-Anweisung

Der Kontrollfluss von `Sychronized`-Anweisungen verzweigt sich nicht. Da sie jedoch eine Blockabstraktion besitzen, werden sie zu den Kontrollanweisungen gezählt. `Sychronized`-Anweisungen bestehen lediglich aus einem Operandenfragment, dessen Operand das Lesen der zu synchronisierenden Mutexvariablen repräsentiert. Außerdem besitzen `Sychronized`-Anweisungen eine Blockabstraktion.

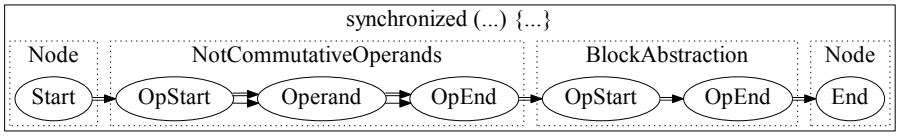


Abbildung 4.14: Abstraktion einer `Sychronized`-Anweisung

$$
\begin{aligned}
&\forall sys \in SynchronizedStatements \\
&\exists \\
&c \in Compositions, \{n_s, n_e\} \subseteq Nodes, \{f_s, f_e\} \subseteq Fragments|^{N}, \\
&mtx, \in Fragments|^{NCO}, b \in Fragments|^{BA} \\
&: \\
&n_s|^{Start} \wedge n_e|^{End} \wedge f_s.N = \{n_s\} \wedge f_e.N = \{n_e\} \wedge \\
&c.F = \{f_s, mtx, b, f_e\} \wedge \\
&n_s \underset{IS}{\rightarrow} mtx \underset{IS}{\rightarrow} b \underset{IS}{\rightarrow} n_e \subseteq Edges|^{IS} \wedge \\
&n_s \rightarrow mtx \dashrightarrow b \dashrightarrow n_e \subseteq ControlFlow
\end{aligned}
\tag{4.15}
$$

4.2.5 Fragmente

In diesem Unterkapitel werden die einzelnen Fragmenttypen genauer erläutert. Zunächst wird auf Fragmente vom Typ *Node* eingegangen. Anschließend werden die Operandenfragmente betrachtet. Dies sind Fragmente der Typen *CommutativeOperands*, *NotCommutativeOperands* und *BlockAbstraction*. Zuletzt wird noch genauer auf Blockabstraktionen und ihre Operanden eingegangen.

Knotenfragmente

Knotenfragmente sind Fragmente vom Typ Knoten, die lediglich einen einzigen Knoten enthalten. Sie werden für die Start- und Endknoten einer Komposition verwendet. Außerdem werden sie genutzt, falls zusätzliche Knoten gebraucht werden, um Programmeffekte zu modellieren. Programmeffekte entstehen, wenn bspw. Variablen gelesen oder geschrieben werden. Wenn Knotenfragmente genutzt werden, müssen für sie keine speziellen Kanten oder Abhängigkeiten erzeugt werden.

Operandenfragmente

Der Quelltext besteht u. a. aus Quelltextelementen, die Unterelemente besitzen. Hierzu gehören bspw. Alternativen, die Bedingungen und Dann- bzw. Sonstanweisungen haben, aber auch Ausdrücke wie Additionsausdrücke, die Unterausdrücke besitzen können. Ein Unterelement kann ein Terminal sein wie ein Literal, das nicht durch eine eigene Komposition in der Quelltextabstraktion repräsentiert wird, oder es kann ein Unterausdruck wie eine Addition sein, die ihrerseits durch eine eigene Komposition in der Abstraktion repräsentiert wird. Egal ob ein Unterelement eine eigene Komposition in der Abstraktion besitzt oder nicht, muss die Abstraktion mindestens eine Information über seine Existenz besitzen. Denn wenn bspw. die Unterelemente bzw. Operanden einer Addition vertauscht werden sollen, ist wichtig, dass alle Operanden in der Abstraktion modelliert werden, d. h. sowohl Literale ohne eigene Komposition als auch beliebige andere Unterausdrücke mit eigener Komposition. Zur Modellierung der Operanden werden Operandknoten innerhalb von Operandenfragmenten genutzt.

```
abstract int foo();
void nesting(int i) {
  i = foo() + 1;
}
```

Quelltext 4.1: Quelltext mit verschachteltem Additionsausdruck

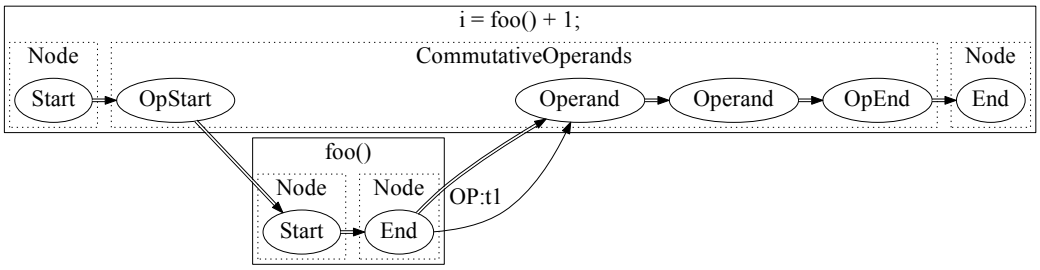


Abbildung 4.15: Operandenfragment einer Addition

Das aus Kapitel 3 bekannte Quelltextbeispiel 4.1 zeigt die Operanden einer Addition. Einer der Operanden ist ein Methodenaufruf, während der andere ein Literal und damit ein Terminalausdruck ist. Die Abstraktion in Abbildung 4.15 zeigt, wie die Unterelemente der Addition in der Abstraktion behandelt werden. Für den Methodenaufruf existiert eine eigene Komposition, während das Literal keine Komposition besitzt. Für beide Operanden der Addition werden außerdem Operandknoten im Operandenfragment erstellt. Da die Addition kommutativ ist und somit beide Operanden vertauscht werden können, ist das Operandenfragment vom Typ *CommutativeOperands*.

Die Operanden von Blockabstraktionen werden durch eine Analyse festgestellt, die im Abschnitt Blockabstraktionen beschrieben wird. Die Operanden von Blockabstraktionen ermöglichen die Einfügen-Umgestaltungen über Blockgrenzen hinweg. Grundsätzlich besitzt eine Blockabstraktion immer dann für eine Variable v einen Operanden, wenn v innerhalb des zugehörigen Quelltextblocks verwendet wird und ein Einfügen des v definierenden Ausdrucks möglich ist.

Kommutative Gruppen und Blockabstraktionen lassen Änderungen an der Operandenreihenfolge zu (siehe Transformation *Operanden umsortieren*, Abschnitt 3.6), während nicht-kommutative Gruppen ihre Ausdrücke/Deklarationen fest anordnen. Blockabstraktionen sind ebenfalls kommutativ. In Definition 4.16 werden Abhängigkeitskanten (CD-Kanten) verwendet, um ein Vertauschen der Operanden in nicht-kommutativen Blöcken zu verhindern. Welche Gruppen kommutativ sind und welche nicht, zeigt Tabelle 4.2.

Jedes Operandenfragment besteht laut Definition 4.16 aus mehreren Knoten, startet mit dem Startknoten vom Typ *OpStart* und endet mit dem Endknoten vom Typ *OpEnd*. Startet bzw. endet bedeutet hier, dass der Startknoten den kleinsten *op*-Wert und der Endknoten den größten *op*-Wert des Fragments besitzt. Die Operandknoten einer Gruppe repräsentieren Deklarations- und Leseoperationen. Bei Variablendeklarationen bilden bspw. die Operanden die Deklarations- und Initialisierungsreihenfolge der Variablen aus dem Quelltext ab. Bei Ausdrücken repräsentieren die Operandknoten die Leseoperation eines Operanden, wie eines Literals oder einer Variablen. Auch die Leseoperation des Rückgabewerts von Unterausdrücken wird von Operandknoten repräsentiert. In diesem Fall

wird allerdings zusätzlich eine OP-Datenflusskante vom Unterausdruck zum Operandknoten erzeugt. Bei Blockabstraktionen repräsentieren die Operanden die Leseoperation der im Block verwendeten Variablen.

$$
\begin{aligned}
&\forall f \in Fragments|^{\neg Node} \\
&\exists \\
&\{n_{OS}, n_{OE}\} \cup OperandNodes(f) \subseteq Nodes \\
&: \\
n_{OS}|^{OpStart} \wedge n_{OE}|^{OpEnd} \wedge \{n_{OS}, n_{OE}\} \cup OperandNodes(f) = f.N \wedge& \\
&\forall (n_i, n_j) \in SuccPairs_{Pos}(f.N) : \\
&(c^? = Nested^?(n_j) \wedge n_i \dashrightarrow c^? \dashrightarrow n_j \subseteq ControlFlow \wedge \\
&f.typ = NotCommutativeOperands \Rightarrow n_i \underset{CD}{\rightarrow} n_j \subseteq Edges|^{CD})
\end{aligned}
\tag{4.16}
$$

In Definition 4.16 steht die Knotenmenge $f.N$ für alle Knoten eines Operandenfragments. Sie enthält die Knoten vom Typ *OpStart*, *OpEnd* und Operandknoten vom Typ *Operand*. Die Operandknoten $n^{Operand}$ werden von der injektiven Funktion $OperandNodes(f)$ für das Fragment f geliefert. Die injektive Funktion $Nested^? : Nodes \rightarrow Compositions$ liefert die Könntekomposition $c^?$. Sie entspricht genau der Komposition c, die zum Unterausdruck des angegebenen Operandknotens gehört. Besitzt der angegebene Operandknoten keinen Unterausdruck, entspricht die Könntekomposition $c^?$ dem Nullelement $\perp$. Um darauf hinzuweisen, dass die Rückgabe der Funktion $Nested^?$ ein Könnteelement ist, wird der Funktionsname mit einem Fragezeichen versehen.

Die Abbildung $OperandNodes : Fragment \rightarrow \mathcal{P}(Nodes)$ liefert für jedes Fragment eine Menge von Operandknoten. Sie orientiert sich dabei an den Quelltexteigenschaften des durch das Fragment f repräsentierten JDT-Elements e. Ist e eine Blockabstraktion, liefert *OperandNodes* eine Menge von Operandknoten, die alle im `Block` gelesenen Variablen repräsentieren. Ist e ein `Ausdruck`, liefert die Abbildung eine Menge von Operandknoten, die sich aus der jeweiligen JDT-Klasse des Ausdrucks ergibt.

Die Klasse `Expression` der Klassenhierarchie von JDT besitzt mehrere, von ihr abgeleitete Klassen. In Tabelle 4.3 werden verschiedene Eigenschaften für den Umgang mit den abgeleiteten Klassen angegeben. Die Spalte *Repräsentiert durch Operandknoten* der Tabelle 4.3 zeigt, dass für jeden `Ausdruck` e in JDT ein Operand erzeugt wird, wenn e Teil eines anderen Ausdrucks ist. Allerdings wird nicht für jeden JDT-Ausdruck auch eine Komposition erstellt. In der Spalte *Repräsentiert durch Komposition* ist angegeben, aus welchen Ausdrücken Kompositionen erzeugt werden. Das Prädikat *Represented*, das in Abbildungsregel 4.3 genutzt wird, wird durch diese Spalte definiert. Für Literale wertet es bspw. zu *falsch* aus, sodass folglich keine Kompositionen für sie erzeugt werden. Literale werden in der Abstraktion lediglich als Operandknoten dargestellt.

Die Tabelle 4.3 gibt in der Spalte *JDT-Unterelemente* die Operanden für jede `Expression`-Klasse an. Bspw. liefert *OperandNodes* für ein Fragment, das einen `ArrayAccess` repräsentiert, die Operanden für den Array-Ausdruck und einen für den Feldindex. Der Zuweisungsausdruck wird als einziger gesondert behandelt, um die Anzahl der Kompositionen zu reduzieren und somit eine Optimierung der Abstraktion zu bewirken. Die Vorbedingung der Optimierung ist, dass die Zuweisung als Anweisung

JDT-Ausdruck	Repräsentiert durch: Operandknoten	Repräsentiert durch: Komposition (*Represented*)	JDT-Unterelemente (*OperandNodes*)
Annotation	-	-	-
ArrayAccess	✓	✓	Array, Index
ArrayCreation	✓	✓	Dimensionen, Initializer
ArrayInitializer	✓	✓	Ausdrücke
Assignment	✓	✓	Linke/rechte[a] Seite
BooleanLiteral	✓	-	-
CastExpression	✓	✓	Ausdruck
CharacterLiteral	✓	-	-
ClassInstanceCreation	✓	✓	Ausdruck, Argumente
ConditionalExpression	✓	✓	Bedingung, Dannausdruck, Sonstausdruck
FieldAccess	✓	✓	Ausdruck
InfixExpression	✓	✓	Operanden
InstanceofExpression	✓	✓	Operand
MethodInvocation	✓	✓	Ausdruck, Argumente
Name	✓	-	-
NullLiteral	✓	-	-
NumberLiteral	✓	-	-
ParenthesizedExpression	✓	✓	Ausdruck
PostfixExpression	✓	✓	Ausdruck
PrefixExpression	✓	✓	Ausdruck
StringLiteral	✓	-	-
SuperFieldAccess	✓	✓	-
SuperMethodInvokation	✓	✓	Argumente
ThisExpression	✓	✓	-
TypeLiteral	✓	-	-
VariableDeclarationExpression	✓	✓	Deklarationsfragmente

Tabelle 4.3: Unterelemente der JDT-Ausdrücke

[a]Bzw. die Unterelemente des Ausdrucks der rechten Seite.

auftritt, also nicht als Unterausdruck in einem geschachtelten Ausdruck. Falls dann der Ausdruck *rhs* der rechten Seite repräsentiert wird (*Represented*), wird *rhs* in der Abstraktion nicht abgebildet. Stattdessen werden die JDT-Unterelemente von *rhs* als direkte JDT-Unterelemente der Zuweisung verwendet. Da die Zuweisungsanweisung häufig in Quelltexten auftritt, kann hierdurch die Zahl der Kompositionen deutlich reduziert werden. Dies ist insbesondere auch aufgrund der Klone wichtig, die die Anzahl von Kompositionen weiter steigern (siehe Unterkapitel 4.4). Durch diese Optimierung ergeben sich keine Einschränkungen bei der Anwendung der Umgestaltungen, da sie durch die in Abschnitt 5.1.4 erklärte Wunschanpassung kompensiert werden kann.

Blockabstraktionen

Die Quelltextblöcke eines abgeschlossenen Quelltextblocks werden jeweils durch partielle Abstraktionen abgebildet. Sind b und b_u zwei Quelltextblöcke, wobei b_u ein direkter Unterblock von b ist, und sind a und a_u ihre zugehörigen Quelltextabstraktionen wird b_u innerhalb von a nur als Blockabstraktion f_u repräsentiert (siehe Definitionen in Unterkapitel 3.2). Zu den durch eine Blockabstraktion abstrahierten Quelltextblöcken gehören genau die in Tabelle 4.4 angegebenen JDT-Elemente.

JDT-Anweisung	Unterelement als Blockabstraktion
DoWhileStatement	Schleifenrumpf
EnhancedForStatement	Schleifenrumpf
ForStatement	Initialisierung, Aktualisierung, Schleifenrumpf
IfStatement	Dann- und Sonstblock
SynchronizedStatement	Block
WhileStatement	Schleifenrumpf

Tabelle 4.4: Durch eine Blockabstraktion abstrahierte JDT-Elemente

Blockabstraktionen besitzen wie kommutative Gruppen Operandknoten. Die Anzahl der Operandknoten ergibt sich aus den Operanden, die innerhalb des abstrahierten Blocks b existieren, wie folgt:

> **Operanden von Blockabstraktionen:** *Ist b ein Quelltextblock und B_u die Menge aller Unterblöcke von b inklusive b, wird ferner b durch die Blockabstraktion f abstrahiert und wird B_u durch die Menge der partiellen Abstraktionen A_u abstrahiert, ist außerdem $Edges_{A_u}$ die Kantenmenge aller partiellen Abstraktionen in A_u, ergibt sich die Menge aller Operandknoten von f aus der Menge der Symbole aller OP-Kanten von A_u wie folgt.*

$$\forall s \in \{s \in Symbols | \exists \overset{s}{\underset{OP}{\rightarrow}} \in Edges_{A_u} \wedge \neg DeclaredIn(s, A_u)\} : \exists n \in f.N|^{Operand}$$

Diese Definition schließt keine Variablen mit ein, die innerhalb eines Unterblocks B_u deklariert wurden. Denn solche Variablen müssen nicht als Operanden von f angegeben werden, da sie innerhalb von b unbekannt sind. Die Funktion $DeclaredIn : Symbol \times \mathcal{P}(PartialAbstraction) \rightarrow Boolean$ ermittelt für ein Symbol s und eine Menge von partiellen Abstraktionen A_u, ob s in einer partiellen Abstraktion von A_u deklariert wurde. Nur wenn ein s außerhalb der Quelltextunterblöcke von b deklariert wurde, wird für s ein Operand in f erzeugt.

4.2.6 Transformierbarkeit: Extrahieren/Einfügen

Die Abbildungsregeln erzeugen aus allen Anweisungen und Ausdrücken Kompositionen in der Quelltextabstraktion. Einige von diesen Kompositionen repräsentieren Ausdrücke im

Quelltext wie bspw. Methodenaufrufe und sind eingebettet in übergeordnete Ausdrücke. Das Extrahieren von Unterausdrücken ist in den meisten Fällen ohne Probleme möglich. Eine Ausnahme bilden hier Pre-/Post-de- und Inkrementoperationen (kurz: PI-Operationen) sowie die Zuweisung. Die linke Seite einer Zuweisung darf nicht extrahiert und durch eine temporäre Variable ersetzt werden. Ebenfalls darf die Variable der PI-Operationen nicht extrahiert und ersetzt werden. Allerdings ist in beiden Fällen weiterhin gestattet, Unterausdrücke der genannten Ausdrücke zu extrahieren. Das folgende Quelltextbeispiel 4.2 verdeutlicht die Situation.

```
1  class MyClass {
2    MyClass mc = this;
3    int i;
4    void trns(MyClass c) {
5      c.i++; // Extrahieren von c.i nicht möglich
6      c.mc.i++; // Extrahieren von c.mc möglich
7    }
8  }
```

Quelltext 4.2: Extrahieren der Variablen des Postinkrementausdrucks

Die Variable der Postinkrementanweisung im Beispiel 4.2 kann in Zeile 5 nicht extrahiert werden. Würde sie extrahiert und durch eine temporäre Variable `t` ersetzt, entstünde der Ausdruck `t++`, der den Wert von `c.i` nicht inkrementieren würde. In Zeile 6 kann dagegen der Unterausdruck `c.mc` der Postinkrementvariablen extrahiert und durch eine temporäre Variable `t` ersetzt werden. Denn der Ausdruck `t.i++` inkrementiert weiterhin denselben Wert wie vor der Extraktion.

Neben den Ausnahmen, die das Extrahieren einschränken, existieren auch Ausnahmen, die das Einfügen verhindern. Das Einfügen besteht aus drei Teilen. Erstens wird eine Variable v betrachtet, der zweitens ein Ausdruck a einer Initialisierung oder rechten Seite einer Zuweisung zugewiesen wird. Drittens wird a an einer Stelle im Quelltext eingefügt, an der zuvor v gelesen wurde. Die Voraussetzung dafür ist, dass ein Datenfluss von der Initialisierung bzw. Zuweisung zur Lesestelle existiert.

Im Quelltext existieren aber neben Initialisierungen und Zuweisungen noch andere Formen des Schreibens einer Variablen. Hierzu zählen PI-Ausdrücke und Zuweisungen, die als Unterausdrücke verwendet werden, sowie andere Operationen, die Seiteneffekte besitzen. Ferner kann das Lesen einer Variablen auch über Seiteneffekte geschehen. Sofern das Lesen und/oder Schreiben einer Variablen aufgrund eines Seiteneffekts geschieht, ist eine Einfügen-Umgestaltung nicht möglich. Zur genauen Unterscheidung der Lese- und Schreiboperationen werden im Unterkapitel 4.3 Programmeffekte eingeführt, aus denen Lese-/Schreibabhängigkeiten erzeugt werden. Hierdurch entstehen auch Abhängigkeitskanten vom Typ OP. Der OP-Kantentyp besagt, dass weder die Lese- noch die Schreibstelle einer Variablen durch einen Seiteneffekt verursacht wurde. Ferner besagt er, dass der Wert einer Variablen an ihrer Lesestelle auf genau einen Ausdruck zurück geführt werden kann. Dies ist für eine Einfügen-Umgestaltung notwendig, da in Fällen, in denen durch Kontrollflussverzweigung mehrere Schreibstellen im Quelltext existieren, nicht klar ist, welcher Ausdruck zum Einfügen verwendet werden soll.

Damit im Constraint-Problem klar ist, ob der Freiheitsgrad des Einfügens/Extrahierens überhaupt anwendbar ist, wird das Attribut *trn* genutzt. Es ist ein fixes Attribut jeder Komposition vom Typen *Boolean*. Das Attribut *inl* jeder Komposition hingegen ist ein

variables Attribut, dass als CSP-Variable im Constraint-Problem verwendet wird und seinen Wert während der Lösungssuche ändern kann. Der Zustand von *inl* gibt an, ob seine Komposition in einen übergeordneten Ausdruck eingebettet ist ($inl = wahr$) oder ob seine Komposition als einzeln stehende Zuweisung im Quelltext steht ($inl = false$). Mithilfe von *trn* wird daher angegeben, ob der Freiheitsgrad des Extrahierens/Einfügens einer Komposition zur Verfügung steht. Nur dann, wenn *trn wahr* ist, darf *inl* seinen initialen Wert ändern. Im Kapitel 5 wird die Generatorregel 5.17 dies angeben. Die Variable *trn* einer Komposition c ergibt sich aus dem JDT-Element, das von c repräsentiert wird.

> **Transformierbarkeit:** *Sei n_{Op} ein Knoten mit der Eigenschaft $n_{Op}|^{Operand}$ und $c_p = n_{Op}.fgm.cmp$ seine Komposition, die das JDT-Element e_p abbildet. Das Attribut c.trns einer Komposition c, die das JDT-Element e abbildet, ist genau dann wahr, wenn gilt:*
>
> 1. $\exists c \underset{OP}{\rightarrow} n_{Op} \in Edges$ und
> 2. e_p ist keine Pre- oder Post-de- oder -inkrementoperation und
> 3. e ist nicht die linke Seite einer Zuweisung.

Eine Komposition c muss in der Abstraktion repräsentiert werden, damit sie extrahiert oder eingefügt werden kann. Da sich das Extrahieren bzw. Einfügen nur entlang von Operandenabhängigkeiten (OP-Kanten) vollziehen kann, muss ferner eine solche Kante existieren, die von c aus startet. Sie zielt auf den Operandknoten n_{Op} einer Komposition c_p, die ein JDT-Element e_p repräsentiert. Damit ein Extrahieren möglich ist, darf weder der zu extrahierende Ausdruck e die linke Seite einer Zuweisung, noch der übergeordnete Ausdruck e_p von e eine Pre-, Post-de- oder -inkrementoperation sein.

4.3 Abhängigkeiten im Quelltext

Die Kontrollflusskanten werden genutzt, um die Abhängigkeiten zwischen Knoten der Abstraktion zu repräsentieren. Zusätzlich zum Kontrollfluss ist dazu jedoch auch die Information über die Lese- und Schreibeffekte der Knoten notwendig. Im Folgenden wird präsentiert, wie der JDT-AST genutzt wird, um derartige Abhängigkeiten im Quelltext zu identifizieren. Hierzu werden die Bindungsinformationen genutzt, die JDT bereitstellt. Die Präsentation zeigt die wichtigsten Zusammenhänge der Vorgehensweise, nutzt jedoch häufig auch Abbildungsfunktionen, die weitere Details der Vorgehensweise abstrahieren. In diesem Unterkapitel wird die genutzte interprozedurale Analyse vorgestellt. Die vorliegende Arbeit orientiert sich dabei an Bernsteins Definition von Datenabhängigkeiten [14], die auf Ausdrücken basiert, um Programmabhängigkeiten festzustellen.

> **Datenabhängigkeit:** *Für einen Ausdruck A sind $R(A)$ die Menge der Variablen, die vom Ausdruck gelesen werden und $W(A)$ die Menge der Variablen, die vom Ausdruck A geschrieben werden. Zwei Ausdrücke A_1 und A_2 besitzen eine Datenabhängigkeit genau dann, wenn:*
>
> $$((W(A_1) \cap W(A_2)) \cup (R(A_1) \cap W(A_2)) \cup (W(A_1) \cap R(A_2))) \neq \emptyset$$

Über Bernsteins Definition hinaus sollen hier jedoch auch Abhängigkeiten beachtet werden, die aufgrund von Sprüngen entstehen. Im folgenden Beispiel sollen kurz die Effekte erklärt werden, die in Java-Programmen auftreten können. Diese Effekte verursachen entlang des Kontrollflusses Abhängigkeiten im Quelltext und somit auch in der Abstraktion.

```
1  void foo() {
2    int i;
3    i = 1;
4    while (true) {
5      if (i == 1) {
6        continue;
7      }
8    }
9  }
```

Quelltext 4.3: Effekte verursachen entlang des Kontrollflusses Abhängigkeiten im Quelltext

Der Quelltext 4.3 zeigt mehrere Anweisungen, die verschiedene Effekte bewirken können. Die Deklaration in Zeile 2 bewirkt einen Deklarationseffekt und die sich anschließende Zuweisung in Zeile 3 bewirkt einen Schreibeffekt auf die Variable `i`. Ferner erzeugt der Äquivalenzausdruck in der Alternative (Zeile 5) einen Leseeffekt auf die Variable `i` und schließlich verursacht die Sprunganweisung in Zeile 6 einen Sprungeffekt. Schließlich besitzt die `While`-Schleife in Zeile 4 auch einen Fangeffekt, der den Kontrollflusssprung auffängt, der innerhalb der Schleife durch den Sprungeffekt der `Continue`-Anweisung verursacht wurde. Diese Effekte werden in Kombination mit dem Kontrollfluss betrachtet, um Abhängigkeiten im Quelltext festzustellen.

Zunächst werden die dafür relevanten Programmeffekte, wie das Lesen oder Schreiben einer Variablen, eingeführt. Diese werden dann auf Knoten abgebildet, um Abhängigkeitskanten zwischen zwei Knoten eines Kontrollflusspfades zu erzeugen. Schließlich werden Beispiele für verschiedene Programmabhängigkeiten vorgestellt.

4.3.1 Programmeffekte

Programmeffekte ergeben sich aus Variablenzugriffen und -deklarationen sowie Programmsprüngen. Die Menge der Variablen wird mit $Variables$ angegeben und kann auf Symbole der Quelltextabstraktion abgebildet werden. Zugriffe auf eine Variablen v entstehen, wenn ihre Symbole gelesen oder geschrieben werden. Sie können genauer unterschieden werden, je nachdem, auf welche Art das Lesen oder Schreiben durchgeführt wird. Leseoperationen entstehen direkt durch das Referenzieren einer Variablen. Schreibende Seiteneffekte entstehen durch Methodenaufrufe, aber auch z. B. durch die Postinkrementoperation. Ferner wird eine Variable geschrieben, wenn ihr ein neuer Wert durch einen `Zuweisungsausdruck` zugewiesen wird. Neben Abhängigkeiten, die durch Lese- und Schreibeffekte entstehen, werden auch Sprung-, Fang- und Deklarationseffekte beachtet, die ebenfalls Abhängigkeiten im Quelltext verursachen. Im Folgenden werden die verschiedenen Effekte definiert.

> **Effekt:** *Ein Effekt wird als Tupel angegeben, wobei das erste Element den Effekttyp und das zweite Element die Variable bestimmt. Der Effekttyp ist ein Element aus* $\{Jump, Decl, Ref, Asgn, SE_W, Write\}$.
>
> *Die Abbildung* $Typ : Effekt \rightarrow Effekttyp$ *bildet einen Effekt auf dessen Typ ab.*

Die Abbildung $Var : Effekt \rightarrow Variable$ bildet einen Effekt auf dessen Variable ab.

Sprungeffekt ($Jump$)**:** *Ist ein JDT-Element e, abstrahiert durch die Komposition c, eine Sprunganweisung, dann besitzt c einen Sprungeffekt. Ferner besitzt c einen Sprungeffekt, wenn e ein Methodenaufruf ist, dessen Methodendeklaration das Werfen einer Ausnahme definiert.*

Fangeffekt ($Catch$)**:** *Ist ein JDT-Element e, abstrahiert durch die Komposition c, eine CatchClause oder eine Schleife, dann besitzt c einen Fangeffekt.*

Deklarationseffekt ($Decl, v$)**:** *Ist ein JDT-Element e, abstrahiert durch die Komposition c, ein JDT-VariableDeclarationStatement oder VariableDeclarationExpression, und besitzt e Unterelemente vdf_i der Klasse VariableDeclarationFragment, dann besitzt c für jede von vdf_i deklarierte Variable v einen Deklarationseffekt mit der Deklarationsposition i.*

Referenzeffekt (Ref, v)**:** *Besitzt ein JDT-Element e, abstrahiert durch die Komposition c, als direktes Unterelement das Symbol s_i mit der Variablen v, dann besitzt c einen Referenzeffekt auf v. Die Referenzposition i des Symbols ergibt sich aus der Abbildungsregel von e auf c.*

Zuweisungseffekt ($Asgn, v$)**:** *Ist e ein JDT-Assignment, abstrahiert durch die Komposition c und bildet das Symbol s mit der Variablen v die linke Seite von e, dann besitzt c einen Zuweisungseffekt auf v.*

Seiteneffekt Schreiben (SE_W, v)**:** *Ist e ein JDT-Element, abstrahiert durch die Komposition c, ist s ein Symbol mit der Variablen v, dann besitzt c einen schreibenden Seiteneffekt auf v wenn gilt, dass e eine JDT-Pre- und Post-de- und inkrementoperation ist, oder dass e eine Zuweisung ist, wobei s die linke Seite von e bildet.*

Schreibeffekt ($Write, v$)**:** *Ist ef ein Zuweisungseffekt oder schreibender Seiteneffekt, wird ef auch Schreibeffekt genannt.*

Die verschiedenen Effekte von Ausdrücken und Anweisungen werden auf die Abstraktion abgebildet. Die Menge aller Effekte wird im Folgenden $Effects$ genannt. Die Funktion $CEffects : Compositions \rightarrow \mathcal{P}(Effects)$ liefert für eine Komposition die Menge ihrer Effekte. Da jede Anweisung und jeder Ausdruck auf Kompositionen abgebildet werden, die aus mehreren Knoten bestehen, werden die Effekte auf die einzelnen Knoten abgebildet. Hierfür wird die Funktion $NEffects : Compositions \times Effects \rightarrow Nodes$ genutzt, die für eine Komposition und einen zugehörigen Effekt den Knoten liefert, auf den der Effekt abgebildet wird. Um eine Vorstellung davon zu vermitteln, auf welche Knoten die Effekte abgebildet werden, zeigt die Tabelle 4.5 allgemein, auf welche Knotentypen ein Effekt für eine beliebige Komposition abgebildet wird. Im Falle des Knotentyps *End* ist die Abbildung eindeutig, da jede Komposition immer genau einen Knoten dieses Typs besitzt. Knoten der Typen *Decl* und *Operand* können jedoch mehr als einmal in c auftreten. Falls

$Typ(Effekt)$	Knotentyp
$Asgn$	End
Ref	Operand[a]
SE_W	End
$Decl$	Decl[b]
$Jump$	End
$Catch$	End

Tabelle 4.5: Die Tabelle gibt die Knotentypen der Effekte an.

[a]Der Knoten, dessen JDT-Element den Referenzeffekt besitzt.
[b]Der Knoten, dessen JDT-Element den Deklarationseffekt besitzt.

mehrere Operand- bzw. Deklarationsknoten in c existieren, wird der Effekt auf denjenigen Knoten abgebildet, der das effektauslösende JDT-Element repräsentiert.

Mithilfe der Abbildung $NTEffect$ und der evtl. notwendigen Zuordnung von Effekten auf Knoten einer Komposition kann für jeden Knoten angegeben werden, welche Knoteneffekte er besitzt. Hierfür wird die Abbildung Ef genutzt. Sie ist im Folgenden wichtig, um Abhängigkeitskanten zu definieren.

> **Knoteneffekt:** *Ist n ein Knoten einer Komposition c und ist $ef \in Effects(c)$ ein Effekt von c, der auf n abgebildet wird, wird ef Knoteneffekt von n genannt.*
>
> *Die Abbildung $Ef : Nodes \to \mathcal{P}(Effects)$ gibt die Knoteneffekte für einen Knoten n an.*

4.3.2 Effekte von Blockabstraktionen

Blockabstraktionen abstrahieren die Anweisungen, Ausdrücke und Effekte von Unterblöcken im Quelltext. Die Effekte einer Blockabstraktion umfassen die Menge aller Effekte der Quelltextunterblöcke. Schließlich können in Unterblöcken Variablen gelesen, geschrieben oder Sprunganweisungen aufgerufen werden. Aufgrund einer Sprunganweisung kann sich der Kontrollfluss an Blockabstraktionen verzweigen. Diese Effekte beeinflussen die Datenflüsse und damit die Abhängigkeiten einer partiellen Abstraktion. Um diese Effektmenge zu identifizieren, wird die Menge aller partiellen Blockabstraktionen analysiert, die den Quelltextblock der untersuchten Blockabstraktion und seine Unterblöcke abstrahieren.

Sprungeffekte werden jedoch nicht unbedingt aus der Menge aller partiellen Abstraktionen zur Blockabstraktion propagiert. Es ist möglich, dass ein Sprung innerhalb eines Unterblocks im Quelltext durch eine andere Anweisung aufgefangen wird. Bspw. werden die durch `Throw`-Anweisungen geworfenen Ausnahmen durch `CatchClauses` gefangen, wenn der Typ der Ausnahme akzeptiert wird (§11.3). Ebenso lenken die durch `Break`- und `Continue`-Anweisungen ausgelösten Sprünge den Kontrollfluss zum Anfang oder Ende der nächsten äußeren Schleife. Falls ein `Label` angegeben wurde, wird der Kontrollfluss zum Anfang bzw. Ende der entsprechend markierten Schleife gelenkt. Die Fähigkeit einer Anweisung, einen Sprungeffekt zu fangen, wird mithilfe des Fangeffekts abgebildet. Nur wenn ein Sprung eines Unterblocks nicht gefangen wird und somit den Quelltextblock b selbst abrupt beenden kann (§14.1), wird der Sprungeffekt auf den Quelltextblock b

propagiert. Die Funktion $Catches : Effects \times Effects \rightarrow Boolean$ (siehe Tabelle 4.6) gibt an, ob ein Sprungeffekt durch einen Fangeffekt gefangen werden kann. Sie berücksichtigt dabei auch die Blockhierarchie.

Sprungeffekt fangen: *Eine Komposition c, die das JDT-Element e abstrahiert, das den durch abrupte Beendigung einer Anweisung ausgelösten Kontrollflusssprung j fortsetzt, besitzt einen Fangeffekt, der den Sprungeffekt von j auffängt, sofern die Funktion Catches zu wahr auswertet.*

Auslöser des Sprungeffekts ef_s	Auslöser des Fangeffekts ef_f	$Catches(ef_s, ef_f)$
ReturnStatement	-	✗
BreakStatement	Schleife[a]	✓
ContinueStatement	Schleife[a]	✓
ThrowStatement	CatchClause[b]	✓

Tabelle 4.6: Abbildung *Catches* definiert wann ein Sprung gefangen wird

[a] ggf. gleiches Label
[b] sofern der Ausnahmetyp akzeptiert wird

Mithilfe der Funktion *Catches* werden nun die resultierenden Sprungeffekte einer Blockabstraktion definiert.

Blockabstraktionseffekte: *Ist b ein Quelltextblock, abstrahiert durch die Blockabstraktion f und ist B_u die Menge aller Quelltextunterblöcke von b, die ferner f enthält, wird außerdem B_u durch die Menge der partiellen Abstraktionen A_u abstrahiert, ergeben sich die Mengen aller Sprungeffekte (Jumps), Referenzeffekte (Refs), Seiteneffekte (SE_W) und Gesamteffekte (Effects) von f wie folgt.*

$$Jumps(f) = \{ef_j | \exists c_j \in A_u.C : ef_j \in Effects(c_j) \wedge Typ(ef_j) = Jump \wedge \forall c_c \in A_u.C : \nexists ef_c \in Effects(c_c) : Catches(ef_j, ef_c)\}$$

$$Refs(f) = \{(Ref, s)| \wedge \exists \xrightarrow[OP]{s} \in Edges_{A_u} \wedge \neg DeclaredIn(s, A_u)\}$$

$$SE_W(f) = \{ef_{se} | \exists c_{se} \in A_u.C : ef_{se} \in Effects(c_{se}) \wedge Typ(ef_{se}) = Write\}$$

$$Effects(f) = Jumps(f) \cup Refs(f) \cup SE_W(f)$$

4.3.3 Abhängigkeitskanten

Wenn die Knoteneffekte bekannt sind, können entlang der Kontrollflusskanten Abhängigkeitskanten erstellt werden. Dabei sind solche Kontrollflusspfade zwischen zwei Knoten n_s

und n_e interessant, bei denen n_s und n_e Knoteneffekte besitzen. Je nach den Effekten, die die Knoten n_s und n_e eines Kontrollflusspfads besitzen, können Programmabhängigkeiten und folglich Abhängigkeitskanten in der Abstraktion existieren.

Kontrollflusspfad: *Ein Pfad* $n_1 \dashrightarrow n_2 \dashrightarrow \ldots \dashrightarrow n_k$ *wobei* $k > 1$ *und* $n_i \in Nodes$ *für* $i \in \{1, \ldots, k\}$*, heißt Kontrollflusspfad, falls jede Kante* $n_i \rightarrow n_{i+1}$*,* $i \in \{1, \ldots, k-1\}$ *eine Kontrollflusskante ist.*

Kontrollflusspfadtupel: *Ein Tupel* (n_s, n_e) *von Knoten heißt Kontrollflusspfadtupel, falls es einen Kontrollflusspfad* $n_1 \dashrightarrow n_2 \dashrightarrow \ldots \dashrightarrow n_k$*,* $k > 1$ *gibt, sodass* $n_s = n_1$ *und* $n_e = n_k$ *ist.*

Knoteneffekt		$ET(n_s, n_e)$
$Ef(n_s)$	$Ef(n_e)$	$(Kantentyp, Variable)$
$(Asgn, v)$	(Ref, v)	$(OP, v)^a$
$(Asgn, v)$	(Ref, v)	$(FD, v)^b$
(SE_W, v)	(Ref, v)	(SE, v)
(Ref, v)	$(Write, v)$	(AD, v)
$(Decl, v)$	$(Write, v)$	$(INIT, v)$
$(Write, v)$	$(Write, v)$	(DOD, v)
$(Write, v)$	$(Jump, -)$	(IO, v)
$(Jump, -)$	$(Write, v)$	(IO, v)

Tabelle 4.7: Abbildung ET

[a]Bedingung: Es existiert genau ein OP-/FD-Datenfluss zu Ref und die Komposition von n_s ist klonbar.
[b]Sonst

Die Tabelle 4.7 definiert die Abbildung $ET : Nodes \times Nodes \rightarrow EdgeType \times Variables$. Sofern in ET eine Abbildung für zwei Knoten n_s und n_e eines Kontrollflusspfads auf einen Kantentyp existiert, wird eine Abhängigkeitskante mit dem ermittelten Typ erzeugt. Voraussetzung dafür ist, dass die Knoten die angegebenen Effekte besitzen. Bspw. werden Operandenkanten (Typ: OP) genau dann erstellt, wenn ein Datenfluss existiert, der bei einer Zuweisung zur Variablen v beginnt und bei einer Referenz auf v endet. Eine weitere Einschränkung in diesem Fall ist, dass nur exakt einer dieser Datenflüsse hin zur Referenz fließen darf. Würden bspw. zwei getrennte Datenflüsse von unterschiedlichen Zuweisungen zu ein und derselben Referenz fließen, wird statt einer OP-Kante eine FD-Kante erzeugt, wie in Zeile 2 der Tabelle spezifiziert ist. Die folgende Definition gibt an, wie Abhängigkeitskanten in der Abstraktion erzeugt werden.

Kantenbildungsregel: *Ist* $CFPTupel$ *die Menge aller Kontrollflusspfadtupel und ist* $ET(n_s, n_e)$ *definiert, wird eine Abhängigkeitskante zwischen* n_s *und* n_e *wie folgt definiert.*

$$\forall (n_s, n_e) \in \{(n_s, n_e) \in CFPTupel | n_s.fgm.cmp \neq n_e.fgm.cmp\}$$
$$\exists$$
$$n_s \overset{Var(ET(n_s,n_e))}{\underset{Typ(ET(n_s,n_e))}{\rightarrow}} n_e \in Edges$$

4.3.4 Beispielabhängigkeiten

Abhängigkeitskanten entstehen sowohl aufgrund von Datenflüssen, als auch aufgrund von Sprunganweisungen. Im Folgenden werden Beispiele für beide Fälle präsentiert.

Datenabhängigkeiten

Bereits wenige Anweisungen erzeugen in der Abstraktion mehrere Knoten, die je nach den zugrunde liegenden Quelltextelementen Lese- und Schreibeffekte besitzen können. Im folgenden Quelltextbeispiel 4.4 wird die Postinkrementanweisung auf der Variablen `a` durchgeführt, bevor der Variablen `b` der Wert von `a` zugewiesen wird.

```
void foo(int a, int b) {
  a++;
  b = a;
}
```

Quelltext 4.4: Die Anweisungen erzeugen eine Datenflussabhängigkeit.

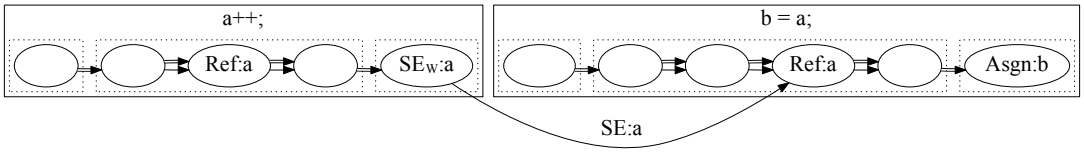


Abbildung 4.16: Abhängigkeiten in der Abstraktion von Quelltext 4.4

Abbildung 4.16 zeigt die Quelltextabstraktion des Beispielprogramms. Der Kontrollfluss verbindet im Beispiel alle Knoten von links nach rechts miteinander. Die Knoten der Abstraktion sind mit ihren Lese-/Schreibeffekten und den zugehörigen Variablen beschriftet, die durch Zugriffe wie Definitionen (Schreiben einer Variablen) oder Verwendungen (Lesen einer Variablen) entstehen. Der letzte Knoten der Zuweisung `a++;` ist mit $SE_W : a$ beschriftet, da er den Wert der Variablen `a` durch einen Seiteneffekt definiert. In der Zuweisung `b=a;` bewirkt der Knoten mit der Beschriftung $Ref : a$ das Lesen des Werts der Variablen `a`. Folglich werden die beiden Knoten daher durch eine Datenflusskante miteinander verbunden.

