

ulm university universität
uulm

Universität Ulm | 89069 Ulm | Germany

Fakultät für
Ingenieurwissenschaften
und Informatik
Institut für
Programmiermethodik und
Compilerbau

2009

Verallgemeinertes Rahmenwerk zur constraintbasierten Testdatenerzeugung aus Programmflussgraphen

Dissertation zur Erlangung des akademischen Grades Dr. rer. nat.
der Fakultät für Ingenieurwissenschaften und Informatik
der Universität Ulm

Vorgelegt von:
Ralf Gerlich
aus Friedrichshafen

Amtierender Dekan:

Prof. Dr.-Ing. Michael Weber

Gutachter:

Prof. Dr. Helmuth Partsch

Prof. Dr. Franz Schweiggert

Datum der Disputation:

28.10.2009

Inhaltsverzeichnis

1 Einleitung **1**

1.1 Aufbau der Arbeit 2

2 Grundlagen **5**

2.1 Softwaretest 5

2.1.1 Teststrategie 6

2.1.2 Qualität von Testfallmengen 7

2.1.3 Testkriterien 8

2.1.4 Statistische Aspekte beim Softwaretest 12

2.2 Relationale Semantik 12

2.3 Kontrollflussgraphen 14

2.3.1 Dominatoren und Postdominatoren 16

2.3.2 Kontrollabhängigkeiten 18

2.3.3 Schleifen 19

2.3.4 Semantik von Kontrollflussgraphen 20

2.3.5 Datenfluss 25

2.3.6 Ausführbarkeit von Wegen 26

2.4 CHR 27

2.4.1 Suche 29

2.4.2 Probabilistic CHR 30

2.5 Static Single Assignment Form 31

3 Motivation **33**

3.1 Grenzen von Testkriterien 34

3.2 Überblick 35

3.3 Stand der Wissenschaft 37

3.3.1 Zufallstest 37

3.3.2 Zielgerichtete Verfahren 39

3.3.3 Lösung der Wegconstraints 42

3.4 Diskussion 43

3.4.1 Klassische und individuelle Testkriterien 43

3.4.2 Verarbeitung von Programmflussgraphen 44

3.4.3 Behandlung nicht ausführbarer Wege 46
3.4.4 Vorwärts- und rückwärtsgerichtete Einschränkungen 48
3.5 Schlussfolgerungen 50

4 Formalisierung von Testkriterien 53
4.1 Grundlagen formaler Testkriterien 53
4.1.1 Testfallsuche 54
4.2 Rückführung auf Ausführungsfolgen 56
4.3 Vereinfachung von *all-edges* 58
4.4 Vereinfachung von *all-nodes* 58
4.5 Definitionsfreie Wege 60
4.6 Allgemeinere Testvorgaben 61
4.7 Schleifenfreie Wege 64

5 Konstruktion von Wegen und Eingaben 67
5.1 Umformung der Spezifikation 68
5.2 Rekursiver Ansatz 70
5.3 Wegvorhersage 72
5.4 Datenflussvorhersage 73
5.4.1 Variablenvorhersage 75
5.4.2 Weitergehende Vorhersagen 76
5.5 Iterativer Ansatz 78

6 Implementierung 81
6.1 Formulierung in CHR 83
6.1.1 Auflösung von Nichtdeterminismen 85
6.1.2 Beispiel 87
6.1.3 Probabilistische Erweiterung 88
6.2 Speicherzugriffe, Arrays und Referenzen 89
6.3 Optimierter Finite-Domain-Löser 92
6.4 Übersetzung in Prolog-Code 94

7 Eigenschaften 95
7.1 Testfallqualität 96
7.2 Terminierung 98
7.2.1 Theoretisches Zufallsverhalten 98
7.2.2 Monte-Carlo-Experimente 102
7.2.3 Allgemeine Terminierung 105
7.2.4 Schlussfolgerungen 107
7.3 Weglängen 108

7.3.1 Theoretisches Verhalten 108
7.3.2 ggT-Algorithmus 111
7.3.3 Schleifen 111
7.3.4 Wegvorhersage 113
7.3.5 Schlussfolgerungen 115
7.4 Analyse der laufzeitbestimmenden Faktoren 116
7.4.1 Datenstrukturen 117
7.4.2 Laufzeit ohne Vorhersage 119
7.4.3 Laufzeit mit Wegvorhersage 120
7.4.4 Laufzeit mit Variablenvorhersage 120
7.4.5 Laufzeit mit Knotenvermeidung 121
7.4.6 Schlussfolgerungen 122
7.5 Beispiele 122
7.5.1 Berechnung der Fibonacci-Folge 123
7.5.2 Verschachtelte Schleife 128
7.5.3 Vorgegebener Rückgabewert 131
7.5.4 Knotenabdeckung 137
7.5.5 Dreiecksklassifizierung 140
7.5.6 Schlussfolgerungen 141

8 Schlussfolgerungen und Ausblick **145**

Literaturverzeichnis **149**

1 Einleitung

Testen ist eine der wichtigsten analytischen Qualitätssicherungsmaßnahmen im Rahmen der Softwareentwicklung. Systematisches Vorgehen, Reproduzierbarkeit und repräsentative Testvorgaben vorausgesetzt, kann durch Softwaretest die Qualität der zu prüfenden Software in einer realen oder äußerst realitätsnahen Umgebung eingeschätzt werden.