Sprungabhängigkeiten

Durch einen bedingten Sprung entsteht eine Abzweigung des Kontrollflusses, entlang derer Daten fließen können. Auf diesem Weg können folglich zusätzliche Datenabhängigkeiten entstehen. Würden Sprünge entfernt oder verschoben, könnten die durch sie bestehenden Abhängigkeiten beeinträchtigt werden.

```
1       void foo(int i) {
2         while (...) {
3           i = 1;
4           if (...) break;
5           i = 2;
6         }
7         print(i);
8       }
```

Quelltext 4.5: Datenflusskritischer Sprung durch Abbruchanweisung `break`

Im Quelltext 4.5 wird ein Programm gezeigt, dessen Kontrollfluss sich aufgrund einer Abbruchanweisung teilt. Dabei wird sowohl vor als auch nach dem Sprung eine Definition der Variablen `i` durchgeführt. Im Anschluss an die Schleife wird `i` als Methodenparameter gelesen. Daher existieren zwei Datenflussabhängigkeiten, die sowohl in Zeile 3 als auch in Zeile 5 starten und bei `print(i)` enden. Wenn in dieser Situation die `Alternative` mit einer der Zuweisungen vertauscht wird, wird eine der Abhängigkeiten zerstört. Eine ähnliche Situation entsteht, wenn statt der `Break`-Anweisung eine andere Sprunganweisung verwendet wird. Wenn bspw. ein Methodenaufruf verwendet wird, dessen Methodendeklaration das Werfen einer Ausnahme definiert, kann sich der Kontrollfluss ebenfalls teilen. Wird eine Ausnahme geworfen, verzweigt der Kontrollfluss in den `Catch`-Block. Existiert kein `Catch`-Block, wird die Ausführung der Methode abrupt beendet (§14.1).

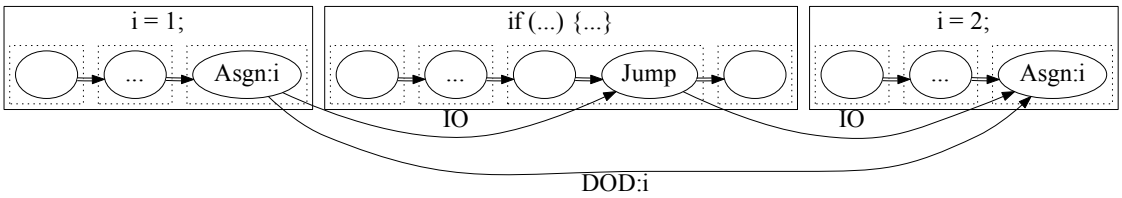


Abbildung 4.17: Abhängigkeiten durch einen Sprungeffekt

In Abbildung 4.17 wird die partielle Abstraktion des Schleifenrumpfs dargestellt. Die Knoten der nicht kommutativen Gruppen sind aus Platzgründen verkürzt dargestellt. In der Abbildung wird ersichtlich, dass der Sprungeffekt der `Break`-Anweisung auf den Endknoten der Blockabstraktion des Dannblocks der Alternative propagiert wurde. Hierdurch entstehen entsprechend der Kantenbildungsregeln (Unterkapitel 4.3.3) zwei IO-Kanten zwischen den Definitionen der Variablen `i` und der Blockabstraktion.

4.4 Klonerzeugung

Im abgeschlossenen Quelltextblöcken werden häufig Variablen genutzt, die an einer Stelle geschrieben und an verschiedenen, nachfolgenden Stellen innerhalb des Methodenrumpfs gelesen werden. Hierdurch ergeben sich zwei mögliche Situationen, in denen die bisher vorgestellte Quelltextabstraktion nicht ausreicht, um die Einfügen-Umgestaltung durchführen zu können. Die Einfügen-Umgestaltung kann dazu führen, dass der einzufügende Ausdruck kopiert werden muss. Dies geschieht, wenn er an mehreren Stellen im Quelltext eingefügt werden soll. In Hinblick darauf, dass die Abstraktion in ein Constraint-Problem

überführt wird, entsteht jedoch ein Problem. Der zu kopierende Ausdruck wird bisher lediglich von einer Komposition c abstrahiert. Die variablen Attribute von c werden später in Constraint-Variablen umgewandelt und geben u. a. über das Attribut *cp* an, an welcher Position sich die Komposition befindet. Wenn der Umgestaltungswunsch jedoch angibt, dass sich ein und dieselbe Komposition an mehreren Positionen gleichzeitig befinden soll, wird das Constraint-Problem unlösbar. Daher müssen Kopien der Komposition erstellt werden, die eigene Constraint-Variablen besitzen. Diese können dann so an alle Stellen verschoben werden und Positionswerte annehmen, dass der Umgestaltungswunsch erfüllbar ist. Die Kopien, auch *Klone* genannt, werden mithilfe der Transformation *Datendistribution* (siehe Abschnitt 3.6) erstellt.

Klone müssen ebenfalls erstellt werden, wenn sich aufgrund der Einfügen-Umgestaltung ein Ausdruck über Blockgrenzen hinweg bewegt. Das Ergebnis dieser Bewegung ist, dass sich Klone der zugehörigen Komposition des Ausdrucks in allen partiellen Abstraktionen der betroffenen Blöcke befinden. Diese Art der Klonerstellung wird *Propagierung von Kompositionen* genannt und im folgenden Abschnitt vorgestellt. Anschließend wird präsentiert, wie Klone erstellt werden, die dazu dienen, dass eine Komposition innerhalb einer partiellen Abstraktion eingefügt werden kann. Die Erzeugung der Klone wird *Expansion* genannt und im letzten Abschnitt erklärt.

Zunächst wird jedoch geklärt, wann eine Komposition klonbar ist. Denn nicht jeder Ausdruck im Quelltext und folglich nicht jede Komposition darf kopiert werden.

> **Klonbarkeit:** *Ist e ein JDT-Element und c seine Komposition, ist ferner s die JDT-Anweisung, die e enthält, so ist c klonbar, wenn e ein JDT-Ausdruck ist und die folgenden Bedingungen erfüllt sind:*
>
> 1. *e ist vom JDT-Typ Expression.*
> 2. *Falls e vom JDT-Typ Assignment ist, ist die rechte Seite von e klonbar.*
> 3. *Kein Unterelement von s ist eine Variablendeklaration oder eine Objekt- bzw. Felderzeugung.*

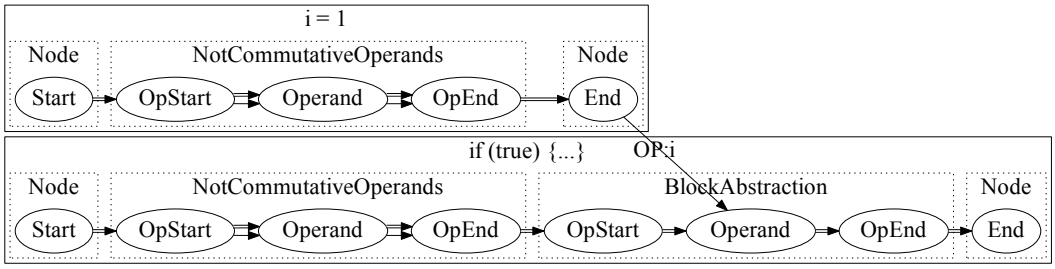
4.4.1 Propagierung von Kompositionen

Zunächst abstrahiert eine partielle Abstraktion a eines Quelltextblocks b nur JDT-Elemente, die in b liegen. Unter bestimmten Bedingungen werden jedoch auch JDT-Elemente von a abstrahiert, die sich in einem übergeordneten Quelltextblock von b befinden. Der Grund dafür ergibt sich aus der Einfügen-Umgestaltung. Die Einfügen-Umgestaltung fügt den einer Variablen v zugewiesenen Ausdruck e an allen Stellen s im Quelltext ein, an welchen v gelesen wird. Allerdings können Situationen im Quelltext vorliegen, in denen e und s in unterschiedlichen Quelltextblöcken liegen. Dementsprechend befänden sich die Kompositionen zu e und s in unterschiedlichen partiellen Abstraktionen und eine Berechnung der Einfügen-Umgestaltung an der Stelle s wäre unmöglich.

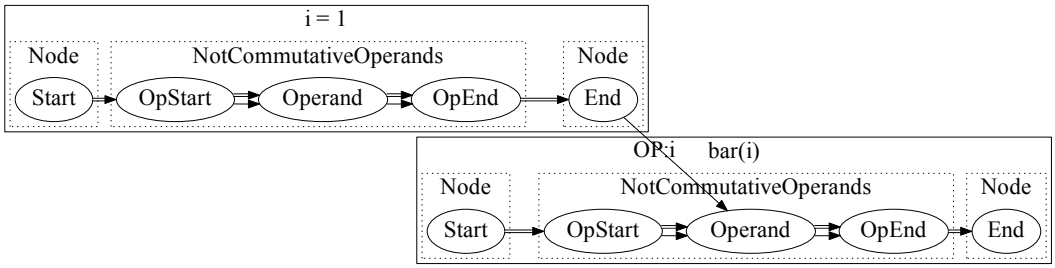
Das Quelltextbeispiel 4.6 veranschaulicht die Situation. Im Beispiel wird die Variable `i` im Methodenblock durch das Literal `1` definiert. Die Verwendungsstelle von `i` liegt jedoch innerhalb des Dannblocks der Alternative. Die zugehörigen partiellen Abstraktionen des Beispiels zeigen, dass der Zuweisungsausdruck nicht nur im Methodenblock, sondern auch im Dannblock existiert.

```
abstract void bar(int i);
void nestingBA(int i) {
  i = 1;
  if (true) {
    bar(i);
  }
}
```

Quelltext 4.6: Quelltext mit verschachteltem Additionsausdruck



(a) Die partielle Abstraktion des Methodenblocks zeigt die Blockabstraktion des Dannblocks, der einen Operanden für die Variable `i` besitzt.



(b) Die partielle Abstraktion des Dannblocks zeigt den Methodenaufruf sowie die Zuweisungsanweisung, die in die Abstraktion des Dannblocks propagiert wurde.

Abbildung 4.18: Quelltextabstraktion des Quelltexts 4.6.

Damit der Freiheitsgrad des Einfügens von Ausdrücken nicht aufgrund von Blockgrenzen eingeschränkt wird, werden Kompositionen in andere partielle Abstraktionen propagiert. Die Komposition c des JDT-Elements e existiert daher nicht nur innerhalb der partiellen Abstraktion des Quelltextblocks b. Stattdessen wird ein Klon von c erzeugt, der Teil jener partiellen Abstraktion b_s ist, deren Quelltextblock s enthält. Ist e selbst ebenfalls ein verschachtelter Ausdruck, der die Unterausdrücke e_u besitzt, wird nicht nur von c sondern auch von allen Unterkompositionen c_u ein Klon erstellt und in die partielle Abstraktion b_s propagiert. Darüberhinaus existieren c bzw. Klone von c und deren Unterkompositionen nicht nur in den partiellen Abstraktionen zu b und b_s, sondern auch in allen anderen partiellen Abstraktionen, deren Quelltextblöcke hierarchisch zwischen b und b_s liegen.

Propagierung von Kompositionen: *Ist v eine Variable, die vom JDT-Element s verwendet wird und ist e der Ausdruck, der genau den von s gelesenen Wert von v definiert, ist ferner b der Quelltextblock von e und b_s der*

Quelltextblock von s, ist außerdem c die klonbare Komposition von e oder die Komposition eines Unterausdrucks von e und ist B die Menge aller Elternblöcke von b_s abzüglich der Elternblöcke von b aber inklusive b und b_s, wird c in alle partiellen Abstraktionen von B propagiert.

4.4.2 Klone innerhalb von partiellen Abstraktionen

Wird eine Variable an einer Stelle w geschrieben und an verschiedenen, nachfolgenden Stellen r_i im selben Block gelesen, ergeben sich mehrere Datenflussabhängigkeiten von w zu den r_i. Klongruppen sind für diese mehrmaligen Verwendungen einer Variablen wichtig. Ob der Variablen anschließend ein neuer Wert zugewiesen wird, ist nicht ausschlaggebend. An jeder Einfügen-Stelle im Quelltext muss der zugewiesene Ausdruck statt der Variablen eingefügt werden. Folglich muss der zugewiesene Ausdruck kopiert werden.

```
i = 1;
f(i);
f(i);
```

Quelltext 4.7: Einfügen an mehreren Stellen möglich

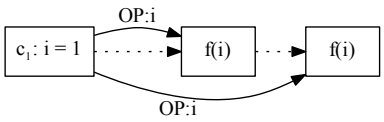
```
j = 1;
f(j)
f(j++);
```

Quelltext 4.8: Einfügen nur einmal möglich

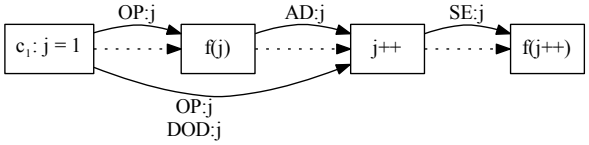
Wird eine Variable an k Stellen im Quelltext gelesen, so muss für $k-1$ Stellen eine Kopie des ihr zuvor zugewiesenen Ausdrucks erstellt werden. Für Variable `i` in Quelltext 4.7 ist dies das Literal `1`. Der ursprüngliche Ausdruck kann schließlich an der k-ten Stelle eingesetzt werden. Unter Umständen kann jedoch der Ausdruck nicht an allen k Stellen im Quelltext eingesetzt werden, sondern aufgrund von Abhängigkeiten nur an einigen, wie Variable `j` in Quelltext 4.8. In solchen Fällen verbleibt der ursprüngliche Ausdruck bei der Initialisierung. Außerdem wird für jede Stelle, an der ein Einfügen möglich ist, eine Kopie des ursprünglichen Ausdrucks erstellt. Für jede Lesestelle muss ermittelt werden, ob eine Verschiebung des Ausdrucks von der Schreib- zur Lesestelle möglich ist, ohne bestehende Abhängigkeiten zu verletzen. Um diese Analyse durchführen zu können, werden für einen Definitionsausdruck *e Klone* für jede Lesestelle von *e* erstellt.

Die Abbildungen 4.19c und 4.19d zeigen die Abstraktionen für die Lesestellen der beiden Variablen `i` und `j` aus den Quelltexten 4.7 und 4.8. Die gestrichelten Kanten stellen Kontrollflusskanten dar. Für die Initialisierungen beider Variablen werden jeweils zwei Klone erstellt. Für Variable `i` ist dies notwendig, da `i` an beiden Lesestellen eingefügt werden kann. Im Gegensatz dazu kann Variable `j` nicht an beiden Stellen eingefügt werden, dennoch sind zwei Klone der Zuweisung `j = 1;` notwendig. Der Grund dafür ist, dass `j` an einer Stelle eingesetzt werden kann und die Initialisierung von `j` weiterhin notwendig bleibt; nur so kann die Abhängigkeit zur Postinkrementanweisung erhalten bleiben.

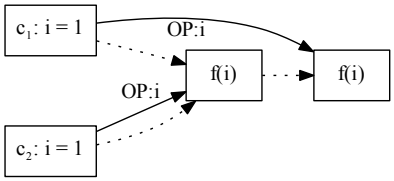
Klone entstehen entlang bestimmter Abhängigkeitskanten in der Quelltextabstraktion. Die einzelnen Kantentypen und ihre Erzeugung wurden im Unterkapitel 4.3 präsentiert: OP-Kanten repräsentieren eine Abhängigkeit, entlang derer eine Einfügen-Transformation möglich ist, sofern die einzufügende Komposition transformierbar ist (siehe Abschnitt 4.2.6). D. h. dass der Ausdruck der Komposition des Quellknotens im Ausdruck der Komposition des Zielknotens eingefügt werden könnte. FD- und SE-Kanten repräsentieren genau wie OP-Kanten einen Datenfluss von der Schreibstelle einer Variablen zur Lesestelle. Sie unterscheiden sich allerdings von OP-Kanten dadurch, dass entlang dieser Kanten keine



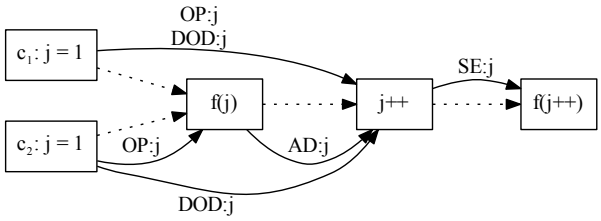
(a) Abstraktion ohne Klone (Quelltext 4.7)



(b) Abstraktion ohne Klone (Quelltext 4.8)



(c) Expandierte Abstraktion (Quelltext 4.7)



(d) Expandierte Abstraktion (Quelltext 4.8)

Abbildung 4.19: Abstraktionen vor und nach der Expansion

Einfügen-Transformation durchgeführt werden kann. Die Ursache ist bspw. dass das einzufügende JDT-Element eine Schleife ist, die eine Variable ändert, und syntaktisch nicht eingefügt werden kann. Ein anderer Grund könnte sein, dass das Schreiben der Variablen durch einen Seiteneffekt eines Postinkrementausdrucks a ausgelöst wird, und a ebenfalls nicht eingefügt werden kann.

Das Gleichungssystem wird aus dem Quelltext erstellt und kann während des Lösens nicht manipuliert werden. Enthielte es – wie der Quelltext – den einzufügenden Ausdruck nur ein einziges Mal, könnte es nicht feststellen, ob dieser Ausdruck an allen Lesestellen zugleich stehen könnte. Aus diesem Grund wird für jede mögliche Lesestelle ein Klon des einzufügenden Ausdrucks erstellt. Soll dann eine Einfügen-Transformation durchgeführt werden, können diese Klone an alle möglichen Stellen verschoben werden. So wird festgestellt, ob bzw. an welchen Stellen das Einfügen möglich ist. Ausdrücke, die zueinander einen Klon darstellen, werden über Klongruppen gruppiert. Alle Ausdrücke, die eine Referenz zur selben Klongruppe haben, sind ein Klon zueinander.

Klon: *In der Quelltextabstraktion sind zwei Kompositionen $c_1, c_2 \in Compositions$ genau dann Klone voneinander, wenn ihre Klongruppen gleich sind: $c_1.cgr = c_2.cgr$.*

Sind zwei Kompositionen c_1 und c_2 Klone voneinander, haben sie die folgenden Eigenschaften:

1. *Sie sind klonbar.*
2. *Höchstens eine OP-Kante startet von einem Knoten von c_1 bzw. c_2.*
3. *Alle Eigenschaften von c_1 und c_2 und die Eigenschaften ihrer Fragmente ($c.F$) und Knoten ($c.F.N$) sind gleich.*

4. *Alle Kanten, die einen Knoten von c_1 bzw. c_2 zum Quell- oder Zielknoten haben ($c.F.N.S$ und $c.F.N.T$), sind gleich, sofern sie nicht vom Typ FD oder OP sind.*

Die Klongruppe besitzt das Attribut *ext*. Es gibt darüber Aufschluss, ob der Originalausdruck im modifizierten Quelltext weiterhin existieren soll. Falls ein Einfügen eines Ausdrucks an allen k Lesestellen der Variablen möglich ist, müssen später genau $k-1$ Kopien des einzufügenden Ausdrucks im Quelltext erzeugt werden. Eine k-te Kopie muss nicht erzeugt werden, da der Originalausdruck im Quelltext stattdessen wiederverwendet werden kann. Wird jedoch nicht an allen Verwendungen der Ausdruck eingefügt, sondern an weniger als k Stellen, ist für jede dieser Stellen eine Kopie notwendig, da der originale Ausdruck weiterhin in der Variablenzuweisung verwendet wird.

Im Quelltext können mehrere OP-Kanten von einer Variablen aus starten (siehe Variable `i` in Quelltext 4.7). Durch das Erstellen von Klonen in der Abstraktion wird jedoch gewährleistet, dass von jeder Komposition maximal eine OP- bzw. FD-Kante startet. Dies wird in der folgenden Invariante (Gleichung 4.17) festgehalten:

Invariante für OP-Kanten: *Zwei verschiedene OP-Kanten starten nie vom selben Knoten.*

$$\forall op_1, op_2 \in Edges|^{Op} : op_1 \neq op_2 \Rightarrow op_1.src \neq op_2.src \tag{4.17}$$

4.4.3 Abstraktion expandieren

Um die Klone zu erstellen, wird zunächst die Quelltextabstraktion eines abgeschlossenen Quelltextblocks ohne Klone erstellt. Diese kann Kompositionen enthalten, von denen aus mehrere OP-Kanten starten. Die Propagierung von Kompositionen wird als erstes durchgeführt. Darauf aufbauend werden mit dem folgenden Verfahren (Quelltext 4.9) die Klone innerhalb jeder partiellen Abstraktion erstellt. Der Pseudocode nutzt die Elemente der Quelltextabstraktion. Auf Attribute wie bspw. die Menge aller startenden Kanten eines Knotens *endNode* wird mit $endNode.S$ zugegriffen.

Zunächst wird die Arbeitsliste in Quelltext 4.9 mit allen Kompositionen gefüllt. Diese Menge wird anschließend in Form eines Arbeitslistenalgorithmus' abgearbeitet, bis die Arbeitsliste leer ist. Während der Abarbeitung der Liste `C` werden immer dann Kompositionen zur Liste hinzugefügt, wenn ein Klon erstellt wurde. Hinzugefügt werden dabei immer die Vorgängerkompositionen eines Klons. Mit Vorgängern sind dabei Kompositionen gemeint, die durch eine Datenflusskante mit einem Klon verbunden sind, wobei der Datenfluss beim Vorgänger startet. Der Grund für das erneute Hinzufügen des Vorgängers ist, dass der Vorgänger selbst eventuell geklont werden muss. Im Quelltext entspricht das Klonen eines Vorgängers bspw. dem Klonen eines Unterausdrucks: Wird beim Einfügen einer Variablen ein Ausdruck an verschiedene Stellen kopiert, so müssen auch seine Unterausdrücke geklont werden, um an allen Stellen eingefügt zu werden. In Algorithmus 4.9 wird anschließend überprüft, ob eine Komposition klonbar ist. Verlassen mehrere Datenflusskanten dieselbe Komposition, kann diese Komposition geklont werden. Die genaue Bedingung dafür wird in `isForkable` beschrieben: Eine Komposition ist genau dann klonbar, wenn (1) das zugehörige Java-Element ein Ausdruck ist, (2) der zugehörige Ausdruck keinen Speicher

```
void : createClones(Blockabstraction : ba) {
  Set of Compositions : C = compositionsOf(ba);

  while (|C| > 0) {
    Composition : c = C.remove();
    Composition : copyC = null;

    do {
      copyC = forkOneEdge(c);

      if (copyC != null) {
        // add predecessing compositions
        for (Edge e : c.F.N.T) {
          C.add(e.src.fgm.cmp);
        }
      }
    } while (copyC != null);
  }
}

ComplexNode : forkOneEdge(Composition : c) {
  Node : endNode = endNode(c);

  for (Edge : e in endNode.S) {
    if (isForkable(e)) {
      // create a clone of c that has the edge e
      forkOneEdge = cloneComposition(c, e);
      return;
    }
  }
}

Boolean : isForkable(Edge : e) {
  Set of Edges : E = e.src.S;
  Boolean : isForkableSet = false;

  isForkableSet |= |E of Type OpEdge| > 1;
  isForkableSet |= |E of Type FDEdge| > 0;

  isForkable = e.typ == OpEdge;
  isForkable &= e.src.fgm.cmp.clonable;
  isForkable &= forkableSet;
}
```

Quelltext 4.9: Klonerstellung

alloziert (z. B. durch einen `Creator`-Ausdruck) und (3) sie eine OP-Kante und eine weitere OP- oder FD-Kante hat.

Beim Erstellen eines Klons ändern sich die ausgehenden Kanten sowohl vom Original als auch vom Klon einer Komposition. Das Original behält alle Kanten bis auf eine OP-Kante. Der Klon erhält diese OP-Kante und alle anderen Kanten, die weder vom Typ OP oder FD sind. Wurde ein Klon erstellt, werden weitere Klone der selben Komposition erstellt, wenn dies möglich ist. Falls dies nicht möglich ist, wird die Arbeitsliste um die Vorgänger der Komposition erweitert. Der Algorithmus endet, wenn die Arbeitsliste leer ist.

Zusammenfassung

In diesem Kapitel wurde gezeigt, wie ein Java-7-Quelltextblock auf eine Abstraktion abgebildet wird. Hierzu wurde in mehreren Schritten vorgegangen. Die initiale Abstraktion wurde ggf. durch die Wunschanpassung modifiziert, um den Umgestaltungswunsch zu ermöglichen. Anschließend wurden die Abhängigkeiten identifiziert und entsprechende Abhängigkeitskanten eingefügt. Schließlich wurden Klone erzeugt, um das Einfügen von Ausdrücken auch dann zu ermöglichen, wenn ein Ausdruck an mehreren Stellen eingefügt werden soll. Die so erzeugte Abstraktion repräsentiert alle Quelltextelemente und Abhängigkeiten, um daraus ein Constraint-Problem zu erzeugen, dass eine Umgestaltungslösung berechnen kann.

Kapitel 5

Lösungsfindung

Ein Umgestaltungsproblem besteht ursprünglich aus einem Quelltext und dem Umgestaltungswunsch des Entwicklers. Die Umgestaltungslösung ist im Idealfall ein modifizierter Quelltext, der den Umgestaltungswunsch erfüllt, aber sein Programmverhalten behalten hat. Im letzten Kapitel wurde gezeigt, wie ein Quelltext in eine Quelltextabstraktion A_3 überführt wird. Diese Quelltextabstraktion wird in diesem Kapitel auf ein Constraint-Problem abgebildet. Die Lösung des Constraint-Problems gibt an, welche Programmtransformationen notwendig sind, um den modifizierten Quelltext zu erhalten. Da die Abbildung der Quelltextabstraktion allein keine Lösung produziert, die den Umgestaltungswunsch beinhaltet, wird auch der Umgestaltungswunsch in das Constraint-Problem überführt.

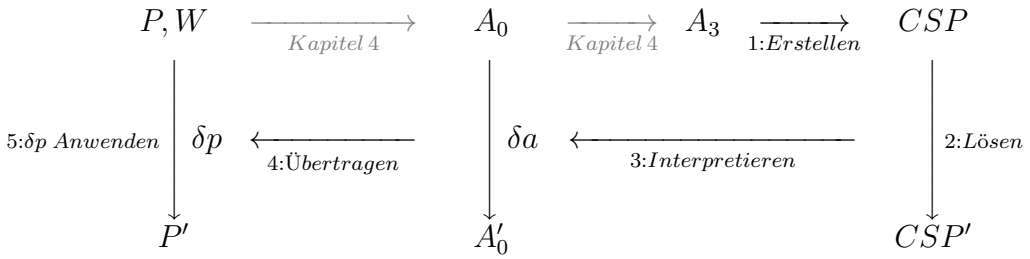


Abbildung 5.1: Aus der Abstraktion A_3 und dem Umgestaltungswunsch W werden die Transformationen δp berechnet und auf den Quelltext P angewendet.

In Abbildung 5.1 sind die insgesamt fünf Schritte dargestellt, die eine Abstraktion und einen Umgestaltungswunsch so bearbeiten, dass ein Ursprungsprogramm wie gewünscht modifiziert wird. Im ersten Schritt wird zunächst ein Constraint-Problem erzeugt (Unterkapitel 5.1), das die Freiheitsgrade und Bedingungsgleichungen enthält, um eine Menge von Transformationen zu berechnen. Das erstellte Constraint-Problem wird im zweiten Schritt gelöst. Hierfür wird ein optimiertes Lösungsverfahren in Kapitel 5.2 vorgestellt. Sofern eine Lösung des Constraint-Problems berechnet werden konnte, wird diese im dritten Schritt interpretiert, um eine Menge an Transformationen δa zu erhalten (Abschnitt 5.3.1). In Unterkapitel 5.3 werden ebenfalls die letzten beiden Schritte vorgestellt, die erläutern, wie die Menge der Transformationen auf Programmtransformationen δp abgebildet werden (Abschnitt 5.3.2) und was bei deren Anwendung auf ein gegebenes Programm beachtet werden muss (Abschnitt 5.3.3).

5.1 Constraint-Problem

Das Constraint-Problem besteht aus Constraint-Variablen, -Gleichungen und einem Optimierungskriterium. In den folgenden Abschnitten wird zunächst erklärt, wie die Variablen der Quelltextabstraktion auf Constraint-Variablen (kurz: CSP-Variablen) überführt werden. Anschließend wird darauf eingegangen, wie aus den Zusammenhängen und Abhängigkeiten einer Quelltextabstraktion Constraint-Gleichungen (kurz: CSP-Gleichungen) erzeugt werden. Ferner wird gezeigt, wie der Umgestaltungswunsch in das Constraint-Problem abgebildet wird. Damit unter den möglichen Umgestaltungslösungen die optimale im Sinne des Entwicklers ausgewählt wird, wird schließlich das Optimierungskriterium des Constraint-Problems präsentiert.

5.1.1 Variablen

Umgestaltungen des Quelltexts machen Transformationen des Quelltexts erforderlich. Im Gleichungssystem werden Transformationen durch Wertänderungen von CSP-Variablen repräsentiert. Die Wertänderungen entstehen dadurch, dass sich die Lösungswerte des Gleichungssystems von den initialen Werten, basierend auf dem Ursprungsprogramm, unterscheiden. Tabelle 5.1 zeigt alle acht Attribute, die als CSP-Variablen verwendet werden. Dabei wird neben dem Attribut das zugehörige Element und der Wertebereich des Attributs dargestellt.

Die Variable *op* gibt die Operandenposition eines Knotens an, sofern dieser einen Knoten eines Operandenfragments modelliert. Modelliert der Knoten keinen Knoten eines Operandenfragments, wird der Wert von *op* nicht weiter interpretiert. Die Variable *np* spezifiziert die eindeutige Position eines Knotens innerhalb der partiellen Abstraktion eines Blocks. Aus dieser Position wird die Position der zugehörigen Komposition ermittelt und in *cp* gespeichert. Existiert zu einer Komposition eine Klongruppe, definiert die Variable *cgp* statt *cp* der zugehörigen Kompositionen die Position im Block. Die Variable *inl* einer Komposition hat den Wert *wahr* wenn der zugehörige Ausdruck eingebettet/eingefügt werden soll. Soll der Ausdruck extrahiert werden bzw. bleiben, hat sie den Wert $falsch$. Die Klongruppe einer Menge von geklonten Kompositionen besitzt die Variable asx. Die Kompositionsklone können möglicherweise an unterschiedliche Stellen in der Abstraktion verschoben werden, sodass alle Klone in übergeordneten Kompositionen eingefügt sind. In einem solchen Fall wird für jede eingefügte Komposition eine Kopie des einzufügenden Ausdrucks im Quelltext erzeugt. Außerdem wird der Originalausdruck gelöscht. Die Variable asx speichert dann, ob der Originalausdruck gelöscht werden soll. Wenn Ausdrücke im Quelltext eingefügt werden, entfällt unter Umständen die Notwendigkeit, die zugehörige Variable zu deklarieren, da sie nun nicht mehr verwendet wird. Daher besitzen Symbole in der Abstraktion die Variable exs. Ihr Wert ist genau dann $wahr$, wenn die Deklaration weiterhin im Quelltext verbleiben soll.

5.1.2 Notation

Die Gleichungen eines Constraint-Problems ergeben sich aus den Zusammenhängen und Abhängigkeiten, die in der Quelltextabstraktion modelliert sind. Weitere Gleichungen entstehen aus dem Umgestaltungswunsch des Entwicklers. Ferner wird auch das Optimie-

Klasse	CSP-Variable	Wertebereich	Beschreibung
Komposition	inl	$\{true, false\}$	Ausdruck eingebettet
Komposition	cp	$\{1..\|N\|\}$	Position im Block
Knoten	np	$\{1..\|N\|\}$	Position im Block
Knoten	op	$\{1..\|n.fgm.N\|\}$	Operandenposition
Symbol	exs	$\{true, false\}$	Deklaration notwendig
Klongruppe	asx	$\{true, false\}$	Originalausdruck existiert
Klongruppe	cgp	$\{1..\|N\|\}$	Position im Block

Tabelle 5.1: CSP-Variablen

rungskriterium mithilfe von Gleichungen angegeben. All diese Gleichungen werden in den folgenden Abschnitten erläutert. Die dazu notwendige Notation wird in diesem Abschnitt eingeführt.

Die Constraint-Gleichungen einer Abstraktion werden mithilfe von *Generatorregeln* beschrieben. Zum Definieren der Generatorregeln wird eine Notation genutzt, die sich an eine Notation von Steimann et al. [110] anlehnt und aus drei wesentlichen Teilen besteht: Allquantoren, Filter/Selektor und Constraint-Gleichung.

> **Generatorregel:** *Eine Generatorregel, bestehend aus Allquantor $\forall$, Filter/Selektor FS und Constraint-Gleichung G wird wie folgt notiert. Der Filter/Selektor ist optional.*
>
> $$\forall, FS : G$$

Die *Allquantoren* dienen der Auswahl von Abstraktionselementen. Durch zusätzliche, optionale Prädikate im *Filter/Selektor*-Teil, der mit Komma hinzugefügt wird, kann die Auswahl des Allquantors eingeschränkt werden. Hierbei beziehen sich die Prädikate nur auf Attribute, die keine CSP-Variablen sind. Desweiteren können über Prädikate weitere Elemente/Attribute spezifiziert werden, die in der Constraint-Gleichung verwendet werden. Der Filter wird wie ein Prolog-Term [36, 21, 114] auf alle Elemente des Allquantors einer konkreten Abstraktion angewendet. Die *Constraint-Gleichung* wird schließlich durch einen Doppelpunkt vom vorherigen Teil getrennt. Die Constraint-Gleichung besteht nur aus Prädikaten, die sich letztlich auf Constraint-Variablen beziehen. Diese Prädikate werden in eine Constraint-Gleichung [52] und den zugehörigen Constraint-Variablen umgewandelt. Während Allquantor und Filter/Selektor bestimmte Elemente und/oder Attribute herausfiltern oder -suchen, werden im Constraint-Gleichungsteil CSP-Variablen miteinander in Beziehung gesetzt. Auf diese Art generiert jede Generatorregel für eine gegebene konkrete Abstraktion eine Menge von Constraint-Gleichungen. Die Gesamtheit aller Generatorregeln wird *Regelsatz* genannt.

Transformationsprogrammierung

In den Arbeiten der Transformationsprogrammierung [86, 10] werden Transformationsregeln genutzt, um die Semantik von äquivalenten Programmumformungen zu spezifizieren. Diese Regeln bestehen aus drei Teilen: einer Bedingung sowie einem abstrakten Vor-

und Nachschema des Programmtexts. Sofern die Bedingung zutrifft, kann der Teil des Programmtexts, der durch das Vorschema spezifiziert wird, durch das Nachschema ersetzt werden. Diese Transformationsregeln können reversibel sein, sodass sie auch rückwärts angewendet werden können. Die folgende Transformationsregel 5.1 aus [88] beschreibt, wann eine Alternative, deren seiteneffektfreie Bedingung immer wahr ist, durch ihren Dann-Ausdruck A ersetzt werden kann.

$$\frac{if\ E\ then\ A\ else\ B}{A} \Big[\!\Big[\ Bool'scher\ Ausdruck\ E\ ist\ eine\ Tautologie \tag{5.1}$$

Im Vergleich zum Ansatz der vorliegenden Arbeit verzichten Transformationsregeln auf einen expliziten Allquantor. Der Filter/Selektor-Teil steht im Vorschema und die Constraint-Gleichung in der Bedingung. Der größte Unterschied beider Notationen ergibt sich daraus, dass eine Transformationsregel eine Strukturelle Änderung beschreibt: Im obigen Beispiel verschwindet die Alternativanweisung komplett durch Anwendung der Regel. Im Gegensatz dazu beschreibt der Constraint-Teil von Generatorregeln, welchen Wert welche Variablen besitzen. Sie werden nicht sofort und nicht einzeln angewendet. Wenn die Werte letztendlich ausgewertet werden, können Anweisungen im Anschluss natürlich auch gelöscht werden. Aber während die Constraint-Teile der Generatorregeln im Constraint-Lösungsverfahren genutzt werden, ändert sich die Struktur des Problems – also die Struktur des Programms – nicht.

Refacola

In der Arbeit von Steimann, Thies et al. [113, 118] werden die Generatorregeln Constraint-Regeln genannt und in der Prämisse-/Konklusionsschreibweise notiert. Jede Regeln unterteilt sich dadurch ebenfalls in die Teile Prämisse und Konklusion. Die Prämisse stellt eine Vorbedingung für die Regel dar, die sehr ähnlich zum hier vorgestellten Filter/Selektor-Teil ist. Die Konklusion repräsentiert die Constraint-Gleichung, die erzeugt wird, wenn die Prämisse gültig war. Die Konklusion kann in beiden Notationen jedoch auch Bedingungen in Form von Implikationen enthalten. In der Prämisse-/Konklusionsschreibweise wird auf einen Allquantor verzichtet. Stattdessen wird angegeben, dass jede Regel implizit über alle Elemente der Abstraktion quantifiziert wird [110]. Die folgende Constraint-Regel 5.2 aus [110] definiert, dass eine Deklaration d und eine Referenz r, die eine Bindung zueinander besitzen, den gleichen Namen (id) und Typen (t) haben. Wenn in einem konkreten Programm insgesamt zwei solche Paare (r_1, d_1) und (r_2, d_2) gefunden werden, erzeugt diese Constraint-Regel vier konjunktive Bedingungsgleichungen: $r_1.id = d_1.id$, $r_1.t = d_1.t$, $r_2.id = d_2.id$ und $r_1.t = d_1.t$.

$$\frac{binds(r, d)}{r.id = d.id \;\; r.t = r.t} \tag{5.2}$$

Allerdings wird in der von Steimann, von Pilgrim et al. eingeführten und zur Spezifikation ihres Ansatzes genutzten Sprache Refacola jedoch ein Allquantor explizit angegeben [110]. Diese zweite Form ihrer Notation besteht neben dem Allquantor aus einem `if`- und einem `then`-Teil, wobei ersterer die Prämisse und letzterer die Konklusion umsetzen.

```
for all r : Reference, d : DeclaredEntity
if binds(r, d)
```

```
then r.id = d.id and r.t = d.t
```

Quelltext 5.1: Constraint-Regel 5.2 in der Refacola-Notation.

In dieser Arbeit wird eine dreiteilige Notation gewählt, deren Namen Allquantor, Filter/Selektor und Constraint-Gleichung sind. Sie lehnt sich an die dreiteilige Notation von Steimann et al. an und wird aus den folgenden Gründen genutzt: Erstens sind die Generatorregeln sehr umfangreich und werden in der Prämisse-/Konklusionsschreibweise schnell unübersichtlich. Zweitens erleichtert die zusätzliche Unterteilung der Prämisse in Allquantor und Filter/Selektor die Verständlichkeit der Regeln. Denn häufig müssen spezielle Elemente der Abstraktion im Filter/Selektor-Teil zunächst gefunden oder aussortiert werden, damit dann auf Constraint-Variablen der gewünschten Elemente im Constraint-Gleichungsteil zugegriffen werden kann. Ferner könnte der Begriff Konklusion im Sinne einer Schlussfolgerung für den Constraint-Lösungsprozess verstanden werden, obwohl das in der Konklusion spezifizierte Prädikat lediglich als Constraint-Gleichung erzeugt werden soll. Erst in einem nächsten Schritt würde es mit allen anderen, so erzeugten Constraint-Gleichungen gelöst werden. Steimanns Constraint-Regeln werden ferner hier *Generatorregeln* genannt, da sie Constraint-Gleichungen erzeugen.

5.1.3 Generatorregeln

Die folgenden Generatorregeln werden auf alle Elemente einer expandierten Quelltextabstraktion angewendet. Dabei werden die Mengen der Kompositionen, Fragmente, Knoten, Klongruppen, Kanten und Symbole wieder mit $Compositions$, $Fragments$, $Nodes$, $CloneGroups$, $Edges$ und $Symbols$ bezeichnet.

IS-Kanten von Operandenfragmenten

Operandenfragmente bspw. vom Typ *CommutativeOperands* besitzen mehr als einen Knoten. Die Abbildungsregeln von Quelltext auf die Abstraktion sehen vor, dass die Knoten von Operandenfragmenten mit IS-Kanten verbunden sind (siehe Abschnitt 4.2.5). Der Grund dafür ist, dass kein Knoten einer anderen Komposition mit den Knoten eines Fragments interferieren darf. Eine Vermischung von Knoten verschiedener Kompositionen könnte nicht auf den Quelltext zurück abgebildet werden. Dennoch ermöglichen zwei Transformationen, dass die IS-Kanten innerhalb von Operandenfragmenten ihren Quell- und/oder Zielknoten verändern. Dies geschieht durch die Transformationen *Operanden umsortieren* (siehe Abschnitt 3.6) oder das *Einfügen/Extrahieren von Kompositionen* (siehe Abschnitt 3.6).

Im Folgenden werden die Abhängigkeitskanten in Form von Gleichungen in das Constraint-Problem überführt. Dabei wird für die Quell- und Zielknoten einer Kante jeweils eine Bedingungsgleichung erstellt. Diese Gleichungen sind jedoch während der Lösungsfindung nicht modifizierbar. Da sich die Quell- und Zielknoten der IS-Kanten innerhalb von Operandenfragmenten jedoch aufgrund der genannten Transformationen ändern können, müssen diese IS-Kanten besonders behandelt werden.

IS-Kanten innerhalb von Operandenfragmenten werden im Gegensatz zu allen anderen Kanten nicht in den Generatorregeln beachtet. Bei der Erzeugung des Constraint-Problems werden sie über mehrere Generatorregeln hinzugefügt, die das Verhalten der Kanten replizieren. Daher sind solche IS-Kanten im Folgenden nicht Teil mehr der Abstraktion und

insbesondere nicht mehr Element der Menge *Edges*. Mithilfe dieser Generatorregeln (siehe folgende Abschnitte IS-Kanten in Operandfragmenten und Veränderliche IS-Kanten) wird ermöglicht, dass die IS-Kanten ihre Quell- und Zielknoten aneinander binden und ferner, dass IS-Kanten ihre Quell- und Zielknoten entsprechend der genannten Transformationen ändern können. Hierdurch werden die Freiheitsgrade zum Vertauschen von Operanden sowie zum Extrahieren/Einfügen von Ausdrücken ermöglicht, die ohne das Ersetzen der IS-Kanten durch Generatorregeln nicht hätten genutzt werden können.

Eindeutigkeit

Die erste und eine der einfachsten Generatorregeln erzeugt Constraint-Gleichungen, die besagen, dass jeder Knoten eine eindeutige Position besitzt. Hierzu werden alle Knoten der Abstraktion ausgewertet. Sofern zwei nicht identische Knoten ausgewählt werden, gilt für sie, dass ihre Positionswerte nicht gleich sein dürfen. Für jedes Paar ungleicher Knoten wird eine solche Constraint-Gleichung generiert (Generatorregel 5.3).[1] Analog wird für alle Operanden eines Operandenfragments festgelegt, dass ihre Operandenpositionen von einander unterschiedlich sein müssen. (Generatorregel 5.4).

$$\forall n_1, n_2 \in Nodes, n_1 \neq n_2 : n_1.np \neq n_2.np \tag{5.3}$$

$$\forall f \in Fragments|^{\neg Node}\ \forall n_1, n_2 \in f.N, n_1 \neq n_2 : n_1.op \neq n_2.op \tag{5.4}$$

Kanten

Aus jeder Kante der Abstraktion wird eine Constraint-Gleichung erzeugt. Sie bewirkt, dass der Quellknoten jeder Kante vor ihrem Zielknoten liegt, wenn sie nicht loop-carried (Abschnitt 2.3.1) ist. Ist sie loop-carried, liegt umgekehrt der Zielknoten vor dem Quellknoten.

$$\forall e \in Edges|^{\neg.lc} : e.src.np \leq e.tgt.np \tag{5.5}$$

$$\forall e \in Edges|^{.lc} : e.tgt.np \leq e.src.np \tag{5.6}$$

IS-Kanten

IS-Kanten bewirken eine strengere CSP-Gleichung, da der Quellknoten von IS-Kanten direkt vor ihrem Zielknoten liegen muss. Daher darf kein anderer Knoten zwischen Quell- und Zielknoten gelegt werden. Da IS-Kanten, die innerhalb von Operandenfragmenten liegen, kein Teil der Abstraktion sind, wirkt sich die folgende Generatorregel sich nur auf IS-Kanten aus, die Knoten verschiedener Fragmente verbinden.

$$\forall e_{IS} \in Edges|^{IS} : e_{IS}.src.np + 1 = e_{IS}.tgt.np \tag{5.7}$$

[1] Im Prototypen CodeShaker wird hierzu die `allDifferent`-Bedingung von `choco` verwendet, um die Gleichungsanzahl zu verringern.

IS-Kanten in Operandfragmenten

Die in der Quelltextabstraktion modellierten IS-Kanten zwischen Knoten von Operandfragmenten werden beim Erzeugen der Constraint-Gleichungen ignoriert (siehe Abschnitt IS-Kanten von Operandenfragmenten in diesem Unterkapitel). Nichtsdestotrotz müssen diese Knoten mithilfe von CSP-Gleichungen aneinander gekettet werden, damit keine anderen Knoten eine Position zwischen ihnen einnehmen. Daher erzeugt die folgende Generatorregel Constraint-Gleichungen, die wie die Gleichungen von IS-Kanten wirken. Die Generatorregel orientiert sich dabei an der Reihenfolge der Knoten eines Operandfragments, die durch die Operandenposition *op* gegeben ist. Je nachdem, in welcher Reihenfolge die Knoten geordnet sind, werden Gleichungen erzeugt, deren Wirkung analog zu den Gleichungen der Generatorregel *IS-Kanten* ist. Dadurch können die IS-Kanten dieser Generatorregel als veränderlich aufgefasst werden und ihre Quell- und Zielknoten verändern. Die folgende Gleichung 5.8 repräsentiert IS-Kanten, die zwei Kompositionen verbinden. Im Beispiel in den Abbildungen 5.2 werden die durch Generatorregel 5.8 erzeugten IS-Kanten grau dargestellt und mit e_2 und e_3 markiert.

$$\begin{aligned} \forall f \in Fragment|^{\neg Node}\ \forall n_1, n_2 \in f.N, \\ n_1 \neq n_2 \wedge \underset{Op}{\twoheadrightarrow} n_2 = \emptyset : \\ n_1.op + 1 = n_2.op \Rightarrow n_1.np + 1 = n_2.np \end{aligned} \tag{5.8}$$

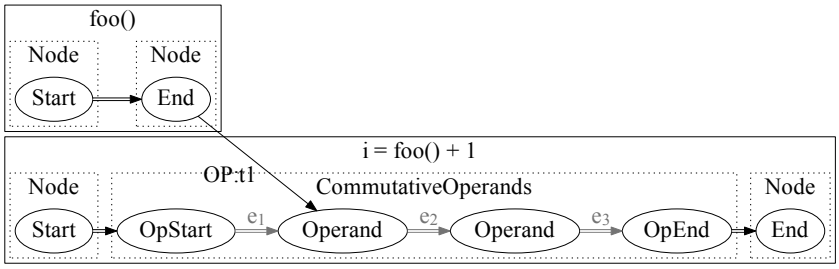
Veränderliche IS-Kanten

Die vorherige Generatorregel erzeugt eine Constraint-Gleichung einer IS-Kante, die aktiv wird, wenn keine OP-Kanten auf den Startknoten zielen. Existiert allerdings eine OP-Kante, so ist ein Einfügen oder Extrahieren möglich und eine Fallunterscheidung muss durchgeführt werden. Der Hintergrund der Fallunterscheidung ist der Freiheitsgrad des Einfügens/Extrahierens. Für diese Fallunterscheidungen werden die folgenden beiden Generatorregeln 5.9 und 5.10 genutzt.

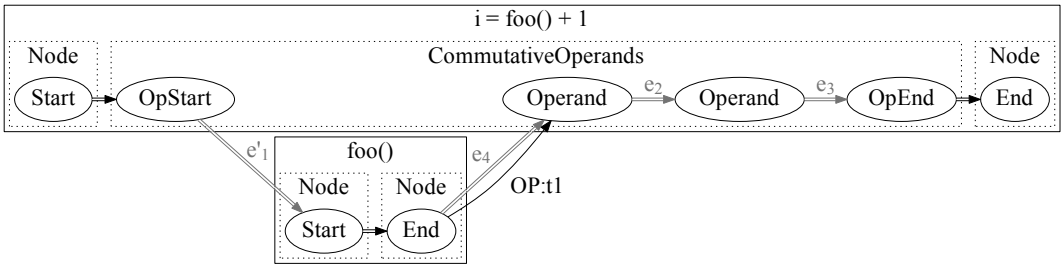
Das Quelltextbeispiel 5.2 zeigt eine Addition, die aus zwei Operanden besteht, wobei der Methodenaufruf extrahiert werden kann. In den Abbildungen 5.2 wird gezeigt, welche IS-Kanten in Abhängigkeit der *inl*-Variable der `foo()`-Komposition aktiv sind. Wenn wie in Abbildung 5.2a die Komposition `foo()` extrahiert ist, ist die IS-Kante e_1 aktiv, die die Knoten des Operandfragments direkt verbindet. Wenn die Komposition jedoch eingefügt werden soll, ist die IS-Kante e_1 deaktiviert. Stattdessen existieren die IS-Kanten e_1' und e_4. Die Kante e_4 wird von der Generatorregel 5.10 und die Kanten e_1 bzw. e_1' werden von der Generatorregel 5.9 erzeugt. Je nachdem, ob die Komposition von `foo()` eingefügt bzw. extrahiert werden soll, bewirken die jeweils aktiven Kanten einen engen Zusammenhalt der Knoten beider bzw. jeder einzelnen Komposition.

```
abstract int foo();
void nesting(int i) {
  i = foo() + 1;
}
```

Quelltext 5.2: Quelltext mit verschachteltem Additionsausdruck



(a) Der Unterausdruck ist extrahiert und die Variable *inl* hat den Wert *falsch*.



(b) Der Unterausdruck ist eingefügt und die Variable *inl* hat den Wert *wahr*.

Abbildung 5.2: Repräsentation der Extrahieren/Einfügen-Transformation der Komposition von `foo()` im Constraint-Problem. Grau dargestellt sind die IS-Kantengleichungen, die durch die Generatorregeln 5.8, 5.9 und 5.10 erzeugt werden.

In der folgenden Generatorregel 5.9 werden die Constraint-Gleichungen zweier IS-Kanten erzeugt, von denen immer genau eine aktiv ist. Wenn ein Unterausdruck eingefügt wird, ist der übergeordnete Ausdruck durch eine IS-Kante mit seinem Unterausdruck verbunden. Dabei startet die IS-Kante an einem Knoten des Operandenfragments des Überausdrucks und endet beim Startknoten des Unterausdrucks. Wenn der Unterausdruck herausgelöst wird, zeigt die IS-Kante nicht mehr auf den Unterausdruck. Stattdessen zeigt sie[2] auf den nächsten Knoten des Operandenfragments im Sinne der Operandenordnung. Die Generatorregel 5.9 gibt dieses Verhalten an.

$$
\begin{aligned}
\forall f \in Fragments|^{\neg Node} \;\; \forall n_1, n_2 \in f.N \;\; \forall e_{Op} \in \underset{Op}{\twoheadrightarrow} n_2 \\
\forall c \in Compositions \;\; \forall n_S \in c.F.N, \\
n_1 \neq n_2 \wedge c = e_{Op}.src.fgm.cmp \wedge n_S|^{Start} : \\
n_1.op + 1 = n_2.op \Rightarrow (\neg c.inl \wedge n_1.np + 1 = n_2.np) \\
\vee (c.inl \wedge n_1.np + 1 = n_S.np)
\end{aligned}
\tag{5.9}
$$

[2] Versteckt ganz tief im Werke drin, wo's immer ruhig ist und still, sitzt die Banane gelb und dünn, die auch gesehen werden will.

Veränderliche OP-Kanten

Wenn ein Unterausdruck eingefügt wird, wird entlang seiner OP-Kante eine weitere IS-Kante im Constraint-Problem erzeugt. Dadurch wird der Unterausdruck mit seinem übergeordnetem Ausdruck verbunden. Im Beispiel 5.2 mit den Abbildungen 5.2 wird die Kante e_4 genau dann erzeugt, wenn die Komposition des Methodenaufrufs `foo()` eingefügt werden soll. Andernfalls existiert die Kante e_4 nicht. Die folgende Gleichung gibt an, wie sich die parallel zur OP-Kante verlaufende IS-Kante verhält, je nachdem, ob der Unterausdruck eingefügt werden soll oder nicht.

$$\begin{aligned} &\forall e_{op} \in Edges|^{Op} : \\ e_{Op}.src.fgm.cmp.inl \Rightarrow e_{Op}.src.np + 1 &= e_{Op}.tgt.np \end{aligned} \tag{5.10}$$

Veränderliche Deklarationen

Deklarationen existieren nur dann, wenn die deklarierte Variable lokal ist und verwendet wird. Sie existieren außerdem, wenn die deklarierte Variable bereits vor der Umgestaltung existierte und nicht verwendet wurde (Gl. 5.11). Dies soll verhindern, dass alle ungenutzten Variablen, die der Entwickler jedoch explizit deklariert hat, von der Umgestaltung gelöscht werden. Variablen werden immer verwendet, wenn eine FD-Kante existiert, die das Symbol der Variablen besitzt. Da FD-Kanten einen expliziten Datenfluss über eine Variable repräsentieren (Gl. 5.12), darf die Deklaration der Variablen nicht entfernt werden. OP-Kanten können neben expliziten allerdings auch implizite Datenflüsse zwischen verschachtelten Ausdrücken repräsentieren. Eine Variable wird immer dann bei OP-Kanten genutzt, wenn sie anstelle eines Unterausdrucks eingefügt wird (Gl. 5.13). Diese Sachverhalte werden von den folgenden drei Gleichungen ausgedrückt.

$$\forall s \in Symbols|^{Local}, \overset{s}{\twoheadrightarrow} = \emptyset : s.exs \tag{5.11}$$

$$\forall s \in Symbols|^{Local}, \overset{s}{\underset{FD}{\twoheadrightarrow}} \neq \emptyset : s.exs \tag{5.12}$$

$$\begin{aligned} &\forall s \in Symbols|^{Local}, \overset{s}{\underset{FD}{\twoheadrightarrow}} = \emptyset : \\ (\overset{s}{\underset{Op}{\twoheadrightarrow}} |^{\neg.src.fgm.cmp.inl} \neq \emptyset) &= s.exs \end{aligned} \tag{5.13}$$

Positionen

Die folgende Gleichung 5.14 ist eine Vereinfachung, um bequemer auf die Position einer Komposition zugreifen zu können. Die Positionen der Komposition bzw. Klongruppe werden genutzt, um Anweisungen verschieben zu können und die neuen Positionen von Anweisungen zu ermitteln. Wie angegeben, ergibt sich die Position einer Komposition aus der Position ihres Startknotens. Sofern die Komposition Teil einer Klongruppe ist, beeinflusst die Position der Komposition auch die der Klongruppe. Gleichung 5.15 definiert diesbezüglich, dass sich die Position der Zuweisung *cgp* aus der Menge aller zugehörigen Kompositionen

ergibt. Dabei ist die Zuweisungsposition die kleinste, d. h. dass die Zuweisung immer vor allen anderen Klonen im Quelltext steht.

$$\forall c \in Compositions \;\; \forall n_S \in c.F.N, n_S|^{Start} : c.cp = n_S.np \tag{5.14}$$

$$\forall g \in CloneGroups : g.cgp = min(\{c.cp | c \in g.C\}) \tag{5.15}$$

Originalausdruck

Sofern Klone einer Komposition existieren, existiert auch eine zugehörige Klongruppe. Die Kompositionen der Klongruppe repräsentieren alle denselben Quelltextausdruck. Falls eine der Kompositionen, aber nicht alle, eingefügt werden, muss für jede Einfügestelle eine Kopie des Quelltextausdrucks erzeugt werden. Der originale Quelltextausdruck bleibt jedoch bestehen, da er auf der rechten Seite einer Zuweisung steht, die noch verwendet wird. Erst wenn alle Kompositionen einer Klongruppe eingefügt werden, besteht keine Notwendigkeit mehr für den originalen Quelltextausdruck sowie für die Zuweisung, auf deren rechter Seite er steht. Folglich können die Zuweisung und der originale Quelltextausdruck gelöscht werden, sofern alle Kompositionen einer Klongruppe eingefügt werden. Gleichung 5.16 stellt sicher, dass die Zuweisung und der Originalausdruck in einem solchen Fall gelöscht werden. Nur wenn der Ausdruck an irgendeiner Stelle extrahiert ist, d. h. die Variable verwendet wird, hat das Attribut *asx* den Wert *wahr*.

$$\forall g \in CloneGroups : (|g.C|^{\neg.inl}| = 0) \neq g.asx \tag{5.16}$$

Transformierbarkeit

Kompositionen geben an, ob der durch sie repräsentierte Quelltext ein Ausdruck ist, der an bestimmter Stelle als Unterausdruck eingefügt bzw. herausgelöst werden kann. Hierzu wird das Attribut *trn* genutzt. Ist dieses Attribut *wahr*, so kann die Komposition als Unterausdruck eingefügt bzw. herausgelöst und der Wert ihres Attributs *inl* modifiziert werden. Wenn der Wert von *inl* nicht modifiziert werden kann, wird er auf seinen Initialwert inl_{INIT} festgelegt. Dieser Sachverhalt wird durch die folgende Gleichung 5.17 ausgedrückt.

$$\forall c \in Compositions, \neg c.trn : (c.inl = c.inl_{INIT}) \tag{5.17}$$

5.1.4 Gleichungen des Änderungswunsches

Die Änderungswünsche ergeben sich aus den Umgestaltungen (Kapitel 1.4) und müssen in das Gleichungssystem abgebildet werden. Das Gleichungssystem, das durch die Quelltextabstraktion erstellt wird, beschreibt alle relevanten Abhängigkeiten. Es berücksichtigt außerdem die Eigenheiten der Quellsprache, sodass jede Lösung des Gleichungssystems wieder in ein Programm der Quellsprache rücktransformiert werden kann. Jedoch beinhaltet die Lösungsmenge solcher Gleichungssysteme immer noch die Triviallösung: Die Werte der Triviallösungen unterscheiden sich nicht von den ursprünglichen Werten. D. h. dass durch die Triviallösung keine Änderungen am ursprünglichen Quelltext durchgeführt würde.

Damit dennoch die gewünschte Änderung der Umgestaltung am Quelltext durchgeführt wird, müssen zusätzliche Gleichungen erstellt werden. Diese zusätzlichen Gleichungen leiten sich aus dem Umgestaltungswunsch des Nutzers ab. Sie beschränken den Lösungsraum und bewirken so, dass die Triviallösung nicht mehr Teil der Lösungen des Gleichungssystems ist. Durch sie wird also sichergestellt, dass eine Lösung des Gleichungssystems den Umgestaltungswunsch umsetzt, oder aber, dass keine Lösung existiert. Die Gleichungen unterscheiden sich in ihrem Grundprinzip je nachdem, welchen Umgestaltungswunsch sie umsetzen sollen. Während einige Umgestaltungen nur dann sinnvoll sind, wenn sie ganz und gar umgesetzt werden können (z. B. Neuordnen von Parametern/Argumenten), können andere Umgestaltungen auch dann sinnvoll sein, wenn sie nur teilweise umgesetzt werden können (z. B. Einfügen von Ausdrücken). In Situationen, in denen auch eine teilweise Umsetzung des Umgestaltungswunsches für den Anwender sinnvoll ist, wird keine Gleichung erstellt, die die komplette Umsetzung des Umgestaltungswunsch erzwingt. Dies betrifft die Einfügen-Umgestaltung, bei der das Einfügen eines Ausdrucks auch dann sinnvoll ist, wenn aufgrund von Abhängigkeiten der Ausdruck nicht an allen möglichen Stellen eingefügt werden kann. Würde die komplette Umsetzung des Einfügen-Wunsches erzwungen, müsste der komplette Wunsch abgelehnt werden, sofern einer der Ausdrücke nicht eingefügt werden kann. Stattdessen wird das Optimierungskriterium so verändert, damit es eine größtmögliche Umsetzung des Wunsches bewirkt. Dadurch kann eine Constraint-Lösung berechnet werden, die den Ausdruck überall dort einfügt, wo es möglich ist.

Grundsätzlich orientiert sich die Formulierung der Umgestaltungswünsche an der Repräsentation des Quelltexts als Gleichungssystem, d. h. dass die Wünsche auf die Gleichungsvariablen abgebildet werden. Je nachdem wie konkret der Umgestaltungswunsch spezifiziert wird, werden die Wertebereiche der Variablen mehr oder minder eingeschränkt. Wenn Umgestaltungswünsche sehr konkret festgelegt werden, werden bspw. exakte Position von Operanden angegeben. Werden die Wünsche weniger strikt spezifiziert, wie beim Verschieben von Ausdrücken oder Anweisungen, werden stattdessen nur Größer/Kleiner-Relationen genutzt, um die Positionswerte einzuschränken. Im Folgenden wird aufgeführt, welche zusätzlichen Gleichungen durch die verschiedenen Umgestaltungswünsche erzeugt werden.

In bestimmten Situationen muss die Quelltextabstraktion modifiziert werden, um die gewünschte Umgestaltung zu ermöglichen. Diese Wunschanpassung wird beim Erzeugen der Quelltextabstraktion durchgeführt (siehe Einleitung von Kapitel 4) und kann beim Einfügen/Extrahieren von Ausdrücken notwendig werden. Alle anderen Umgestaltungen erfordern keine Wunschanpassung.

Wunschanpassungen zum Einfügen/Extrahieren von Ausdrücken

Wunschanpassungen müssen in zwei Situationen vorgenommen werden. Die erste Situation tritt beim Extrahieren eines Ausdrucks auf. Ein Ausdruck, der ein `Literal`, `Name` oder eine `Variablenreferenz` ist, wird von der Abstraktion nicht als eigene Komposition abgebildet. Folglich kann ein Literal bspw. nicht aus dem Ausdruck `f(1);` extrahiert werden. Falls der Nutzer aber dennoch wünscht, ein Literal zu extrahieren, muss die Abstraktion dementsprechend angepasst werden. Ebenfalls kann die rechte Seite einer Zuweisung nicht extrahiert werden, da sie aufgrund einer Optimierung in den Zuweisungsausdruck integriert ist (siehe Abschnitt 4.2.5). In solchen Fällen kann durch die Transformation *Transitivität*

Abhilfe geschaffen werden. Wenn bspw. der Umgestaltungswunsch, das Literal `42` aus der Anweisung `a = 42;` zu extrahieren, betrachtet wird, wird die Zuweisung als einzelne Komposition in der Abstraktion dargestellt. Wird nun die Transformation *Transitivität* (siehe Abschnitt 3.6) angewendet, wird eine zusätzliche Komposition erzeugt. Auf diese Weise ist ein Extrahieren der `42` möglich und es wird eine neue Zwischenvariable eingefügt. Analog verhält sich die Vorgehensweise, wenn statt der `42` bspw. eine Addition auf der rechten Seite der Zuweisung extrahiert werden soll.

Die zweite Situation, in der Wunschanpassungen vorgenommen werden müssen, tritt beim Einfügen von Ausdrücken auf. Soll der Initialisierungsausdruck einer Deklaration eingefügt werden, muss die Abstraktion transformiert werden. Der Grund dafür ist, dass die Initialisierung ein fester Bestandteil der Deklarationskomposition ist und eine Einfügen-Transformation nicht möglich ist. Ist der Umgestaltungswunsch, dass die Initialisierung der Deklaration `int t = 47;` überall dort eingefügt werden soll, wo `t` gelesen wird, führt dies zu einem Problem, da der Wunsch in der Abstraktion so nicht unterstützt wird. Daher wird die Abstraktion mithilfe der Transformation *Deklaration umwandeln 1* (siehe Abschnitt 3.6) modifiziert. Diese Transformation spaltet die Deklarationsanweisung so in zwei Anweisungen auf, dass die Initialisierung entfernt und in eine neue Zuweisungsanweisung übernommen wird. Diese Zuweisungsanweisung wird dann genutzt, um den Umgestaltungswunsch umzusetzen. Falls eine Deklaration mehrere Variablen deklariert und der Initialisierungsausdruck von einer der deklarierten Variablen eingefügt werden soll, wird die Transformation *Deklaration umwandeln 2* (siehe Abschnitt 3.6) angewendet. Bspw. könnte der Umgestaltungswunsch sein, den Initialisierungsausdruck der Variablen `b` der Deklaration `int a, b = 1` an all seinen Verwendungsstellen einzufügen. Die Transformation *Deklaration umwandeln 2* spaltet eine Deklaration so in mehrere Deklarationen auf, dass für jede deklarierte Variable eine eigene Deklarationsanweisung erzeugt wird. Anschließend wird die Transformation *Deklaration umwandeln 1* genutzt, um wie zuvor den Initialisierungsausdruck in eine eigene Zuweisungsanweisung zu überführen. Auf der durch diese Wunschanpassungen vorgenommenen Abstraktion wird schließlich die Lösungssuche durchgeführt.

Ausdrücke einfügen/extrahieren

Soll ein Ausdruck aus einer Anweisung oder einem übergeordneten Ausdruck extrahiert werden, wird für den zu extrahierenden Ausdruck eine Gleichung erstellt. Diese Gleichung legt für die Komposition des Ausdrucks fest, dass das Attribut *inl* den Wert *falsch* haben muss. Beim Einfügen einer Variablenzuweisung, deren Variable nur ein einziges Mal verwendet wird, wird der Ausdruck auf der rechten Seite betrachtet. Dieser soll an die Lesestelle statt der Variablen eingesetzt werden. Dazu wird das *inl*-Attribut der zugehörigen Komposition in einer Gleichung auf *wahr* gesetzt.

> **Ausdruck extrahieren:** *Ist e ein Ausdruck, der extrahiert werden soll und ist C die Menge aller Kompositionen, die e in der Quelltextabstraktion repräsentieren, wird die folgende Generatorregel genutzt, um den Umgestaltungswunsch zu erzeugen.*
>
> $$\forall c \in C : c.inl = false$$

Bei einzufügenden Variablenzuweisungen, deren Variable an mehreren Stellen im Programm verwendet wird, muss anders verfahren werden. Denn würde für jede der einzufügenden Variablenstellen eine Gleichung erstellt, könnte der Constraint-Löser nur dann eine Lösung berechnen, wenn der Ausdruck auch tatsächlich an allen Stellen eingefügt werden könnte. Jedoch wünscht sich der Nutzer vielleicht auch eine Lösung, bei der der Ausdruck nur an allen tatsächlich möglichen Stellen eingefügt wird. Aus diesem Grund ändert die Einfügen-Umgestaltung bei mehreren Einfügestellen das Optimierungskriterium, statt Bedingungsgleichungen zu nutzen.

Ausdruck einfügen 1: *Ist e ein Ausdruck, der eingefügt werden soll, C die Menge aller Kompositionen, die e in der Quelltextabstraktion repräsentieren und enthält C nur ein einziges Element, wird die folgende Generatorregel genutzt, um den Umgestaltungswunsch zu erzeugen.*

$$\forall c \in C : c.inl = true$$

Ausdruck einfügen 2: *Ist e ein Ausdruck, der eingefügt werden soll, C die Menge aller Kompositionen, die e in der Quelltextabstraktion repräsentieren und enthält C mehr als ein Element, wird die folgende Kostengleichung genutzt, um den Umgestaltungswunsch in das Gleichungssystem zu integrieren.*

$$inlCosts = \sum_{c \in C} \begin{cases} 0, & falls\ c.inl \\ 1, & falls\ \neg c.inl \end{cases} \tag{5.18}$$

Gleichung 5.18 definiert die Kostenvariable *inlCosts*, die im Optimierungskriterium wiederverwendet wird (siehe Kapitel 5.1.5). Soll ein Ausdruck an mehreren Stellen eingefügt werden, wird die zugehörige Klongruppe g des Ausdrucks betrachtet. Die Klongruppe gruppiert alle Kompositionen, die einen Ausdruck repräsentieren, der an mehreren Stellen im Quelltext verwendet werden könnte (siehe Unterkapitel 4.4). Damit jede der zu g gehörenden Kompositionen eingefügt wird, erzeugt die Gleichung 5.18 Kosten, sofern die *inl*-Variable einer Komposition den Wert *falsch* hat. Ist der Wert *wahr*, so entstehen keine Kosten. Falls die Generatorregel 5.18 nicht angewendet wird, weil ein anderer Umgestaltungswunsch zur Anwendung kommt, wird der Wert der Variablen *inlCosts* auf 0 definiert.

Operanden verschieben/neuordnen

Die Position von Operanden eines Ausdrucks kann durch zwei unterschiedliche Umgestaltungen verändert werden: das Neuordnen oder das Verschieben von Operanden. Beim Neuordnen gibt der Nutzer explizit eine gewünschte Reihenfolge der Operanden an. Im Gegensatz dazu gibt der Nutzer beim Verschieben von Operanden lediglich an, welcher Operand nach links oder rechts verschoben werden soll. Je nachdem, welche Art der Umgestaltung gewählt wird, werden unterschiedliche Gleichungen im Gleichungssystem erzeugt. Sollen die Operanden neu angeordnet werden, wird für jeden Operanden festgelegt, welche Position er bekommen soll. Dazu wird für jeden Operanden eine Gleichung erzeugt.

Diese Gleichungen legen über das Knotenattribut Operandenposition (*op*) fest, auf welche Position er verschoben werden soll. Existiert dann eine Lösung des Gleichungssystems, so können alle Operanden auf die gewünschten Positionen verschoben werden.

> **Operanden vertauschen:** *Sei e ein Infixausdruck, dessen Operanden vertauscht werden sollen, und der durch die Menge aller Kompositionen C in der Abstraktion repräsentiert wird. Sei ferner* $cg_c \in c.F|^{CommutativeOperands}$ *mit* $c \in C$ *das Fragment, das die zu vertauschenden kommutativen Operanden repräsentiert und seien* $n \in cg_c.N|^{Operand}$ *die Operandenknoten von* cg_c*. Sei außerdem* i_n *die neue Operandenposition des Knoten n, wird die folgende Generatorregel genutzt, um den Umgestaltungswunsch zu erzeugen.*
>
> $$\forall n \in cg_c.N|^{Operand}, c \in C : n.op = i_n$$

Im Gegensatz dazu können auch lediglich einzelne Operanden verschoben werden, ohne Wünsche für die anderen Operanden eines Ausdrucks zu nennen. Diese Art der Umgestaltung ist sinnvoll wenn bspw. per Tastatureingabe ein ausgewählter Ausdruck nach rechts oder links verschoben werden soll. Dieser Umgestaltungswunsch wird in insgesamt eine Gleichung übersetzt. Diese Gleichung gibt an, dass die gewünschte Operandenposition größer bzw. kleiner als die ursprüngliche Position sein soll.

> **Operanden verschieben:** *Sei e ein Infixausdruck, dessen Operanden vertauscht werden sollen, und der durch die Menge aller Kompositionen C in der Abstraktion repräsentiert wird. Sei ferner* $cg_c \in c.F|^{CommutativeOperands}$ *mit* $c \in C$ *das Fragment, das die zu vertauschenden kommutativen Operanden repräsentiert und sei* $n \in cg_c.N|^{Operand}$ *der zu verschiebende Operandenknoten von* cg_c*. Dann wird die folgende Generatorregel genutzt, um den Umgestaltungswunsch zu erzeugen und den Operanden nach rechts zu verschieben. Soll der Operand nach links verschoben werden, wird das Größerals- durch ein Kleineralszeichen ersetzt.*
>
> $$\forall c \in C : n.op > n.op_{INIT}$$

Anweisungen verschieben

In Hinblick auf die Position einer Anweisung im Quelltext kann der Nutzer den Wunsch haben, die Anweisung auf- oder abwärts zu verschieben. Durch Auswahl einer Anweisung und Betätigung einer Tastenkombination soll dann die Anweisung Semantik erhaltend verschoben werden. Folglich muss der Nutzer lediglich die zu verschiebende Anweisung und die Verschiebungsrichtung angeben. Sofern die zu verschiebende Anweisung eine Klongruppe *g* besitzt, wird die Gleichungsvariable *g.cgp* betrachtet. Besitzt sie keine Klongruppe, wird stattdessen die Gleichungsvariable *cp* der zugehörigen Komposition betrachtet. Die Gleichung des Umgestaltungswunsches gibt an, dass die gewünschte Position der Variablen größer bzw. kleiner als die ursprüngliche Position ist. Aufgrund der Optimierungskriterien wird sichergestellt, dass die Anweisung tatsächlich verschoben wird und stattdessen keine Transformation wie das Extrahieren stattfindet, die die Position *cp* bzw. *cgp* ebenfalls ändern würde.

Anweisung verschieben: *Ist s eine Anweisung, die abwärts verschoben werden soll, und C die Menge aller Kompositionen, die s in der Quelltextabstraktion repräsentieren, wird die folgende Generatorregel genutzt, um den Umgestaltungswunsch zu erzeugen. Soll die Anweisung s aufwärts verschoben werden, wird das Größerals- durch ein Kleineralszeichen ersetzt.*

$$\forall c \in C : c.cp > c.cp_{INIT}$$

Parameter/Argumente neuordnen

Wenn die Argumente einer Methode vertauscht werden sollen, muss eine Berechnung für jeden Methodenaufruf dieser Methode durchgeführt werden. Nur wenn alle Methodenaufrufe umgestaltet werden können, ohne dass dies durch Programmabhängigkeiten verhindert wird, darf die Methodensignatur wie gewünscht modifiziert werden. Daher werden beim Vertauschen von Argumenten Gleichungssysteme für jede Methode im Programm erstellt, die einen zu ändernden Methodenaufruf enthalten. Die Generatorregeln sind analog zu den Generatorregeln des Vertauschens von Operanden aufgebaut. Alle Gleichungssysteme müssen eine Lösung berechnen können, damit die Umgestaltung umgesetzt werden kann.

Parameter/Argumente vertauschen: *Sei M die Menge aller Methodenaufrufe, deren Argumente vertauscht werden sollen, innerhalb eines abgeschlossenen Quelltextblocks und sei C die Menge aller Kompositionen C, die $m \in M$ in der Abstraktion repräsentieren. Sei ferner $ncg_c \in c.F|^{NotCommutativeOperands}$ mit $c \in C$ das Fragment, das die zu vertauschenden kommutativen Operanden repräsentiert und seien $n \in ncg_c.N|^{Operand}$ die Operandenknoten von ncg_c. Sei außerdem i_n die neue Operandenposition von n, wird die folgende Generatorregel genutzt, um den Umgestaltungswunsch zu erzeugen.*

$$\forall c \in C \;\; \forall n \in ncg_c.N|^{Operand} : n.op = n_i$$

Diese Gleichung legt die Position des Operanden durch das Knotenattribut *op* auf die gewünschte Position des Arguments fest. D. h. dass die Positionen der Argumente aller Aufrufe kongruent zu den Positionen der Parameter der gewünschten Methodensignatur sind. Eine Besonderheit dieses Umgestaltungswunschs ist, dass die Abstraktion bzw. das Gleichungssystem zusätzlich modifiziert werden muss. Denn üblicherweise sind die Argumente eines Methodenaufrufs auf ihre Position fixiert. Dies wird sichergestellt, indem sie in Fragmente vom Typ *NotCommutativeOperandss* gruppiert werden. Da hier jedoch die Neuordnung der Argumente ausdrücklich erwünscht ist, werden die CD-Kanten der *NotCommutativeOperands* der Methodenaufrufe m ausnahmsweise entfernt.

Durch das Gleichungssystem werden allerdings nur Transformationen berechnet, die die Argumente der Aufrufe vertauschen. Die Parameter der Signatur(en) müssen zusätzlich angepasst werden. Die dadurch möglicherweise entstehenden Signaturkonflikte werden in der Arbeit nicht betrachtet, da diese von den verwandten Arbeiten bereits genau untersucht wurden [121, 110].[3]

[3] In der Evaluation konnte kein Fehler beobachtet werden, der aufgrund von Signaturkonflikten entstanden ist.

5.1.5 Optimierungskriterium

Das Lösen von Gleichungssystemen kann zu drei unterschiedlichen Resultaten führen: Entweder es existiert keine Lösung, nur eine Lösung oder es existieren mehrere Lösungen. Existiert keine Lösung, ist dies gleichbedeutend damit, dass der Ansatz keine Quelltextumgestaltung findet, die Semantik erhaltend ist und die gewünschte Umgestaltung umsetzt. Existiert nur eine Lösung, kann diese verwendet werden. Existieren hingegen mehrere Lösungen, muss entschieden werden, welche davon gewählt werden soll. Tatsächlich existieren sogar in den allermeisten Fällen mehrere Lösungen, sofern überhaupt eine Lösung berechnet werden konnte (siehe Evaluation, Kapitel 6.3). Welche der berechneten Lösungen für die Umgestaltung verwendet wird, wählt der Löser daher nach einem Optimierungskriterium aus. Optimal ist eine Lösung genau dann, wenn sie das Umgestaltungsziel umsetzt, aber so wenige zusätzliche Änderungen wie möglich durchführt.

$$min(ieCount + roCount + 10 \cdot inlCosts) \tag{5.19}$$

$$ieCount = \sum_{c \in Compositions} \begin{cases} 0, & falls\ c.inl = c.inl_{INIT} \\ 1, & falls\ c.inl \neq c.inl_{INIT} \end{cases} \tag{5.20}$$

$$roCount = \sum_{n \in Nodes|^{Operand}} \begin{cases} 0, & falls\ n.op = n.op_{INIT} \\ 1, & falls\ n.op \neq n.op_{INIT} \end{cases} \tag{5.21}$$

In den Gleichungen 5.19, 5.20 und 5.21 wird dies spezifiziert. Im Optimierungskriterium werden alle Transformationen, die ein Einfügen oder Extrahieren von Ausdrücken bedeuten, einbezogen. Ferner werden Neuordnen von Operanden mitgezählt. Je geringer die Anzahl dieser Änderungen ist, desto besser ist die Lösung.

Darüber hinaus kann die Einfügen-Umgestaltung das Optimierungskriterium beeinflussen, wie in Kapitel 5.1.4 beschrieben. Eventuell können im Zuge solch einer gewünschten Umgestaltung nicht alle Ausdrücke eingefügt werden. In diesem Fall werden so viele Ausdrücke wie möglich eingefügt oder extrahiert. Dies wird durch den dritten Parameter *inlCosts* in der Gleichung 5.19 bewirkt, der in Gleichung 5.18 definiert wird. Je mehr Ausdrücke eingefügt werden, desto geringer ist der Wert von *inlCosts*. Er geht hier mit einem 10-fach höheren Gewicht in die Gleichung ein, das sich in den Tests als sinnvoll erwiesen hat. Dadurch wird die gewünschte Umgestaltung im Vergleich zu unerwünschten Nebeneffekten bevorzugt. Wird der Parameter *inlCosts* nicht verwendet, soll er mit dem Wert 0 definiert sein.

5.2 Suchstrategie

Ziel der Suchstrategie ist, für jede Gleichungsvariable einen Wert zu finden, sodass alle Bedingungsgleichungen erfüllt sind. Dafür wählen übliche Suchstrategien abwechselnd erst je eine Gleichungsvariable und dann ihren Wert aus. Die Wahl der Variablen orientiert sich an verschiedenen, allgemeinen Kriterien wie bspw. der Anzahl der Bedingungen, die an die Variable geknüpft sind. Die Wahl des Werts ist auf-, absteigend oder zufällig. Diese allgemeinen Standardsuchstrategien können jedoch nicht die Eigenheiten von domänenspezifischen Problemen berücksichtigen und erkunden daher unnötig große Suchräume. Das

heißt, dass sich die aufgrund der Bedingungsgleichungen zuvor festgelegten Variablenwerte später in der Suche als ungültig bzw. widersprüchlich herausstellen und Rücksprünge aus diesen Sackgassen erforderlich sind. Wenn solche Irrwege während der Suche vermieden werden können, kann die Suche insgesamt beachtlich beschleunigt werden.

5.2.1 Teilprobleme

Die hier verwendete Suchstrategie orientiert sich an der Struktur des Gleichungssystems; also am Aufbau der Gleichungen und Variablen. Die Struktur ergibt sich dabei aus der Programmabstraktion, ihren Blöcken und Gleichungsvariablen. Das Gesamtproblem wird daher einerseits in verschiedenartige Teilprobleme zerlegt, die unterschiedliche Gleichungsvariablen lösen. Andererseits wird für jede Art von Teilproblem auch eine bestimmte Suchstrategie genutzt, die Irrwege während der Suche so gut wie möglich vermeidet. Die Teilprobleme sind in die folgenden zwei Arten unterteilt: Block- und Restteilproblem. Während ein Gleichungssystem aus mehreren Blockteilproblemen bestehen kann, gehört allerdings immer nur ein Restteilproblem zu jedem Gleichungssystem. Blockteilprobleme gehen aus einzelnen partiellen Abstraktionen hervor und das Restteilproblem beherbergt alle restlichen Variablen, die nicht durch Blockteilprobleme gelöst werden.

Element	CSP-Variable	Wertebereich	Teilproblem
Komposition	inl	$\{true, false\}$	Block
Knoten	np	$\{1..\|N\|\}$	Block
Knoten	op	$\{1..\|n.fgm.N\|\}$	Block
Symbol	exs	$\{true, false\}$	Rest
Klongruppe	asx	$\{true, false\}$	Rest
Klongruppe	cgp	$\{1..\|N\|\}$	Rest

Tabelle 5.2: CSP-Variablen der Teilprobleme

Jede Art von Teilproblem konzentriert sich auf die Wertebelegung von spezifischen Gleichungsvariablen des Gleichungssystems. Die Tabelle 5.2 führt alle Gleichungsvariablen auf und weist in der Spalte *Teilproblem* aus, welche Variablenart in welcher Art von Teilproblem gelöst wird. Das bedeutet, dass beim Lösen von Blockteilproblemen festgelegt wird, welche Reihenfolgen Anweisungen und Ausdrücke haben und ob Ausdrücke eingefügt oder extrahiert werden sollen. Beim Lösen von Restteilproblemen wird hingegen bestimmt, ob Symbole oder Zuweisungen existieren, oder an welcher Position Zuweisungen im Quelltext stehen.

Die Abbildung 5.3 zeigt die Aufteilung des gesamten Gleichungsproblems in Teilprobleme. Die Blockteilprobleme werden in der Abbildung mit $TP_1 \dots TP_n$ bezeichnet und das Restteilproblem wird TP_m genannt. Jedes der Teilprobleme kann einzeln gelöst werden. Jedoch existieren zwischen den Teilproblemen auch Abhängigkeiten, die in zwei Arten unterteilt werden können. Zwischen zwei Blockteilproblemen können dann Abhängigkeiten bestehen, wenn ihre zugehörigen Blöcke Abhängigkeiten zueinander besitzen. Solche Abhängigkeiten können dann entstehen, wenn Datenflüsse von einem über- in einen untergeordneten Block existieren. Das Restteilproblem besitzt Abhängigkeiten zu allen Blockteilproblemen, da seine Variablenwerte nur dann in Übereinstimmung mit allen

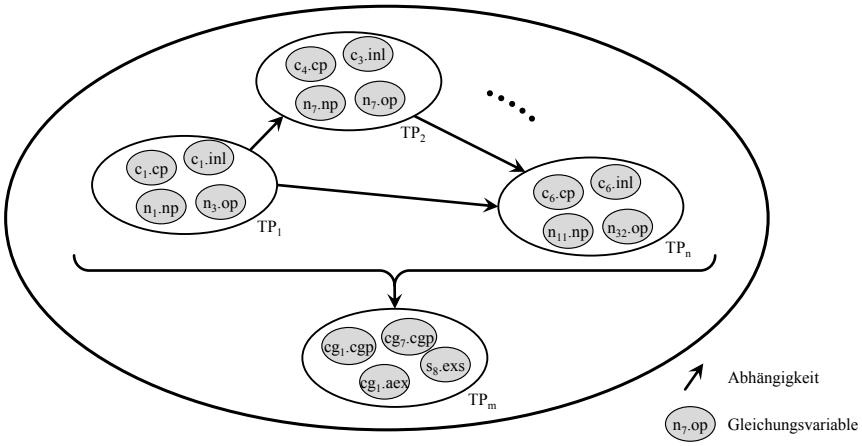


Abbildung 5.3: Abhängigkeiten und Gleichungsvariablen der Teilprobleme (TP)

Gleichungen berechnet werden können, wenn die Blockteilprobleme zuvor gelöst wurden. Aufgrund der Abhängigkeiten zwischen den Teilproblemen wird ihre Lösungsreihenfolge eingeschränkt. Zunächst müssen die Blockteilprobleme der Reihe nach gelöst werden, so wie sie im Quelltext vom Methodenbeginn bis zum Methodenende definiert sind. Sind alle Blockteilprobleme gelöst, wird schließlich auch das Restteilproblem gelöst.

Ausgangsprogramm	Umgestaltungswunsch	Erwartetes Programm

`int y, z;`
`void foo() {`
`  int x, t, a;`
`  t = a++;`
`  x = a + 2;`
`  if (...) {`
`  y = x;`
`  z = t;`
`  }`
`}`
	Zuweisung `t = a++;` einfügen	`int y, z;`
		`void foo() {`
		`  int x, t, a;`
		
		`  if (...) {`
		`  z = a++;`
		`  y = a + 2;`
		`  }`
		`}`

Tabelle 5.3: Abhängigkeit zwischen Blockteilproblemen

Das Umformungsbeispiel in Tabelle 5.3 zeigt ein Beispiel für eine Abhängigkeit zwischen zwei Blockteilproblemen. Beide Programme besitzen die gleichen Vor- und Nachzustände und werden daher als semantisch äquivalent angesehen. Die Abstraktion besteht aus zwei Blöcken: dem Methodenrumpf von `foo()` und dem Dannblock der Alternative. Aufgrund des Umgestaltungswunsches einerseits und der Programmabhängigkeiten andererseits wird neben der Zuweisung `t=a++;` auch die Zuweisung `x=a+2;` eingefügt. Dies geschieht, da die letztgenannte Zuweisung weder vor die erstgenannte, noch hinter die Alternative verschoben werden kann.

Zur Lösung der Umgestaltung des Beispiels wird das Gesamtproblem in zwei Blockteilprobleme und ein Restteilproblem aufgeteilt. Dabei wird erst das Blockteilproblem des

Methodenrumpfs, dann das des Dannblocks und schließlich das Restteilproblem gelöst. Die Lösung des ersten Blockteilproblems legt jedoch fest, dass die gewünschte Umformung nur dann durchgeführt werden kann, wenn die Zuweisung `x=a+2;` ebenfalls eingefügt wird. Folglich werden im Anschluss an die Lösung des ersten Blockteilproblems weitere Gleichungen erzeugt, die dazu führen, dass die zweite Zuweisung eingefügt werden muss. Diese zusätzlichen Gleichungen können sich so auf alle folgenden Blockteilprobleme auswirken.

$$\begin{aligned} \forall g \in CloneGroups \;\; \forall c, c_b \in Compositions, \\ \{c, c_b\} \subseteq g.C \wedge \neg(Intent\ is\ to\ inline\ c_b) \wedge \\ Solved(c_b) \wedge \neg Solved(c) \wedge \\ \wedge c_b.inl_{SOLVED} \wedge \neg c_b.inl_{INIT} : \\ c.inl = true \end{aligned} \tag{5.22}$$

Gleichung 5.22 beschreibt die zusätzlichen Bedingungsgleichungen, die nach dem Lösen jedes Blocks erzeugt werden. Sie trifft zu, falls laut Lösung eine Komposition eingefügt werden soll, die zuvor nicht eingefügt wurde und deren Einfügen auch kein Teil des Umgestaltungswunsches war. Das Prädikat *Solved* gibt an, ob ein Abstraktionselement Teil eines Blockteilproblems ist, das bereits gelöst wurde. Die Komposition c_b ist folglich Teil eines bereits gelösten Blockteilproblems, während c eine Komposition eines Blockteilproblems ist, das noch gelöst werden soll. Die Variable $c_b.inl_{SOLVED}$ spezifiziert hierbei den durch die Suchstrategie ermittelten sowie optimalen Wert des Teilproblems. Ferner müssen weitere Kompositionen existieren, die Teil derselben Klongruppe sind. In solchen Fällen wird für alle Kompositionen der zugehörigen Klongruppe festgelegt, dass sie eingefügt werden müssen.