Manuelle Vorbereitung und Ausführung eines Softwaretests sind recht aufwändig. Üblicherweise wird der Testaufwand auf etwa die Hälfte des Gesamtaufwands für ein Softwareprojekt geschätzt[FPB95]. Dementsprechend groß ist das Potenzial zur Aufwandseinsparung durch Automation im Bereich des Softwaretests.

Wir stellen in dieser Arbeit ein neues theoretisches Rahmenwerk und ein Verfahren für die automatisierte Auswahl von Testeingaben für den Modultest vor. Dafür definiert der Testingenieur Vorgaben, anhand derer ein Ausführungsweg durch den Prüfling und eine entsprechende Eingabe ausgewählt wird. Eine Testvorgabe kann zum Beispiel der Durchlauf durch einen bestimmten Knoten sein.

Zu vielen der konstruierbaren Wege existiert jedoch keine Eingabe, die zu ihrer Ausführung führt. Diese Wege werden als nicht-ausführbar bezeichnet. Das Verfahren muss solange wiederholt Wege auswählen, bis es die gewünschte Zahl von Eingaben geliefert hat. Je besser ein Verfahren also die Konstruktion nicht-ausführbarer Wege vermeiden kann, umso geringer ist die Laufzeit, in der die gewünschten Eingaben erzeugt werden können.

In einigen bekannten Verfahren sind bereits Strategien zur Vermeidung solcher nicht-ausführbaren Wege umgesetzt. Das eine Verfahren, das am weitesten in dieser Hinsicht fortgeschritten ist, erfordert jedoch die Darstellung des zu testenden Programms in einer in der Praxis nicht üblichen `while-if`-Form, was den Einsatz für Sprachen wie Java oder C erschwert. Die anderen Verfahren akzeptieren zwar den Kontrollflussgraphen des Programms als Eingabe, bieten dafür aber keine so guten Strategien zur Vermeidung nicht-ausführbarer Wege.

Zudem berücksichtigt keines der bekannten Verfahren die Möglichkeit, die Wegkonstruktion und damit die Analyse auch entgegen der normalen Ausführungs-

richtung durchzuführen. Diese Analyserichtung bietet jedoch bei vielen Arten von Testvorgaben einen erheblichen Vorteil.

Die bekannten Verfahren sind außerdem meist auf ein einzelnes Testkriterium ausgelegt und andere Kriterien sind schwer oder gar nicht als Vorgabe für diese Verfahren darstellbar.

Unser Verfahren akzeptiert die zu analysierenden Programme in Form von erweiterten Kontrollflussgraphen, wie sie als Basisblockgraphen auch direkt von heutigen Compilern erzeugt werden. Es unterstützt die meisten der bekannten Vermeidungstechniken ebenso wie die Konstruktion der Wege in und entgegen der normalen Ausführungsrichtung.

Wir untersuchen in dieser Arbeit auch erstmals ausführlich die Auswirkungen dieser Strategien auf Laufzeit und Art der gewählten Wege aus theoretischer Sicht ebenso wie auf Basis von Experimenten. Eine ähnliche qualitative und quantitative Betrachtung dieser Strategien ist aus der Literatur nicht bekannt. Erst damit wird aber eine Aussage möglich, ob überhaupt eine allgemeingültige optimale Strategie zur constraintbasierten Testdatenerzeugung existiert.

Weiterhin ermöglicht unser Verfahren die Anwendung aller klassischen Testkriterien wie Knoten- und Kantenabdeckung sowie der üblichen datenflussorientierten Kriterien ebenso wie domänen- oder anwendungsspezifische Testvorgaben, die über diese bekannten Kriterien hinausgehen.

Das Verfahren ist offen für Erweiterungen, die auf Basis des ebenfalls eingeführten theoretischen Rahmenwerks als korrekt bewiesen und auf ihre Eigenschaften im Bezug auf die Laufzeit und das Profil der erzeugten Testeingaben untersucht werden können.

1.1 Aufbau der Arbeit

Die vorliegende Arbeit ist in acht Kapitel aufgeteilt, wovon diese *Einleitung* das erste ist. Kapitel 2 enthält eine Darstellung der wesentlichen für das Verständnis der Arbeit notwendigen *Grundlagen*.

In Kapitel 3 wird die *Motivation* für diese Arbeit und der *Stand der Wissenschaft* dargelegt. Wir betrachten einige wichtige *Anforderungen* für die constraintbasierte Testdatenerzeugung in der industriellen Praxis und diskutieren diese in Bezug auf die bekannten Verfahren.

Unser Beitrag ist in den restlichen Kapiteln dargestellt.

Wir formalisieren in Kapitel 4 einige weit verbreitete *Testkriterien* und führen sie auf Konjunktionen einer kleinen Menge von Grundrelationen zurück. Ebenso führen wir dort *Hilfsmittel* zur Darstellung weiterer Testvorgaben ein.

In Kapitel 5 leiten wir das *theoretische Rahmenwerk* zur Lösung von Systemen der zuvor eingeführten Grundrelationen her.

Die Umsetzung dieses Rahmenwerks als CHR-Programm und die wesentlichen Details einer prototypischen *Implementierung* stellen wir in Kapitel 6 vor.

Das *Terminierungsverhalten*, die *Länge der erzeugten Wege* und die *Laufzeit* ermitteln wir in Kapitel 7 auf Basis theoretischer Analysen und experimenteller Messungen, und vergleichen diese Eigenschaften mit denen der anderen Verfahren, soweit bekannt.

Schließlich fassen wir die *Ergebnisse* der Arbeit in Kapitel 8 zusammen und bieten einen Ausblick auf mögliche Erweiterungen und Anwendungen.

2 Grundlagen

Wir betrachten nun die Grundlagen des Softwaretests und führen einige Hilfsmittel zur Darstellung der Semantik von Programmen ein. Weiterhin bieten wir einen kurzen Überblick über die Sprache CHR und erläutern die Static-Single-Assignment-Form als besondere Darstellung für imperative Programme.

2.1 Softwaretest

Unter Softwaretest versteht man die Ausführung von Software mit dem Ziel, Fehler zu finden[MBTS04]. Dazu wird das zu prüfende System – kurz: *Prüfling* – durch Eingaben stimuliert und seine Reaktionen werden aufgezeichnet. Das System gilt dann als fehlerhaft, wenn es sich nicht entsprechend der Erwartungen verhält.

Diese Erwartungen können etwa in Form eines Spezifikationsdokuments exakt festgelegt sein, aber sich auch implizit aus *Vorstellungen* des Kunden ergeben.

Der gesamte Testablauf wird in Testfälle unterteilt. Zu einem *Testfall* gehört dabei sowohl die Eingabe, mit der der Prüfling zu stimulieren ist, als auch die erwartete Reaktion des Systems.

Da jeder Testfall lediglich ein Beispiel einer möglichen Ausführung darstellt, kann durch Test nur die Anwesenheit von Fehlern, nicht jedoch ihre Abwesenheit nachgewiesen werden[Dij72]. Für einen Korrektheitsnachweis durch Test müsste das System mit sämtlichen möglichen Eingaben stimuliert und die Korrektheit jeder einzelnen Reaktion festgestellt werden. Die Menge dieser Eingaben ist in der Praxis jedoch unendlich oder zumindest zu groß, um sie vollständig durchgehen zu können.

Somit ist der Softwaretest ein *Experiment* – ähnlich naturwissenschaftlichen Experimenten –, dessen Ziel die Prüfung der These von der Korrektheit der stimulierten Software ist. Wird ein Fehler gefunden, ist die These eindeutig widerlegt. Wird jedoch kein Fehler gefunden, so kann die These abhängig von der Repräsentativität der gewählten Testfälle als gestützt, nicht jedoch als bestätigt gelten.

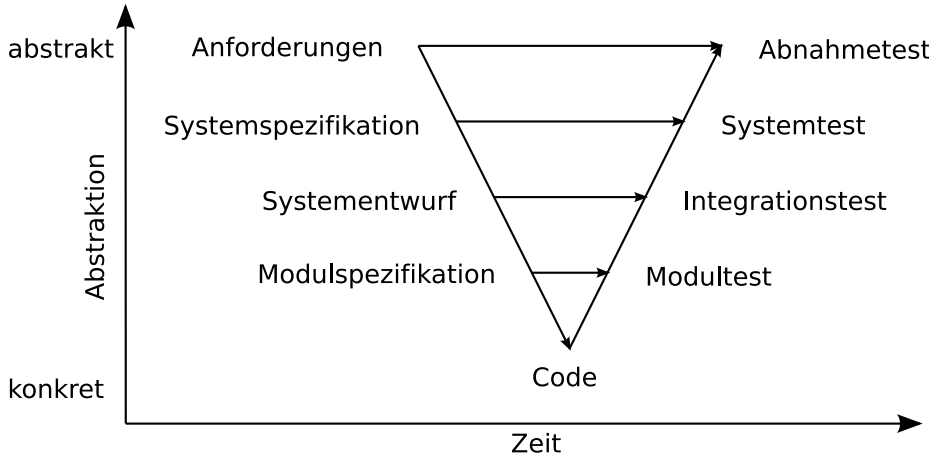


Abbildung 2.1: Beziehungen zwischen Entwicklung und Test

Zum Softwaretest gehören als Teilaktivitäten die Planung im Rahmen des Projektplans, die Vorbereitung durch Auswahl der Testfälle, die Ausführung des Tests durch Stimulierung des Prüflings und die Auswertung der Ergebnisse.

2.1.1 Teststrategie

Üblicherweise wird beim Softwaretest eine Bottom-Up-Strategie angewendet, d.h. es werden zunächst einzelne Module auf ihre Funktion getestet und erst dann Schritt für Schritt zum Gesamtsystem zusammengesetzt und jeweils im Teilverbund getestet. So ist jedem Verfeinerungsschritt im Vorgehensmodell ein Testschritt gegenübergestellt[MBTS04] (s. Abbildung 2.1).

Diese Vorgehensweise ergibt sich aus der Erkenntnis, dass für die Erfüllung der Spezifikation einer Abstraktionsebene zunächst notwendigerweise die Spezifikationen der darunterliegenden Abstraktionsstufen erfüllt sein müssen. Es ist somit nicht sinnvoll, das gesamte System zu testen, bevor nicht die einzelnen Module geprüft wurden.