In den folgenden beiden Unterkapiteln werden die Lösungsstrategien der beiden Teilprobleme beschrieben. Da Blockteilprobleme wesentlich komplexer im Aufbau und Funktionsweise sind, werden sie anhand der folgenden Regeln und eines Beispiels erklärt.

5.2.2 Blockteilproblem

Allgemeiner Ansatz

Die Suchstrategie funktioniert wie eine Tiefensuche. Jeder Schritt der Tiefensuche entspricht in der Suchstrategie der Wahl einer Variablen und einem zugehörigen Wert. Mit jedem solchen Schritt wird die Zahl der undefinierten Variablen kleiner, solange bis jeder Variablen ein Wert zugewiesen wurde. Ist dies der Fall, konnte eine Lösung gefunden werden. Allerdings wird nach jedem Schritt geprüft, ob alle Constraint-Gleichungen erfüllt sind. Nur wenn dies zutrifft, wird das nächste Paar aus Variable und Wert belegt. Falls die Constraint-Gleichungen einen Widerspruch der bereits definierten Variablen aufdecken, befindet sich die Suche in einer Sackgasse und es wird ein Backtracking-Schritt durchgeführt. Falls im Suchpfad alle Variablen mit Werten belegt werden können und die Constraint-Gleichungen keinen Widerspruch der Wertebelegung erzeugen, ist eine Lösung gefunden. Die Wertebelegung wird dann Lösung des Constraint-Problems genannt. Welche Transformationen aus einer solchen Lösung abgelesen werden und ob diese die Programmsemantik erhalten, wird im Unterkapitel 5.3 erklärt. Im Folgenden werden Tupel verwendet, um die Paare von Variablen und zugehörigen Wert zu notieren.

Die Suchstrategie wird vom Constraint-Löser durchgeführt. Zur Definition der Vorgehensweise der Suchstrategie wird ein Regelsatz von Entscheidungsregeln verwendet (Tabelle 5.4). Die Notation des Regelsatzes basiert auf der in Kapitel 3 vorgestellten Notation. Allerdings werden zwei weitere Symbole eingeführt, um die *Entscheidungsregel* zu kennzeichnen. Eine Entscheidungsregel besteht aus zwei Tupeln, die durch ein Pfeilsymbol miteinander verbunden sind. Ausgehend vom linken Tupel wird das rechte Tupel als nächster Schritt im Suchbaum gewählt. Dadurch kann ein Suchbaum von Tupeln aufgespannt werden, der für jede Gleichungsvariable einen Wert festlegt. Jedes dieser Tupel besteht daher aus einer Gleichungsvariablen (erstes Element) und ihrem konkreten Belegungswert (zweites Element). Das Starttupel des Suchbaums wird mit ϵ bezeichnet. Hat ein Tupel lediglich ein Nachfolgertupel, so wird das Symbol $\longmapsto$ genutzt, um auf das Zieltupel zu zeigen. Besitzt ein Tupel allerdings potentiell mehrere Nachfolger, wird das Symbol $\Longrightarrow$ genutzt.

Immer wenn ein Tupel gewählt wurde, wird vom Constraint-Löser überprüft, ob weiterhin alle Constraint-Gleichungen erfüllt sind. Falls nicht, wird ein Backtracking-Schritt durchgeführt und das Verfahren springt zurück zur letzten Verzweigung, an der mehrere Zieltupel zur Auswahl standen. Kann ein Ast des Suchbaums so weit erkundet werden, dass jede Variable des Gleichungssystems belegt ist, ist eine gültige Lösung gefunden. Andernfalls konnte keine Lösung gefunden werden und das Constraint-Problem ist nicht lösbar. Folglich ist der Umgestaltungswunsch nicht durchführbar auf dem gegebenen Quelltextblock.

Die Tabelle 5.4 führt alle Entscheidungsregeln auf, die die Suchstrategie in Blockteilproblemen verwendet. Dabei gilt, dass für ein beliebiges Starttupel immer genau eine Entscheidungsregel gültig ist. Mit Komma getrennt stehen hinter den Entscheidungsregeln die Bedingung, die gelten muss, damit eine Entscheidungsregel aktiv ist. Wird ein $*$ als Variablenwert angegeben, so ist der Variablenwert unerheblich für die Entscheidungsregel. Für Regeln mit identischen Starttupeln gilt, dass die Bedingung die Wahl der Entscheidungsregeln auf eine Regel einschränkt. Eine Eigenschaft des Suchpfads ist, dass keine CSP-Variable auf zwei unterschiedliche Werte festgelegt wird. Zusätzlich gelten die folgenden Einschränkungen der Entscheidungsregeln.

Für die Entscheidungsregeln(n)

- (2) und (3) gilt, dass val_{np} der um eins erhöhte Wert des vorangegangenen *c.cp*-Werts ist.
- (4) bis (8) sowie (19) und (20) gilt, dass die *id*-Werte der Knoten wie folgt geordnet sind: Die Nummerierung beginnt beim Startknoten und endet beim Endknoten. Außerdem ist sie kongruent zum ursprünglichen Quelltext und der daraus erzeugten Fragmente.
- (6) und (7) gilt: $val_{op} \in \{1..(|n_1.fgm.N| - 1)\}$.
- (8) gilt: $val_{op} = |n_1.fgm.N|$.
- (9) und (10) gilt: $val_{np} = n^{OpStart}.np \wedge \{n, n_1\} = n_1.fgm.N$.

(1)	$\epsilon \Longrightarrow$	$(c.inl, false)$,	$\exists c \in C$
(2)	$(c.inl, false) \longmapsto$	$(n_S.np, val_{np})$,	$\exists c \in C, n_S \in N : n_S^{Start} \wedge n_S \in c.F.N$
(3)	$(c.inl, true) \longmapsto$	$(n_S.np, val_{np})$,	$\exists c \in C, n_S \in N : n_S^{Start} \wedge n_S \in c.F.N$
(4)	$(c.cp, val) \longmapsto$	$(n_2.np, val + 1)$,	$\exists c \in C, n_2, n_S \in N : n_S^{Start} \wedge \{n_2, n_S\} \subset c.F.N \wedge n_S.id + 1 = n_2.id$
(5)	$(n_1.op, 0) \longmapsto$	$(n_2.op, 1)$,	$\exists n_1, n_2 \in N : n_1^{OpStart} \wedge n_2^{OpEnd} \wedge n_1.id + 1 = n_2.id$
(6)	$(n_1.op, 0) \Longrightarrow$	$(n_2.op, val_{op})$,	$\exists n_1, n_2 \in N : n_1^{OpStart} \wedge n_2^{Operand} \wedge n_1.id + 1 = n_2.id$
(7)	$(n_1.op, *) \Longrightarrow$	$(n_2.op, val_{op})$,	$\exists n_1, n_2 \in N : n_1^{Operand} \wedge n_2^{Operand} \wedge n_1.id + 1 = n_2.id$
(8)	$(n_1.op, *) \longmapsto$	$(n_2.op, val_{op})$,	$\exists n_1, n_2 \in N : n_1^{Operand} \wedge n_2^{OpEnd} \wedge n_1.id + 1 = n_2.id$
(9)	$(n_1.op, 1) \longmapsto$	$(n_1.np, val_{np})$,	$\exists n_1 \in N : n_1^{OpEnd}$
(10)	$(n_1.op, *) \longmapsto$	$(n_{OP1}.np, val_{np})$,	$\exists n_1, n_{OP1} \in N : n_1^{OpEnd} \wedge n_{OP1}^{Operand} \wedge \{n_1, n_{OP1}\} \subset n_1.fgm.N \wedge n_{OP1}.op = 1$
(11)	$(n_1.op, *) \longmapsto$	$(c_{OP1}.inl, true)$,	$\exists c_{OP1} \in C, n_1, n_{OP1} \in N : n_1 \underset{OP}{\rightarrow} n_{OP1} \in E : n_1^{OpEnd} \wedge n_{OP1}^{Operand} \wedge n \in c_{OP1}.F.N \wedge n_{OP1}.op = 1$
(12)	$(n_1.np, val) \longmapsto$	$(n_2.np, val + 1)$	$\exists n_1, n_2 \in N : \{n_1, n_2\} \subset n_1.fgm.N \wedge n_1^{Operand} \wedge n_2^{OpEnd} \wedge n_2.op = n_1.op + 1$
(13)	$(n_1.np, val) \longmapsto$	$(n_{OP1}.np, val + 1)$,	$\exists n_1, n_{OP1} \in N : \{n_1, n_{OP1}\} \subset n_1.fgm.N \wedge n_1^{Operand} \wedge n_{OP1}^{OpEnd} \wedge n_{OP1}.op = n_1.op + 1$
(14)	$(n_1.np, val) \longmapsto$	$(c_{OP1}.inl, true)$,	$\exists c_{OP1} \in C, n_1, n_{OP1} \in N : n_1 \underset{OP}{\rightarrow} n_{OP1} \in E : n_1^{Operand} \wedge n_{OP1}^{Operand} \wedge n_1 \in c_{OP1}.F.N \wedge n_{OP1}.op = n_1.op + 1$
(15)	$(n_1.np, val) \longmapsto$	$(c.cp, val)$,	$\exists n_1 \in N : n_1^{Start} \wedge n_1 \in c.F.N$
(16)	$(n_1.np, *) \Longrightarrow$	$(c.inl, false)$,	$\exists n_1, n_2 \in N, c \in C : n_1^{End} \wedge \neg(n_1.fgm.cmp.inl)$
(17)	$(n_1.np, val) \longmapsto$	$(n_{OP}.np, val + 1)$,	$\exists n_1, n_{OP} \in N : \exists n_1 \underset{OP}{\rightarrow} n_{OP} \in E : n_1^{End} \wedge n_{OP}^{Operand} \wedge (n_1.fgm.cmp.inl)$
(18)	$(n_1.np, *) \longmapsto$	$(n_1.op, 0)$,	$\exists n_1 \in N : n_1^{OpStart}$
(19)	$(n_1.np, val) \longmapsto$	$(n_2.np, val + 1)$,	$\exists n_1, n_2 \in N : n_1^{OpEnd} \wedge n_1.id + 1 = n_2.id$
(20)	$(n_1.np, val) \longmapsto$	$(n_2.np, val + 1)$,	$\exists n_1, n_2 \in N : n_1^{Aux} \wedge n_1.id + 1 = n_2.id$

Tabelle 5.4: Entscheidungsregeln der Suchstrategie

- (10) und (13) gilt: Sie werden angewendet, wenn der Unterausdruck nicht eingebettet wird, da der *inl*-Wert von c im aktuellen Suchpfad bereits auf $false$ festgelegt wurde.
- (11) und (14) gilt: Sie werden angewendet, wenn der Unterausdruck c eingebettet wird, da der *inl*-Wert von c im aktuellen Suchpfad nicht auf $false$ festgelegt wurde.
- (12) gilt: Sie wird angewendet, wenn die *np*-Variablen aller Operandenknoten des Fragments bereits festgelegt wurden.

Beispiel

Die Lösungsstrategie für Blockteilprobleme soll nun anhand eines Beispiels verdeutlicht werden. Hierzu wird der folgende Quelltext 5.3 genutzt, dessen Methode `foo()` zwei Zuweisungsanweisungen enthält. Ein hier anwendbarer Umgestaltungswunsch könnte die erste Zuweisungsanweisung einfügen, sodass nur noch die Zuweisung `y=1+2;` nach der Umgestaltung existiert.

```
int x, y;
void foo() {
  x = 1;
  y = x + 2;
}
```

Quelltext 5.3: Erklärungsbeispiel der Suchstrategie

Die Abstraktion des Beispiels ist in Abbildung 5.4 dargestellt. Die Abstraktion besteht lediglich aus zwei Kompositionen und den dazu gehörigen Fragmenten und Knoten. Sie besitzt eine Datenflussabhängigkeit, die mit e^{x}_{Op} beschriftet ist. Alle Kompositionen und Knoten sind zur eindeutigen Referenzierung durchnummeriert.

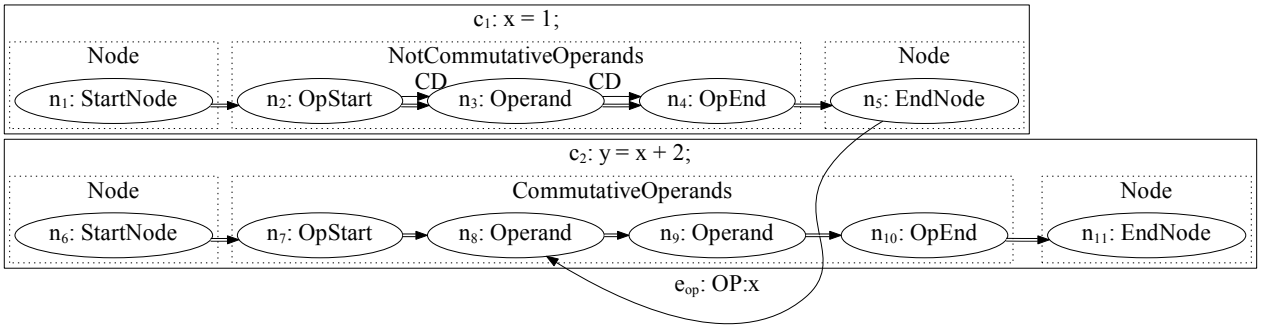


Abbildung 5.4: Abstraktion des Beispiels

Wenn dieses Beispiel in ein Gleichungssystem überführt und gelöst werden soll, wird die oben beschriebene Lösungsstrategie angewendet. In der Abbildung 5.5 wird der Suchbaum graphisch dargestellt. Jeder Knoten symbolisiert ein Entscheidungstupel, bestehend aus einer Gleichungsvariablen und ihrem zugewiesenen Wert. Zwischen zwei Entscheidungstupeln existiert eine Kante, die mit einer Entscheidungsregel beschriftet ist (siehe Gleichungen 5.4). Starten mehrere Kanten vom selben Entscheidungstupel, so ist an dieser Stelle eine Verzweigung. Durch Verzweigungen entstehen Suchbäume, die möglicherweise

mehrere Lösungen besitzen. Während der Suchbaum aufgespannt wird, werden die Variablen eines Teilproblems festgelegt. Dabei kann es passieren, dass die Bedingungsgleichungen und die festgelegten Variablenwerte von mindestens zwei Tupeln einen Widerspruch ergeben. Passiert dies, so ist der aktuelle Teilbaum unlösbar und ein Rückfallen zur letzten Verzweigung wird durchgeführt. Kann im Gegensatz dazu ein Teilbaum soweit aufgespannt werden, bis kein weiteres Tupel existiert, so ist eine Lösung gefunden. Um die optimale Lösung zu finden, werden alle Teilbäume aufgespannt und der Suchraum komplett erkundet. Die resultierende Menge an gültigen Teilbäumen beschreibt die Variablenbelegungen aller Lösungen. Unter ihnen wird mithilfe des Optimierungskriteriums die beste ausgewählt.

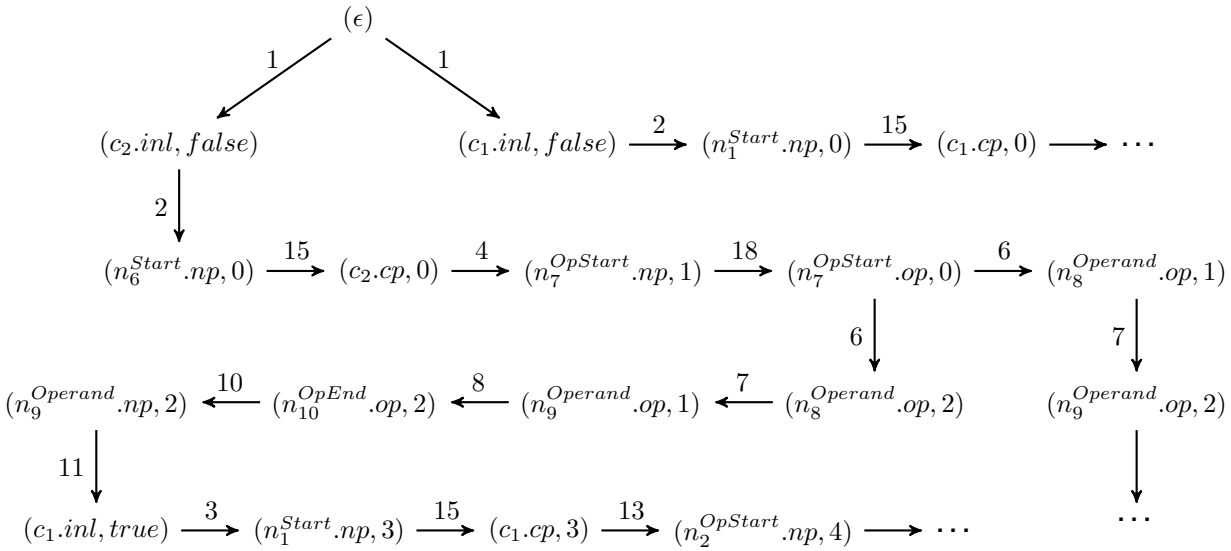


Abbildung 5.5: Teilgraph des Suchbaums

In Abbildung 5.5 startet der Suchbaum mit dem ϵ-Knoten und wendet die erste Entscheidungsregel an. Da in der Abstraktion zwei Kompositionen existieren, kann diese Regel zwei Mal angewendet werden und der ϵ-Knoten verfügt daher über zwei Nachfolger. Auf diese beiden Nachfolger werden jeweils die gleichen Regeln angewendet. Als nächstes verzweigt sich der Suchbaum an der Stelle, wo die Reihenfolge der Additionsoperanden festgelegt werden muss. Entweder die Reihenfolge bleibt so wie sie war (rechter Ast), oder beide Operanden werden vertauscht (absteigender Ast). Bleibt es bei der ursprünglichen Operandenreihenfolge, so sind die Werte der *op*-Variablen 1 und dann 2. Werden die Operanden vertauscht, sind ihre Werte nacheinander 2 und dann 1. Da der zweite Operand lediglich ein Literal ist, wird im folgenden die Entscheidungsregel 10 angewendet. Auf das Literal folgt als Operand die Variablenreferenz `x`, die eingebettet werden kann. An dieser Stelle des Suchbaums muss sie eingebettet werden, weshalb Entscheidungsregel 11 angewendet wird. In dem anderen Suchbaum, der vom ϵ-Knoten aus startet, würde `x` nicht eingebettet werden. Schließlich folgen noch alle weiteren Variablen, die in Blockteilproblemen belegt werden müssen. Die hier begonnenen Suchbäume führen dabei alle zu einer gültigen Lösung.

5.2.3 Restteilproblem

Für die Suchstrategie des Restteilproblems wird eine Standardstrategie des Lösers genutzt. Da keine gegenseitigen Abhängigkeiten zwischen den Variablen des Restteilproblems bestehen, führt die Standardsuchstrategie zu keinen merkbaren Geschwindigkeitseinbußen. Prinzipiell ist allerdings auch eine direkte Berechnung der Variablenwerte des Restteilproblems möglich, um ein Ausprobieren aller Werte während der Suchstrategie zu vermeiden. Denn die Variablenwerte des Restteilproblems resultieren aus den bereits festgelegten Werten der Blockteilprobleme und können daher einfach ermittelt werden. Wie anhand der Gleichungen in Abschnitt 5.1.3 nachvollzogen werden kann, resultiert bspw. der Wert der *cgp* aus den Positionen aller zugehörigen Kompositionen (Gleichung 5.15).

5.3 Quelltexttransformation

Aus dem Vergleich der Variablenwerte des gelösten Gleichungssystems mit den ursprünglichen Variablenwerten wird eine Menge an sequenzkongruenten Transformationen abgelesen, die in Unterkapitel 3.6 vorgestellt wurden. Da die Transformationen sequenzkongruent sind, ist auch eine beliebige Komposition der Transformationen sequenzkongruent. Da die Transformationen allerdings nur die in der Abstraktion bekannten Abhängigkeiten beachten, gilt die Sequenzkongruenz nur insoweit, wie die Programmabhängigkeiten in die Abstraktion abgebildet wurden. Wie in der Einleitung erwähnt, ist bei der Abbildung der Programmabhängigkeiten jedoch ein Kompromiss sinnvoll, damit einerseits genügend Freiheitsgrade existieren, um die gewünschten Umgestaltungen anzuwenden, und andererseits die Programmsemantik so gut wie möglich bewahrt wird. Unter der Annahme, dass die Abstraktion alle Programmabhängigkeiten abbildet, ändert eine beliebige Menge der Transformationen die Programmsemantik nicht. Aufgrund des Kompromisses werden einige Programmabhängigkeiten in der Abstraktion jedoch bewusst ausgespart, wie bspw. Abhängigkeiten, die durch Aliasbeziehungen entstehen. Daher wird hier – wie in vergleichbaren Arbeiten – der Erhalt der Semantik nicht garantiert. Dennoch berücksichtigt die vorliegende Arbeit immer mehr Abhängigkeiten im Vergleich zu verwandten Arbeiten und erhält somit die Semantik besser (siehe Evaluation, Kapitel 6.

5.3.1 Lösung des Gleichungssystems interpretieren

Um eine Menge an Transformationen zu erhalten, werden aus einem gelösten Gleichungssystem die Variablenwerte interpretiert. Jede Transformation ergibt sich dabei aus Informationen über die Art der Variablen und ihrer Änderung. Die Transformationen, die aus dem Gleichungssystem abgelesen werden, können in zwei Gruppen unterteilt werden. Die erste Gruppe besteht aus allen Transformationen, die im Zuge des Einfügens/Extrahierens von Ausdrücken notwendig sind: *Datendistribution*, *Komposition einfügen/extrahieren* und *Deklaration hinzufügen/entfernen*. Die zweite Gruppe besteht aus den Transformationen, die durch das Verschieben/Neuordnen von Anweisungen oder Operanden entstehen. Das sind *Operanden umsortieren* und *Anweisungen umsortieren*. Diese Transformationen werden auf die Abstraktion A_0 angewendet, die direkt aus dem Quelltext durch die Abbildungsvorschriften erzeugt wurde (siehe Unterkapitel 4.2). Die Transformationen müssen nicht auf die modifizierte Abstraktion A_3 angewendet werden, da A_3 die Zusatzinformationen,

wie Abhängigkeitskanten und auch die Modifikationen, die durch die Wunschanpassung und Klonerzeugung entstanden sind, für die Transformationen irrelevant sind.

Gleichungsvariable	Wert_{INIT}	Wert_{SOLVED}	Transformation
inl	$falsch$	$wahr$	Datendistribution, Komposition einfügen
inl	$wahr$	$falsch$	Datendistribution^{-1}, Komposition extrahieren
asx	$falsch$	$wahr$	Datendistribution
asx	$wahr$	$falsch$	Datendistribution^{-1}
exs	$falsch$	$wahr$	Deklaration hinzufügen
exs	$wahr$	$falsch$	Deklaration entfernen
op	–	–[a]	Operanden umsortieren[b]
cp, cgp	–	–[a]	Anweisungen umsortieren[c]

Tabelle 5.5: Transformationen aus der Constraint-Lösung interpretieren

[a]Wertänderung hat stattgefunden
[b]entsprechend der *op*-Werte
[c]entsprechend der *cp*- und *cgp*-Werte

Tabelle 5.5 gibt an, welche Transformationen aus welchen Variablen abgelesen werden. Dabei wird überprüft, welche Wertänderung stattgefunden hat, da die Transformationen nur dann erzeugt werden, wenn sich der Variablenwert auf bestimmte Weise geändert hat. Dieser Vorher-/Nachhervergleich wird bei den Variablen inl, asx und esx durchgeführt. Anhand des Vergleichs werden alle Transformationen erzeugt, die für ein Einfügen oder Extrahieren von Ausdrücken notwendig sind.

Die Transformationen, die Operanden oder Anweisungen verschieben, werden immer dann erzeugt, wenn sich die zugehörigen Positionswerte verändert haben. Dies betrifft die Variable *op*, die die Positionswerte der Operanden angibt und die Variablen *cp* und *cgp*, die die Positionswerte der Kompositionen, bzw. Klongruppen angeben. Die Quelltextblöcke werden durch partielle Abstraktionen repräsentiert. Innerhalb einer partiellen Abstraktion sind die Positionswerte, die die Position einer Anweisung repräsentieren können, paarweise verschieden. Demnach ist die Position einer Anweisung innerhalb ihres Blocks eindeutig bekannt. Die Transformation *Anweisungen umsortieren* wird daher für jede partielle Abstraktion und somit für jeden Quelltextblock erstellt. Die Position einer Anweisung, die selbst einen Block b enthält, ergibt sich dann aus der Positionsvariablen der Komposition, die die Blockabstraktion für b enthält.

Die im Unterkapitel 3.6 vorgestellten Transformationen werden auf Abstraktionen angewandt, die abgeschlossene Quelltextblöcke widerspiegeln. Mit Ausnahme der Umgestaltung *Parameter umsortieren* ist diese Abbildung des Quelltexts und der notwendigen Transformationen auch ausreichend. Das Umsortieren von Parametern erfordert jedoch auch Änderungen an der Methodensignatur. Sofern das Umsortieren der Argumente aller Methodenaufrufe erfolgreich berechnet werden konnte, muss auch eine Transformation des Quelltexts durchgeführt werden, die die Methodensignatur ändert. Unter Umständen können aufgrund von Vererbung von Klassen oder Schnittstellen auch weitere Signaturen von der Änderung betroffen sein, falls die Methode andere Methoden oder Methodende-

klarationen überschreibt, oder selbst überschrieben wird. Somit betrifft das Umsortieren der Parameter nicht nur die Methodendeklaration *md* innerhalb der Klasse, die auch den Methodenrumpf definiert, sondern auch alle anderen Klassen, die eine an *md* bindende Methodendeklaration enthalten. Wird eine dieser Deklarationen entsprechend des Umgestaltungswunsches verändert, können Konflikte zu anderen, bereits bestehenden Methodendeklarationen entstehen. Diese Konflikte resultieren bspw. daraus, dass eine äquivalente Signatur bereits vorhanden ist und so eine Kollision entsteht. Konflikte dieser Art wurden bereits in vielen Arbeiten untersucht und auf verschiedene Art gelöst [84, 121, 110, 118] und daher in dieser Arbeit nicht genauer untersucht.

5.3.2 Transformationen anwenden

Um die Umgestaltung *Ausdruck einfügen* durchführen zu können, werden mehrere Transformationen benötigt. Diese werden zusammengefasst, bevor sie auf den Quelltext angewendet werden. Die drei Transformationen (1) *Datendistribution*, (2) *Komposition einfügen* und (3) *Deklaration entfernen* werden daher gemeinsam ausgewertet. Dabei werden nicht alle Transformationen dieser Arten sondern nur solche Transformationen gemeinsam ausgewertet, die sich auf dieselbe Variable beziehen. Die Bezugsvariable ergibt sich bei den drei Transformationen wie folgt: Bei Transformation (1) wird die Distributionsvariable, bei (2) die Substitutionsvariable und bei (3) die Deklarationsvariable betrachtet. Die Deklarationsvariable ist diejenige, die von der Deklarationsanweisung deklariert wird. Die Substitutionsvariable ist diejenige, die durch einen Ausdruck substituiert wird und die Distributionsvariable ist diejenige, die an mehreren Stellen im Quelltext gelesen wird. Die Datendistribution und das Einfügen/Extrahieren von Kompositionen treten nie einzeln auf (siehe Tabelle 5.5). Ob eine Deklaration hinzugefügt oder entfernt wird, ist je nach Fall unterschiedlich. Durch die gemeinsame Auswertung der Transformationen bzgl. derselben Variablen können redundante Quelltextänderungen vermieden und Quelltextelemente wiederverwendet werden.

Ausdruck einfügen

Wird ein Ausdruck eingefügt, kann dadurch die Zuweisung der Variablen und womöglich auch ihre Deklaration überflüssig werden. Beim Einfügen wird eine Kopie des ursprünglichen Ausdrucks für jede Stelle erzeugt. Die Zuweisung wird gelöscht, sofern der Ausdruck an jede abhängige Stelle eingefügt werden kann. Beim Löschen wird auch der Ausdruck selbst gelöscht. Werden die Transformationen gemeinsam ausgewertet, so kann auf das Löschen des ursprünglichen Ausdrucks verzichtet werden. Stattdessen wird dieser wiederverwendet und an einer Stelle eingefügt, für die sonst eine Kopie des Ausdrucks erzeugt worden wäre. Die Deklaration der Variablen wird gelöscht, wenn die Variable lokal ist und nicht mehr verwendet wird.

```
void foo() {
  int a;
  a = 1;
  f(a);
  f(a);
}
```

Quelltext 5.4: Literal `1` einfügen

Das Quelltextbeispiel 5.4 zeigt eine Deklaration, eine Zuweisung zu `a` und zwei Methodenaufrufe. Der Umgestaltungswunsch, den Literalausdruck `1` an beiden Lesestellen von `a` einzufügen, würde hier zu den folgenden Transformationen führen, die sich auf die Variable `a` beziehen: Komposition einfügen, Deklaration entfernen und Datendistribution. Außerdem werden die beiden Transformationen *Datendistribution* und *Datendistribution*$^{-1}$ für die Zuweisung `a = 1;` erzeugt, die sich jedoch gegenseitig aufheben. Die übrig gebliebenen drei Transformationen bewirken das Entfernen der Deklaration, das Substituieren des Arguments `a` durch das Literal `1` und das Löschen der Zuweisung. Damit das Literal an beiden Stellen eingefügt wird, wird es mithilfe der Transformation *Datendistribution* kopiert. Folglich werden die Methodenargumente durch den ursprünglichen Literalausdruck `1` bzw. durch dessen Kopie substituiert.

Ausdruck extrahieren

Soll ein Ausdruck aufgrund einer Umgestaltung extrahiert werden, werden die zu *Ausdruck Einfügen* jeweils komplementären Transformationen verwendet. Diese sind: (1) *Datendistribution*$^{-1}$, (2) *Komposition extrahieren* und (3) *Deklaration hinzufügen*. Wird ein Ausdruck extrahiert, dann werden mehrere Quelltextänderungen durchgeführt. Zunächst wird der betroffene Ausdruck gelöscht und durch eine neue Variable substituiert. Ferner werden eine Deklaration der neuen Variablen und eine Zuweisung erstellt. Die linke Seite der Zuweisung ist die neue Variable und die rechte ist eine Kopie des extrahierten Ausdrucks.

```
void bar() {
  f(1+2);
}
```

Quelltext 5.5: Additionsausdruck extrahieren

Der Quelltext 5.5 zeigt einen Methodenaufruf, dessen Argument ein Additionsausdruck ist. Dieser Ausdruck soll extrahiert und durch die neue Variable `t` substituiert werden. Die Menge der berechneten Transformationen enthält die folgenden vier Transformationen: Deklaration hinzufügen, Komposition extrahieren sowie Datendistribution und Datendistribution^{-1}. Die letzten beiden Transformationen heben sich gegenseitig auf, sodass nur die zwei erstgenannten übrig bleiben. Anstatt den Additionsausdruck zu löschen und eine Kopie von ihm für die notwendige Zuweisung zu erzeugen, wird dadurch der Ausdruck wiederverwendet.

Wie im Abschnitt Wunschanpassungen zum Einfügen/Extrahieren von Ausdrücken (in Abschnitt 5.1.4) erläutert wurde, sind u. U. Wunschanpassungen notwendig, damit eine Umgestaltung berechnet werden kann. Hierzu gehört die Transformation *Transitivität*, die bspw. dann durchgeführt wird, wenn ein `Literal`, `Name` oder eine `Variablenreferenz` extrahiert werden soll. Da die berechneten Transformationen auf der Abstraktion A_0 angewendet werden, die die Wunschanpassung noch nicht enthält, müssen die Transformationen, die durch die Wunschanpassung durchgeführt wurden, ebenfalls auf A_0 bzw. dem Quelltext durchgeführt werden.

Deklarationen anpassen

Wenn ein Ausdruck extrahiert wird, erzeugen die berechneten Transformationen Deklarationsanweisungen ohne Initialisierung der deklarierten Variable. Zusätzlich wird eine Zuweisungsanweisung erzeugt, um die Variable zu initialisieren. Beide Anweisungen sind notwendig, aber etwas unelegant, wie im Quelltextbeispiel 5.4 an der Variablen `a` zu sehen ist. Um beide Anweisungen zusammenzufassen, wird daher auch die Transformationen *Umgestaltung umwandeln 1* angewendet. Sie fasst die Deklaration und die Zuweisung so zusammen, dass die Zuweisung als Initialisierung in der Deklaration genutzt wird und auf die Zuweisungsanweisung verzichtet werden kann.

Umgekehrt werden Variablen nicht immer einzeln und ohne Initialisierung deklariert. Die Transformation *Deklaration entfernen* basiert jedoch auf Deklarationen, die diese einfache Struktur besitzen. Hier müssen dann die Transformationen der Wunschanpassung erneut durchgeführt werden (siehe Abschnitt 5.1.4). Falls eine Variable entfernt werden soll, die eine Initialisierung besitzt oder in einer Deklaration steht, die noch weitere Variablen deklariert, werden die Transformationen *Deklaration umwandeln 1* bzw. *2* angewendet. Durch sie können Deklarationen so herausgelöst werden, dass die Deklaration einer einzelnen Variablen durchgeführt werden kann. Gegebenenfalls müssen übrig gebliebene, aufgespaltene Deklarationen wieder zusammengefügt werden, um einen Quelltext zu erzeugen, der dem ursprünglichen Quelltext möglichst ähnlich sieht.

Reihenfolgen von Operanden und Anweisungen

Die Reihenfolge der Anweisungen wird nach jeder Umgestaltung angepasst. Fand keine Änderung der Reihenfolge statt, so hat diese Anpassung keine Auswirkung. Andernfalls werden die Anweisungen der neuen Reihenfolge entsprechend angeordnet.

Anweisungen und Operanden werden dann umsortiert, wenn eine Transformation zum Ändern der jeweiligen Reihenfolge existiert. Dabei wird auf möglicherweise entfernte oder neu hinzugekommene Anweisungen Rücksicht genommen. Falls Operanden substituiert wurden, wird dies ebenfalls beim Umsortieren berücksichtigt.

Reihenfolge der Transformationen

Die Anwendungsreihenfolge der Transformationen wird in der vorgestellten Arbeit entsprechend Tabelle 5.6 durchgeführt. Dabei werden die Transformationen anhand ihres Typs der Priorität nach geordnet (hoch (1) bis gering (4)) und anschließend durchgeführt. Je nach Implementierung kann auch eine andere Priorität der Transformationen festgelegt werden. Diese Reihenfolge hat sich jedoch als diejenige herausgestellt, die die wenigsten Implementierungsabhängigkeiten zwischen den Transformationen verursacht.

5.3.3 Zusätzliche Quelltextsemantik

Neben den genannten Transformationen sind unter bestimmten Bedingungen noch weitere Transformationen notwendig, um die Programmsemantik zu erhalten. Diese Transformationen wurden bisher nicht erwähnt, da sie keinen gewichtigen Teil an dieser Arbeit ausmachen.

Priorität (1 = höchste)	Transformation
1	Transitivität
1	Deklaration umwandeln 1, 2
1	Deklaration hinzufügen
1	Datendistribution
2	Komposition einfügen
2	Komposition extrahieren
2	Deklaration entfernen
3	Deklaration umwandeln^{-1} 1, 2
3	Datendistribution^{-1}
4	Operanden umsortieren
4	Anweisungen umsortieren

Tabelle 5.6: Reihenfolge der Transformationen

Operatorpräzedenz

Die Auswertungsreihenfolge eines geschachtelten Ausdrucks richtet sich nach der Operatorpräzedenz in Java (§15.7.3). Wird die Umgestaltung *Ausdruck einfügen* angewendet, kann ein neuer, geschachtelter Ausdruck entstehen. Seine Auswertungsreihenfolge muss der ursprünglichen Auswertungsreihenfolge entsprechen, damit sich das Programmverhalten nicht ändert.

```
int a,b;
void foo() {
  a = 1 + 2;
  b = a * 3;
}
```

Quelltext 5.6: Einfügen der Addition ändert das Programmverhalten

Das Quelltextbeispiel 5.6 besteht aus den Zuweisungen zu den Variablen `a` und `b`. Wird die Umgestaltung *Ausdruck einfügen* auf die Addition angewendet, entsteht der geschachtelte Ausdruck `1+2*3`. In der umgestalteten AST-Struktur ist die Addition der linke Operand der Multiplikation. Dessen Ergebnis weicht jedoch vom ursprünglichen Ergebnis von `a+3` ab, da die Operatorpräzedenz der AST-Struktur des umgestalteten ASTs widerspricht.

Damit sich Operatorpräzedenz und AST-Struktur nicht widersprechen, wird eine Klammerung eingeführt, die die bisherige Präzedenz erzwingt. Im Beispiel wird anstelle der Variablen `a` ein zusätzlicher Klammerungsausdruck im AST eingefügt, dessen Unterknoten die Addition ist, sodass der Ausdruck `(1+2)*3` entsteht.

Typimporte

Beim Extrahieren von Ausdrücken wird die Deklaration einer neuen Variablen notwendig. Diese neue Variable erhält den Typ entsprechend des Rückgabetyps des extrahierten Ausdrucks. Ist dieser Typ eine Klasse, die noch nicht sichtbar in der aktuellen Compilation-Unit ist, muss sie importiert oder auf sie mittels voll qualifizierender Namen zugegriffen

werden. Üblicherweise soll jedoch auf voll qualifizierende Namen zugunsten der Lesbarkeit und Praktikabilität [123, S. 188] verzichtet werden. Daher wird der Klassenimport bevorzugt durchgeführt, sofern er möglich ist. Entsteht durch den Import ein Konflikt, werden voll qualifizierende Namen verwendet.

Überladungsproblem

Schäfer hat in seiner Arbeit [95, S. 91] ein Überladungsproblem beim Einfügen von Ausdrücken vorgestellt, dass bedingt durch das automatische Typumwandeln in Java entstehen kann. Dabei kann durch Änderung der Methodenbindung eine andere Methode als die ursprüngliche aufgerufen werden. Dies kann zu einer Änderung der Programmsemantik führen und muss daher vermieden werden.

```
1  class Dummy {
2    void foo() {
3      long l;      // Wunsch: Einfügen von l
4      l = 42;
5      bar(l);
6    }
7    void bar(int i) {
8      System.out.println("int");
9    }
10   void bar(long l) {
11     System.out.println("long");
12   }
13 }
```

Quelltext 5.7: Überladungsproblem nach Schäfer [95, S. 91]

In Quelltext 5.7 soll die in Zeile 4 definierte Variable `l` in Zeile 5 eingefügt werden. Die resultierende Anweisung dieser Umgestaltung wäre `bar(42);`. Diese Anweisung bindet jedoch nun an die in Zeile 10 deklarierte Methode `bar` und nicht mehr wie zuvor an die Methode in Zeile 7. Durch zusätzliches Hinzufügen des Cast-Operators kann diese Semantikänderung verhindert werden. Korrekterweise entsteht durch die Umgestaltung die folgende Anweisung: `bar((long)l)`. Korrekturen wie diese führt der Ansatz in Anschluss an das Lösen des Gleichungssystems separat durch.

5.3.4 Verbindlichkeit

Das Ergebnis einer Umgestaltung besteht aus mehreren Transformationen, die die intendierte Änderung des Nutzers umsetzen. Ob der Umgestaltungswunsch umgesetzt wird, wird aus dem Gleichungssystem mitsamt Umgestaltungswunsch sowie den berechneten Transformationen ersichtlich.

Der Umgestaltungswunsch ist dann Teil des berechneten Ergebnisses, wenn aus ihm eine Bedingungsgleichung erzeugt wird, die den Wert einer CSP-Variablen auf einen neuen Wert fixiert (siehe Abschnitt 5.1.4). Bei der Auswertung wird diese Gleichungsvariable schließlich interpretiert und aufgrund der Wertänderung eine Transformation erzeugt (siehe Abschnitt 5.3.1). Sofern eine Lösung der Umgestaltung ermittelt werden konnte, entsteht folglich immer mindestens eine Transformation, die direkt aus dem Umgestaltungswunsch resultiert.

Wurde der Wunsch stattdessen mithilfe des Optimierungskriteriums spezifiziert, werden die berechneten Transformationen anschließend überprüft. Es gilt, dass mindestens eine der berechneten Transformationen einen Teil des Umgestaltungswunsches umsetzen muss, damit das Ergebnis als Lösung betrachtet werden kann. Andernfalls wird die berechnete Lösung als *keine Lösung* interpretiert.

Umgestaltungswunsch	Fixierte Variable	Wert	Erzeugte Transformation
Ausdruck einfügen	$c.inl^a$	$wahr$	Komposition einfügen[b]
Ausdruck extrahieren	$c.inl^a$	$falsch$	Komposition extrahieren
Operanden neuordnen	$n.op^c$	anders	Operanden umsortieren
Argumente neuordnen	$n.op^c$	anders	Argumente umsortieren
Anweisungen neuordnen	$c.cp^d$	-	Anweisungen umsortieren

Tabelle 5.7: Vom Wunsch zur Transformation

[a]Die Komposition c repräsentiert den betrachteten Ausdruck.
[b]Soll mehr als ein Ausdruck eingefügt werden, wird stattdessen das Optimierungskriterium modifiziert.
[c]Der Knoten n repräsentiert den betrachteten Operanden.
[d]Die Komposition c repräsentiert die betrachtete Anweisung.

Die oben stehende Tabelle 5.7 zeigt den Weg vom Umgestaltungswunsch zur Transformation, sofern der Wunsch als Bedingungsgleichung umgesetzt wird und somit eine Variable fixiert. Erkennbar ist, dass aus jedem Wunsch eine Transformation erzeugt wird, die den Wunsch umsetzt, sofern eine Lösung des Gleichungssystems berechnet werden kann. Häufig werden aber noch weitere Transformationen erzeugt, die dazu dienen, die Programmsemantik sowie Quelltextkonstrukte zu erhalten.

Zusammenfassung

In diesem Kapitel wurde vorgestellt, wie eine Abstraktion auf ein Constraint-Problem überführt wird. Dabei wurde präsentiert, wie die Freiheitsgrade mithilfe der Constraint-Variablen und -Gleichungen umgesetzt werden. Die Einschränkungen der Freiheitsgrade entstehen durch die Quelltextabhängigkeiten, die in der Abstraktion als Kanten modelliert werden. Der Umgestaltungswunsch ist als eigene Bedingungsgleichung in das Constraint-Problem eingeflossen oder wurde als Modifikation des Optimierungskriteriums umgesetzt. Mithilfe der vorgestellten Lösungsstrategie kann eine Constraint-Lösung gefunden werden, die eine Lösung des Umgestaltungswunsches repräsentiert. Die Constraint-Lösung wird dann interpretiert, um eine Menge von sequenzkongruenten Transformationen zu erhalten, deren Anwendung auf das Ursprungsprogramm schließlich das umgestaltete Programm erzeugt.