Man unterscheidet zwischen

- dem *Modultest*, bei dem zunächst einzelne Module – also Funktionen, Klassen oder Pakete – gegenüber ihrer *Modulspezifikation* getestet werden,
- dem *Integrationstest*, bei dem die bereits getesteten Module schrittweise zu größeren Einheiten zusammengesetzt werden und ihr Zusammenwirken gegenüber dem *Systementwurf* getestet wird,
- dem *Systemtest*, bei dem das Gesamtsystem gegenüber der *Systemspezifikation* getestet wird, und

- dem *Abnahmetest*, bei dem das Gesamtsystem gegenüber den *Systemanforderungen* getestet wird.

Diesen einzelnen Testschritten liegen unterschiedliche Anforderungen zur Testprozedur zugrunde. Während beim Modultest auf Funktionsebene meist einzelne Funktionen durch Stimulation ihrer Parameter getestet werden, stehen beim Integrations-, System- und Abnahmetest die Interaktion zwischen den Modulen sowie die Schnittstellen im Vordergrund.

Unterschieden wird außerdem in *Blackbox-* und *Glassbox-* bzw. *Whitebox-Test*. Beim Blackboxtest werden die Testfälle nur auf Grundlage der nach außen sichtbaren Schnittstelle und der Spezifikation des Prüflings festgelegt, während beim Glassboxtest auch die innere Struktur des Codes mit in die Auswahl einbezogen wird.

2.1.2 Qualität von Testfallmengen

Da nicht alle möglichen Eingaben im Softwaretest durchprobiert werden können, bleibt lediglich lediglich die Möglichkeit, die Korrektheit des Prüflings mit einer möglichst geringen Irrtumswahrscheinlichkeit plausibel zu machen.

Bei einem Test eines großen und komplexen Systems durch eine kleine Anzahl von Testfällen ist selbst ohne Auftreten eines Fehlers diese Irrtumswahrscheinlichkeit als groß anzunehmen. Ebenso kann eine große Anzahl von Testfällen allein keine kleine Irrtumswahrscheinlichkeit begründen, wenn die Testfälle ungünstig verteilt werden.

Die Qualität oder Eignung einer Testfallmenge ist durch zwei wesentliche Anforderungen bestimmt:

- Durch die Testfallmenge sollen möglichst viele unterschiedliche Fehler aufgedeckt werden, und
- eine geringe Irrtumswahrscheinlichkeit für die Annahme der Fehlerfreiheit begründet werden, sofern keine Fehler gefunden werden.

Aufgrund der äußeren Umstände in einem Projekt – begrenzte Ressourcen, Ökonomieerfordernis – soll zudem der Aufwand zur Erfüllung dieser zwei Anforderungen möglichst gering sein.

Beim manuellen Softwaretest kann der Aufwand nur durch die Minimierung der Testfallmengen reduziert werden. Dabei ist die Abwägung zwischen dieser Reduktion und der Erfüllung der primären Anforderungen an die Testfallmengen kritisch.

2.1.3 Testkriterien

Da weder vor noch nach dem Test abschließend bekannt ist, welche und wie viele Fehler der Prüfling enthält, kann nicht direkt beurteilt werden, ob eine Testfallmenge die richtigen, zu viele oder zu wenige Testfälle enthält. Stattdessen muss auf indirekte Kriterien zurückgegriffen werden.

Zwei wesentliche Ansätze zur Beurteilung der Eignung einer Testfallmenge sind die Mutationsanalyse und die Abdeckungsanalyse, von denen die letztere in der Praxis aufgrund des geringeren Aufwands häufiger zum Einsatz kommt.

Für die *Mutationsanalyse* wird der ursprüngliche Prüfling modifiziert und dem Test unterzogen[ORZ93, OLR+94]. Die modifizierten Versionen des Prüflings – sogenannte *Mutanten* – sollten bei mindestens einem Testfall eine vom eigentlichen Prüfling abweichende Reaktion hervorrufen. Ist dies nicht der Fall, kann die Mutation von der Testfallmenge nicht erkannt werden. Dies ist entweder auf eine ungeeignete Testfallmenge oder auf eine tatsächliche Äquivalenz zwischen Mutant und Prüfling zurückzuführen. Da letzteres nicht entscheidbar ist, müssen nicht erkannte Mutanten oft einer manuellen Analyse unterzogen werden.

Bei der *Abdeckungsanalyse* wird durch Kriterien festgelegt und entsprechende Metriken festgestellt, welche Elemente der Spezifikation oder des Programms während des Tests untersucht werden[ZHM97]. Dabei sollen diese Elemente meist vollständig abgedeckt werden, was aber bei einigen Kriterien nicht möglich ist.

Spezifikationsorientierte Abdeckungskriterien basieren meist auf der Zuordnung der Anforderungen zu Testfällen, die speziell ihre Erfüllung bzw. Nichterfüllung überprüfen.

Betrachten wir als Beispiel eine Spezifikation für ein Programm, das ein Dreieck anhand seiner Seitenlängen a, b und c als gleichseitig, gleichschenklig, unregelmäßig oder unzulässig klassifiziert. Eine der Anforderungen ist, dass die Eingabe genau dann als gleichschenkliges Dreieck eingestuft werden muss, wenn die Längen ein Dreieck beschreiben und genau zwei der Seitenlängen gleich sind. Hierfür könnten etwa die Eingaben $a = 2$, $b = 2$, $c = 1$ und $a = 2$, $b = 2$, $c = 0$ verwendet werden. Erstere muss als gleichschenklig eingestuft werden, während dies bei letzterer nicht der Fall sein darf, da es sich schon nicht um die Seiten eines Dreiecks handelt.

Bei der *Äquivalenzklassenanalyse* wird die Eingabemenge laut Spezifikation so in Äquivalenzklassen eingeteilt, dass eine Eingabe aus einer Klasse bezüglich ihrer Möglichkeiten, einen Fehler zu identifizieren, als äquivalent zu den anderen Eingaben ihrer Klasse angesehen werden kann[MBTS04]. Aus jeder dieser Äquivalenz-

klassen ist ein Testfall zu wählen. In obigem Beispiel bietet die Spezifikation selbst bereits die Äquivalenzklassen in Form der unterschiedlichen Klassifikationen der Eingabe an.

Die *Grenzfallanalyse* ist ein Spezialfall der Äquivalenzklassenanalyse. Hier werden Repräsentanten der Äquivalenzklassen als Testfälle gewählt, die möglichst nah an deren Grenzen im Eingaberaum liegen[MBTS04]. Unterscheidet etwa die Spezifikation danach, ob ein Eingabewert `a` positiv oder nicht ist, im Prüfling wird aber die Bedingung `a>=0` (nicht-negativ) verwendet, so soll dies durch die Grenzfallanalyse festgestellt werden.

Strukturbasierte Abdeckungskriterien werden auf Basis des Kontrollflussgraphen eines Programms definiert (sog. *Codeabdeckung*). In diesem werden Anweisungen als Knoten und die Abfolgen dieser Anweisungen im Programm durch gerichtete Kanten zwischen den entsprechenden Knoten beschrieben. Ein Start- und ein Endknoten definieren Einsprung- und Schlusspunkt des Programms.

Jedem Testfall ist jener Weg zugeordnet, der bei Ausführung des Programms mit der jeweiligen Eingabe durchlaufen wird. Die Menge der Wege, die bei Ausführung einer bestimmten Testfallmenge durchlaufen werden, bezeichnen wir mit W.

Weiterhin kann der Datenfluss betrachtet werden, also die Zusammenhänge zwischen Zuweisungen und Verwendungen von Variablen. Dabei wird eine Zuweisung einer Variablen als *def* (definition), eine Verwendung im Rahmen einer Verzweigungsbedingung als *p-use* (predicate use) und eine Verwendung im Rahmen einer Berechnung, die keine Verzweigungsbedingung ist, als *c-use* (computational use) bezeichnet[RW82].

Ein Weg wird als *definitionsfrei* bzgl. einer Variablen x bezeichnet, wenn er außer seinen Start- und Endknoten keinen Knoten durchläuft, der eine Definition von x enthält. Wir sagen, dass eine Definition d eine Nutzung u *erreicht*, wenn es einen definitionsfreien Weg von d nach u gibt. Wird eine Nutzung von keiner Definition erreicht, so ist dies mit großer Wahrscheinlichkeit bereits ein Programmfehler.

Hierauf aufbauend können die folgenden Kriterien definiert werden[RW82]:

- *all-nodes*: Für jeden Knoten n im Kontrollflussgraphen existiert mindestens ein Weg in W, in dem n durchlaufen wird.
- *all-edges*: Für jede Kante e im Kontrollflussgraphen existiert mindestens ein Weg in W, in dem e durchlaufen wird. Dieses Kriterium wird auch als *Verzweigungsabdeckung* oder *DC* (*decision coverage*) bezeichnet.

- *all-defs*: Für jede Variable x und jede Definition d von x existiert mindestens ein Weg in W, der einen definitionsfreien Teilweg von d zu einer Verwendung von x enthält.
- *all-p-uses*: Für jede Variable x, jede Definition d von x und jede von d erreichte p-use p von x existiert mindestens ein Weg in W, der einen definitionsfreien Teilweg von d zu p enthält.
- *all-c-uses/some-p-uses*: Für jede Variable x, jede Definition d von x und jede von d erreichte c-use c von x existiert mindestens ein Weg in W, der einen definitionsfreien Teilweg von d zu c enthält. Erreicht d keine c-use von x, so muss für mindestens eine von d erreichte p-use p von x ein Weg in W existieren, der einen definitionsfreien Teilweg von d zu p enthält.
- *all-p-uses/some-c-uses*: Für jede Variable x, jede Definition d von x und jede von d erreichte p-use p von x existiert mindestens ein Weg in W, der einen definitionsfreien Teilweg von d zu p enthält. Erreicht d keine p-use von x, so muss für mindestens eine von d erreichte c-use c von x ein Weg in W existieren, der einen definitionsfreien Teilweg von d zu c enthält.
- *all-uses*: Für jede Variable x, jede Definition d von x und jede von d erreichte Verwendung u von x muss ein Weg in W existieren, der einen definitionsfreien Teilweg von d nach u enthält.
- *all-du-paths*: Für jede Variable x, jede Definition d von x, jede von d erreichte Verwendung u von x und jeden schleifenfreien Teilweg P von d nach u muss ein Weg in W existieren, der P enthält.
- *all-paths*: W muss jeden Weg enthalten, der vom Start- zum Endknoten des Programmflussgraphen führt.