Kapitel 6

Evaluation

Die Evaluation ist ein elementarer Bestandteil der Arbeit, da die Fehlerfreiheit des Ansatzes nicht über Beweise gezeigt werden kann. In diesem Kapitel wird anhand von sehr umfangreichen Tests und Vergleichen evaluiert, wie sich der Ansatz im Vergleich zu bestehenden Ansätzen verhält. In Unterkapitel 6.2 wird der Ansatz zunächst auf Grundlage verschiedener Beispiele exemplarisch mit bestehenden Werkzeugen verglichen. Hierfür wird an insgesamt 15 ausgewählten Beispielen gezeigt, wie sich der Ansatz im Vergleich zu Eclipse, IntelliJ IDEA und NetBeans verhält. Diese Vergleiche zeigen einerseits die Vorteile des Ansatzes bezogen auf konkrete Beispiele. Andererseits zeigen sie auch sein Potential für Umgestaltungen im Allgemeinen. Der hier vorgestellte Ansatz wurde in einem Prototypen namens *CodeShaker*[1] implementiert. Anhand von Tests an mehr als 183246 Beispielen aus fünf quelloffenen Programmen wird ausgewertet, wie schnell, verlässlich und praktikabel der Ansatz im Vergleich zu Eclipse ist (Unterkapitel 6.3).

Zunächst wird im Methodikteil (Unterkapitel 6.1) vorgestellt, welchen Forschungsfragen in der Evaluation nachgegangen wird und mit welcher Herangehensweise sie beantwortet werden sollen. Anschließend werden die Ergebnisse der Evaluation vorgestellt. Die Grenzen des Ansatzes werden in Unterkapitel 6.4 erläutert. Im letzten Unterkapitel werden die zuvor vorgestellten Forschungsfragen beantwortet.

CodeShaker

Die Evaluation dieser Arbeit wurde anhand des Prototypen CodeShaker durchgeführt. CodeShaker kann wie folgt vom Evaluationsprototypen des nächstverwandten Ansatzes abgegrenzt werden.

Die von Steimann, von Pilgrim, Thies et al. [113, 110, 111, 118] genutzte Implementierung zum Auswerten ihres Ansatzes heißt Refacola. Refacola besteht aus mehreren Komponenten, wobei der Kern eine Domänensprache ist, die der Abstraktion der vorliegenden Arbeit entspricht. Diese Domänensprache muss selbst zunächst definiert werden und wird dann genutzt, um aus Java-6-Quelltext eine Abstraktion zu erzeugen. Die nächste Komponente von Refacola ist der Regelsatz, der in der Domänensprache spezifiziert wird. Schließlich wird die Domänensprache auch dazu genutzt, die gewünschten Änderungen

[1] Die Implementierung von CodeShaker umfasst 21428 Quelltextzeilen in Java und ca. 400 Zeilen in Prolog. Hinzu kommen 5085 Java-Quelltextzeilen, um den Ansatz systematisch zu testen (jeweils ohne Testfälle).

einer Umgestaltung zu definieren. Bevor die Programmkomponenten genutzt werden können, müssen sie jedoch erst anhand der Definition der Domänensprache generiert werden. CodeShaker verzichtet auf eine eigene Domänensprache und nutzt als Abstraktion stattdessen eine Datenbank von Prolog-Fakten. Auf ihrer Basis werden sowohl der Regelsatz als auch die Umgestaltungswünsche und Optimierungskriterien spezifiziert.

Refacola nutzt eine eigens implementierte Constraint-Optimierung, um einen standardmäßigen Constraint-Löser in akzeptabler Zeit die Umgestaltung berechnen zu lassen. CodeShaker verwendet zum Lösen eines Constraint-Problems einen standardmäßigen Constraint-Löser. Über eine Standardschnittstelle des Constraint-Lösers wird eine eigens auf die Abstraktion angepasste Suchstrategie implementiert, um die Lösungssuche zu optimieren.

Die Vorteile von CodeShaker sind, dass erstens das Generieren von Software wegfällt, die durch die in der Domänensprache formulierten Definitionen spezifiziert wird. Hierdurch vereinfacht sich der Entwicklungsprozess bezogen auf Verständlichkeit, Einarbeitungsaufwand und Änderungsaufwand stark. Zweitens können verschiedene Prolog- und Constraint-Löser genutzt werden, um die Generatorregeln auf die Prolog-Fakten anzuwenden bzw. das Constraint-Problem zu lösen. Hierdurch kann die jeweils leistungsfähigste oder technisch passendste Implementierung eines Lösers auf jeder Plattform genutzt werden. Ferner unterstützt CodeShaker Java 7.

6.1 Methodik

Die Evaluation dieser Arbeit lehnt sich an die Arbeit von Thies [118] an, die deklarative Umgestaltungen mithilfe von Constraint-Programmierung vorstellt. Sie richtet sich ferner nach dem *Ziel-Frage-Metrik*-Ansatz von Basili [8]. Das *Ziel* wird laut Basili auf konzeptioneller Ebene für ein Objekt in einem bestimmten Kontext und mit bestimmten Gründen definiert, wobei das Objekt bspw. ein Produkt oder Prozess ist. Die *Frage* charakterisiert auf operationaler Ebene wie ein bestimmtes Ziel gemessen oder erreicht wird. Fragen beziehen sich dabei auf eine Qualitätseigenschaft des Objekts unter einem bestimmten Gesichtspunkt. Es können mehrere Fragen für ein Ziel definiert werden. Die *Metrik* befindet sich auf der quantitativen Ebene und bezeichnet die Menge aller Daten um die Fragen zu beantworten. Die Daten können sowohl subjektiv (Einschätzung zur Lesbarkeit von Quelltext) als auch objektiv (Zeilenanzahl eines Programm) sein.

Die Ziele der Arbeit sind, einen *korrekten* und *praxistauglichen* Ansatz vorzustellen, der *ohne große Einschränkungen* eingesetzt werden kann und einen *Mehrwert* im Vergleich zu bestehenden Ansätzen liefert. Aus diesen Zielen ergeben sich die folgenden Forschungsfragen.

6.1.1 Forschungsfragen

Die Evaluation soll die folgenden Forschungsfragen klären.

Frage 1: Werden die Umgestaltungen korrekt durchgeführt?

In der Evaluation soll überprüft werden, ob oder inwieweit sich das Programmverhalten der Projekte durch das Umgestalten ändert. Notwendige Eigenschaften für diese Überprüfung

sind, ob die umgestalteten Programme weiterhin fehlerfrei kompiliert werden können, ob sie zur Laufzeit keine neuen oder anderen Ausnahmen werfen und ob ihre Terminierungseigenschaften [24] unverändert geblieben sind. Ferner ist das Ein-/Ausgabeverhalten d. h. die Programmsemantik wichtig und soll nicht geändert werden. Ferner soll überprüft werden, ob der Wunsch der Umgestaltung im neuen Programm umgesetzt wurde.

Frage 2: Unterliegen die Umgestaltungen übermäßigen Einschränkungen?

Ein Teil der Programmsemantik wird durch einen Regelsatz abgebildet. Falls dieser Regelsatz zu restriktiv ist, würde er das Finden der Lösung übermäßig einschränken [105]. Dadurch wären seine Ergebnisse sicherlich weiterhin interessant, aber nur bedingt praxistauglich, da dem Entwickler dann in vielen Situationen keine Lösung angeboten würde. Stattdessen müsste er diese Umgestaltungen dann selbst durchführen. Die Evaluation soll feststellen, ob bzw. inwieweit der Ansatz zu restriktiv ist.

Frage 3: Wie groß ist der Mehrwert der Constraint-Programmierung?

Etablierte Verfahren setzen häufig schablonenartige Lösungsstrategien mit Vor- und Nachbedingungen ein [84, 95], um eine Umgestaltung durchzuführen. Dabei werden notwendige Folgeänderungen nicht gefunden und Fehler erzeugt. Der Constraint-basierte Ansatz berechnet stattdessen eine Lösung unter Kenntnis aller Programmabhängigkeiten und kann somit die notwendige Menge an Transformationen ermitteln, um fehlerfreie Umgestaltungen zu ermöglichen. In der Evaluation soll überprüft werden, wie häufig die Berechnung von Folgeänderungen überhaupt notwendig ist. Falls Folgeänderungen in der Praxis nur äußerst selten benötigt werden, wäre ihr Mehrwert für den Entwickler sehr gering.

Frage 4: Wie hoch sind praxisnahe Laufzeiten?

Die Berechnung von Umgestaltungen mithilfe von Constraint-Programmierung führt zu höherem Berechnungsaufwand, der unter Umständen lange Wartezeiten für den Entwickler bedeutet. Daher soll die Evaluation auch klären, wie komplex typische Umgestaltungsprobleme werden und wie hoch ihr Berechnungsaufwand in der Praxis ist. Bereits jetzt stehen Umgestaltungswerkzeuge in Konkurrenz zur manuellen Durchführung [79, 124] und müssen daher schnell, d. h. innerhalb weniger Sekunden, vom Werkzeug durchgeführt werden können.

6.1.2 Korrektheit

Idealerweise sollten Programmtransformationen wie Umgestaltungen formal auf ihre Korrektheit untersucht werden. Voraussetzung für eine solche Untersuchung wäre die formale Beschreibung der Ausführungssemantik von Java-7-Programmen. Diese könnte dann mit der hier vorgestellten Quelltextabstraktion und dem Regelsatz derart verglichen werden, dass beide Formalismen keine Widersprüche zueinander erzeugen. Tatsächlich liegt eine vollständige Beschreibung der Java-Semantik aber nur textuell in Form der Java-Sprachspezifikation vor [43]. Bestehende Arbeiten zur Formalisierung von Java beziehen sich (a) auf alte Versionen von Java (bspw. Version 1.1 von Nipkow et al. [81]) und sind

bezogen auf diese Version unvollständig und unterstützen (b) Sprachkonstrukte aktueller Versionen nicht (z. B. Generics).

Ferner ist unklar, ob die Formalisierung genügend mit der informellen Spezifikation validiert wurde. Tips Arbeit [121], die sehr ähnlich zu Nipkows Arbeit ist, stellt einen Ansatz für die Umgestaltung *Extrahiere Interface* vor und beweist dessen Korrektheit. Bei der Formalisierung berücksichtigt Tip die Spezifikationen zu Methodenüberschreibungen der Java-Spezifikation §8.4.8.1, übersieht jedoch Sonderfälle in der Spezifikation §8.4.8.4 (siehe Beispiel 6.1) [118, 121]. Auf diese Weise konstruiert Tip einen Beweis, der in seinen zu engen Grenzen korrekt ist, aber nicht die gesamte Spezifikation abdeckt.

Ausgangsprogramm	Umgestaltungswunsch	Erwartetes Programm

`class A {`
`  public void m(A a) {`
`  a.m(a);`
`  }`
`}`
`interface I {`
`  void m( A a);`
`}`
 	
`class B extends A`
`  implements I {}`
	Extrahiere Interface aus `A`	`class A implements J {`
		`  public void m(A a) {`
		`  a.m(a);`
		`  }`
		`}`
		`interface I {`
		`  void m( A a);`
		`}`
		`interface J { }`
		`class B extends A`
		`  implements I {}`

Tabelle 6.1: Beispiel aus Thies [118] für fehlerhafte Umgestaltungen aufgrund unvollständiger Formalisierung

Im Themengebiet der Programmumgestaltungen hat sich daher der Testansatz etabliert [23, 110, 105, 104, 41, 118], um die Korrektheit ohne formale Beweise zu untermauern. Hier wird auf die Durchführung von Tests zurückgegriffen, um die Forschungsfragen eins bis vier zu beantworten. Der Testansatz unterteilt sich dabei in Regressionstests und das systematische Testen von quelloffenen Projekten.

Die Regressionstests werden für die Qualitätssicherung beim Entwickeln von CodeShaker genutzt. Sie stellen exemplarisch sicher, dass der Umgestaltungswunsch vom Ansatz berücksichtigt und im modifizierten Programm eingefügt wird (zu Forschungsfrage 1). Die insgesamt 271 Regressionstests wurden auf verschiedenen Ebenen von CodeShaker erstellt. 24 dieser Testfälle beziehen sich lediglich auf die Quelltextabstraktion und den Regelsatz. Um sie ausführen zu können, ist weder ein Java-Programm notwendig, aus dem die Quelltextabstraktion erzeugt werden würde, noch wird ein Java-Programm oder eine modifizierte Abstraktion erzeugt. Das Ergebnis dieser Testfälle ist ein gelöstes Gleichungssystem, dessen Variablen auf die im Test angegebenen Erwartungswerte überprüft werden.

183 Testfälle nutzen als Eingabe ein kleines Java-Programm, das in die Quelltextabstraktion überführt wird. An jeder der so erzeugten Abstraktionen werden statische Eigenschaften wie die Anzahl der Abstraktionselemente (Kompositionen, Knoten, etc.) und die Zahl aller Typen von Abhängigkeitskanten überprüft.

64 Testfälle testen die Umgestaltungen von Ende zu Ende, d. h. sie erhalten als Eingabe ein Java-Programm und erzeugen ein modifiziertes Programm als Ausgabe, das mit dem erwarteten Programm verglichen wird. Genau wie bei den 24 Testfällen zu Abstraktion und Regelsatz konzentrieren sich die Testfälle darauf, einzelne Regeln oder Sonderfälle zu testen. Dabei wird überprüft, ob die Freiheitsgrade und Beschränkungen des Regelsatzes richtig ausgeprägt sind. Die Testfälle testen sowohl den Fall, wenn es eine korrekte Lösung geben muss, als auch, wenn es keine korrekte Lösung geben darf. 15 dieser Regressionstestfälle werden in Unterkapitel 6.2 vorgestellt und die Ergebnisse von CodeShaker werden mit denen von anderen Werkzeugen wie Eclipse verglichen. Im Vergleich der Werkzeuge kann auch die Forschungsfrage 2 (Existieren übermäßige Einschränkungen?) beantwortet werden.

Das systematische Testen von quelloffenen Projekten soll reale und praxisnahe Ergebnisse liefern und damit Antwort auf die Fragen eins bis vier geben. Die Vorgehensweise ist dabei an die von Steimann, Thies et al. angelehnt [110, 118]. Insgesamt wurden fünf quelloffene Projekte untersucht, die aus verschiedenen Entwicklerkreisen stammen. Durch diese Diversifizierung soll das Risiko verringert werden, dass manche Programmiermuster überdurchschnittlich und andere unterdurchschnittlich oder gar nicht enthalten sind. Damit Fehler erkannt werden können, die möglicherweise durch eine Umgestaltung entstehen, wurde außerdem darauf geachtet, dass die Projekte über eine möglichst hohe Zahl an Testfällen verfügen. Insgesamt wurden 133768 Tests dieser Art durchgeführt und werden genauer in Unterkapitel 6.3 präsentiert.

6.1.3 Übermäßige Einschränkungen

Ob der Ansatz übermäßigen Einschränkungen beim Finden einer Umgestaltung unterliegt, lässt sich am ehesten durch manuelle Inspektion der ursprünglichen Programme erkennen. Falls der Ansatz nicht im Stande ist, eine Lösung zu berechnen, müsste manuell überprüft werden, ob die Umgestaltung tatsächlich nicht durchgeführt werden kann. Ein Teil der Regressionstests wurde dahingehend untersucht, sodass zur Beantwortung der zweiten Frage insgesamt 87 Tests genutzt werden können. Für eine sehr viel größere Zahl an Testfällen ist die manuelle Inspektion jedoch ungeeignet.

Opdyke [84] nutzt stattdessen in seiner Arbeit an Umgestaltungen die Eigenschaft seines Ansatzes, Umgestaltungsprobleme in Vorbedingungen und Transformationen zu unterteilen. Um zu überprüfen, ob seine Vorbedingungen zu restriktiv sind, deaktiviert er jede Vorbedingung einzeln und zeigt so, dass die Transformationen dann zu Fehlern im erzeugten Programm führen. Die Fehler werden nach der Transformation durch den Compiler aufgedeckt. Dieser Ansatz kann auf diese Constraint-basierte Arbeit nicht übertragen werden, da die Regeln des Regelsatzes oft zu komplex sind. Wenn einzelne von ihnen deaktiviert werden, ist außerdem ersichtlich, dass durch ihr Fehlen Fehler bei der Umgestaltung entstehen. Stattdessen müssten die komplexen Regeln in wesentlich feingranularere Teilregeln zerlegt werden, die jede für sich aus der Lösungsberechnung ausgeschlossen werden müssten. Allerdings würde sich eine Synthese zu komplexen Regeln aus eine Untermenge der feingranularen Regeln wahrscheinlich sehr schwierig gestalten.

Wie Soares [102] vorschlägt, wird dieser Ansatz in Unterkapitel 6.3 ebenfalls systematisch mit dem bestehenden Werkzeug Eclipse verglichen. Hierbei wird ausgewertet, wie häufig welcher der beiden Ansätze restriktiver als der jeweilig andere ist und warum. Notwendig für diese Auswertung ist, dass bei beiden Werkzeugen die gleichen Eingabear-

gumente angegeben werden können und auch die jeweiligen Vorbedingungen erfüllt sind. Daher werden in Unterkapitel 6.3 nur die Umgestaltungen *Ausdruck Einfügen*, *Ausdruck Extrahieren* und *Methodensignatur Ändern* betrachtet. Zusätzlich wird die Umgestaltung *Ausdruck Einfügen* in *Ausdruck Einfügen I* und *II* unterteilt. Sie haben verschiedene Vorbedingungen, wobei Variante *I* mit der Umgestaltung in Eclipse vergleichbar ist.

Thies [118] testet seinen Ansatz, indem er für alle Umgestaltungsbeispiele ein Constraint-Problem erzeugt und ohne Umgestaltungswunsch löst. Für jedes Beispiel muss dann eine der so berechneten Lösung die Triviallösung sein, die zu keiner Transformation des ursprünglichen Programms führt. Andernfalls hieße das, dass das Ursprungsprogramm gegen den Regelsatz verstößt und zwangsläufig durch ihn verändert werden würde, selbst ohne Umgestaltungswunsch. In der vorliegenden Arbeit wird ebenfalls eine Umgestaltung ohne Änderungswunsch durchgeführt, die im Folgenden die *Neutrale Umgestaltung* genannt wird. Ihr Optimierungskriterium ist wie bei allen anderen hier genutzten Umgestaltungen, dass sich die Lösung so wenig wie möglich vom ursprünglichen Programm unterscheiden soll. Die Erwartung an die neutrale Umgestaltung ist, dass eine Lösung existiert, aber die von ihr berechnete Transformationsmenge leer ist und sie damit das Programm nicht modifiziert. In Unterkapitel 6.3 wird die neutrale Umgestaltung für alle untersuchten Projekte durchgeführt.

Eine inhärente Beschränkung des Ansatzes ergibt sich aus dem Aufbau und der Struktur der Quelltextabstraktion und des Regelsatzes. Je präziser sie Java modellieren, desto mehr Freiheitsgrade können sie abbilden und bei der Lösungsfindung nutzen. Wenn jedoch die Abstraktion nicht alle Eigenheiten abbildet und an einigen Stellen Näherungen verwendet oder manche Modifikationen gar nicht in Betracht zieht, werden manche Umgestaltungswünsche folglich abgelehnt. Die Frage bleibt dabei allerdings stets, welche Folgeänderungen durch den Entwickler überhaupt toleriert werden und welche nicht. Darüberhinaus muss angenommen werden, dass ihre Antwort zudem vom Geschmack und Programmierstil eines Entwicklers abhängt. Die strukturellen Einschränkungen des Ansatzes werden in Unterkapitel 6.4 genauer erläutert.

6.1.4 Mehrwert durch Constraint-Programmierung

Der Mehrwert der Constraint-Programmierung wird deutlich, wenn er deaktiviert werden kann. Hierzu werden in Anlehnung an Thies [118] Modi genutzt, die unterschiedliche Freiheitsgrade für die Umgestaltungen festlegen. Dabei werden die folgenden beiden Modi verwendet: *Modus I* erlaubt alle bekannten Freiheitsgrade (Operanden/Anweisungen vertauschen, Ausdrücke einfügen/extrahieren). *Modus II* dagegen beschränkt die Freiheitsgrade auf das Vertauschen von Anweisungen. Die anderen drei Freiheitsgrade sind im *Modus II* deaktiviert, sodass hierdurch eine Umgestaltungslösung vom Constraint-Löser entweder nicht oder nur auf einem anderen Wege gefunden werden kann. Die Modi werden in den Tabellen mit *M1* und *M2* abgekürzt.

Ein weiterer Untersuchungsgegenstand zur Beantwortung von Frage 3 sind die tatsächlich berechneten Transformationen jeder Umgestaltungslösung. Bei der Durchführung der systematischen Tests wird ausgewertet, welche Transformationen einen Teil der Lösung bilden. Sofern neben derjenigen Transformation, die den Umgestaltungswunsch repräsentiert, noch weitere Transformationen berechnet wurden, ist dies ein Anhaltspunkt darauf, dass die Freiheitsgrade und die Constraint-Programmierung einen Mehrwert bedeuten.

6.1.5 Laufzeiten

Im Rahmen der systematischen Tests werden neben den Ergebnissen, wie den berechneten Transformationen, auch die Laufzeiten aufgezeichnet. Diese werden zunächst in Form eines Histogramms ausgewertet, um zu ermitteln, wie häufig eine zeitnahe Berechnung einer Umgestaltung möglich ist. Außerdem werden die Ergebnisse in Abhängigkeit der Komplexität der Umgestaltung dargestellt. Da die Berechnungsdauer stark von der Größe des Umgestaltungsproblems abhängt, soll ausgewertet werden, wie stark dieser Zusammenhang in Bezug auf die Methodengröße bzw. Größe der abgeschlossenen Blöcke ist. Je größer eine Methode ist, desto mehr Anweisungen und Ausdrücke enthält sie. Dementsprechend besitzt sie ebenfalls eine umfangreichere Abstraktion mit vielen Knoten und Kanten, sodass das erzeugte Constraint-Problem ebenfalls wächst. Interessant ist, bis zu welcher Methodengröße im Allgemeinen die Umgestaltungsprobleme zeitnah gelöst werden können.

Die Laufzeitmessungen von Java-Programmen unterliegen allerdings einigen Einschränkungen. Der *Garbage Collector* in Java [55] partitioniert den Hauptspeicher in unregelmäßigen Abständen und beeinflusst dadurch die Lösungsberechnung zeitlich [15]. Auch ohne den Garbage Collector wurde von Georges et al. gezeigt, dass reproduzierbare Laufzeiten in Java generell schwierig umgesetzt werden können [40]. Durch Optimierungen in Java (laut Gu et al. [49] und Caches in JDT [34] ergeben sich weitere Ungenauigkeiten bei der Laufzeitmessung.

6.2 Gegenüberstellung verschiedener Werkzeuge

Bei der Gegenüberstellung des Ansatzes werden die einzelnen Umgestaltungen anhand von Beispielen untersucht, die die Feinheiten der Java-Semantik hervorheben, auf die bei Umgestaltungen geachtet werden muss. Der Vergleich wird mit den etablierten Werkzeugen Eclipse[2], NetBeans[3] und IntelliJ IDEA[4] geführt. Die hier dargestellten Resultate lassen sich jedoch auch auf C# in der Entwicklungsumgebung *Microsoft Visual Studio* [73] mit der Erweiterung *ReSharper*[5] [92] übertragen, denn *ReSharper* verhält sich vergleichbar wie die genannten Java-Entwicklungsumgebungen. Andere Umgestaltungswerkzeuge wie RefactorIT [91] werden hier nicht betrachtet, da sie entweder nicht weiter gepflegt wurden oder die hier untersuchten Umgestaltungen nur teilweise oder gar nicht unterstützen. Bei der Auswertung wird das Kreuzsymbol (✗) verwendet um anzuzeigen, dass sich die Programmsemantik durch eine Umgestaltung verändert hat. Das Häkchensymbol (✓) wird genutzt um anzuzeigen, dass die Programmsemantik durch eine Umgestaltung nicht verändert wurde. Das Häkchensymbol wird auch dann verwendet, wenn eine Umgestaltung abgelehnt wird, falls keine semantikerhaltende Lösung gefunden wurde. Bietet ein Werkzeug eine Umgestaltung nicht an und kann es sie daher nicht durchführen, wird dies mit einem Querstrich (-) gekennzeichnet.

In den folgenden Unterkapiteln wird nun einzeln auf die im Ansatz betrachteten Umgestaltungen eingegangen.

[2]Eclipse-Version: 4.4.1
[3]NetBeans-Version: 8.0.1
[4]IntelliJ-IDEA-Version: 13.1.1
[5]Visual-Studio-Version: 2013, ReSharper-Version: 8.0

6.2.1 Ausdrücke einfügen

Beim Einfügen von Ausdrücken verhalten sich die genannten Entwicklungsumgebungen untereinander sehr ähnlich. Sie führen die Umgestaltung nur unter bestimmten Vorbedingungen durch, dazu zählen: (1) Die einzuführende Variable muss eine lokale Variable sein. (2) Sie darf nur ein einziges Mal geschrieben werden. (3) Ihre Definition muss als Initialisierung in der Deklarationsanweisung durchgeführt werden. Bspw. kann im Quelltext 6.1 wegen (3) nicht die Umgestaltung *Ausdruck einfügen* von `x` durchgeführt werden. Die in CodeShaker implementierte Umgestaltung besitzt jedoch keine der genannten Einschränkungen. Sie ist daher an wesentlich mehr Stellen im Programm anwendbar und somit flexibler einsetzbar. Um allerdings den Vergleich zu bestehenden IDEs durchführen zu können, werden in diesem Abschnitt nur solche Umgestaltungen betrachtet, die den genannten Restriktionen genügen. Daher wurde der Quelltext in Tabelle 6.2 entsprechend angepasst.

```
int x;
x = 1;
int y = x;
```

Quelltext 6.1: Einfügen von `x` in etablierten IDEs nicht möglich.

Ausgangsprogramm	Umgestaltungswunsch	Erwartetes Programm

1 `int x = 1;`
2 `int y = x;`
3 `x = 3;`
4 `int z = f(y);`
	Zuweisung `y = x;` einfügen	`int x = 1;`
		`int z = f(x);`
		`x = 3;`

Auswertung	CodeShaker (✓), Eclipse (✗), IntelliJ (✗), NetBeans (✗)	

Tabelle 6.2: Beispiel: Ausdruck einfügen 1 (nach [95, S. 45])

Das erste Beispiel (Tabelle 6.2) besteht aus vier Anweisungen, wobei der Umgestaltungswunsch die Variable `y` der letzten Anweisung durch `x` ersetzen soll. Sie wird kurz *Ausdruck einfügen* genannt, da die rechte Seite der Zuweisung `y = x;` dort eingefügt werden soll, wo `y` verwendet wird. Etablierte Entwicklungsumgebungen ersetzen die Verwendung der Variable `y` einfach mit `x`. Dadurch entsteht in der letzten Zeile des Programms die Anweisung `int z = f(x);`. Da jedoch `x` zuvor in Zeile 3 redefiniert wird, ändert sich durch dieses direkte Ersetzen die Semantik des Programms. CodeShaker vertauscht hingegen zusätzlich die letzten beiden, voneinander unabhängigen Anweisungen. Dadurch werden bestehende Datenabhängigkeiten bewahrt und die Programmsemantik erhalten.

Wie Schäfer in [95, S. 50] zeigt, können ähnliche Abhängigkeiten auch innerhalb von Schleifen entstehen. Diese Situation entsteht, wenn die Anweisungen zwei bis vier innerhalb einer Schleife stünden. Hinzu käme eine schleifenabhängige Abhängigkeit, die von der Definition von `x` (Zeile 3) zur Verwendung in Zeile 2 verliefe (siehe loop-carried Abhängigkeiten, Abschnitt 2.3.1).

Ausgangsprogramm	Umgestaltungswunsch	Erwartetes Programm

`int x = 1;`
`int y = x;`
`int z = x++ + y;`
	Zuweisung `y = x;` einfügen	`int x = 1;`
		`int z = x + x++;`

Auswertung	CodeShaker (✓), Eclipse (✗), IntelliJ (✗), NetBeans (✗)	

Tabelle 6.3: Beispiel: Ausdruck einfügen 2

Ausgangsprogramm	Umgestaltungswunsch	Erwartetes Programm

`int x = 1;`
`int y = x;`
`int z = x++ - g(y);`
	Zuweisung `y = x;` einfügen	`int x = 1;`
		`int t = g(x);`
		`int z = x++ - t;`

Auswertung	CodeShaker (✓), Eclipse (✗), IntelliJ (✗), NetBeans (✗)	

Tabelle 6.4: Beispiel: Ausdruck einfügen 3

Das zweite Beispiel (Tabelle 6.3) besteht aus drei Deklarationsanweisungen, deren Variablen jeweils initialisiert werden. Ziel der Umgestaltung ist, die Verwendung der Variablen `y` durch ihre Initialisierung, also `x`, zu ersetzen. Die etablierten und hier aufgeführten IDEs erzeugen bei dieser Umgestaltung das gleiche Ergebnis mit dem Ausdruck: `x++ + x`. Dieses Ergebnis entsteht, da die Variable durch Ersetzung einfach mit ihrem Initialisierungsausdruck ausgetauscht wird. Dabei werden jedoch Datenabhängigkeiten missachtet und die Semantik verändert. Denn die Auswertungsreihenfolge von Operanden ist in Java von links nach rechts spezifiziert [43]. CodeShaker vertauscht stattdessen die Operanden der Addition und kann dadurch die Umgestaltung semantikerhaltend durchführen.

Das dritte Beispiel (Tabelle 6.4) besteht wieder aus drei Deklarationen, wobei die Variable `y` ersetzt werden soll. Etablierte Werkzeuge ersetzen, wie in den Beispielen zuvor, die Variable `x` einfach durch `y`. Durch die zwischenzeitliche Erhöhung des Werts von `x` wird dabei wieder die Programmsemantik verändert. Im Gegensatz zu dieser Lösung berechnet CodeShaker eine Umgestaltung, bei der zunächst die Variable `y` ersetzt wird. Allerdings wird außerdem der Ausdruck `g(y)` extrahiert und durch die neue Variable `t` ersetzt. Dies ist notwendig, da erstens die Postdekrementoperation vor der (erneuten) Verwendung von `x` stehen muss und zweitens, die Operanden der Subtraktion in diesem Ansatz nicht vertauscht werden können.

Beim vierten Beispiel (Tabelle 6.5) existieren zusätzliche Abhängigkeiten auf Kontrollflusspfaden, die durch Ausnahmen entstehen. Ziel der Umgestaltung ist, die Variable `y` an ihrer Verwendung mit dem Ausdruck `1/i` zu ersetzen. Die üblichen Werkzeuge führen diese Umgestaltung anstandslos durch. Allerdings missachten sie, dass der eingefügte Ausdruck eine Ausnahme produzieren kann, die nun von einer anderen `Try`-Anweisung gefangen wird. Da sich in diesem Beispiel die Ausgabe des Programms ändert, ändert sich folglich die

Ausgangsprogramm	Umgestaltungswunsch	Erwartetes Programm

`try {`
`  int y = 1/i;`
`  try {`
`  int z = y;`
`  }catch(ArithEx e){`
`  println(23);`
`  }`
`}catch(ArithEx e){`
`  println(42);`
`}`
	Zuweisung `y = 1/i;` einfügen	*(Umgestaltung nicht möglich)*

Auswertung	CodeShaker (✓), Eclipse (✗), IntelliJ (✗), NetBeans (✗)	

Tabelle 6.5: Beispiel: Ausdruck einfügen 4 (nach [95, S. 50])

Ausgangsprogramm	Umgestaltungswunsch	Erwartetes Programm

`1 void foo() {`
`2  long l = 42;`
`3  bar(l);`
`4 }`
`5 void bar(int i) {`
`6  println("int");`
`7 }`
`8 void bar(long l) {`
`9  println("long");`
`10 }`
	Zuweisung `l = 42;` einfügen	`void foo() {`
		`  bar ((long) 42);`
		`}`
		`void bar(int i) {`
		`  println("int");`
		`}`
		`void bar(long l) {`
		`  println("long");`
		`}`

Auswertung	CodeShaker (✓), Eclipse (✓), IntelliJ (✓), NetBeans (✗)	

Tabelle 6.6: Beispiel: Ausdruck einfügen 5 (nach [95, S. 94])

Programmsemantik durch die Umgestaltung. CodeShaker kann hier daher keine Lösung des Umgestaltungswunsches anbieten.

Laufzeitausnahmen (engl. runtime exceptions) werden üblicherweise nicht in Programmen behandelt. CodeShaker bezieht allerdings jede explizit definierte Ausnahmebehandlung bei der Betrachtung der Kontroll-/Datenflüsse und somit der Programmabhängigkeiten mit ein. Laufzeitausnahmen, die nicht behandelt werden, werden dementsprechend nicht zur Semantik des Programms gezählt.

Das fünfte Einfügen-Beispiel (Tabelle 6.6) geht auf die Problematik bei der impliziten Typumwandlung ein. Üblicherweise werden ganzzahlige Zahlenliterale als Typ `integer` ausgewertet. Wird ein solches Literal einer Variablen vom Typ `long` zugewiesen, führt Java eine passende, aber implizite Typumwandlung durch. Durch Weglassen dieser Umwandlung kann sich, wie im Beispiel gezeigt, die Semantik ändern.

Ausgangsprogramm	Umgestaltungswunsch	Erwartetes Programm

1 **void** foo() {
2 **int** t = a++;
3 **int** x = a + 2;
4 **if** (...) {
5 **int** y = x;
6 **int** z = t;
7 }
8 }
	Zuweisung `t = a++;` einfügen	**void** foo() {
		
		if (...) {
		int z = a++;
		int y = a + 2;
		}
		}

Auswertung	CodeShaker (✓), Eclipse (✗), IntelliJ (✗), NetBeans (✗)	

Tabelle 6.7: Beispiel: Ausdruck einfügen 6

Der Methodenaufruf in Zeile 3 bindet an die Methode mit der Signatur `bar(long l)`. Wird das Literal `42` einfach anstelle der Variablen `l` im Methodenaufruf eingefügt, so wie es NetBeans tut, ändert sich ihre Bindung: Sie bindet anschließend zur Methode mit der Deklaration `bar(int l)`. Die notwendige, explizite Typumwandlung wird in CodeShaker im Anschluss an das Finden einer Lösung umgesetzt.

Das letzte Einfügen-Beispiel (Tabelle 6.7) zeigt, dass einerseits Blöcke zusätzliche Abhängigkeiten erzeugen und andererseits Abhängigkeiten auch Blockgrenzen überschreiten können. Es wurde bereits im Abschnitt 5.2.1 vorgestellt. Dort wurde es als Beispiel 5.3 zur Verdeutlichung von Abhängigkeiten zwischen verschiedenen Blöcken genutzt.

Wird beim Einfügen der Ausdruck `a++` in den Block verschoben, muss die Datenflussabhängigkeit zum Ausdruck `a + 2` (Zeile 3) beachtet werden. Das Einfügen des Ausdrucks in Zeile 2 bewirkt daher, dass die folgende Zuweisung in Zeile 3 ebenfalls eingefügt wird. Dies geschieht, weil sie nicht hinter den Block verschoben werden kann, da sie bereits innerhalb des Blocks verwendet wird. Zusätzlich muss die Reihenfolge der Zuweisungen im Block (Zeilen 5 und 6) angepasst werden, damit die Datenflussabhängigkeit erhalten bleibt.

6.2.2 Ausdrücke extrahieren

Ausdrücke können in den betrachteten Werkzeugen an vielen Programmstellen ohne Vorbedingungen extrahiert werden. Üblicherweise gibt der Nutzer dabei den Namen der neu eingeführten Variablen an. Ferner arbeiten die Werkzeuge mit einer Analyse zur Klonerkennung. Dadurch wird dann nicht nur der markierte Ausdruck extrahiert, sondern ebenfalls auch äquivalente Ausdrücke im selben Block oder in Unterblöcken. Die folgenden Beispiele gehen auf Semantikänderungen ein, die entstehen, wenn Ausdrücke mit Seiteneffekten extrahiert werden.

Das erste Beispiel (Tabelle 6.8) zeigt, dass durch Seiteneffekte Abhängigkeiten entstehen können, die ihrerseits zu weiteren Änderungen im umgestalteten Programm führen können. Etablierte Werkzeuge erkennen diese Abhängigkeiten nicht und extrahieren die Multiplikation ohne weitere Änderungen. Dadurch wird die ursprünglich vor der Mul-

Ausgangsprogramm	Umgestaltungswunsch	Erwartetes Programm

`int x = 1;`
`int z = x— + x*2;`
	Ausdruck `x*2` extrahieren	`int x = 1;`
		`int t0 = x--;`
		`int t1 = x*2;`
		`int z = t0 + t1;`

Auswertung	CodeShaker (✓), Eclipse (✗), IntelliJ (✗), NetBeans (✗)	

Tabelle 6.8: Beispiel: Ausdruck extrahieren 1

Ausgangsprogramm	Umgestaltungswunsch	Erwartetes Programm

`int x`
`int bar()`
`  throws Exception {`
`  // ...`
`}`
`void foo() {`
`  x = x++ + 2*bar();`
`}`
	Ausdruck `2*bar()` extrahieren	`int x`
		`int bar()`
		`  throws Exception {`
		`  // ...`
		`}`
		`void foo() {`
		`  int t0 = x++;`
		`  int t1 = 2*bar();`
		`  x = t0 + t1;`
		`}`

Auswertung	CodeShaker (✓), Eclipse (✗), IntelliJ (✗), NetBeans (✗)	

Tabelle 6.9: Beispiel: Ausdruck extrahieren 2

tiplikation durchgeführte Postdekrementoperation nun nach ihr durchgeführt. Folglich ändert sich der Wert der Variablen `x` durch diese Änderung. CodeShaker beachtet hingegen diese Abhängigkeit und extrahiert ebenfalls die Postdekrementoperation. Hierdurch wird gewährleistet, dass die Datenabhängigkeit von `x` weiterhin besteht.

Das zweite Beispiel (Tabelle 6.9) verhält sich ähnlich zum ersten. Durch eine Abhängigkeit darf nicht nur die Multiplikation allein extrahiert werden. Hier entsteht diese Abhängigkeit allerdings durch den Ausnahmekontrollfluss, der durch den Methodenaufruf `bar()` ausgelöst werden kann. Dadurch kann die Methode `foo()` ebenfalls außerordentlich abgebrochen werden. Ändert sich jedoch die Reihenfolge der Additionsoperanden, würde sich auch der Nachzustand von `x` ändern, sofern eine Ausnahme entsteht. Folglich extrahiert CodeShaker den ersten Additionsoperanden und behält damit ihre Ausführungsreihenfolge und Semantik bei.

Beispiel 3 (Tabelle 6.10) betrachtet die Grenzen von Blöcken beim Extrahieren von Ausdrücken. Bei dieser Umgestaltung muss neben der Extraktion eine Blockanweisung erstellt werden. Dieser neue Block ersetzt und enthält den Methodenaufruf `foo(x++);`. Der Block enthält außerdem die Deklaration und Zuweisung der neuen Variablen. Denn würden beide neuen Elemente außerhalb bzw. vor der Alternativanweisung stehen, wird

Ausgangsprogramm	Umgestaltungswunsch	Erwartetes Programm

`void foo(int x) {`
`  x = 0;`
`  if (...)`
`  foo(x++);`
`  foo(x);`
`}`
	Ausdruck `x++` extrahieren	`void foo (int x) {`
		`  x = 0;`
		`  if (...) {`
		`  int t = x++;`
		`  foo(t);`
		`  }`
		`  foo(x);`
		`}`

Auswertung	CodeShaker (✓), Eclipse (✓), IntelliJ (✓), NetBeans (✗)	

Tabelle 6.10: Beispiel: Ausdruck extrahieren 3

Ausgangsprogramm	Umgestaltungswunsch	Erwartetes Programm

`x = 1;`
`y = (x==1?2:5) + x++;`
	Operanden der Addition vertauschen	`x = 1;`
		`int t = (x==1?2:5);`
		`y = x++ + t;`

Auswertung	CodeShaker (✓), Eclipse (-), IntelliJ (✗), NetBeans (✗)	
Bemerkung	Umgestaltung nicht verfügbar in Eclipse	

Tabelle 6.11: Beispiel: Operanden vertauschen 1

die Datenabhängigkeit zum zweiten Methodenaufruf `foo(x)` verletzt.

6.2.3 Operanden vertauschen

Das Vertauschen von Operanden wird nicht von allen Entwicklungsumgebungen unterstützt. Wie bei den bisherigen Umgestaltungen ändert sich dabei die Ausführungsreihenfolge von Operationen. Damit einhergehend müssen die Abhängigkeiten unter Operanden derselben Operation beachtet werden. Die folgenden Beispiele zeigen exemplarisch, wie solche Abhängigkeiten aussehen und beim Umgestalten beachtet werden können.

Im ersten der beiden Beispiele (Tabelle 6.11) werden die Operanden einer Addition vertauscht. Der zweite Operand hat einen Seiteneffekt auf die Variable `x`, die jedoch vom ersten Operanden gelesen wird. Die zwischen beiden Operanden bestehende Datenabhängigkeit wird von CodeShaker beachtet, indem zusätzlich zum Vertauschen ein Extrahieren des ersten Operanden durchgeführt wird. Diese Umgestaltung ist daher komplementär zum Beispiel in Tabelle 6.3.

Das zweite Beispiel (Tabelle 6.12) zum Operanden-Vertauschen verhält sich analog zum Ersten. Jedoch kann hier eine Datenabhängigkeit nur durch eine Aliasanalyse erkannt werden. Darauf aufbauend wird die Abhängigkeit bzgl. der Variablen `x` in der Methode `bar` festgestellt: Sie beginnt beim Lesen von `x` und endet beim Methodenaufruf `foo()`.

Ausgangsprogramm	Umgestaltungswunsch	Erwartetes Programm

`int x = 0;`
`int foo() {`
`  return ++x;`
`}`
`int bar() {`
`  return foo() + x;`
`}`
	Operanden der Rückgabeanweisung von `bar()` vertauschen	`int x = 0;`
		`int foo() {`
		`  return ++x;`
		`}`
		`int bar() {`
		`  int t = foo();`
		`  return x + t;`
		`}`

Auswertung	CodeShaker (✓), Eclipse (-), IntelliJ (✗), NetBeans (✗)	
Bemerkung	• Umgestaltung nicht verfügbar in Eclipse. • Aliasanalyse notwendig	

Tabelle 6.12: Beispiel: Operanden vertauschen 2

Ausgangsprogramm	Umgestaltungswunsch	Erwartetes Programm

`int x = 1;`
`int y = 2*x;`
`x++;`
	Deklaration von `y` nach unten verschieben	`int x = 1;`
		`int t = 2*x;`
		`x++;`
		`int y = t;`

Auswertung	CodeShaker (✓), Eclipse (-), IntelliJ (✗), NetBeans (✗)	
Bemerkung	Umgestaltung nicht verfügbar in Eclipse.	

Tabelle 6.13: Beispiel: Anweisungen verschieben 1

Ihretwegen extrahiert CodeShaker beim Vertauschen beider Additionsoperanden zusätzlich den Methodenaufruf. So wird sichergestellt, dass die Datenabhängigkeit weiterhin besteht.

6.2.4 Anweisung verschieben

Wie das Vertauschen von Operanden wird das Verschieben von Anweisungen nicht von allen Entwicklungswerkzeugen unterstützt. Eclipse bietet zwar an, einzelne Zeilen zu verschieben, jedoch ist dies nur eine Editorfunktionalität auf Textbasis. Meistens kann mit ihr eine Anweisung verschoben werden, die sich nur über eine Zeile erstreckt. Ist sie länger, müssen die zugehörigen Zeilen zusätzlich manuell ausgewählt werden. Da diese Editorfunktionalität keinerlei Syntax oder gar Programmabhängigkeiten beachtet, wird sie hier nicht als Umgestaltung eingestuft. Analog zu den vorangegangenen Beispielen gehen die folgenden drei Tabellen exemplarisch auf die Probleme beim Verschieben von Anweisungen ein.

Im ersten Beispiel (Tabelle 6.13) soll die mittlere Anweisung nach unten verschoben werden. Die mittlere Anweisung ist eine Deklaration, die eine Initialisierung besitzt. Jedoch

Ausgangsprogramm	Umgestaltungswunsch	Erwartetes Programm
1 **int** x = −1; 2 **while** (**true**) { 3 **if** (...) 4 **break**; 5 x = 2; 6 } 7 f(x);	Alternative nach unten verschieben	*(Umgestaltung nicht möglich)*
Auswertung	CodeShaker (✓), Eclipse (-), IntelliJ (✗), NetBeans (✗)	
Bemerkung	Umgestaltung nicht verfügbar in Eclipse	

Tabelle 6.14: Beispiel: Anweisungen verschieben 2

Ausgangsprogramm	Umgestaltungswunsch	Erwartetes Programm

int x = 0;
void calc(**int** y) {
x = x * y − 1;
}
void bar () {
calc(1);
calc(2);
}
	Methodenaufruf `calc(1)` nach unten verschieben	*(Umgestaltung nicht möglich)*

Auswertung	CodeShaker (✓), Eclipse (-), IntelliJ (✗), NetBeans (✗)	
Bemerkung	• Umgestaltung nicht verfügbar in Eclipse. • Aliasanalyse notwendig	

Tabelle 6.15: Beispiel: Anweisungen verschieben 3

existiert eine Datenabhängigkeit zwischen der Initialisierung und der letzten Anweisung im Programm (`x++;`). Folglich wird der zugewiesene Ausdruck der Initialisierung extrahiert, sodass die Inkrementanweisung vor die Deklaration der Variablen gezogen werden kann.

Das zweite Beispiel (Tabelle 6.14) zeigt, dass abzweigende Kontrollflüsse zu Datenabhängigkeiten führen können. Eine Datenabhängigkeit beginnt bei der Definition von `x` in Zeile 1 und endet beim Methodenaufruf in Zeile 7. Sie entsteht, wenn die Schleife bereits beim ersten Durchlauf durch die `Break`-Anweisung beendet wird. Eine weitere Datenabhängigkeit beginnt bei der zweiten Zuweisung in Zeile 5 und endet wieder beim Methodenaufruf. Sie entsteht, wenn die Schleife mindestens einmal komplett durchlaufen wird. Aufgrund der zweiten Abhängigkeit darf die Zuweisung `x = 2;` nicht über die Kontrollflussverzweigung nach oben verschoben werden. Folglich ist das Verschieben der Alternative nach unten hier nicht umsetzbar. Komplementär dazu dürfte diese Zuweisung auch nicht nach unten verschoben werden, wenn sie ursprünglich vor der Sprunganweisung gestanden hätte. Ana-

Ausgangsprogramm	Umgestaltungswunsch	Erwartetes Programm

`void s(A a, A b){...}`
 	
`void f(A[] p, A[] q){`
`  int i = 0;`
`  while(i < p.length)`
`  s(p[i], q[i++]);`
`}`
	Parameter von `s(A a, A b)` vertauschen	`void s(A b, A a){...}`
		
		`void f(A[] p, A[] q){`
		`  int i = 0;`
		`  while(i < p.length)`
		`  {`
		`  A t = p[i];`
		`  s(q[i++], t);`
		`  }`
		`}`

Auswertung	CodeShaker (✓), Eclipse (✗), IntelliJ (✗), NetBeans (✗)	

Tabelle 6.16: Beispiel: Parameter/Argumente vertauschen 1

loge Problemstellungen können sich auch bei anderen Sprunganweisungen ergeben (siehe [95, S. 51]). Verzweigt sich der Kontrollfluss aufgrund einer Ausnahme, kann eine ähnliche Situation entstehen. Ebenfalls können `Continue`-Anweisungen eine Barriere bilden, über die eine Anweisung nicht verschoben werden darf.

Auf die Relevanz der Informationen einer statischen Programmanalyse wird im letzten Beispiel (Tabelle 6.15) dieses Abschnitts eingegangen. Der Umgestaltungswunsch ist, beide Methodenaufrufe der Methode `calc(...)` zu vertauschen. Allerdings hat die Methode Seiteneffekte auf die Variable `x`, sodass ein Vertauschen den Endzustand der Variablen `x` verändert. Folglich ist die Durchführung des Umgestaltungswunsches nicht möglich und muss von CodeShaker abgelehnt werden.

6.2.5 Parameter/Argumente vertauschen

Das Vertauschen von Parametern einer Methodendeklaration verhält sich sehr ähnlich zum Vertauschen von Operanden. Wurde eine neue Reihenfolge der Parameter festgelegt, wird für jeden zugehörigen Methodenaufruf geprüft, ob seine Argumente entsprechend der neuen Signatur vertauscht werden können. Ist das Vertauschen aller Parameter und Argumente einer Methode möglich, können die Methodensignatur(en) und -aufrufe wie gewünscht geändert werden. Probleme, die beim Tausch von Parametern durch die Typhierarchie entstehen, werden hier nicht betrachtet.

Im Beispiel (Tabelle 6.16) wird ersichtlich, dass das Vertauschen von Argumenten nicht immer problemlos durchgeführt werden kann. Wenn Abhängigkeiten zwischen Argumenten bestehen, müssen zusätzliche Modifikationen am Programm durchgeführt werden, sofern dies überhaupt möglich ist. Im Fall des Beispiels wird ein Argument extrahiert um die Ausführungsreihenfolge der Argumente trotz des Vertauschens beizubehalten. Etablierte Werkzeuge führen die notwendige Überprüfung und zusätzliche Transformation nicht durch.

6.3 Studienergebnisse

Die Gegenüberstellung hat die Eigenschaften und Vorteile des Ansatzes lediglich bezogen auf Spezialfälle gezeigt. Die nun folgenden, systematischen Umgestaltungstests werden an einer Menge von Projekten durchgeführt, die aus dem Open-Source-Bereich stammen. Insgesamt wird gezeigt, dass der Ansatz im Vergleich zu Eclipse häufig besser, aber u. U. auch schlechter abschneidet. Auf die Situationen, in denen der Ansatz schlechter abschneidet, wird im Abschnitt 6.3.6 eingegangen.

6.3.1 Teststrategie

Der Ansatz zum Testen der Umgestaltungen entstand in verschiedenen Vorarbeiten von u. a. Steimann [109, 59, 113] und wurde schließlich von Thies et al. [119] *Umgestaltungswerkzeugtest* (engl. Refactoring Tool Testing), kurz RTT, genannt. In der Nomenklatur des RTT-Ansatzes wird jenes Programm, das genau eine Umgestaltung an einem Quelltext durchführen kann, mit *Umgestaltungswerkzeug* bezeichnet. CodeShaker und Eclipse können verschiedene Umgestaltungen durchführen und sind daher in diesem Sinne keine Umgestaltungswerkzeuge. Unter der Angabe einer konkreten Umgestaltung kann allerdings *Ausdruck einfügen mit CodeShaker* als Umgestaltungswerkzeug verstanden werden. Der RTT-Ansatz testet ein solches Umgestaltungswerkzeug und wird in drei Komponenten aufgeteilt. Als erste Komponente stellt die Ablaufsteuerung eine Ausführungsschleife dar, die alle Werkzeuge an allen potentiellen Umgestaltungsstellen im untersuchten Projekt ausführt und die Ergebnisse protokolliert. Die potentiellen Umgestaltungsstellen findet die zweite Komponente, die als Menge von Adapter bezeichnet wird. Zu jedem Umgestaltungswerkzeug existiert ein Adapter, der die Umgestaltungsstellen im Projekt detektiert. Die dritte Komponente des RTT-Ansatzes ist ein Satz von Testorakeln, der genutzt wird, um die Anwendung der Umgestaltungswerkzeuge auszuwerten und interpretierbare Ergebnisse zu liefern.

Der Ablauf der systematischen Tests beginnt mit der Analyse jedes Projekts auf potentielle Stellen im Quelltext, an denen eine Umgestaltung durchgeführt werden kann. Dabei dienen sowohl die Quelltexte von jedem Projekt als auch die Testfälle seiner Test-Suite als Grundlage. Anschließend wird an jeder dieser Stellen die passende Umgestaltung durchgeführt und zunächst ausgewertet, ob und mit welchem Ergebnis der Quelltext umgestaltet werden konnte. Zum Durchführen der Umgestaltungen werden CodeShaker und Eclipse verwendet. Die Auswertung jedes Tests erfolgt durch mehrere Testorakel, die zu fünf Testergebnissen führen können (siehe Abbildung 6.1). Als Testorakel wird dabei (1) das jeweilige Umgestaltungswerkzeug selbst verwendet, da es eine Umgestaltung liefern, verweigern oder bei der Berechnung abstürzen kann. Ein weiteres Testorakel ist (2) der Compiler, der das modifizierte Programm kompilieren kann oder eine Ausnahme wirft. Anschließend wird (3) die Test-Suite als letztes Testorakel genutzt, die entweder die gleichen oder veränderte Testresultate liefert. Trotzdem das Durchführen der Test-Suite für jede Umgestaltung den Rechenaufwand enorm erhöht[6] , ist dieser Schritt notwendig, um die Korrektheit des Ansatzes einschätzen zu können. Ändert sich das Programmverhalten aus Sicht der zugehörigen Test-Suiten, ist dies ein Indiz auf einen fehlerhaften Umgestaltungsansatz. Im

[6]Die Ausführungsdauer einer Test-Suite betrug bis zu 120 Sekunden. Um alle 183246 Tests durchzuführen, wurden bis zu elf Rechner für ca. fünf Wochen parallel eingesetzt.

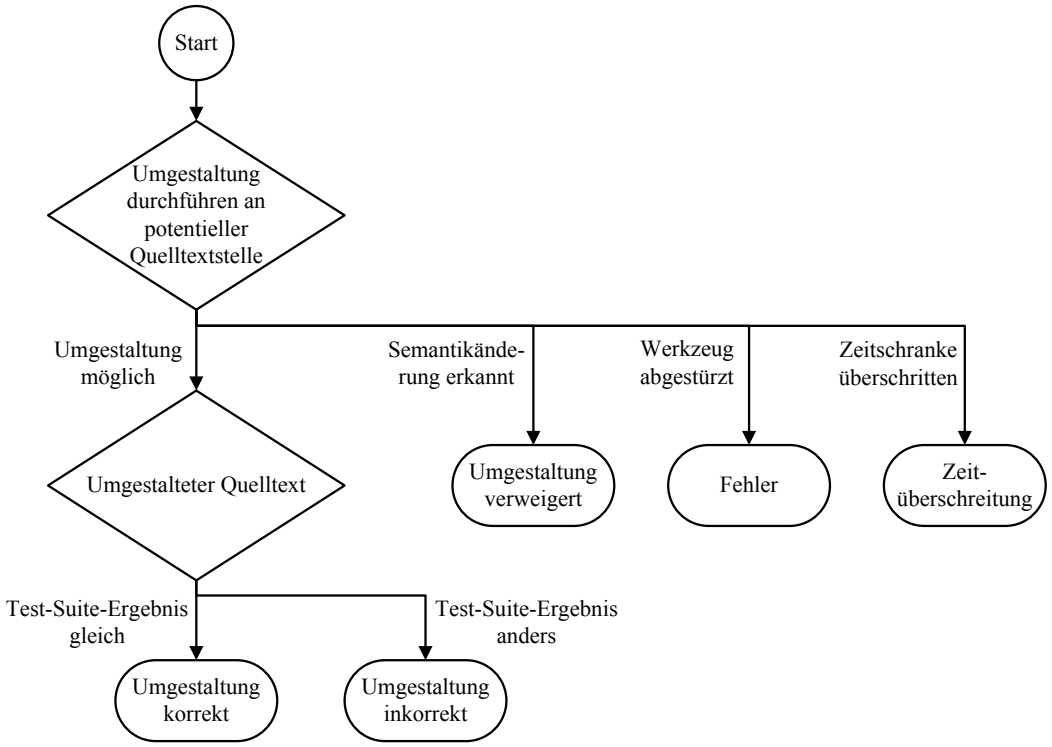


Abbildung 6.1: Evaluationsstrategie

Gegensatz zur vorliegenden Arbeit verzichten Steimann und Thies auf die standardmäßige Durchführung der Test-Suites. Steimann et al. führen gar keine Test-Suiten [110] durch und Thies tut dies nur stichprobenhaft für wenige, ausgewählte Fälle [118].

Stürzt Eclipse oder CodeShaker durch einen Fehler ab oder entsteht eine Ausnahme während der Testdurchführung oder des Build-Vorgangs, lautet das Ergebnis *Fehler*. Wird die Durchführung der Umgestaltung aufgrund einer drohenden Semantikänderung von Eclipse oder CodeShaker abgelehnt, wird ebenfalls kein umgestaltetes Programm erzeugt und das Ergebnis lautet *Umgestaltung verweigert*. Wenn die Lösungssuche des Gleichungssystems eine Zeitschranke überschreitet, wird ebenfalls keine Umgestaltung erzeugt (Ergebnis *Zeitüberschreitung*). Falls jedoch ein umgestaltetes Programm erzeugt werden kann, wird in einem nächsten Schritt überprüft, ob es weiterhin die gleiche Semantik wie das Original hat. Um eine Semantikänderung festzustellen, wird die zum Programm gehörende Test-Suite sowohl am Originalprogramm als auch am umgestalteten Programm durchgeführt. Ergibt der Vergleich der Ergebnisse keine Unterschiede, wird von einer erfolgreichen Umgestaltung ausgegangen und das Ergebnis lautet *Umgestaltung korrekt*. Andernfalls ist das Ergebnis *Umgestaltung inkorrekt*, da eine Semantikänderung festgestellt werden konnte. Für die neutrale Umgestaltung gilt eine weitere Bedingung: Wenn beim Durchführen einer neutralen Umgestaltung eine Lösung gefunden wird, die eine Änderung enthält, wird die Umgestaltung mit *Umgestaltung verweigert* bewertet.

6.3.2 Untersuchte Projekte

Die untersuchten Projekte wurden fast nicht[7] modifiziert und sind in Tabelle 6.17 aufgelistet. Die einzigen Änderungen des Quelltexts betreffen Dateipfadanpassungen und das Hinzufügen einer `AllTests`-Klasse, mit deren Hilfe alle JUnit-Testfälle gestartet werden können. Neben den Testabdeckungsmaßen wurde außerdem untersucht, wie hoch der Anteil der Implementierung ist, der sich innerhalb von Methoden mit mehr als 40 Anweisungen befindet; dies kann Einfluss auf die Berechnungsdauer haben. Die Tabelle 6.18 listet die fünf untersuchten Projekte samt ihrer Größen in Quelltextzeilen auf.

Projektname	SLOC		Anteil langer[a]	Testabdeckung	
	Programm	Test-Suite	Methoden	Instruktionen	Zweige
Collections[b]	27,3k	32,8k	11,4%	80,3%	71%
Math[c]	82,8k	88,4k	21,1%	93%	84%
Google Gson[d]	7,2k	13k	10,7%	87,9%	76,3%
Joda Convert[e]	2k	2,1k	12,0%	69,4%	84,3%
JSoup[f]	10,3k	4,6k	6,5%	83,4%	71,8%

Tabelle 6.17: Untersuchte Projekte und ihre Größe

[a]mehr als 40 Anweisungen
[b]Collection-Bibliothek von Apache, Version 4.0 [32]
[c]Mathe-Bibliothek von Apache, Version 3.4.1 [33]
[d]Ehemaliges Google-Projekt zur Anbindung von JSON, Version 2.3.1 [42]
[e]Serialisierungs- und Konvertierungsbibliothek, Version 1.7 [57]
[f]Java Html-Parser, Version 1.8.2 [58]

Abdeckungsmaß	Abdeckung				
	∅	2%-Quantil	5%-Quantil	10%-Quantil	Median
Instruktionen	96,1%	85,2%	86,1%	88,7%	98,1%
Zweige	94,9%	81,2%	83,3%	84,8%	98,1%

Tabelle 6.18: Tatsächliche Abdeckung bezogen auf die umgestaltete Java-Klasse

Die Schlussfolgerung, dass bei unveränderten Testergebnissen eine erfolgreiche Umgestaltung vorliegt, ist allerdings nur bedingt verlässlich. Denn die Qualität der Test-Suite bestimmt, ob *jede* mögliche Semantikänderung überhaupt festgestellt werden kann. Um die Verlässlichkeit dieser Schlussfolgerung zu erhöhen, wurden in der Untersuchung nur solche quelloffenen Programme verwendet, die über eine umfangreiche Test-Suite verfügen. Als Kriterium zur Wahl der Projekte wurden Testabdeckungsmetriken gewählt, wobei die Test-Suiten jedes Projekts eine Mindestabdeckung der Instruktionen und Zweige von je 70% besitzen müssen. Darüberhinaus werden nur solche Java-Klassen auf mögliche Stellen zur Umgestaltung untersucht, die ein Abdeckung beider Metriken von mindestens 80% erreichen. In Tabelle 6.18 werden die tatsächlichen Abdeckungsmaße angegeben, die aus der Menge aller Java-Klassen hervorgeht, in denen eine Umgestaltung durchgeführt wurde.

[7] Nur beim Projekt Math wurden permanent insgesamt 65 der 5835 Testfälle deaktiviert, um die Laufzeit der Test-Suite auf ein Siebentel zu reduzieren. Hierdurch verringerten sich die Testabdeckungsmaße um ca. 1-2%.

6.3.3 Umgestaltungsstellen

Durch eine Suche auf dem abstrakten Syntaxbaum (kurz: AST) mit geeigneten Filtern wurden solche Stellen im Quelltext gefunden, die passende AST-Knoten für jede Umgestaltung bieten. Hierdurch konnte die Gesamtmenge aller AST-Knoten auf solche Knoten eingeschränkt werden, die potentiell für die jeweilige Umgestaltung geeignet sind. Zu jeder dieser Umgestaltungsstellen müssen ggf. Umgestaltungsparameter erzeugt werden, sodass möglicherweise auf jeder Quelltextstelle mehrere Umgestaltungen durchgeführt werden können. Eine Quelltextstelle inklusive ihrer Umgestaltungsparameter wird *Umgestaltungsstelle* genannt.

Umgestaltung	Anzahl der untersuchten Stellen[a]					
	Projekt					Gesamt
	Collections	Math	Google Gson	Joda Convert	JSoup	
Ausdruck einfügen	1138	10k	1096	73	327	12634
Ausdruck extrahieren	10k	10k	2774	321	1468	24563
Operanden vertauschen	36	1568	16	0	8	1628
Anweisungen verschieben	10k	10k	5280	356	1447	27083
Parameter vertauschen	104	804	48	4	16	976
Neutrale Umgestaltung	2840	7165	786	92	422	11305

Tabelle 6.19: Anzahl der untersuchten Stellen nach Umgestaltung und Projekt

[a]Alle Stellen im Projekt, aber max. 10000

Die Tabelle 6.19 zeigt die Anzahl der untersuchten Umgestaltungsstellen in den Projekten. Um die Suche durchzuführen und die potentiellen Umgestaltungsstellen zu finden, muss jede Stelle innerhalb eines `Methodenrumpfs` oder in einem `Initializer`-Block stehen. Außerdem dürfen die Methodenrümpfe oder `Initializer`-Blöcke keine `Blockanweisungen` enthalten, da diese noch nicht unterstützt werden (siehe Unterkapitel 4.2.1). Ferner mussten stets die Mindestgrößen der Testabdeckungsmaße eingehalten werden. Die folgenden spezifischen Filter wurden außerdem für die einzelnen RTT-Adapter genutzt.

RTT-Adapter: Ausdruck einfügen

Die Umgestaltung *Ausdruck einfügen* wird für alle `Deklarationsfragmente`, die über eine Initialisierung verfügen, und für alle Zuweisungsanweisungen, (`ExpressionStatements` deren Ausdruck eine `Zuweisung` ist) durchgeführt. In beiden Fällen darf die zugewiesene

Variable nicht mit dem Schlüsselwort `final` ausgezeichnet sein, denn `Final`-Variablen werden häufig in anderen Kontexten verwendet, wie anonymen Threads. Würden solche Variablen eingefügt und ihre Deklaration und Initialisierung entfernt werden, könnten sie innerhalb eines anonymen Threads nicht mehr genutzt werden. Sofern die rechte Seite der Zuweisung oder der Initialisierung ein Methodenaufruf ist, gilt, dass JDT dessen Bindungsinformation des Aufrufs zur Deklaration ermitteln können muss. Das ist notwendig, da in manchen Fällen beim Einfügen eine `Cast`-Operation erstellt werden muss, damit die Typinformationen nicht verloren gehen. Aus jeder so gefundenen Umgestaltungsstelle wird der Wunsch erstellt, dass der zugewiesene Ausdruck an allen Lesestellen des Datenflusses eingefügt werden soll.

RTT-Adapter: Ausdruck extrahieren

Ausdrücke wurden an allen Stellen im Quelltext extrahiert, die einen der folgenden JDT-Ausdrücke darstellen: `Array"-Creation`, `Assign"-ment`, `Cast"-Ex"-pression`, `Class"-Instance"-Creation`, `Con"-di"-tional"-Ex"-pression`, `In"-fix"-Ex"-pression`, `Instance"-of"-Ex"-pression`, `Me"-thod"-In"-vo"-ka"-tion`, `Paren"-thesized"-Ex"-pression`, `Post"-fix"-Ex"-pression`, `Pre"-fix"-Ex"-pression`, `Su"-per"-Field"-Access`, `Su"-per"-Me"-thod"-In"-vo"-ca"-tion`, und `Va"-ria"-ble"-De"-clara"-tion"-Ex"-pression`. Diese Ausdrücke werden jedoch nicht zwangsläufig als mögliche Extraktionsstellen angesehen. Für jeden Ausdruck A müssen außerdem die folgenden Bedingungen erfüllt sein. A selbst darf kein `Super-` oder `This`-Konstruktoraufruf sein, da Konstruktoraufrufe immer die erste Anweisung eines Konstruktorrumpfs sein müssen. Ein Elternausdruck von A darf keine `Annotation`, Post- oder Preinkrement- oder -dekrementausdruck sein, da diese Elternausdrücke entweder kein Teil des ausgeführten Quelltexts sind oder ihre direkten Kindausdrücke syntaktisch nicht extrahiert werden können. Aus dem zuletzt genannten Grund darf der direkte Elternausdruck ebenfalls keine `Ausdrucksanweisung` oder ein `Variablendeklarationsfragment` sein. Falls der direkte Elternausdruck von A eine `Zuweisung` ist, darf der Elternausdruck dieser Zuweisung keine Ausdrucksanweisung sein, da ein Extrahieren der rechten Seite nur eine weitere Zuweisung einfügen würde und daher nicht unterstützt wird. Falls die direkte Elternanweisung von A eine `For-Schleife` ist, darf A der zu extrahierende Ausdruck A ausschließlich ein Unterausdruck der Schleifenanweisung sein. Ist A ein Teil der Initialisierung oder der Aktualisierung, wird die Extraktion von A hier nicht unterstützt. Diese Bedingungen sind vergleichbar mit den Vorbedingungen der Eclipse-Umgestaltung *Extrahiere lokale Variable*. Für jeden Ausdruck, der diesen Bedingungen genügt, wird der Umgestaltungswunsch erstellt, dass er extrahiert und statt seiner eine neue, lokale Variable eingefügt und zuvor erzeugt, deklariert und initialisiert werden soll.

RTT-Adapter: Operanden vertauschen

Um Stellen in den Projekten zu finden, die für die Umgestaltung *Operanden vertauschen* geeignet sind, werden alle `Infixausdrücke` I in Betracht gezogen. Falls der Rückgabetyp von I `String` oder `char` ist, wird I ausgeschlossen, da Operationen mit diesen Typen nicht kommutativ sind und folglich ein Vertauschen nicht möglich ist. I wird ferner nur dann einbezogen, wenn der Operator von I für eine Addition, Multiplikation, Gleichheit, Ungleichheit oder für die bedingten oder unbedingten logischen Operationen Und oder

Oder steht. In JDT können Infixausdrücke auch über mehr als zwei Operanden verfügen. Sie repräsentieren dann bspw. eine Addition von mehr als zwei Variablen. Verfügt der Infixausdruck lediglich über zwei Operanden, wird aus der Umgestaltungsstelle der Wunsch erzeugt, dass beide Operanden vertauscht werden sollen. Besitzt der Infixausdruck drei Operanden, wird für jede Permutation der Operanden (mit Ausnahme der initialen Permutation) ein Umgestaltungswunsch erstellt. Hat I mehr als drei Operanden, werden die Operanden an erster, letzter und mittlerer Stelle permutiert. Die Operanden dazwischen verbleiben an ihren Ausgangspositionen im Umgestaltungswunsch. Auf diese Weise werden höchstens fünf Umgestaltungswünsche pro Umgestaltungsstelle erzeugt. Andernfalls könnte die Zahl der Wünsche schnell sehr hoch (exponentieller Anstieg) und damit kaum beherrschbar werden.

RTT-Adapter: Anweisung verschieben

Zu den verschiebbaren Anweisungen, für die Umgestaltungsstellen gefunden werden, gehören fast alle Unterklassen der `Statement`-Klasse von JDT mit Ausnahme der `Blockanweisung`, da einzeln stehende Blockanweisungen nicht unterstützt werden. Für jede verschiebbare Anweisung werden – falls möglich – zwei Umgestaltungswünsche erstellt, sodass die Anweisung einmal auf- und das andere Mal abwärts um jeweils eine Anweisung im Quelltext verschoben werden soll. Wenn die Anweisung bereits am Anfang oder Ende ihres Blocks steht, kann nur ein Wunsch von beiden erstellt werden.

RTT-Adapter: Parameter vertauschen

Aus der Menge aller Methodendeklarationen werden Umgestaltungsstellen gefiltert, die die folgenden Eigenschaften besitzen: Die Methodendeklarationen müssen mehr als einen Parameter besitzen und unter ihnen darf kein Parameter mit variabler Stellenanzahl sein (`§8.4.1`). Außerdem darf die Klasse der Methodendeklaration weder eine überschreibende noch eine überschriebene Klasse sein. Diese Einschränkung hat den Grund, dass eine überschreibende Klasse möglicherweise eine Klasse aus einer externen Bibliothek überschreibt. Eine Änderung der Methodensignatur wäre hier nicht möglich. Andernfalls könnte die Umgestaltung natürlich durchgeführt werden, die jedoch eine Analyse der Klassenhierarchie notwendig macht. In dieser Analyse müsste ermittelt werden, welche anderen Methodensignaturen aus überschriebenen und überschreibenden Klassen und Schnittstellen durch die Umgestaltung angepasst werden müssten. Auf diese Analyse wurde in der Implementierung verzichtet. Sie schränkt jedoch die Aussagekraft der Tests nicht ein und kann später nachgerüstet werden. Analog zum Vertauschen von Operanden werden beim Vertauschen von Parametern zu jeder Umgestaltungsstelle möglicherweise mehrere Wünsche erstellt. Besitzt die Methode lediglich zwei Parameter, wird nur ein Wunsch erstellt. Bei drei oder mehr Parameter werden fünf Umgestaltungswünsche erstellt. Im Fall von mehr als drei Parametern werden der erste, der letzte und ein mittlerer Parameter miteinander vertauscht, wobei die anderen Parameter ihre Position nicht ändern sollen.

RTT-Adapter: Neutrale Umgestaltung

Die Probanden der neutralen Umgestaltung sind die Gesamtmenge der `Methodenrümpfe` und `Initializer`-Blöcke. Für sie gelten nur die oben genannten und für alle RTT-

Adapter gültigen Einschränkungen.

6.3.4 Ergebnis und Vergleich

In diesem Unterkapitel werden die Ergebnisse der Umgestaltungen ausgewertet. Die Gesamtzahl der 183245 Ergebnisse besteht aus 133768 Ergebnissen von CodeShaker in den Modi der Freiheitsgrade M1 und M2. Hinzu kommen 11305 Ergebnisse, die mit der neutralen Umgestaltung berechnet wurden. Die restlichen 38172 Ergebnisse wurden mit Eclipse erzeugt.

Die Umgestaltung *Ausdruck einfügen* wurde in zwei Teile untergliedert. Die Version I steht für das Einfügen von Initialisierungsausdrücken, die Teil einer Variablendeklaration sind. Dies entspricht dem Einfügen von Ausdrücken so wie es in Eclipse implementiert ist. Version I ermöglicht somit einen Vergleich der Umgestaltung *Ausdruck einfügen* zwischen Eclipse und CodeShaker. Version II steht für das Einfügen von Ausdrücken, die die rechte Seite einer Zuweisungsanweisung sind. Diese Art des Einfügens von Ausdrücken wird in Eclipse nicht unterstützt und daher hier einzeln aufgeführt.

Ergebnisart	Korrekt	Verweigert	Inkorrekt	Zeit-überschreitung	Fehler
Ausdruck einfügen I	3367 (43,8%)	2070 (26,9%)	378 (4,9%)	1860 (24,2%)	2 (0%)
Ausdruck einfügen II	523 (10,5%)	2896 (58,2%)	302 (6,1%)	1254 (25,2%)	5 (0,1%)
Ausdruck extrahieren	17733 (71,2%)	771 (3,1%)	152 (0,6%)	6235 (25,0%)	9 (0%)
Operanden vertauschen	1147 (70,4%)	19 (1,2%)	19 (1,2%)	407 (25,0%)	36 (2,2%)
Anweisungen verschieben	3067 (11,3%)	20764 (76,7%)	644 (2,4%)	2592 (9,6%)	17 (0%)
Parameter vertauschen	786 (79,5%)	0 (0%)	0 (0%)	200 (20,5%)	0 (0%)
Neutrale Umgestaltung	11040 (97,7%)	0 (0%)	0 (0%)	265 (2,3%)	0 (0%)

Tabelle 6.20: Ergebnisse der Umgestaltungen in CodeShaker M1

Die Gesamtergebnisse (Tabelle 6.20) variieren je nach Umgestaltung stark. Während die Erfolgsquote (korrektes Ergebnis) beim Extrahieren von Ausdrücken mit 71,2% sehr hoch ist, ist sie beim Verschieben von Anweisungen mit 11,3% sehr gering und das obwohl jede Umgestaltungsstelle die Umgestaltung grundsätzlich ermöglichen würde. Das bedeutet, dass bspw. nur solche Anweisungen verschoben werden sollen, die auch tatsächlich in einem Block mit mehreren Anweisungen stehen. Der Grund für die Spannbreite der Erfolgsquote sind daher nur die tatsächlichen Quelltextabhängigkeiten. Das Extrahieren wird wesentlich seltener durch sie behindert als das Verschieben von Anweisungen.

Interessant sind die Ergebnisse beim Einfügen von Ausdrücken. Die Ergebnisse beider Varianten unterscheiden sich hauptsächlich durch eine Differenz von ungefähr 33% der

Testresultate Korrekt und Verweigert. In einem Drittel der Fälle verweigert Variante II also die Umgestaltung, die von Variante I jedoch durchgeführt werden. Sofern die Ergebnisse einen Rückschluss auf die reale Möglichkeit zulassen, ist das Einfügen von Zuweisungen folglich wesentlich seltener in der Praxis möglich als das Einfügen von Deklarationen.

Die Auswertung der neutralen Umgestaltung zeigt, dass abgesehen von einigen Zeitüberschreitungen immer eine Lösung berechnet wurde, die keine Änderung bewirkt.

Umgestaltung	Ergebnisart	Anzahl der Vorkommnisse (relativ)[a]		
		CodeShaker M1	CodeShaker M2	Eclipse
Ausdruck einfügen I	Korrekt	3361 (57,9%)	2232 (38,4%)	3384 (58,2%)
	Verweigert	2069 (35,6%)	2944 (50,7%)	1555 (26,8%)
	Inkorrekt	379 (6,5%)	633 (10,9%)	870 (15,0%)
Ausdruck einfügen II	Korrekt	520 (14,0%)	475 (12,8%)	-
	Verweigert	2894 (77,9%)	2942 (79,2%)	-
	Inkorrekt	302 (8,1%)	299 (8,0%)	-
Ausdruck extrahieren	Korrekt	17665 (95,1%)	17468 (94,0%)	17522 (94,3%)
	Verweigert	766 (4,1%)	771 (4,1%)	654 (3,5%)
	Inkorrekt	152 (0,8%)	344 (1,9%)	407 (2,2%)
Operanden vertauschen	Korrekt	1147 (96,9%)	1148 (96,9%)	-
	Verweigert	19 (1,6%)	19 (1,6%)	-
	Inkorrekt	19 (1,6%)	18 (1,6%)	-
Anweisung verschieben	Korrekt	3067 (12,5%)	1325 (5,4%)	-
	Verweigert	20755 (84,8%)	22139 (90,5%)	-
	Inkorrekt	644 (2,6%)	1002 (4,1%)	-
Parameter vertauschen	Korrekt	744 (100%)	744 (100%)	744 (100%)
	Verweigert	0 (0%)	0 (0%)	0 (0%)
	Inkorrekt	0 (0%)	0 (0%)	0 (0%)

Tabelle 6.21: Ergebnissvergleich von CodeShaker M1, M2 und Eclipse

[a]Betrachtet werden hier nur Ergebnisse, deren Testergebnis sowohl in CodeShaker als auch in Eclipse korrekt, inkorrekt oder verweigert ist.

Ein Vergleich der Ergebnisse von CodeShaker M1 und M2 sowie Eclipse, falls möglich, wird in Tabelle 6.21 präsentiert. Die eingangs beschriebene Evaluationsstrategie wurde für jede potentielle Stelle einmal mit Eclipse und einmal mit CodeShaker M1 sowie M2 durchgeführt. Diese Tabelle fasst nur Ergebnisse zusammen, die sowohl im Eclipse- als auch in den CodeShaker-Umgestaltungswerkzeugen vom Typ *korrekt*, *inkorrekt* oder *verweigert* sind.

Beim Vertauschen von Parametern verhalten sich Eclipse und CodeShaker identisch. Die Zahl der korrekten Ergebnisse der beiden anderen mit Eclipse vergleichbaren Umgestaltungen ist in CodeShaker ähnlich gut wie in Eclipse: Das Einfügen von Ausdrücken I hat mit 57,9 zu 58,2% im Vergleich zu Eclipse leicht schlechtere Ergebnisse und das Extrahieren von Ausdrücken profitiert leicht in CodeShaker M1 mit 95,1 zu 94,3% bei den erfolgreichen Umgestaltungen. Deutlich spürbarer sind die Verbesserung in Bezug auf die Verringerung der inkorrekten Ergebnisse im Vergleich zu Eclipse. Das Einfügen von Ausdrücken I erzeugt in CodeShaker M1 nur 6,5% inkorrekte Ergebnisse, während es in

Eclipse 15% sind. Obwohl die nominalen Werte beim Extrahieren mit 0,8 zu 2,2% geringer sind, bleibt die relative Minimierung der inkorrekten Ergebnisse von ca. 2,5 ähnlich hoch.

Interessant ist auch der Vergleich der beiden Modi von CodeShaker. Bei den Umgestaltungen *Ausdruck einfügen I* und *Anweisung verschieben* ist klar ersichtlich, dass Modus 1 wesentlich häufiger korrekte Ergebnisse berechnen kann als Modus 2 (57,9% zu 38,4% und 12,5% zu 5,4%). Im Gegenzug steigt die Zahl der verweigerten Umgestaltungen in Modus 2 beinah in gleicher Höhe, wie die Zahl der korrekten Ergebnisse abnimmt. Allerdings kann auch beobachtet werden, dass die Zahl der inkorrekten Ergebnisse in Modus 2 gleich hoch oder merkbar höher liegt als in Modus 1. Ursache für diese Verschiebung ist, dass in Modus 2 alle Freiheitsgrade, bis auf das Verschieben von Anweisungen, deaktiviert sind. Statt Operanden zu tauschen oder Ausdrücke zu extrahieren oder einzufügen, kompensiert CodeShaker in Modus 2 diese fehlenden Freiheitsgrade durch das Verschieben von Anweisungen. Die hierdurch entstehenden Änderungen sind wesentlich umständlicher und führen dadurch häufiger zu inkorrekten Umgestaltungen. Beim Vertauschen von Anweisungen können sehr schnell inkorrekte Ergebnisse entstehen, da Seiteneffekte und die aus ihnen resultierenden Datenflüsse nicht in der Abstraktion repräsentiert sind.

Umgestaltung	Eclipse	CodeShaker M1		
		Korrekt	Verweigert	Inkorrekt
Ausdruck einfügen I	Korrekt	2999 (51,6%)	326 (5,6%)	64 (1,1%)
	Verweigert	266 (4,6%)	1260 (21,7%)	31 (0,5%)
	Inkorrekt	103 (1,7%)	484 (8,3%)	284 (4,9%)
Ausdruck extrahieren	Korrekt	17531 (94,1%)	25 (0,1%)	10 (0%)
	Verweigert	16 (0%)	637 (3,4%)	3 (0%)
	Inkorrekt	162 (0,9%)	106 (0,6%)	139 (0,7%)
Parameter vertauschen	Korrekt	776 (100%)	0 (0%)	0 (0%)
	Verweigert	0 (0%)	0 (0%)	0 (0%)
	Inkorrekt	0 (0%)	0 (0%)	0 (0%)

Tabelle 6.22: Ergebnisvergleich von CodeShaker und Eclipse in gegenseitiger Abhängigkeit

Einen genaueren Vergleich zwischen Eclipse und CodeShaker M1 zeigt Tabelle 6.22. Im direkten Vergleich von Eclipse und CodeShaker zeigt die Tabelle, dass in den meisten Fällen CodeShaker bessere Ergebnisse erzielt als Eclipse. Dargestellt wird die Ergebnismatrix beider Werkzeuge, die angibt, wie sich ein Werkzeug im Vergleich zum jeweils anderen verhält. Bspw. erzeugt Eclipse in 103 Fällen inkorrekte Ausdruck-Einfügen-Ergebnisse, unter der Voraussetzung, dass CodeShaker M1 korrekte Ergebnisse berechnet hat. Im Folgenden wird die Kurzschreibweise in Tupelnotation (ECL:inkorrekt, CS:korrekt) genutzt, um einzelne Zellen einer Umgestaltung zu referenzieren. Die Tupelelemente setzen sich aus einem Kurzbezeichner für die Umgebung Eclipse (ECL) oder CodeShaker (CS) und durch Doppelpunkt getrennt dem Ergebnis der Umgestaltung zusammen.

Interessant bei der Auswertung ist die Gegenüberstellung der Zellen (ECL:inkorrekt, CS:korrekt) mit (ECL:korrekt, CS:inkorrekt) und (ECL:verweigert, CS:inkorrekt) und ihrem vertauschten Pendant (ECL:inkorrekt, CS:verweigert). Das erste Zellenpaar gibt Auskunft darüber, ob eine inkorrekte Umgestaltung in dem jeweils anderen Werkzeug korrekt durchgeführt werden konnte. Das zweite Zellenpaar zeigt an, ob Umgestaltungen,

die in einem Werkzeug inkorrekt durchgeführt wurden, im jeweils anderen Werkzeug verweigert wurden. Wünschenswert ist natürlich, dass ehemals inkorrekte Umgestaltungen korrekt durchgeführt werden. Dieses Ziel wird im Folgenden als Ziel I bezeichnet. Wenn dies jedoch nicht möglich ist, sollte das Werkzeug als alternatives Ziel II die Umgestaltung verweigern. Ein inkorrektes Ergebnis ist am wenigsten gewünscht, da es zu Verhaltensänderungen im Programm verursacht, die später gefunden und behoben werden müssen. Der Zellenpaarvergleich zeigt, dass die Umgestaltungen Ausdruck einfügen bzw. extrahieren das Ziel I leicht bzw. signifikant verbessern konnten. Während in 103 zu 64 Fällen beim Ausdruck Einfügen das Werkzeug CodeShaker M1 häufiger bessere Ergebnisse berechnet, gelingt dies beim Extrahieren von Ausdrücken sogar in 162 zu 10 Fällen besser. Der Gewinn von CodeShaker M1 ist beim Einfügen/Extrahieren von Ausdrücken deutlich merkbar. Hier wird Ziel II von 484 auf 31 und von 106 auf 3 Fälle jeweils stark verbessert. Eclipse und CodeShaker M1 verhalten sich jedoch bezüglich des Vertauschens von Parametern gleich. Die Unterschiede in den absoluten Zahlen zwischen den Tabellen ergeben sich daraus, dass CodeShaker M1 und M2 unterschiedlich viele Zeitüberschreitungen besitzen, die jeweils herausgefiltert wurden.

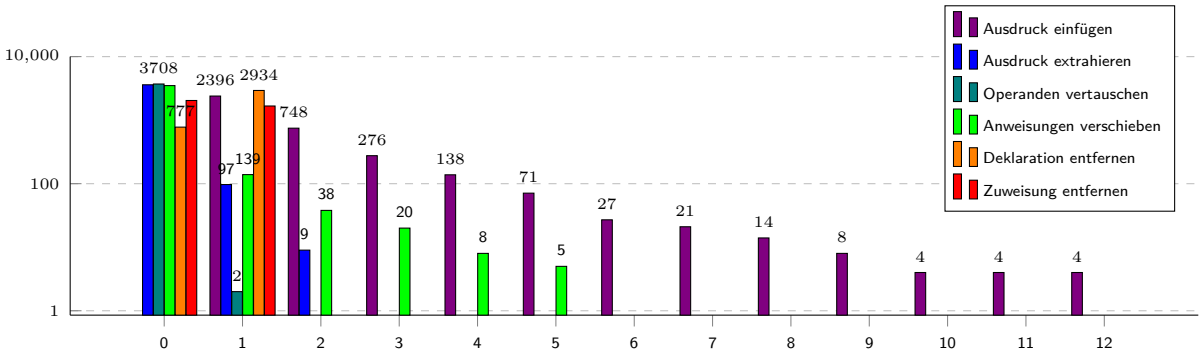
Nun stellt sich die Frage, warum CodeShaker überhaupt schlechtere Ergebnisse im Sinne der Ziele I und II berechnen kann. Schließlich beachtet es wesentlich mehr Programmabhängigkeiten und mindestens jene, die Eclipse auch beachtet. Die zwei Hauptgründe dafür sind, dass CodeShaker (1) in manchen Fällen Typen nicht korrekt inferieren kann und (2) sich Fehler in CodeShaker häufiger auswirken, da durch Folgeänderungen in Umgestaltungen von CodeShaker auch Folgefehler entstehen können. Im Gegensatz zu CodeShaker berechnet Eclipse keine Folgeänderungen und kann daher auch keine Folgefehler erzeugen. Im später folgenden Abschnitt 6.3.6 werden die Ursachen von inkorrekten Umgestaltungen von CodeShaker genauer erläutert.

In Tabelle 6.22 kann schließlich auch noch das Zellenpaar mit seinen Zellen (ECL:korrekt, CS:verweigert) und (ECL:verweigert, CS:korrekt) verglichen werden. Festzustellen ist, dass CodeShaker M1 häufiger als Eclipse Umgestaltungen verweigert (326 zu 266, 25 zu 16 und 49 zu 0). Allerdings wurde diese Entwicklung erwartet, da CodeShaker wesentlich mehr Abhängigkeiten beachtet und somit auch wesentlich vorsichtiger beim Umgestalten vorgeht. Letztendlich kann auch nicht ausgeschlossen werden, dass einige dieser Ergebnisse doch vielleicht inkorrekt sind und nur durch eine mangelhafte Testabdeckung als korrekt eingeschätzt wurden.