Wir unterteilen weiterhin die mit den Verzweigungen im Kontrollflussgraphen verbundenen logischen Ausdrücke in Bedingungen. Eine *Bedingung* ist ein atomares Element eines logischen Ausdrucks, der nicht mehr weiter in logische Teilausdrücke unterteilt werden kann. So sind etwa im Ausdruck $a < b \vee a = 0$ die Bedingungen $a < b$ und $a = 0$ enthalten.

Auf dieser Basis lassen sich weitere Kriterien definieren:

- *all-conditions*: Jede Bedingung im Programm muss in je mindestens einem Testfall erfüllt bzw. nicht erfüllt sein. Dieses Kriterium wird auch als *Bedingungsabdeckung* oder *C/DC* (*condition/decision coverage*) bezeichnet.
- *MC/DC (modified condition/decision coverage)*: Für jede Bedingung muss es je einen Testfall geben, in dem diese Bedingung die Entscheidung unabhängig beeinflusst und diese je einmal erfüllt und einmal nicht erfüllt ist. Dieses

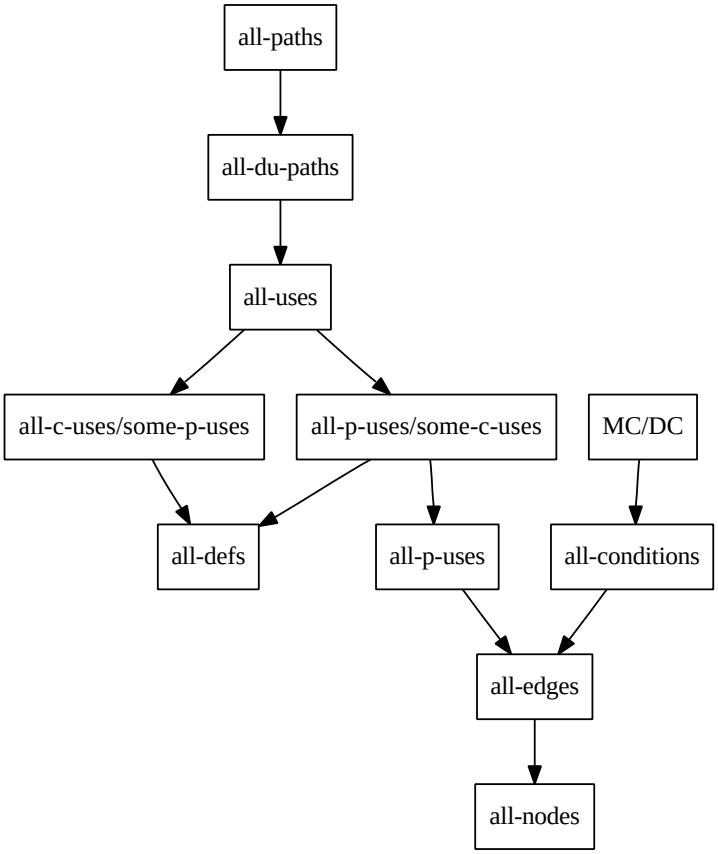


Abbildung 2.2: Implikationsbeziehungen zwischen Abdeckungskriterien[RW82]

> Kriterium ist bei Verwendung von Kurzschlusscode[ASU97] äquivalent zu *all-edges*.

Das Kriterium all-paths ist bei Existenz von Schleifen im Kontrollflussgraphen nicht durch eine endliche Testfallmenge erfüllbar.

Auch die anderen Kriterien können unerfüllbar sein, da bestimmte der geforderten Wege ggf. für keine der möglichen Eingaben ausgeführt werden. Im Fall von all-nodes, all-edges, all-conditions und MC/DC ist dies nur durch einen logischen Fehler zu erklären. Entweder sind die betroffenen Knoten, Kanten oder Bedingungen überflüssig, oder die Bedingungen sind fehlerhaft formuliert.

Zwischen den Kriterien besteht eine Implikationsbeziehung in dem Sinne, dass Testfallmengen, die eines der Kriterien erfüllen, auch bestimmte andere Kriterien erfüllen. Ein Kriterium c_1 impliziert ein Kriterium c_2 genau dann, wenn es in Abbildung 2.2 einen Weg von c_1 nach c_2 gibt.

2.1.4 Statistische Aspekte beim Softwaretest

Analog zu naturwissenschaftlichen Experimenten können auch beim Softwaretest statistische Analysen sinnvoll sein. Hierfür werden meist Methoden des *Zufallstest* angewendet[Ham94]. Dabei werden die Testeingaben zufällig und unabhängig voneinander ausgewählt, gleichverteilt oder nach einem vorgegebenen Eingabeprofil, das beispielsweise die vorherrschenden Bedingungen im täglichen Betrieb formalisiert.

Auf Basis der Ergebnisse eines solchen Zufallstests kann ggf. eine quantitative Abschätzung der *Restfehlerwahrscheinlichkeit* vorgenommen werden[HT90]. Dazu wird angenommen, dass das Programm auf eine Eingabe mit einer Wahrscheinlichkeit kleiner als eine oberen Schranke p_0 fehlerhaft reagiert, sofern die Eingabe entsprechend der angenommenen Eingabeverteilung ausgewählt wird.

Testen wir das Programm und finden keinen Fehler, so können wir die Hypothese mit einer Irrtumswahrscheinlichkeit α als korrekt annehmen. Dabei besteht ein Zusammenhang zwischen der Testfallmenge, p_0 und α.