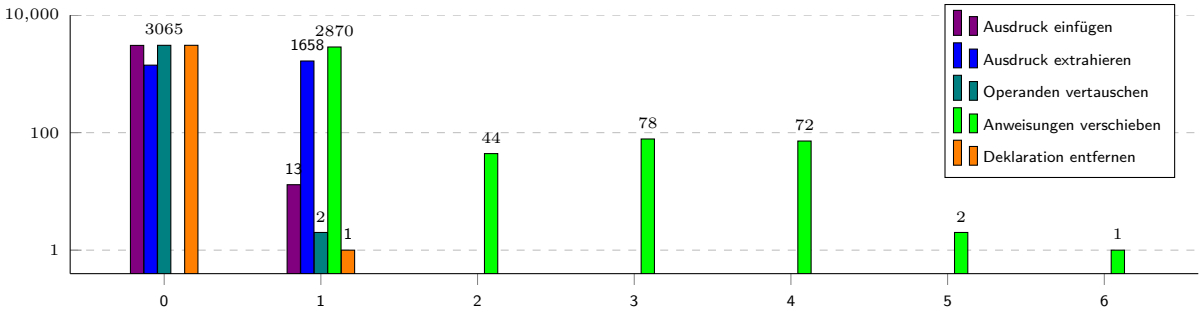
Wie die Histogramme in Abbildung 6.2 zeigen, werden für jeden erfolgreichen Umgestaltungswunsch die zugehörigen Transformationen erzeugt. Zusätzlich entstehen meist noch Folgetransformationen, die dabei helfen, die Quelltextsemantik unverändert zu lassen. Die Zahlen in den Histogrammen geben für jede Programmtransformation an, wie häufig sie für eine Umgestaltungsart aufgetreten sind. In den Histogrammen 6.2a und 6.2c konnten bezogen auf das Verschieben von Anweisungen bzw. das Vertauschen von Operanden bis zur Häufigkeit 17 bzw. 26 noch weitere Datenpunkte beobachtet werden. Alle anderen Graphiken sind vollständig.

Die Umgestaltung *Ausdruck einfügen* (I und II) erzeugt die meisten Folgeumgestaltungen bezogen auf die Anzahl der Typen und deren Anzahl. Im Zuge dieser Umgestaltung wird bspw. in 97 Fällen ein weiterer Ausdruck extrahiert und in 38 Fällen ändert sich in zwei Programmblöcken die Reihenfolge von Anweisungen.

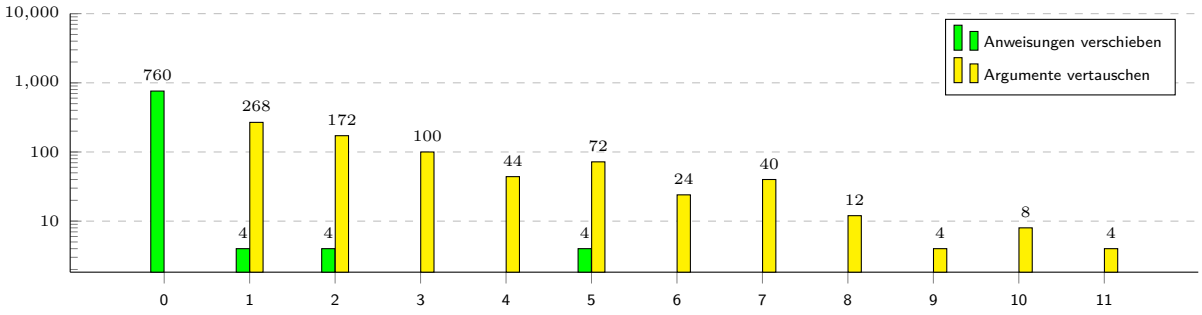
Andere Umgestaltungen bewirken eher selten Folgetransformationen. Hierzu zählen das



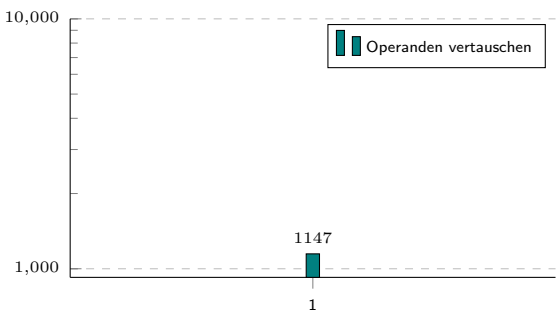
(a) Wunsch: Ausdruck einfügen



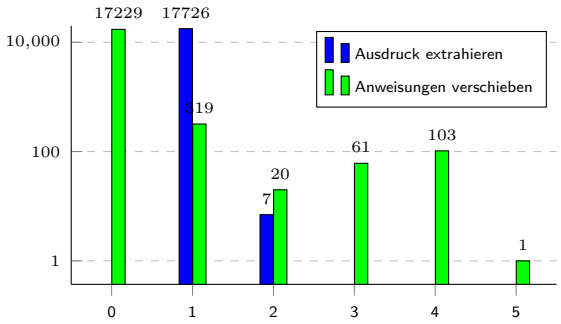
(b) Wunsch: Anweisung verschieben



(c) Wunsch: Argumente vertauschen



(d) Wunsch: Operanden vertauschen



(e) Wunsch: Ausdruck extrahieren

Abbildung 6.2: Die Histogramme beziehen sich auf die Ergebnisse von CodeShaker. Die Ordinate gibt die Zahl der erfolgreichen Tests und die Abszisse gibt die Häufigkeit der Transformationen an.

Extrahieren von Ausdrücken und das Vertauschen von Operanden, die entweder lediglich Anweisungen verschieben oder gar keine Folgeänderungen verursachen. Möglicherweise wären jedoch bei einer größeren Menge an Umgestaltungsstellen für das Vertauschen von Operanden weitere Transformationen entstanden, wie das Extrahieren von Ausdrücken.

6.3.5 Berechnungsdauer und Komplexität

Die Zahl der Zeitüberschreitungen (Tabelle 6.20) liegt bei ca. 25% und ist damit relativ hoch. Das Ergebnis resultiert hauptsächlich daher, dass das größte Projekt, Math, viele sehr lange Methoden besitzt (Tabelle 6.17). Die längste Methode besitzt 641 Anweisungen und die 191 von 7391 (2,6%) Methoden mit 40 oder mehr Anweisungen machen 21,1% des Quelltexts aus. Zum Vergleich implementieren im Projekt Collections die 35 der 2876 (1,2%) Methoden mit mehr 40 Anweisungen nur 11,4% des Programms, also nur rund halb soviel im Vergleich zum Projekt Math. Da die Zahl der Umgestaltungsstellen proportional zur Zahl der Anweisungen ist, liegen dementsprechend im Projekt Math auch wesentlich mehr Umgestaltungsstellen innerhalb langer Methoden und verursachen so Zeitüberschreitungen. Das Einhalten von Code-Richtlinien könnte dabei helfen, dieses Problem zu beheben. Martin schlägt z. B. Methodenlängen von weniger als 20 Anweisungen vor, verweist aber auf für ihn vorbildlichen Quelltext von Kent Beck, der mit 4-5 Anweisungen pro Methode auskommt [70]. Die Richtlinien von CSharp weisen darauf hin, dass Methoden möglichst nicht mehr als sieben Anweisungen besitzen sollen [26, AV1500]. Obwohl auch vagere Aussagen über die empfohlene Länge von Methoden existieren, wie bspw. *«maximal eine Bildschirmseite»*, wird vielerorts zu kurzen Methoden geraten, sodass NetBeans bereits Warnungen anzeigt, wenn eine Methode über mehr als 10 Anweisungen verfügt.

Die Praktikabilität des Ansatzes wurde mit handelsüblichen Rechnern untersucht. Bei der Untersuchung wurden bis zu elf Rechner für ca. fünf Wochen parallel eingesetzt, um die Tests durchzuführen. Aufgrund der hohen Anzahl an Umgestaltungsstellen und der teilweise sehr langen Durchführungszeiten der Test-Suiten[8] konnten nur so die insgesamt 183246 Testfälle durchgeführt werden. Sieben der elf Rechner waren ungefähr gleich leistungsfähig (Unterschied geringer als 2,5%), während die anderen bis zu Faktor drei langsamer waren. Damit die Laufzeiten aller Rechner miteinander verglichen werden konnten, wurde eine spezielle Testreihe erstellt, auf deren Basis relative Laufzeitunterschiede festgestellt und somit die absoluten Laufzeiten normiert werden konnten. Nichtsdestotrotz ergibt sich aus dieser Vorgehensweise eine erhöhte Varianz der gemessenen Zeiten, da mit dem Einsatz vieler verschiedener Rechner mit teilweise unterschiedlicher Hard- und Software auch viele Fehler- und Störquellen einher gehen.

Für jede durchgeführte Umgestaltung wurde in Tabelle 6.23 neben der Lösungsdauer der Umfang des Umgestaltungsproblems und seine Komplexität für das Gleichungssystem festgehalten. Angegeben ist die Komplexität als Zahl der Anweisungen und Ausdrücke für jede umgestaltete Stelle. Ferner ist angegeben, wie viele Kompositionen und Elemente der Abstraktion insgesamt daraus entstanden sind. Darüberhinaus ist die Größe des zugehörigen Gleichungssystems (Anzahl der Variablen und Gleichungen) angegeben. Obwohl die Zahl der erzeugten Kompositionen in der selben Größenordnung wie die Anweisungen/Ausdrücke liegt, steigt die Zahl der Abstraktionselemente stark an. Der Grund ist, dass für die meisten

[8] Das Ausführen der Test-Suite von Math hat selbst nach Entfernen der 65 zeitintensivsten JUnit-Tests immer noch ca. eine Minute auf den schnellsten Rechnern gedauert.

Umgestaltung	Metrik	Ergebnisse					
		∅	Median	80%-Q.	90%-Q.	95%-Q.	98%-Q.
Ausdruck einfügen	#Anw.	13,5	11	19	26	33	47
	#Ausdr.	26,8	21	40	52	66	97
	#Komp.	21,2	18	31	40	48	58
	#Elemente	737,8	496	989	1513	2178	3721
	#Var.	579,5	467	834	1106	1417	2031
	#Gl.	724,3	563	1053	1387	1842	2790
	$T_{Gl.}$ [ms]	1739	687	2065	4296	8765	11968
	#Lsg.	6,7	3	11	18	25	35
Ausdruck extrahieren	#Anw.	11,4	8	18	25	32	53
	#Ausdr.	23,2	16	37	52	69	103
	#Komp.	17,2	13	27	38	47	55
	#Elem.	585,1	317	759	1283	1921	3739
	#Var.	450,3	315	699	1003	1351	1888
	#Gl.	566,7	369	856	1234	1867	2979
	$T_{Gl.}$ [ms]	1169	319	1295	3017	5305	9660
	#Lsg.	5,8	2	9	16	24	33
Operanden vertauschen	#Anw.	14,8	11	20	34	48	60
	#Ausdr.	41,8	31	79	96	112	114
	#Komp.	23,6	22	32	43	48	63
	#Elem.	1150	630	2140	3008	3592	4467
	#Var.	829,7	603	1614	1900	2156	2389
	#Gl.	1040	751	2028	2362	2675	2921
	$T_{Gl.}$ [ms]	1670	671	2519	4265	5897	15880
	#Lsg.	6,6	3	12	21	27	32
Anweisungen verschieben	#Anw.	12,3	8	17	24	39	67
	#Ausdr.	20,8	13	28	45	69	114
	#Komp.	13,7	10	21	29	38	45
	#Elem.	617,2	271	705	1245	2342	4677
	#Var.	423,5	253	580	925	1581	2340
	#Gl.	510,9	295	689	1125	1831	2979
	$T_{Gl.}$ [ms]	771,4	241	780	1656	3387	6750
	#Lsg.	3,4	2	4	7	12	19
Parameter vertauschen	#Anw.	32,8	19	48	88	112	144
	#Ausdr.	65	34	105	164	231	300
	#Komp.	18,3	15	30	36	40	74
	#Elem.	1516	755	2416	3578	5003	9191
	#Var.	1272	693	2124	2975	4126	5345
	#Gl.	1696	981	2559	3693	6058	8290
	$T_{Gl.}$ [ms]	2253	354	2670	4995	11405	15479
	#Lsg.	22,2	6	27	60	119	176

Tabelle 6.23: Komplexität und Praktikabilität

Kompositionen mindestens drei Fragmente, fünf Knoten und zwei Kanten erzeugt werden. Hinzu kommen Symbole, Klongruppen und Abhängigkeitskanten. Da die Constraint-Variablen und -Gleichungen direkt aus der Abstraktion erzeugt werden, liegen ihre Zahlen wiederum dicht bei einander. Die Größe des Gleichungssystems bzw. der Abstraktion ist ein wichtiges Kriterium, um die Berechnungsdauer zu senken. Wenn die Abstraktion insgesamt in einem weiteren Berechnungsschritt komprimiert werden würde, könnten Umgestaltungen schneller berechnet werden.

Die Zeitangabe bezieht sich auf die Dauer, die zum Lösen des Gleichungssystems gebraucht wird. Aus der Tabelle wird ersichtlich, dass der Großteil (90%) der Umgestaltungsprobleme in weniger als fünf Sekunden Gesamtdauer gelöst werden kann und die durchschnittliche Berechnungsdauer bei zwei Sekunden liegt. Ferner existieren durchschnittlich ungefähr fünf Lösungen für jeden Umgestaltungswunsch. Eine Ausnahme bildet hier das Parameter vertauschen, da hier für jeden Methodenaufruf eine Umgestaltung berechnet wird. Jede dieser Berechnungen erhöht somit die Gesamtdauer. Andererseits kann jede Berechnung mehrere Lösungen besitzen, sodass die Gesamtzahl der Lösungen exponentiell steigt. Eine hohe Zahl von Lösungen ist günstig hinsichtlich des Findens einer optimalen Lösung, kostet jedoch auch immer zusätzlichen Berechnungsaufwand.

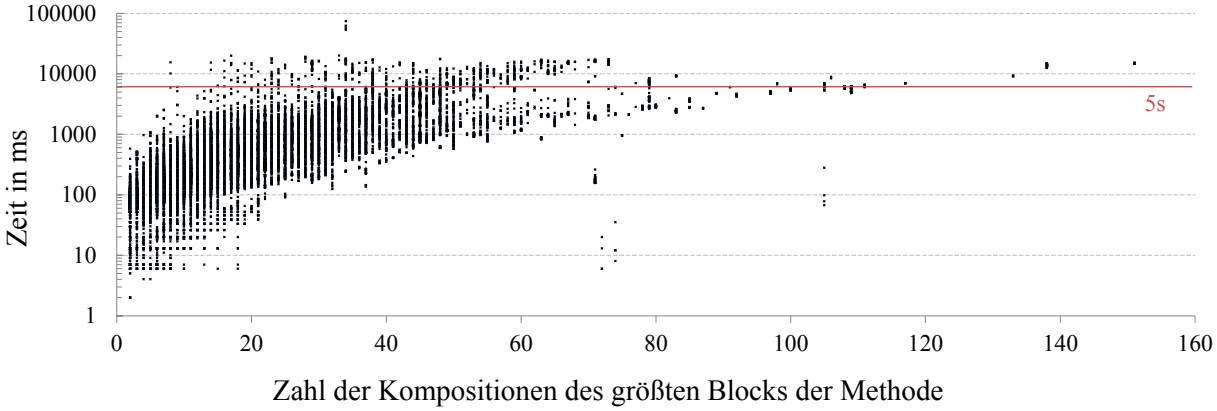


Abbildung 6.3: Zusammenhang zwischen Berechnungsdauer und Größe der umgestalteten Methode.

Die Berechnungsdauer ist direkt abhängig von der Größe der Methode. Da bei der Berechnung für jeden Block einer Methode ein eigenes Teilproblem erstellt wird, richtet sich die Berechnungsdauer vornehmlich nach dem größten Block, da die Dauer exponentiell von der Größe des Blocks abhängt. Abbildung 6.3 stellt die Abhängigkeit der CSP-Berechnungsdauer einer Umgestaltung auf Basis ihrer Komplexität dar. Als Indikator der Komplexität wird die Anzahl der Kompositionen des größten Blocks der Methode verwendet. Bei der Umgestaltung *Parameter vertauschen* wird der größte Block aller berechneten Methoden genutzt. In der Abbildung sind insgesamt 55345 Datenpunkte eingetragen. Aus ihnen kann abgeleitet werden, dass für Methoden, deren größter Block weniger als 40 Kompositionen besitzt, eine Umgestaltung innerhalb von ca. fünf Sekunden berechnet werden kann. Ferner wird ersichtlich, dass der überwiegende Teil aller Methoden in den betrachteten Quelltexten unter dieser Schranke der Ausdrucksanzahl bleibt. Die Zeitschranke für die Berechnungen wurde auf 11 Sekunden[9] festgelegt.

[9] Bezogen auf die sieben schnellsten Rechner. Bei langsameren Rechnern ist die Zeitschranke entspre-

6.3.6 Ursachen für Fehler und inkorrekte Umgestaltungen

Die Tabelle 6.20 zeigt alle Ergebnisse für CodeShaker M1. Die 69 Fehler, die während der Tests entstanden sind, machen einen Anteil von ca. 0,5‰ aus. Die Gründe für die Fehler sind verschieden. Beim Sichten der Fehlerfälle wurde unerwünschtes Verhalten von JDT und dem Java-Compiler entdeckt. Die Typinferenz hat sich als eine der hartnäckigsten Fehlerursachen herausgestellt und war Ursache für einen Großteil der CodeShaker-Regressionstestfälle. In besonderen Situationen sind Ausnahmen oder abweichende Testresultate entstanden, da der erzeugte Quelltext falsche Typen nutzt. Das Beispiel 6.2 zeigt, wie bei einer Extraktion ein falscher Typ hergeleitet wurde.

```
class MyMap<K, V> {
  public void putAll(Map<? extends K, ? extends V> mapToCopy) {
    Set<Map.Entry<? extends K, ? extends V>> map_0 =
      mapToCopy.entrySet();
    for (final Map.Entry<? extends K, ? extends V> e : map_0)
      {...}
  }
}
```

Quelltext 6.2: Extrahieren von mapToCopy.entrySet() erzeugt die Variable map_0 und den Compiler-Fehler *Type mismatch* bei der Initialisierung von map_0.

In anderen Situationen ist eine verlässlich korrekte Typherleitung ebenfalls nur schwer möglich. Dies tritt beispielsweise im Beispiel 6.2 auf, wo nicht nur der Rückgabetyp des extrahierten Ausdrucks `abc()` analysiert, sondern zusätzlich noch der erwartete Typ der Rückgabeanweisung in die Typinferenz miteinbezogen werden muss. Da Typinferenz kein Kernthema dieser Arbeit ist, wurde nur eine vereinfachte Typinferenz implementiert. CodeShaker entscheidet sich in solchen Fällen für einen der möglichen Typen und riskiert damit fehlerhafte Ergebnisse.

```
<T extends Comparable<?>, Serializable> T abc();
Serializable foo() {
  return abc();
}
```

Quelltext 6.3: Extrahieren erzeugt den folgenden Quelltext samt Compiler-Fehler: Type mismatch.

In anderen Fällen hat JDT Fehler verursacht. Das Beispiel 6.4 zeigt eine Situation, in der ein Ausdruck extrahiert werden sollte, der mit Java-Generics arbeitet. Seltsamer Weise wurde beim Ersetzen des Ausdrucks durch eine Variable der Java-Wildcard-Ausdruck nicht entfernt. Folglich ist syntaktisch fehlerhafter Quelltext entstanden, der beim Versuch, ihn zu übersetzen, zum Fehler bei der Auswertung geführt hat. Der Fehler konnte auf Fehlerreport 474922 bei Eclipse zurück geführt werden. Da der Ansatz verschiedene Folgeänderungen zu jeder Umgestaltung berechnen kann, können die genannten Fehler auch bei allen Umgestaltungswünschen entstehen.

```
boolean foo(Object o) {
  boolean b = o instanceof List<?> || o instanceof Set<?>;
  return (b<?>);
```

chend größer.

```
}
```

Quelltext 6.4: Extrahieren mit JDT erzeugt den folgenden, syntaktisch fehlerhaften Quelltext.

Teilweise wurde durch die Umgestaltungen das Programmverhalten so verändert, dass die Test-Suite abstürzte oder nicht mehr terminierte. Im folgenden Beispiel 6.5 konnte beobachtet werden, dass durch Extrahieren der Schleifenbedingung die Test-Suite nicht mehr terminierte. Das Extrahieren führte dazu, dass der Methodenaufruf in der Schleifenbedingung durch eine Variable ersetzt wurde, deren Wert sich innerhalb des Schleifenrumpfs nicht mehr änderte.

```
while (iter.hasNext())
  count++;
```

Quelltext 6.5: Durch Extrahieren der Schleifenbedingung terminierte die Test-Suite nicht mehr.

Eclipse extrahiert ohne Warnung auch die Schleifenbedingung und verhält sich damit genau wie CodeShaker, das den Seiteneffekt nicht erkennt. Aber wenn CodeShaker diese oder ähnliche Extraktionen als Folgetransformationen berechnet, entstehen Fehler, die in Eclipse nicht auftauchen. Im nächsten Beispiel 6.6 ändert sich das Testergebnis, weil die Seiteneffekte nicht beachtet werden. Der Methodenaufruf `l.iterator()` erzeugt eine Instanz, die in der Folge verwendet wird. Wird statt der Verwendung dieser Instanz an jeder ihrer Verwendungsstellen eine neue Instanz erzeugt, ändert sich dadurch das Programmverhalten.

```
void foo(List<E> l1, List<E> l2, E e2) {
  Iterator<E> it = l1.iterator();
  for (E e : l2) {
    it.add(e);
  }
  it.add(e2);
}
```

Quelltext 6.6: Durch Einfügen des Methodenaufrufs `l.iterator()` ändert sich das Programmverhalten.

6.4 Grenzen des Ansatzes

Die Art und Weise, wie die Abstraktion gestaltet ist und auf ein Constraint-Problem abgebildet wird, ist maßgeblich für die Anzahl und Wirkungsweise der Freiheitsgrade. Wenngleich diese Gestaltung der Abstraktion die Freiheitsgrade erzeugt, beschränkt sie diese ebenfalls. In den folgenden Abschnitten soll gezeigt werden, wie sich diese Beschränkungen in Bezug auf konkrete Umgestaltungen auswirken.

6.4.1 Operationen ändern

Der Ansatz beschränkt sich auf Änderungen, die die Reihenfolge von Anweisungen/Operanden betreffen, sowie auf Änderungen bzgl. der Schachtelung von Ausdrücken. Der Ansatz verzichtet bewusst auf das Ändern/Löschen von Operationen und von Abhängigkeiten, da

diese Transformationen nicht sequenzkongruent sind. Daraus resultierend können während einer Umgestaltung die Operationen nicht im Sinne von mathematischen Regeln verändert werden.

```
int a = 1 * (1 - 2);
```
Quelltext 6.7: Ursprung

```
int a = 1 - 2;
```
Quelltext 6.8: Änderung 1

```
int a = -2 + 1;
```
Quelltext 6.9: Änderung 2

In den Quellen 6.7, 6.8 und 6.9 ist dargestellt, wie durch mathematische Umformungen ein Quelltext semantikerhaltend verändert werden könnte. Bei diesen Umformungen werden das Distributivgesetz, die Multiplikation und das Kommutativgesetz angewendet. Außerdem wird im zweiten Schritt die Addition in eine Subtraktion transformiert und der Subtrahend negiert.

```
int a = 1 + 2 - 2;
```
Quelltext 6.10: Vor der Umgestaltung

```
int a = 1; // oder auch + 0
```
Quelltext 6.11: Nach der Umgestaltung

Das Beispiel in den Quellen 6.10 und 6.11 führt eine mathematische Berechnung statisch aus. Dabei wird ein Ausdruck ausgewertet und der verbliebene Ausdruck reduziert, indem eine der Operationen gelöscht wird.

Die in den Quellen 6.7 bis 6.11 präsentierten Beispiele können vom vorgestellten Ansatz nicht umgesetzt werden, da auf eine Manipulation von Ausdrücken/Abhängigkeiten verzichtet wird. Der Vorteil dieser Herangehensweise ist, dass auf diese Weise die Eigenschaft der Sequenzkongruenz genutzt und mit ihrer Hilfe der Erhalt der Semantik gezeigt werden kann.

6.4.2 Komplexe Strukturänderungen

Im Bereich der Compiler-Optimierung existieren vielfältige Umgestaltungen, die dazu führen, den Maschinentext schneller und/oder mit weniger Speicherbedarf ausführen lassen zu können [64, 2, 31, 115]. Nicht alle dieser Änderungen sind wieder zurück in den Quelltext transformierbar.

Ferrante et al. [31, Abb. 12] beschreiben jedoch u. a. die Transformation der *Schleifenvereinigung/-entfaltung* (engl. Loop Fusion/Unswitching). Auf den Quelltext 6.12 wird eine Schleifenentfaltung durchgeführt, sodass der Quelltext 6.13 entsteht. Die Umkehroperation (Schleifenvereinigung) führt beide Schleifen, die keine `if`-Anweisung in sich tragen, wieder auf eine Schleife zurück, die dann jedoch wieder eine Alternativanweisung enthält.

```
for (int i=0; i<100; i++) {
  if (a || b[i]) {
    c[i] = 0;
    d[i] = 0;
  }
}
```

Quelltext 6.12: Ergebnis der *Schleifenvereinigung*

```
if (a) {
  for (int i=0; i<100; i++) {
    c[i] = 0;
    d[i] = 0;
  }
} else {
  for (int i=0; i<100; i++) {
    if (b[i]) {
      c[i] = 0;
      d[i] = 0;
    }
  }
}
```

Quelltext 6.13: Ergebnis der *Schleifenentfaltung*

Bei diesen Transformationen werden keine Operationen manipuliert, sondern nur kopiert bzw. gelöscht. Jedoch ändert sich dabei die Hierarchie von Blöcken und Anweisungen, die sowohl auf- als auch abwärts verschoben und anders geschachtelt werden können. Solche Änderungen werden vom vorgestellten Ansatz nicht unterstützt. Im Ansatz können nur Ausdrücke oder Zuweisungsanweisungen geändert werden, indem sie in der Blockhierarchie abwärts verschoben werden.

6.4.3 Unsichtbare Abhängigkeiten ignorieren

Eine Limitierung des Ansatzes ist die Starrheit des Gleichungssystems, die sich daraus ergibt, dass Variablen und Gleichungen während der Lösungssuche nicht modifiziert werden können. Während sich durch Transformationen der Quelltext tatsächlich ändern kann, wird dies im Gleichungssystem teilweise durch Aktivieren bzw. Deaktivieren von Gleichungen nachempfunden. Dies betrifft bspw. geklonte Ausdrücke und IS-Kanten. Allerdings müssen unter bestimmten Umständen auch noch weitere Abhängigkeiten deaktiviert werden können.

```
1 int a,b;
2 a = 1;
3 b = a; // Wunsch: b einfügen
4 a++;
5 int c = b + a;
```

Quelltext 6.14: Vor der Umgestaltung

```
1 int a = 1;
2
3 a++;
4 int c = 1 + a;
```

Quelltext 6.15: Durch die Abhängigkeiten e_3 und e_4 verhinderte Umgestaltung

Die Quellen 6.14 und 6.15 zeigen eine semantikerhaltende Umgestaltung, die durch die Abhängigkeiten verhindert wird. Ziel der Umgestaltung ist, die Variable `b` einzufügen. Durch den Seiteneffekt auf `a` in Zeile 4 und die Verwendung von `a` in Zeile 5 ist dies jedoch nicht ohne Weiteres möglich. Jedoch könnte der Umgestaltungswunsch erreicht werden, wenn wie in Quelle 6.15, Zeile 4 gezeigt, auch die Variable `a` im einzufügenden Ausdruck durch `1` substituiert wird.

Abbildung 6.4 zeigt die Abstraktion der Quelle 6.14. Sichtbar werden u. a. die Abhängigkeiten mit den IDs e_1 und e_2. Entlang dieser Datenflussabhängigkeiten wird die Einfüge-Umgestaltung für `b` durchgeführt, bei der sowohl `b` als auch `a` eingefügt werden. Jedoch existiert noch die zusätzliche Abhängigkeit e_3 von der Deklaration von `b` zur

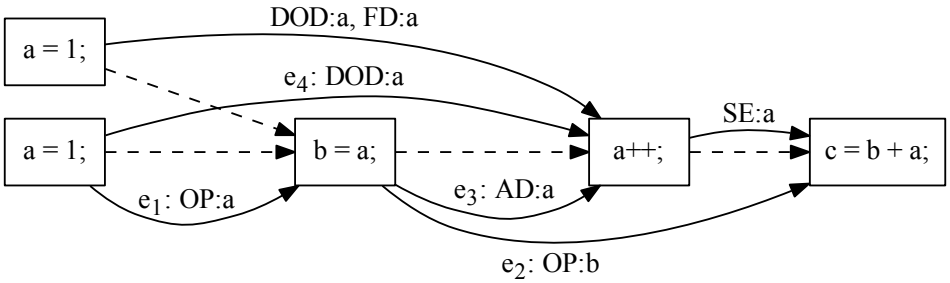


Abbildung 6.4: Abstraktion für Quelltext 6.14

Postinkrementanweisung. Selbst wenn sowohl `a` als auch `b` entlang von e_1 und e_2 eingefügt werden und mit ihnen die Deklaration von `b` überflüssig würde, existieren sie weiterhin im Gleichungssystem. Daraus folgt, dass keine gültige Reihenfolge im Sinne der Positionskongruenz (siehe Unterkapitel 3.5) existiert, bei der sowohl beide Variablen eingefügt, als auch alle Abhängigkeiten richtig herum angeordnet sind. Denn die Abhängigkeit e_3 würde von rechts nach links verlaufen, wenn beide Variablen eingefügt wären. Das gleiche trifft auf die Abhängigkeit e_4 zu. Folglich ist die in Quelltext 6.15 vorgestellte Lösung nicht möglich. Es sei denn der vorgestellte Ansatz würde so modifiziert, dass die Abhängigkeiten e_3 und e_4 deaktiviert werden könnte.

6.5 Schlussauswertung

Die in diesem Kapitel eingangs aufgeworfenen Forschungsfragen sollen im Folgenden beantwortet werden.

Antwort 1: Werden die Umgestaltungen korrekt durchgeführt?

Die überwiegende Mehrheit der Umgestaltungen liefert in der genutzten Teststrategie ein korrektes Ergebnis. Das kann aus den insgesamt 133768 Ergebnissen abgeleitet werden, die mit CodeShaker an insgesamt 66884 Umgestaltungsstellen erzielt wurden. Jedoch sind auch inkorrekte oder verweigerte Ergebnisse entstanden, die hätten vermieden werden können. Werden ausschließlich die korrekten und inkorrekten Ergebnisse in Relation gesetzt, zeigt sich, dass der Ansatz überwiegend korrekt arbeitet. Lediglich die Umgestaltung *Ausdruck einfügen II* führt auch häufiger zu inkorrekten Ergebnisse. Eine ausgiebige, stichprobenhafte Inspektion der insgesamt 74 negativen Ergebnisse, in denen Eclipse korrekte während CodeShaker inkorrekte Ergebnisse berechnete, ergab, dass die Fehlerursachen artfremd zum Forschungsansatz dieser Arbeit waren (Tabelle 6.21). Sie könnten behoben werden, indem mehr Sonderfälle von Inferenzen generischer Typen implementiert und Fehler in JDT behoben würden (Abschnitt 6.3.6). Da für jeden Umgestaltungswunsch auch immer die ihn umsetzende Transformation Teil der Lösungsmenge war, kann ferner davon ausgegangen werden, dass der Wunsch auch Teil des umgestalteten Programms war. Auch bei manuellen Inspektionen konnte nie das Gegenteil festgestellt werden.

Allerdings wurde auch festgestellt, dass nicht alle relevanten Programmabhängigkeiten in der Abstraktion abgebildet und folglich beachtet wurden. Der Vergleich der beiden Modi M1 und M2 von CodeShaker erbrachte, dass CodeShaker M2 häufiger inkorrekte Umgestaltungen berechnet, da der einzige, ihm zur Verfügung stehende Freiheitsgrad das Vertauschen von Anweisungen war (Tabelle 6.21). Das Vertauschen von Anweisungen wurde im Modus 2 zur Kompensation der fehlenden anderen Freiheitsgrade genutzt und daher öfter verwendet. Dabei entstanden häufiger inkorrekte Ergebnisse, da interprozedurale Abhängigkeiten zwischen Anweisungen, die bspw. durch Seiteneffekte entstehen, beim Vertauschen nicht beachtet wurden.

Antwort 2: Unterliegen die Umgestaltungen übermäßigen Einschränkungen?

Die neutrale Umgestaltung wurde an allen 11305 Methoden der untersuchten Projekte erfolgreich durchgeführt. Entweder sie hat eine Lösung berechnet, deren Transformationsmenge erwartungsgemäß leer war oder ihre Berechnung wurde in Ausnahmefällen aufgrund einer Zeitüberschreitung abgebrochen (Tabelle 6.20). Andere Resultate sind nicht vorgekommen, sodass die Abstraktion in Kombination mit dem Regelsatz als passend genug bewertet werden kann, um mit allen Quelltextkonstrukten der untersuchten Projekte umzugehen. Der Vergleich mit Eclipse hat außerdem gezeigt, dass nur in 351 Fällen eine Umgestaltung von CodeShaker verweigert wird, wenn Eclipse diese korrekt durchführt (Tabelle 6.21). Durch manuelle Inspektion konnte ermittelt werden, dass die Ursache dieses Verhaltens Programmabhängigkeiten waren, die Eclipse nicht erkannt und somit nicht beachtet hat.

Antwort 3: Wie groß ist der Mehrwert der Constraint-Programmierung?

Der Mehrwert der Constraint-Programmierung und der Freiheitsgrade wurde zunächst beispielhaft an insgesamt 15 Beispielen gezeigt (Unterkapitel 6.2). Ferner wurden die Studienergebnisse genutzt, um diese Frage für Quelltexte aus der Praxis zu klären. Einerseits zeigt der Vergleich der beiden CodeShaker-Modi, dass mehr korrekte Lösungen ermittelt werden konnten, wenn alle Freiheitsgrade zur Verfügung stehen (Modus 1). Das trifft besonders auf das Einfügen von Ausdrücken (57,8 zu 38,4%) und Verschieben von Anweisungen (12,5 zu 5,4%) zu. Gegenläufig dazu verhält es sich mit den Verweigerungsraten, die im Gegenzug von 35,6 auf 50,7% bzw. von 84,8 auf 90,5% steigen (Tabelle 6.21). Die anderen Umgestaltungen profitieren nur wenig von den zusätzlichen Freiheitsgraden. Andererseits zeigt die Analyse der berechneten Transformationen für jede Umgestaltungsart, dass in vielen Fällen Folgetransformationen berechnet wurden. Zu den häufigsten Folgetransformationen zählen das Verschieben von Anweisungen und das Extrahieren von Ausdrücken (Histogramme 6.2).

Antwort 4: Wie hoch sind praxisnahe Laufzeiten?

Zunächst wurde festgestellt, dass 12548 von 66884 (18,7%) der Ergebnisse von CodeShaker M1 zu einer Zeitüberschreitung führten, also mehr als 11 Sekunden für ihre Berechnung benötigten (Tabelle 6.20). Diese Zahl wurde merklich durch das Projekt Math negativ beeinflusst, da sich im Math-Projekt mehr als 20% der Umgestaltungsstellen in Methoden mit mehr als 40 Anweisungen befinden (Tabelle 6.17). In den anderen, untersuchten

Projekten befinden sich lediglich 10% oder weniger der Implementierung in solch langen Methoden. Von den Ergebnissen, die zu keiner Zeitüberschreitung führten, konnten jedoch 90% in weniger als fünf Sekunden berechnet werden und der Durchschnittswert liegt insgesamt deutlich unter einer Sekunde (Tabelle 6.23). Darüber hinaus kann mittels einer Abschätzung auf Basis der Methodengröße angenommen werden, dass eine Umgestaltung in unter fünf Sekunden berechnet werden kann, wenn ihr größter Block weniger als 40 Kompositionen enthält (Abbildung 6.3).

Kapitel 7

Zusammenfassung

Die Arbeit beschäftigt sich mit der Verbesserung von Quelltextumgestaltungen. Ausgangspunkt waren verschiedene unzulängliche Vorarbeiten aus der Praxis und der Forschung. In der Praxis etablierte Entwicklungsumgebungen zeigten an einigen Stellen Fehler bei der Durchführung von Quelltextumgestaltungen. Aktuelle Forschungsergebnisse konnten diese zwar beheben, waren aber nicht immer in der Lage, eine Lösung für Umgestaltungen zu finden. Dies traf besonders dann zu, wenn die Programmsituation unübersichtlich war. Ziel dieser Arbeit war daher die Entwicklung eines praxistauglichen Ansatzes, der auch in schwierigen Situationen Lösungen für Umgestaltungen findet, die das Ursprungsprogramm fehlerfrei umgestalten. Immer dann, wenn keine fehlerfreie Umgestaltung existiert, soll stattdessen die Umgestaltung verweigert werden.

Als Lösungsansatz wurde eine Abstraktion von Programmen entwickelt, die eine Übertragung des Programms in ein Gleichungssystem ermöglicht. Dieses Gleichungssystem enthält einerseits alle relevanten Programmabhängigkeiten und andererseits Freiheitsgrade, die eine Modifikation erlauben. Die Arbeit beschränkt sich darauf, nur intraprozedurale Programmabhängigkeiten zu finden und zu berücksichtigen, da durch eine interprozedurale Aliasanalyse schnell viele Abhängigkeiten entstehen, die ein Umgestalten erschweren oder sogar unmöglich machen. Zusätzlich enthält das Gleichungssystem den Umgestaltungswunsch, der dazu führt, dass die gewünschte Umgestaltung im Gleichungssystem durchgeführt wird. Beim Lösen des Gleichungssystems wird versucht, eine Lösung zu finden, bei der alle Programmabhängigkeiten beachtet werden und außerdem der Umgestaltungswunsch erfüllt wird. Hierfür werden die Freiheitsgrade genutzt, die den notwendigen Spielraum zur Verfügung stellen, damit eine oder mehrere Lösungen gefunden werden können. Eine Lösung besteht aus einer Menge von sequenzkongruenten Transformationen, die das Programmverhalten im Sinne der identifizierten Programmabhängigkeiten nicht verändern. Da üblicherweise mehrere Lösungen existieren, wird über ein Optimierungskriterium diejenige Lösung ausgewählt, die den ursprünglichen Quelltext minimal modifiziert.

Der entwickelte Ansatz besteht aus insgesamt sechs Beiträgen. Als erstes wurde eine *Quelltextabstraktion* und zugehörige, sequenzkongruente Transformationen entwickelt, die eine Überführung von Programmen auf Gleichungssysteme ermöglicht. Diese Quelltextabstraktion wurde für allgemeine objektorientierte Programmiersprachen entwickelt und in dieser Arbeit für Java 7 ausgelegt (zweiter Beitrag). Sie bietet insgesamt drei Freiheitsgrade, die das Finden von Umgestaltungslösungen ermöglichen. Ferner wurde ein kompletter *Regelsatz* entwickelt, mit dessen Hilfe alle relevanten Programmabhängigkeiten in Gleichungen

des Gleichungssystems überführt werden können. Ebenfalls kann der Umgestaltungswunsch und das Optimierungskriterium mithilfe dieser Regeln im Gleichungssystem abgebildet werden. Der vierte Beitrag ist die problemspezifische *Suchstrategie*, mit deren Hilfe eine Lösung des Gleichungssystems effizient gefunden werden kann. Sie ermöglicht den Einsatz des Ansatzes in der für die Praxis notwendigen Berechnungsgeschwindigkeit. Außerdem wurde eine Transformationsvorschrift für die berechneten, sequenzkongruenten Transformationen präsentiert. Sie erklärt, wie die im Gleichungssystem berechnete Lösung als Umgestaltung für das Ursprungsprogramm interpretiert und angewendet werden soll. Schließlich wurde der praxistaugliche Prototyp *CodeShaker* implementiert, um den Ansatz manuell sowie automatisch ausgiebig testen zu können.

Die Evaluation der Arbeit ist der sechste Beitrag und wurde auf zwei verschiedene Arten durchgeführt. Einerseits wurde anhand von insgesamt fünfzehn Beispielen der vorgestellte Ansatz mit den implementierten Ansätzen in etablierten Entwicklungsumgebungen für Java verglichen. Dabei schnitt der Ansatz in jedem Fall besser als die Konkurrenten ab und konnte die Umgestaltungen stets semantikerhaltend durchführen. Andererseits wurde der Ansatz im Rahmen einer Studie auf insgesamt 133768 Beispiele angewendet und dabei auf mehrere Kriterien hin untersucht. Für die Studie wurden verschiedene quelloffene Projekte herangezogen. Dabei wurde der Ansatz an insgesamt 38173 zusätzlichen Beispielen auch mit dem in der Entwicklungsplattform Eclipse implementierten Verfahren verglichen. Das Ergebnis war, dass der Ansatz im Vergleich zu Eclipse nur sehr selten zu Verhaltensänderungen im Programm führt. In der Mehrzahl der Fälle, in denen Umgestaltungen von Eclipse zu Semantikänderungen führen, verweigert der Ansatz die Umgestaltungen oder führt zusätzliche Änderungen am Quelltext durch, um die Semantik zu erhalten. Ferner konnte durch die Untersuchung festgestellt werden, dass der Ansatz in knapp 90% aller Methoden eines Programms eine Lösung in weniger als fünf Sekunden berechnen konnte und somit als praxistauglich angesehen werden kann. Einschränkend wurde festgestellt, dass lange Methoden, die mehr als 40 Anweisungen oder Ausdrücke in einem Quelltextblock besitzen, das Finden einer Lösung in akzeptabler Zeit verhindern. Letztendlich bleibt es dem Entwickler überlassen, ob er durchschnittlich ein bis zwei Sekunden auf die Berechnung eines verlässlicheren Umgestaltungsergebnisses wartet oder eine möglicherweise wesentlich länger dauernde Fehlersuche riskiert, die durch eine fehlerhafte, aber schneller berechnete Umgestaltung verursacht wurde.

Kapitel 8

Ausblick

Zukünftig kann an verschiedenen Stellen die vorgestellte Arbeit verbessert und fortgeführt werden. Der hier entwickelte Ansatz kann bspw. auf Multi-Thread-Szenarien übertragen werden, da im Kontext mehrerer Threads viele zusätzliche Programmabhängigkeiten entstehen und andererseits auch weitere Umgestaltungen nützlich sein können. Bspw. müssen zwischen mehreren Threads verschränkte und möglicherweise volatile Variablenzugriffe beachtet werden. Lineare Variablenzugriffe würden dann nur noch in synchronisierten Blöcken existieren können. In Multi-Thread-Szenarien könnten außerdem neue Umgestaltungen für Entwickler hilfreich sein, die den Umgang mit mehreren Threads erleichtern. Beispielsweise könnten Umgestaltungen entwickelt werden, die Entwickler unterstützt, indem sie Methoden teil- oder vollautomatisch parallelisiert. Erster Ansatzpunkt dafür könnte sein, die Anweisungen einer Methode oder eines Blocks nach Datenabhängigkeiten disjunkt zu gruppieren, um jede Gruppe in einem zweiten Schritt von einem eigenen Thread ausführen zu lassen.

Eine Erweiterung des Ansatzes kann hinsichtlich der Freiheitsgrade und des Optimierungskriteriums erreicht werden. Durch eine Erweiterung der Abstraktion und des Regelsatzes könnten mehr Freiheitsgrade eingeführt werden, sodass er bessere und mehr Lösungen findet. U. a. könnten Transformationen eingeführt werden, die das Ändern, Hinzufügen oder Entfernen von Operationen erlauben. Dadurch könnten Ausdrücke statisch ausgewertet und nach Bedarf semantikerhaltend verändert werden, um Umgestaltungen zu ermöglichen. Eine weitere Erweiterung könnte sich mit dem Zusammenführen oder Aufspalten von Quelltextblöcken wie Schleifen oder Alternativen befassen. Hierdurch entstünde eine große Flexibilität beim Verschieben oder Gruppieren von Anweisungen.

Die Lösungsberechnung kann optimiert werden, da die Zahl der Lösungen direkt von den Freiheitsgraden der Abstraktion abhängt. Je mehr Freiheitsgrade diese bietet, desto mehr Lösungen existieren und müssen berechnet werden, um die optimale Lösung zu finden. Nicht immer sind alle Freiheitsgrade an allen Orten im Programm sinnvoll oder notwendig. Denkbar wäre, unter der Prämisse einer gewünschten Umgestaltung, die Freiheitsgrade in der Abstraktion gezielt zu minimieren. Aktuell existieren im Gleichungssystem auch immer Freiheitsgrade, die zum Finden einer gewünschten Umgestaltung nicht erforderlich sind. Würden diese deaktiviert, könnte der Suchraum zum Finden einer Lösung sinnvoll reduziert und die Suche somit beschleunigt werden. Wenn bspw. ein Ausdruck A extrahiert werden soll, besteht in der Abstraktion weiterhin die Möglichkeit, auch alle anderen Ausdrücke der Methode zu extrahieren. Inwieweit das notwendig oder überhaupt hilfreich

zum Finden einer Umgestaltungslösung ist, wird nicht ausgewertet. Nichtsdestotrotz verursachen diese Freiheitsgrade einen erhöhten Berechnungsaufwand, der durch eine sinnvolle Optimierung der anwendbaren Freiheitsgrade vermutlich gesenkt werden kann. Steimann et al. [110] nutzen die dafür in Refacola zur Verfügung stehende Möglichkeit, allgemein für Umgestaltungswünsche festzulegen, welche Änderungen erlaubt und welche verboten sind. Ähnlich dazu könnten die hier vorgestellten Filter/Selektoren verwendet werden, auch Aussagen über die Änderbarkeit der Gleichungsvariablen in bestimmten Situationen und Umgestaltungen zu treffen.

Ferner könnten die Abhängigkeitsanalysen des Ansatzes verbessert, oder bestehende Aliasanalysen integriert werden. Mithilfe von inkrementellen, im Hintergrund laufenden Analysen, stünden immer genaue Informationen zur Verfügung, um Umgestaltungen optimal durchführen zu können. Durch eine Klondetektion kann ferner die Extrahieren-Umgestaltung verbessert werden. Beim Durchführen würden dann sowohl der zu extrahierende als auch zu ihm äquivalente Ausdrücke extrahiert. Schließlich kann der Ansatz auf weitere Objekt orientierte Sprachen wie C# übertragen werden.

Abbildungsverzeichnis

1.1 Anteil der Entwickler, die Umgestaltungshilfen nutzen (nach [78]) 4
1.2 Aus Programm P und Umgestaltungswunsch W werden die Transformationen δp berechnet. 10

2.1 PDG-Beispiel: Kontrollflusskante 28
2.2 PDG-Beispiel: Addition 29
2.3 Eine Kantenbeschriftung wird hinzugefügt/entfernt. 31
2.4 Transitivität: Fall 1 32
2.5 Transitivität: Fall 2 32
2.6 Transitivität bei mehreren Definitionsorten nicht möglich 33
2.7 Distribution von Datenoperationen 33
2.8 Distribution von Kontrollprädikaten 34
2.9 Seiteneffekte und Verschachtelung können zu Problemen führen. 36
2.10 Wechselnde Knoten der `if`-Blöcke können nicht direkt auf den Quelltext abgebildet werden. 37
2.11 UML-Objektdiagramm des gekürzten JDT-AST zum Quelltext 2.1 40

3.1 Quelltextabstraktion als UML-Klassendiagramm 47
3.2 Abstraktionsbeispiel für die Addition `i + j` 48
3.3 Verschachtelung in der Abstraktion 49
3.4 Komposition einfügen/extrahieren 64
3.5 Komposition einfügen/extrahieren 65
3.6 Datendistribution 67
3.7 Datendistribution 68
3.8 Anweisungen umsortieren 69
3.9 Deklaration umwandeln 1 70
3.10 Deklaration umwandeln 2 71
3.11 Deklaration hinzufügen/entfernen 72

4.1 Von Programm P und Umgestaltungswunsch W zur Abstraktion A_3. 75
4.2 Abstraktionen von Java-Ausdrücken 82
4.3 Abstraktion einer Zusicherung 83
4.4 Abstraktion einer Rückgabeanweisung 84
4.5 Abstraktion einer Deklaration 85
4.6 Abstraktion einer Deklaration 85
4.7 Abstraktion einer Alternative 86
4.8 Abstraktion einer `For`-Schleife 87
4.9 Abstraktion einer `For-Each`-Schleife 88

4.10 Abstraktion einer `While`-Schleife . . . 89
4.11 Abstraktion einer `Do-While`-Schleife . . . 90
4.12 Abstraktion einer `Try`-Anweisung . . . 91
4.13 Abstraktion einer `Switch`-Anweisung . . . 92
4.14 Abstraktion einer `Sychronized`-Anweisung . . . 92
4.15 Operandenfragment einer Addition . . . 94
4.16 Abhängigkeiten in der Abstraktion von Quelltext 4.4 . . . 105
4.17 Abhängigkeiten durch einen Sprungeffekt . . . 106
4.18 Quelltextabstraktion des Quelltexts 4.6. . . . 108
4.19 Abstraktionen vor und nach der Expansion . . . 110

5.1 Aus der Abstraktion A_3 und dem Umgestaltungswunsch W werden die Transformationen δp berechnet und auf den Quelltext P angewendet. . . . 115
5.2 Repräsentation der Extrahieren/Einfügen-Transformation der Komposition von `foo()` im Constraint-Problem. Grau dargestellt sind die IS-Kantengleichungen, die durch die Generatorregeln 5.8, 5.9 und 5.10 erzeugt werden. . . . 122
5.3 Abhängigkeiten und Gleichungsvariablen der Teilprobleme (TP) . . . 132
5.4 Abstraktion des Beispiels . . . 136
5.5 Teilgraph des Suchbaums . . . 137

6.1 Evaluationsstrategie . . . 164
6.2 Die Histogramme beziehen sich auf die Ergebnisse von CodeShaker. Die Ordinate gibt die Zahl der erfolgreichen Tests und die Abszisse gibt die Häufigkeit der Transformationen an. . . . 173
6.3 Zusammenhang zwischen Berechnungsdauer und Größe der umgestalteten Methode. . . . 176
6.4 Abstraktion für Quelltext 6.14 . . . 181

Tabellenverzeichnis

3.1 Abbildung $E2n$ eines Elements e auf seinen Knoten 55

4.1 Unterstützte JDT-Anweisungen 79
4.2 Kommutativitätsfunktion $IsCommutative$ von JDT-Ausdrücken 81
4.3 Unterelemente der JDT-Ausdrücke 96
4.4 Durch eine Blockabstraktion abstrahierte JDT-Elemente 97
4.5 Die Tabelle gibt die Knotentypen der Effekte an. 102
4.6 Abbildung $Catches$ definiert wann ein Sprung gefangen wird 103
4.7 Abbildung ET 104

5.1 CSP-Variablen 117
5.2 CSP-Variablen der Teilprobleme 131
5.3 Abhängigkeit zwischen Blockteilproblemen 132
5.4 Entscheidungsregeln der Suchstrategie 135
5.5 Transformationen aus der Constraint-Lösung interpretieren 139
5.6 Reihenfolge der Transformationen 143
5.7 Vom Wunsch zur Transformation 145

6.1 Beispiel aus Thies [118] für fehlerhafte Umgestaltungen aufgrund unvollständiger Formalisierung 150
6.2 Beispiel: Ausdruck einfügen 1 (nach [95, S. 45]) 154
6.3 Beispiel: Ausdruck einfügen 2 155
6.4 Beispiel: Ausdruck einfügen 3 155
6.5 Beispiel: Ausdruck einfügen 4 (nach [95, S. 50]) 156
6.6 Beispiel: Ausdruck einfügen 5 (nach [95, S. 94]) 156
6.7 Beispiel: Ausdruck einfügen 6 157
6.8 Beispiel: Ausdruck extrahieren 1 158
6.9 Beispiel: Ausdruck extrahieren 2 158
6.10 Beispiel: Ausdruck extrahieren 3 159
6.11 Beispiel: Operanden vertauschen 1 159
6.12 Beispiel: Operanden vertauschen 2 160
6.13 Beispiel: Anweisungen verschieben 1 160
6.14 Beispiel: Anweisungen verschieben 2 161
6.15 Beispiel: Anweisungen verschieben 3 161
6.16 Beispiel: Parameter/Argumente vertauschen 1 162
6.17 Untersuchte Projekte und ihre Größe 165
6.18 Tatsächliche Abdeckung bezogen auf die umgestaltete Java-Klasse 165
6.19 Anzahl der untersuchten Stellen nach Umgestaltung und Projekt 166

6.20 Ergebnisse der Umgestaltungen in CodeShaker M1 169
6.21 Ergebnissvergleich von CodeShaker M1, M2 und Eclipse 170
6.22 Ergebnisvergleich von CodeShaker und Eclipse in gegenseitiger Abhängigkeit 171
6.23 Komplexität und Praktikabilität 175

Literaturverzeichnis

[1] *7th Annual State of Agile Development Survey*. VersionOne. 2013. URL: http://www.versionone.com/pdf/7th-Annual-State-of-Agile-Development-Survey.pdf.

[2] A. V. Aho, R. Sethi und J. D. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley Series in Computer Science. Addison-Wesley Pub. Co., 1986. ISBN: 9780201100884. URL: http://books.google.de/books?id=-4Q_AQAAIAAJ.

[3] A. J. Albrecht und J. E. Gaffney. "Software Function, Source Lines of Code, and Development Effort Prediction: A Software Science Validation". In: *Software Engineering, IEEE Transactions on* SE-9.6 (1983), S. 639–648. ISSN: 0098-5589. DOI: 10.1109/TSE.1983.235271.

[4] C. Alippi und V. Piuri. *Neural Networks for Identification, Control, Robotics, Signal and Image Processing*. Hrsg. von A. Kent und J. Williams. Encyclopedia of Computer Science and Technology; Ausgabe 35. 0-824-72292-2. New York: Marcel Dekker, 1995, S. 341.

[5] F. E. Allen und J. Cocke. *A Catalogue of Optimizing Transformations*. International Business Machines Corporation., 1971. URL: http://books.google.de/books?id=oeXaZwEACAAJ.

[6] L. O. Andersen. "Program Analysis and Specialization for the C Programming Language". Diss. University of Copenhagen, 1994.

[7] C. Barks und E. Fuchs. *Donald Duck - Donald, der Herr über alle Geschöpfe*. Donald Duck 327. 1956, S. 10.

[8] V. R. Basili, G. Caldiera und H. D. Rombach. "The Goal Question Metric Approach". In: *Encyclopedia of Software Engineering*. Wiley, 1994.

[9] F. L. Bauer und CIP Language Group. *The Munich Project CIP. Volume I. , The Wide Spectrum Language CIP-L*. Lecture Notes in Computer Science. Berlin: Springer-Verlag, 1985. ISBN: 3-540-15187-7. URL: http://opac.inria.fr/record=b1095508.

[10] F. L. Bauer, H. Wössner, H. Partsch und P. Pepper. *Algorithmische Sprache und Programmentwicklung*. Springer Berlin Heidelberg, 1983. ISBN: 9783540129622. URL: https://books.google.de/books?id=V0RWAgAACAAJ.

[11] F. L. Bauer, B. Möller, H. Partsch und P. Pepper. "Formal Program Construction by Transformations-Computer-Aided, Intuition-Guided Programming". In: *IEEE Trans. Softw. Eng.* 15.2 (Feb. 1989), S. 165–180. ISSN: 0098-5589. DOI: 10.1109/32.21743. URL: http://dx.doi.org/10.1109/32.21743.

[12] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna und L. Bier. "Clone Detection Using Abstract Syntax Trees". In: *Proceedings of the International Conference on Software Maintenance*. ICSM '98. Washington, DC, USA: IEEE Computer Society, 1998, S. 368–. ISBN: 0-8186-8779-7. URL: http://dl.acm.org/citation.cfm?id=850947.853341.

[13] K. Beck, M. Beedle, A. v. Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. C. Martin, S. Mellor, K. Schwaber, J. Sutherland und D. Thomas. *Manifesto for Agile Software Development*. 2001. URL: http://agilemanifesto.org/.

[14] A. J. Bernstein. "Analysis of Programs for Parallel Processing". In: *Electronic Computers, IEEE Transactions on* EC-15.5 (1966), S. 757–763. ISSN: 0367-7508. DOI: 10.1109/PGEC.1966.264565.

[15] S. M. Blackburn, P. Cheng und K. S. McKinley. "Myths and Realities: The Performance Impact of Garbage Collection". In: *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*. SIGMETRICS '04/Performance '04. New York, NY, USA: ACM, 2004, S. 25–36. ISBN: 1-58113-873-3. DOI: 10.1145/1005686.1005693. URL: http://doi.acm.org/10.1145/1005686.1005693.

[16] F. P. Brooks. *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley, 1975. ISBN: 0-201-00650-2.

[17] M. Broy, M. Wirsing und P. Pepper. "On the Algebraic Definition of Programming Languages". In: *ACM Trans. Program. Lang. Syst.* 9.1 (Jan. 1987), S. 54–99. ISSN: 0164-0925. DOI: 10.1145/9758.10501. URL: http://doi.acm.org/10.1145/9758.10501.

[18] *C Language Specification WG14 N1570 Committee Draft*. Apr. 2011. URL: http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf.

[19] F. W. Callis. "Problems with Automatic Restructurers". In: Bd. 23. SIGPLAN Notices. ACM Press, 1988, S. 13–21.

[20] E. J. Chikofsky und J. H. Cross II. "Reverse Engineering and Design Recovery: A Taxonomy". In: *IEEE Softw.* 7.1 (Jan. 1990), S. 13–17. ISSN: 0740-7459. DOI: 10.1109/52.43044. URL: http://dx.doi.org/10.1109/52.43044.

[21] W. F. Clocksin und C. S. Mellish. *Programming in Prolog (2Nd Ed.)* New York, NY, USA: Springer-Verlag New York, Inc., 1984. ISBN: 0-387-15011-0.

[22] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman und F. K. Zadeck. "An Efficient Method of Computing Static Single Assignment Form". In: *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '89. Austin, Texas, USA: ACM, 1989, S. 25–35. ISBN: 0-89791-294-2. DOI: 10.1145/75277.75280. URL: http://doi.acm.org/10.1145/75277.75280.

[23] B. Daniel, D. Dig, K. Garcia und D. Marinov. "Automated Testing of Refactoring Engines". In: *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*. ESEC-FSE '07. Dubrovnik, Croatia: ACM, 2007, S. 185–194. ISBN: 978-1-59593-811-4. DOI: 10.1145/1287624.1287651. URL: http://doi.acm.org/10.1145/1287624.1287651.

[24] E. W. Dijkstra. *A Discipline of Programming*. 1st. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1997. ISBN: 013215871X.

[25] A. Donovan, A. Kiežun, M. S. Tschantz und M. D. Ernst. "Converting Java Programs to Use Generic Libraries". In: *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*. OOPSLA '04. Vancouver, BC, Canada: ACM, 2004, S. 15–34. ISBN: 1-58113-831-8. DOI: 10.1145/1028976.1028979. URL: http://doi.acm.org/10.1145/1028976.1028979.

[26] D. Doomen. *Coding Guidelines for C# 3.0, 4.0 and 5.0*. 2014. URL: http://csharpguidelines.codeplex.com/releases/view/46280.

[27] M. Drozdz, D. G. Kourie, B. W. Watson und A. Boake. "Refactoring Tools and Complementary Techniques". In: *2006 IEEE/ACS International Conference on Computer Systems and Applications (AICCSA 2006), March 8-11, Dubai/Sharjah, UAE*. 2006, S. 685–688. DOI: 10.1109/AICCSA.2006.205164. URL: http://dx.doi.org/10.1109/AICCSA.2006.205164.

[28] P. Ebraert, T. D'Hondt und T. Mens. "Enabling Dynamic Software Evolution through Automatic Refactoring". In: *International Workshop on Software Evolution Transformations (SETra)*. Hrsg. von Ying Zhou und James R. Cordy. 2004, S. 3–6.

[29] E. v. Emden und L. Moonen. *jCOSMO*. 2015. URL: https://code.google.com/p/jcosmo/.

[30] J. Ferrante und K. J. Ottenstein. "A Program Form Based on Data Dependency in Predicate Regions". In: *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL '83. Austin, Texas: ACM, 1983, S. 217–236. ISBN: 0-89791-090-7. DOI: 10.1145/567067.567089. URL: http://doi.acm.org/10.1145/567067.567089.

[31] J. Ferrante, K. J. Ottenstein und J. D. Warren. "The Program Dependence Graph and its Use in Optimization". In: *ACM Trans. Program. Lang. Syst.* 9.3 (Juli 1987), S. 319–349. ISSN: 0164-0925. DOI: 10.1145/24039.24041. URL: http://doi.acm.org/10.1145/24039.24041.

[32] The Apache Software Foundation. *Commons Collections Version 4.0*. 2014. URL: https://commons.apache.org/proper/commons-collections/.

[33] The Apache Software Foundation. *Commons Math Version 3.4.1*. 2015. URL: https://commons.apache.org/proper/commons-math/.

[34] The Eclipse Foundation. *Eclipse Java development tools (JDT)*. URL: http://www.eclipse.org/jdt/.

[35] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley, 1999. ISBN: 0-201-48567-2.

[36] N. E. Fuchs. *Kurs in logischer Programmierung*. Springers angewandte Informatik. Springer, 1990. ISBN: 978-3-211-82235-7.

[37] R. M. Fuhrer, M. Keller und A. Kiezun. "Advanced Refactoring in the Eclipse JDT: Past, Present, and Future". In: *1st Workshop on Refactoring Tools, WRT 2007, in conjunction with 21st European Conference on Object-Oriented Programming, July 30 - August 03, 2007, Berlin, Proceedings*. 2007, S. 30–31. URL: http://netfiles.uiuc.edu/dig/RefactoringWorkshop/.

[38] E. Gamma, R. Helm, R. Johnson und J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN: 0-201-63361-2.

[39] J. Garcia, D. Popescu, G. Edwards und N. Medvidovic. "Toward a Catalogue of Architectural Bad Smells". In: *Proceedings of the 5th International Conference on the Quality of Software Architectures: Architectures for Adaptive Software Systems*. QoSA '09. East Stroudsburg, PA, USA: Springer-Verlag, 2009, S. 146–162. ISBN: 978-3-642-02350-7. DOI: 10.1007/978-3-642-02351-4_10. URL: http://dx.doi.org/10.1007/978-3-642-02351-4_10.

[40] A. Georges, D. Buytaert und L. Eeckhout. "Statistically Rigorous Java Performance Evaluation". In: *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications*. OOPSLA '07. Montreal, Quebec, Canada: ACM, 2007, S. 57–76. ISBN: 978-1-59593-786-5. DOI: 10.1145/1297027.1297033. URL: http://doi.acm.org/10.1145/1297027.1297033.

[41] M. Gligoric, F. Behrang, Y. Li, J. Overbey, M. Hafiz und Da. Marinov. "Systematic Testing of Refactoring Engines on Real Software Projects". In: *Proceedings of the 27th European Conference on Object-Oriented Programming*. ECOOP'13. Montpellier, France: Springer-Verlag, 2013, S. 629–653. ISBN: 978-3-642-39037-1. DOI: 10.1007/978-3-642-39038-8_26. URL: http://dx.doi.org/10.1007/978-3-642-39038-8_26.

[42] *Google Gson Version 2.3.1*. 2014. URL: https://github.com/google/gson.

[43] J. Gosling, B. Joy, G. L. Steele Jr., G. Bracha und A. Buckley. *The Java Language Specification, Java SE 7 Edition*. 2013.

[44] H. Görel. *An Introductory Tutorial on JastAdd Attribute Grammars*.

[45] W. G. Griswold. "Program Restructuring as an Aid to Software Maintenance". Diss. University of Washington, 1991.

[46] W. G. Griswold und D. Notkin. "Architectural Tradeoffs for a Meaning-Preserving Program Restructuring Tool". In: *IEEE Trans. Softw. Eng.* 21.4 (Apr. 1995), S. 275–287. ISSN: 0098-5589. DOI: 10.1109/32.385967. URL: http://dx.doi.org/10.1109/32.385967.

[47] W. G. Griswold und D. Notkin. "Automated Assistance for Program Restructuring". In: *ACM Trans. Softw. Eng. Methodol.* 2.3 (Juli 1993), S. 228–269. ISSN: 1049-331X. DOI: 10.1145/152388.152389. URL: http://doi.acm.org/10.1145/152388.152389.

[48] W. G. Griswold und D. Notkin. "Computer-aided vs. Manual Program Restructuring". In: *SIGSOFT Softw. Eng. Notes* 17.1 (Jan. 1992), S. 33–41. ISSN: 0163-5948. DOI: 10.1145/134292.134294. URL: http://doi.acm.org/10.1145/134292.134294.

[49] D. Gu, C. Verbrugge und E. M. Gagnon. "Relative Factors in Performance Analysis of Java Virtual Machines". In: *Proceedings of the 2Nd International Conference on Virtual Execution Environments*. VEE '06. Ottawa, Ontario, Canada: ACM, 2006, S. 111–121. ISBN: 1-59593-332-8. DOI: 10.1145/1134760.1134776. URL: http://doi.acm.org/10.1145/1134760.1134776.

[50] M. Harman und L. Tratt. "Pareto Optimal Search Based Refactoring at the Design Level". In: *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation*. GECCO '07. London, England: ACM, 2007, S. 1106–1113. ISBN: 978-1-59593-697-4. DOI: 10.1145/1276958.1277176. URL: http://doi.acm.org/10.1145/1276958.1277176.

[51] M. Hind. "Pointer Analysis: Haven't We Solved This Problem Yet?" In: *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*. PASTE '01. Snowbird, Utah, United States: ACM, 2001, S. 54–61. ISBN: 1-58113-413-4. DOI: http://doi.acm.org/10.1145/379605.379665. URL: http://doi.acm.org/10.1145/379605.379665.

[52] P. Hofstedt und A. Wolf. *Einführung in die Constraint-Programmierung - Grundlagen, Methoden, Sprachen, Anwendungen*. eXamen.press. Springer, 2007. ISBN: 978-3-540-23184-4. DOI: 10.1007/978-3-540-68194-6. URL: http://dx.doi.org/10.1007/978-3-540-68194-6.

[53] S. Horwitz, J. Prins und T. Reps. "On the Adequacy of Program Dependence Graphs for Representing Programs". In: *in Conference Record of the Fifteenth ACM Symposium on Principles of Programming Languages*. ACM, 1988, S. 146–157.

[54] *IntelliJ IDEA 13.1.1*. JetBrains. 2015. URL: www.jetbrains.com/idea/.

[55] *Java Platform, Standard Edition HotSpot Virtual Machine Garbage Collection Tuning Guide*. Oracle. 2015. URL: https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/.

[56] A. C. Jensen und B. H. C. Cheng. "On the Use of Genetic Programming for Automated Refactoring and the Introduction of Design Patterns". In: *Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation*. GECCO '10. Portland, Oregon, USA: ACM, 2010, S. 1341–1348. ISBN: 978-1-4503-0072-8. DOI: 10.1145/1830483.1830731. URL: http://doi.acm.org/10.1145/1830483.1830731.

[57] *Joda Convert 1.7*. 2014. URL: http://www.joda.org/joda-convert/.

[58] *JSoup 1.8.2*. 2015. URL: http://jsoup.org/.

[59] H. Kegel und F. Steimann. "Systematically Refactoring Inheritance to Delegation in Java". In: *Proceedings of the 30th International Conference on Software Engineering*. ICSE '08. Leipzig, Germany: ACM, 2008, S. 431–440. ISBN: 978-1-60558-079-1. DOI: 10.1145/1368088.1368147. URL: http://doi.acm.org/10.1145/1368088.1368147.

[60] M. Kim, T. Zimmermann und N. Nagappan. "A Field Study of Refactoring Challenges and Benefits". In: *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. FSE '12. Cary, North Carolina: ACM, 2012, 50:1–50:11. ISBN: 978-1-4503-1614-9. DOI: 10.1145/2393596.2393655. URL: http://doi.acm.org/10.1145/2393596.2393655.

[61] S. Kim, E. J. Whitehead und J. Bevan. "Analysis of Signature Change Patterns". In: *Proceedings of the 2005 International Workshop on Mining Software Repositories*. MSR '05. St. Louis, Missouri: ACM, 2005, S. 1–5. ISBN: 1-59593-123-6. DOI: 10.1145/1082983.1083154. URL: http://doi.acm.org/10.1145/1082983.1083154.

[62] A. J. Ko, H. Aung und B. A. Myers. "Eliciting Design Requirements for Maintenance-Oriented IDEs: A Detailed Study of Corrective and Perfective Maintenance Tasks". In: *Proceedings of the 27th International Conference on Software Engineering*. ICSE '05. St. Louis, MO, USA: ACM, 2005, S. 126–135. ISBN: 1-58113-963-2. DOI: 10.1145/1062455.1062492. URL: http://doi.acm.org/10.1145/1062455.1062492.

[63] K. A. Kontogiannis, R. Demori, E. Merlo, M. Galler und M. Bernstein. "Pattern Matching for Clone and Concept Detection". In: *Journal of Automated Software Engineering* 3 (1996), S. 108.

[64] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure und M. Wolfe. "Dependence graphs and compiler optimizations". In: *Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '81. Williamsburg, Virginia: ACM, 1981, S. 207–218. ISBN: 0-89791-029-X. DOI: 10.1145/567532.567555. URL: http://doi.acm.org/10.1145/567532.567555.

[65] W. Landi und B. G. Ryder. "A Safe Approximate Algorithm for Interprocedural Aliasing". In: *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*. PLDI '92. San Francisco, California, United States: ACM, 1992, S. 235–248. ISBN: 0-89791-475-9. DOI: http://doi.acm.org/10.1145/143095.143137. URL: http://doi.acm.org/10.1145/143095.143137.

[66] M. M. Lehman und L. A. Belady. *Program Evolution: Processes of Software Change*. San Diego, CA, USA: Academic Press Professional, Inc., 1985. ISBN: 0-12-442440-6.

[67] M. M. Lehman und J. F. Ramil. "Rules and Tools for Software Evolution Planning and Management". English. In: *Annals of Software Engineering* 11.1 (2001), S. 15–44. ISSN: 1022-7091. DOI: 10.1023/A:1012535017876. URL: http://dx.doi.org/10.1023/A%3A1012535017876.

[68] H. Liu, L. Yang, Z. Niu, Z. Ma und W. Shao. "Facilitating Software Refactoring with Appropriate Resolution Order of Bad Smells". In: *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*. ESEC/FSE '09. Amsterdam, The Netherlands: ACM, 2009, S. 265–268. ISBN: 978-1-60558-001-2.

DOI: 10.1145/1595696.1595738. URL: http://doi.acm.org/10.1145/1595696.1595738.

[69] R. C. Martin. *Agile Software Development: Principles, Patterns, and Practices*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2003. ISBN: 0135974445.

[70] R. C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Robert C. Martin series. Prentice Hall, 2009. ISBN: 9780132350884. URL: https://books.google.de/books?id=dwSfGQAACAAJ.

[71] T. Mens und T. Tourwé. "A Survey of Software Refactoring". In: *IEEE Trans. Software Eng.* 30.2 (2004), S. 126–139.

[72] B. Meyer. *Object-Oriented Software Construction (2Nd Ed.)* Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1997. ISBN: 0-13-629155-4.

[73] *Microsoft Visual Studio 2013*. Microsoft Corporation. 2014. URL: msdn.microsoft.com.

[74] A. Milanova. "Parameterized Object Sensitivity for Points-to Analysis for Java". In: *ACM Trans. Softw. Eng. Methodol* 14 (2005), S. 2005. DOI: http://doi.acm.org/10.1145/1044834.1044835.

[75] I. H. Moghadam und M. O. Cinneide. "Automated Refactoring Using Design Differencing". In: *Proceedings of the 2012 16th European Conference on Software Maintenance and Reengineering*. CSMR '12. Washington, DC, USA: IEEE Computer Society, 2012, S. 43–52. ISBN: 978-0-7695-4666-7. DOI: 10.1109/CSMR.2012.15. URL: http://dx.doi.org/10.1109/CSMR.2012.15.

[76] I. Moore. *Condenser 1.05*. 2015. URL: http://condenser.sourceforge.net/.

[77] I. Moore. *Guru - A Tool for Automatic Restructuring of Self Inheritance Hierarchies*. 1995.

[78] G. C. Murphy, M. Kersten und L. Findlater. "How Are Java Software Developers Using the Eclipse IDE?" In: *IEEE Softw.* 23.4 (Juli 2006), S. 76–83. ISSN: 0740-7459. DOI: 10.1109/MS.2006.105. URL: http://dx.doi.org/10.1109/MS.2006.105.

[79] E. Murphy-Hill, C. Parnin und A. P. Black. "How We Refactor, and How We Know It". In: *Proceedings of the 31st International Conference on Software Engineering*. ICSE '09. Washington, DC, USA: IEEE Computer Society, 2009, S. 287–297. ISBN: 978-1-4244-3453-4. DOI: 10.1109/ICSE.2009.5070529. URL: http://dx.doi.org/10.1109/ICSE.2009.5070529.

[80] *NetBeans IDE 8.0.1*. Oracle Corporation. 2015. URL: www.netbeans.org.

[81] T. Nipkow und D. v. Oheimb. "JavaLight is Type-Safe - Definitely". In: *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '98. San Diego, California, USA: ACM, 1998, S. 161–170. ISBN: 0-89791-979-3. DOI: 10.1145/268946.268960. URL: http://doi.acm.org/10.1145/268946.268960.

[82] M. Ó Cinnéide, L. Tratt, M. Harman, S. Counsell und I. H. Moghadam. "Experimental Assessment of Software Metrics Using Automated Refactoring". In: *Proceedings of the ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*. ESEM '12. Lund, Sweden: ACM, 2012, S. 49–58. ISBN: 978-1-4503-1056-7. DOI: 10.1145/2372251.2372260. URL: http://doi.acm.org/10.1145/2372251.2372260.

[83] M. O'Keeffe und M. í Cinnéide. "Search-Based Refactoring for Software Maintenance". In: *J. Syst. Softw.* 81.4 (Apr. 2008), S. 502–516. ISSN: 0164-1212. DOI: 10.1016/j.jss.2007.06.003. URL: http://dx.doi.org/10.1016/j.jss.2007.06.003.

[84] W. F. Opdyke. "Refactoring Object-Oriented Frameworks". UMI Order No. GAX93-05645. Diss. Champaign, IL, USA: University of Illinois at Urbana-Champaign, 1992.

[85] *Oracle. java.util.concurrent.locks API Specification.* 2010. URL: https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/package-summary.html.

[86] H. Partsch und R. Steinbrüggen. "Program Transformation Systems". In: *ACM Comput. Surv.* 15.3 (Sep. 1983), S. 199–236. ISSN: 0360-0300. DOI: 10.1145/356914.356917. URL: http://doi.acm.org/10.1145/356914.356917.

[87] H. A. Partsch. *Specification and Transformation of Programs: A Formal Approach to Software Development*. New York, NY, USA: Springer-Verlag New York, Inc., 1990. ISBN: 0-387-52356-1.

[88] J. Philipps und B. Rumpe. "Roots of Refactoring". In: *In: Tenth OOPSLA Workshop on Behavioral Semantics*. Northeastern University. 2001, 2001, S. 187–199.

[89] J. v. Pilgrim, B. Ulke, A. Thies und F. Steimann. "Model/Code Co-Refactoring: An MDE Approach". In: *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering (ASE 2013)*. 2013.

[90] A. Podgurski und L. A. Clarke. "A Formal Model of Program Dependences and Its Implications for Software Testing, Debugging, and Maintenance". In: *IEEE Trans. Software Eng.* 16.9 (1990), S. 965–979. DOI: 10.1109/32.58784. URL: http://dx.doi.org/10.1109/32.58784.

[91] *RefactorIT 2.7*. 2015. URL: http://sourceforge.net/projects/refactorit/.

[92] *ReSharper 8.0*. JetBrains. 2014. URL: www.jetbrains.com/resharper/.

[93] C. K. Roy und J. R. Cordy. "A Survey on Software Clone Detection Research". In: *SCHOOL OF COMPUTING TR 2007-541, QUEEN'S UNIVERSITY* 115 (2007).

[94] C. K. Roy, J. R. Cordy und R. Koschke. "Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach". In: *Sci. Comput. Program.* 74.7 (Mai 2009), S. 470–495. ISSN: 0167-6423. DOI: 10.1016/j.scico.2009.02.007. URL: http://dx.doi.org/10.1016/j.scico.2009.02.007.

[95] M. Schäfer. "Specification, Implementation and Verification of Refactorings". Diss. Oxford University Computing Laboratory: Wolfson College, 2010.

[96] M. Schäfer, T. Ekman und O. de Moor. "Formalising and Verifying Reference Attribute Grammars in Coq". In: *Proceedings of the 18th European Symposium on Programming Languages and Systems: Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*. ESOP '09. York, UK: Springer-Verlag, 2009, S. 143–159. ISBN: 978-3-642-00589-3. DOI: 10.1007/978-3-642-00590-9_11. URL: http://dx.doi.org/10.1007/978-3-642-00590-9_11.

[97] M. Schäfer, T. Ekman und O. de Moor. "Sound and Extensible Renaming for Java". In: *Proceedings of the 23rd ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications*. OOPSLA '08. Nashville, TN, USA: ACM, 2008, S. 277–294. ISBN: 978-1-60558-215-3. DOI: 10.1145/1449764.1449787. URL: http://doi.acm.org/10.1145/1449764.1449787.

[98] M. Schäfer, J. Dolby, M. Sridharan, E. Torlak und F. Tip. "Correct Refactoring of Concurrent Java Code". In: *Proceedings of the 24th European Conference on Object-Oriented Programming*. ECOOP'10. Maribor, Slovenia: Springer-Verlag, 2010, S. 225–249. ISBN: 3-642-14106-4, 978-3-642-14106-5. URL: http://dl.acm.org/citation.cfm?id=1883978.1883994.

[99] M. Schäfer, A. Thies, F. Steimann und Tip. F. "A Comprehensive Approach to Naming and Accessibility in Refactoring Java Programs." In: 2012, S. 1233–1257.

[100] M. Schäfer, M. Sridharan, J. Dolby und F. Tip. "Refactoring Java Programs for Flexible Locking". In: *International Conference on Software Engineering (ICSE)*. 2011.

[101] D. R. Smith. "KIDS: A Semiautomatic Program Development System". In: *IEEE Trans. Softw. Eng.* 16.9 (Sep. 1990), S. 1024–1043. ISSN: 0098-5589. DOI: 10.1109/32.58788. URL: http://dx.doi.org/10.1109/32.58788.

[102] G. Soares. "Automated Behavioral Testing of Refactoring Engines". In: *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity*. SPLASH '12. Tucson, Arizona, USA: ACM, 2012, S. 105–106. ISBN: 978-1-4503-1563-0. DOI: 10.1145/2384716.2384760. URL: http://doi.acm.org/10.1145/2384716.2384760.

[103] G. Soares. "Making Program Refactoring Safer". In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*. ICSE '10. Cape Town, South Africa: ACM, 2010, S. 521–522. ISBN: 978-1-60558-719-6. DOI: http://doi.acm.org/10.1145/1810295.1810461. URL: http://doi.acm.org/10.1145/1810295.1810461.

[104] G. Soares, R. Gheyi und T. Massoni. "Automated Behavioral Testing of Refactoring Engines". In: *IEEE Trans. Softw. Eng.* 39.2 (Feb. 2013), S. 147–162. ISSN: 0098-5589. DOI: 10.1109/TSE.2012.19. URL: http://dx.doi.org/10.1109/TSE.2012.19.

[105] G. Soares, M. Mongiovi und R. Gheyi. "Identifying Overly Strong Conditions in Refactoring Implementations". In: *Proceedings of the 2011 27th IEEE International Conference on Software Maintenance*. ICSM '11. Washington, DC, USA: IEEE Computer Society, 2011, S. 173–182. ISBN: 978-1-4577-0663-9. DOI: 10.1109/ICSM.2011.6080784. URL: http://dx.doi.org/10.1109/ICSM.2011.6080784.

[106] *Standard for Programming Language C++, Committee Draft*. Mai 2013. URL: http://open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3690.pdf.

[107] B. Steensgaard. "Points-to Analysis in Almost Linear Time". In: *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '96. St. Petersburg Beach, Florida, United States: ACM, 1996, S. 32–41. ISBN: 0-89791-769-3. DOI: http://doi.acm.org/10.1145/237721.237727. URL: http://doi.acm.org/10.1145/237721.237727.

[108] F. Steimann. "Constraint-based Model Refactoring". In: *Proceedings of the 14th International Conference on Model Driven Engineering Languages and Systems*. MODELS'11. Wellington, New Zealand: Springer-Verlag, 2011, S. 440–454. ISBN: 978-3-642-24484-1. URL: http://dl.acm.org/citation.cfm?id=2050655.2050700.

[109] F. Steimann. "The Infer Type Refactoring and its Use for Interface-Based Programming". In: *Journal of Object Technology* 6.2 (2007), S. 99–120. DOI: 10.5381/jot.2007.6.2.a5. URL: http://dx.doi.org/10.5381/jot.2007.6.2.a5.

[110] F. Steimann, C. Kollee und J v. Pilgrim. "A Refactoring Constraint Language and its Application to Eiffel". In: *Proceedings of the 25th European Conference on Object-Oriented programming*. ECOOP'11. Lancaster, UK: Springer-Verlag, 2011, S. 255–280. ISBN: 978-3-642-22654-0. URL: http://dl.acm.org/citation.cfm?id=2032497.2032516.

[111] F. Steimann und J. v. Pilgrim. "Constraint-Based Refactoring with Foresight". In: *Proceedings of the 26th European Conference on Object-Oriented Programming*. ECOOP'12. Beijing, China: Springer-Verlag, 2012, S. 535–559. ISBN: 978-3-642-31056-0. DOI: 10.1007/978-3-642-31057-7_24. URL: http://dx.doi.org/10.1007/978-3-642-31057-7_24.

[112] F. Steimann und J. v. Pilgrim. "Refactorings Without Names". In: *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. ASE 2012. Essen, Germany: ACM, 2012, S. 290–293. ISBN: 978-1-4503-1204-2. DOI: 10.1145/2351676.2351726. URL: http://doi.acm.org/10.1145/2351676.2351726.

[113] F. Steimann und A. Thies. "From Public to Private to Absent: Refactoring Java Programs Under Constrained Accessibility". In: *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*. Genoa. Italy: Springer-Verlag, 2009, S. 419–443. ISBN: 978-3-642-03012-3. DOI: 10.1007/978-3-642-03013-0_19. URL: http://dx.doi.org/10.1007/978-3-642-03013-0_19.

[114] L. Sterling und E. Shapiro. *The Art of Prolog: Advanced Programming Techniques*. Cambridge, MA, USA: MIT Press, 1986. ISBN: 0-262-19250-0.

[115] P. H. Sweany. "Inter-Block Code Motion Without Copies". UMI Order No. GAX93-11408. Diss. Fort Collins, CO, USA: Colorado State University, 1992.

[116] G. Tassey. *The Economic Impacts of Inadequate Infrastructure for Software Testing*. Techn. Ber. National Institute of Standards und Technology, 2002.

[117] *The Eclipse IDE, Luna Release 4.4.1*. Eclipse Foundation. 2014. URL: www.eclipse.org.

[118] A. Thies. “Constraintbasierte Refaktorisierung von Deklarationen in Java”. Diss. FernUniversität in Hagen, 2014.

[119] A. Thies und Steimann. F. “Systematic Testing of Refactoring Tools”. In: *Poster exhibition at the 5th Workshop on Automation of Software Test. AST'10*. 2010.

[120] F. Tip. “Refactoring Using Type Constraints.” In: *SAS*. Hrsg. von Hanne Riis Nielson und Gilberto Filé. Bd. 4634. Lecture Notes in Computer Science. Springer, 23. Aug. 2007, S. 1–17. ISBN: 978-3-540-74060-5. URL: http://dblp.uni-trier.de/db/conf/sas/sas2007.html#Tip07.

[121] F. Tip, A. Kiezun und D. Bäumer. “Refactoring for Generalization Using Type Constraints”. In: *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications*. OOPSLA '03. Anaheim, California, USA: ACM, 2003, S. 13–26. ISBN: 1-58113-712-5. DOI: 10.1145/949305.949308. URL: http://doi.acm.org/10.1145/949305.949308.

[122] E. P. K. Tsang. *Foundations of Constraint Satisfaction*. Computation in Cognitive Science. Academic Press, 1993. ISBN: 978-0-12-701610-8.

[123] C. Ullenboom. *Java ist auch eine Insel. das umfassende Handbuch ; [Programmieren mit der Java Standard Edition Version 5]*. 5., aktualisierte und erw. Aufl. Galileo computing. Bonn: Galileo Press, 2006. 1456 S. ISBN: 3898427471. URL: http://gso.gbv.de/DB=2.1/CMD?ACT=SRCHA&SRT=YOP&IKT=1016&TRM=ppn+503266825&sourceid=fbw_bibsonomy.

[124] M. Vakilian, N. Chen, S. Negara, B. A. Rajkumar, B. P. Bailey und R. E. Johnson. “Use, Disuse, and Misuse of Automated Refactorings”. In: *Proceedings of the 34th International Conference on Software Engineering*. ICSE '12. Zurich, Switzerland: IEEE Press, 2012, S. 233–243. ISBN: 978-1-4673-1067-3. URL: http://dl.acm.org/citation.cfm?id=2337223.2337251.

[125] J. Warren. “A Hierarchical Basis for Reordering Transformations”. In: *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL '84. Salt Lake City, Utah, USA: ACM, 1984, S. 272–282. ISBN: 0-89791-125-3. DOI: 10.1145/800017.800539. URL: http://doi.acm.org/10.1145/800017.800539.

[126] O. White. *Java Tools and Technologies Landscape for 2014*. 2014. URL: http://zeroturnaround.com/rebellabs/java-tools-and-technologies-landscape-for-2014/.

[127] R. Wilhelm und D. Maurer. *Übersetzerbau: Theorie, Konstruktion, Generierung*. Springer-Lehrbuch. Springer Berlin Heidelberg, 1997. ISBN: 9783540616924. URL: https://books.google.de/books?id=P1279NmA9w8C.

[128] W. Yang. “A New Algorithm for Semantics-Based Program Integration”. Diss. University of Wisconsin-Madison, 1990.

[129] E. Yourdon. *Decline and Fall of the American Programmer*. Yourdon Press Computing Series. Upper Saddle River, NJ, USA: PTR Prentice Hall, 1993. ISBN: 9780131919587.

[130] M. Zhang, T. Hall und N. Baddoo. “Code Bad Smells: A Review of Current Knowledge”. In: *J. Softw. Maint. Evol.* 23.3 (Apr. 2011), S. 179–202. ISSN: 1532-060X. DOI: 10.1002/smr.521. URL: http://dx.doi.org/10.1002/smr.521.