Im Allgemeinen können wir die Irrtumswahrscheinlichkeit α nur reduzieren, indem wir von einer höheren Restfehlerwahrscheinlichkeit p_0 ausgehen und/oder die Größe der Testfallmenge erhöhen.

2.2 Relationale Semantik

Wir führen eine vereinfachte relationale Semantik für imperative Programme ein, analog zu bereits bekannten relationalen[HH98] und funktionalen[Par90] Ansätzen.

Definition 2.1: Eine *Spezifikation* ist eine binäre Relation $\mathcal{S} \subset A \times B$ mit Quelle A und Zielbereich B.

Ein Programm P *erfüllt* eine Spezifikation $\mathcal{S}$, wenn $x\ \mathcal{S}\ y$ für jede Eingabe x und jede zu x gehörende Ausgabe y von P gilt.

Eine Spezifikation $\mathcal{S}$ ist *eindeutig für* x, wenn es maximal ein y gibt mit $x\ \mathcal{S}\ y$.

Ist eine Spezifikation eindeutig für alle x, so wird sie kurz als *eindeutig* bezeichnet.

Eine Spezifikation wird als *mehrdeutig* (für x) bezeichnet, wenn sie nicht eindeutig (für x) ist.

Die Spezifikation für ein Programm kann zum Beispiel in Form eines Spezifikationsdokuments oder des Programmcodes in einer definierten Programmiersprache angegeben werden.

Die Form von Eingabe und Ausgabe ist nicht explizit festgelegt. Möglich ist somit die Definition zum Beispiel als Speicherzustand, als Tupel von Parametern oder als Zeichenstrom.

So kann beispielsweise die Spezifkation einer Anweisung `x:=x+1` mit einer Ganzzahlvariablen `x` durch

$$x \; \mathcal{S}_{x:=x+1} \; x' \Leftrightarrow x, x' \in \mathbb{N} \wedge x' = x + 1$$

beschrieben werden.

Eine Spezifikation entsprechend Definition 2.1 stellt sämtliche möglichen Ein- und Ausgaben des Programms dar. Sogenannte globale Variablen, wie sie in vielen Programmiersprachen zur Verfügung stehen, sind somit bei einer solchen Spezifikation Bestandteil von Ein- bzw. Ausgabe.

Ein Beispiel für ein Programm mit explizit mehrdeutiger Spezifikation sind Zufallszahlengeneratoren und -quellen. Ebenso kann eine Spezifikation lediglich bestimmte Anforderungen bzgl. der Ausgabe formulieren, durch die diese nicht eindeutig bestimmt ist. Sei die Spezifikation $\mathcal{S}_f$ einer Funktion $f\,(x)$ gegeben durch $x \; \mathcal{S}_f \; y \Leftrightarrow x \geq 0 \wedge y^2 = x$, so ist diese Spezifikation auch durch $f\,(x) = -\sqrt{x}$ erfüllt. Solange lediglich irgendeine Zahl y mit $y^2 = x$ gesucht ist, stellt die Mehrdeutigkeit nicht notwendigerweise einen Fehler dar.

Darüberhinaus erfüllt jedes Programm die universelle Spezifikation 1 mit $\forall x, y : x \; 1 \; y$. In dieser Arbeit werden wir Spezifikationen vor allem zur Definition des Ausführungsverhaltens von Programmen verwenden. Je genauer die verwendete Spezifikation das betrachtete Programm beschreibt, umso genauer sind auch die Aussagen, die etwa unser Verfahren zur Testeingabesuche daraus ableiten kann.

Zeitliche Aspekte der Programmausführung sind nur durch Einführung der Zeit als Ein-, Ausgabe- und Zustandselement darstellbar. Die Ausführungszeiten von Programmen auf heutigen Architekturen sind allerdings von einer Vielzahl von Einflüssen der Hardware abhängig, etwa Caching, Sprungvorhersage oder Änderungen der Reihenfolge in der Prozessorpipeline. Somit können die zeitlichen Charakteristika von tatsächlichen Programmen nur näherungsweise beschrieben werden.

Programme können durch Sequenzierung oder Verzweigung kombiniert werden. Die dem kombinierten Programm entsprechenden Spezifikationen können durch Konkatenation bzw. Alternierung der Teil-Spezifikationen konstruiert werden.

Definition 2.2: Die *Konkatenation* $\mathcal{S}_1;\mathcal{S}_2$ bzw. *Alternierung* $\mathcal{S}_1 \cup \mathcal{S}_2$ zweier Spezifikationen $\mathcal{S}_1$ und $\mathcal{S}_2$ sind wie folgt definiert:

$$\begin{aligned} \forall x, z : x\ (\mathcal{S}_1;\mathcal{S}_2)\ z &\quad\Leftrightarrow\quad \exists y : x\ \mathcal{S}_1\ y \wedge y\ \mathcal{S}_2\ z \\ \forall x, y : x\ (\mathcal{S}_1 \cup \mathcal{S}_2)\ y &\quad\Leftrightarrow\quad x\ \mathcal{S}_1\ y \vee x\ \mathcal{S}_2\ y \end{aligned}$$

Die Konkatenation ist assoziativ, die Alternierung ist assoziativ und kommutativ. Wenn keine Verwechslung möglich ist, schreiben wir statt $\mathcal{S}_1;\mathcal{S}_2$ kurz $\mathcal{S}_1\mathcal{S}_2$. Ebenso bindet ; stärker als $\cup$, so dass $\mathcal{S}_1;\mathcal{S}_2 \cup \mathcal{S}_3 = (\mathcal{S}_1;\mathcal{S}_2) \cup \mathcal{S}_3$ und $\mathcal{S}_1 \cup \mathcal{S}_2;\mathcal{S}_3 = \mathcal{S}_1 \cup (\mathcal{S}_2;\mathcal{S}_3)$.

Definition 2.3: Eine Spezifikation $\mathcal{S}$ wird als *Bedingung* bezeichnet, wenn gilt

$$\forall x, y : x\ \mathcal{S}\ y \Rightarrow x = y$$

Für jedes x gilt die Bedingung genau dann als *erfüllt*, wenn $x\ \mathcal{S}\ x$.

Wir führen eine Bedingung $\top$ ein, für die gilt: $\forall x : x\ \top\ x$.

Das Urbild einer Bedingung ist also genau jene Menge der Eingaben, für die die Bedingung erfüllt ist. Die Ausgabe einer Bedingung ist identisch mit ihrer Eingabe. Ihr kommt somit in Kombination mit der Konkatenation eine Filterfunktion zu.

Definition 2.4: Die *Negation* $\overline{\mathcal{S}}$ einer Bedingung $\mathcal{S}$ ist definiert durch

$$\forall x : x\ \overline{\mathcal{S}}\ x \Leftrightarrow \neg\,(x\ \mathcal{S}\ x)$$

Die Spezifikation einer Anweisung `if C then T else E` kann somit dargestellt werden als $\mathcal{S}_C\mathcal{S}_T \cup \overline{\mathcal{S}_C}\mathcal{S}_E$, wobei $\mathcal{S}_C$ die Spezifikation der Bedingung `C` ist.

2.3 Kontrollflussgraphen

Zur Darstellung des Kontrollflusses in einem Programm werden häufig Kontrollflussgraphen verwendet, zum Beispiel im Compilerbau[ASU97] in Form von Basisblockgraphen.

In einem solchen Graphen repräsentieren die Knoten Anweisungen und die Kanten die möglichen Abfolgen zwischen den Knoten. Kontrollflussgraphen besitzen einen Wurzelknoten, der den Einsprungspunkt repräsentiert. Außerdem enthalten sie einen oder mehrere Endknoten, von denen aus keine weitere Kante verläuft.

Unsere Definition lässt ohne Beschränkung der Allgemeinheit nur einen Endknoten zu, um die weiteren Betrachtungen zu vereinfachen. Existieren mehrere Knoten

ohne ausgehende Kanten, so können diese mit einem einzelnen neuen Knoten verbunden werden. Letzterer wird dann zum Endknoten.

Definition 2.5: Ein *Kontrollflussgraph* ist ein gerichteter Graph $CFG\,\langle N, E, r, e\rangle$ mit

- einer endlichen Knotenmenge N,
- einer Kantenmenge E mit $E \subseteq N \times N$,
- einem *Startknoten* r, wobei $r \in N$, und
- einem *Endknoten* e, wobei $e \in N$.

Für zwei Knoten a und b schreiben wir $a \to b$ genau dann, wenn $(a, b) \in E$. Wir bezeichnen in diesem Fall a als einen *Vorgänger* von b und b als einen *Nachfolger* von a.

Der Knoten r hat keine Vorgänger und der Knoten e hat keine Nachfolger.

Eine Folge von Knoten $(v_1, \ldots, v_n)$ mit $n \geq 1$ bezeichnet man als *Weg* (von v_1 nach v_n) in CFG, wenn gilt

$$\forall i \in \{1, \ldots, n-1\} : v_i \to v_{i+1}$$

Die *Länge* des Weges wird mit $n - 1$ angegeben.

Für jeden Knoten $n \in N$ gibt es einen Weg von r nach n und einen Weg von n nach e.

Ein Weg $(w_1, \ldots, w_m)$ ist ein *Teilweg* eines Weges $(v_1, \ldots, v_n)$, wenn es ein $i \in \{1, \ldots, n - m + 1\}$ gibt, so dass für alle $j \in \{1, \ldots, m\}$ gilt $w_j = v_{i+j-1}$.

Für jeden Knoten n in CFG ist insbesondere (n) ein Weg der Länge 0.

Ein Beispiel für einen Kontrollflussgraphen ist in Abbildung 2.3 dargestellt. Der zugehörige Startknoten ist r, der Endknoten ist e und für die Knoten- bzw. Kantenmenge gilt $N = \{r, 1, 2, 3, e\}$ und $E = \{(r, 1), (1, e), (1, 2), (1, 3), (2, 1), (3, 1)\}$.

Definition 2.6: Der Weg $(v_1, \ldots, v_n)$ wird als *schleifenfrei* bezeichnet, wenn es kein Paar (i, j) gibt mit $1 \leq i < j \leq n$ und $v_i = v_j$.

Definition 2.7: Ein Knoten b gilt als *erreichbar* von Knoten a aus in CFG, wenn es einen Weg von a nach b in CFG gibt.

Definition 2.8: Sei CFG ein Kontrollflussgraph und $a, b \in N$.

Dann gilt $a \to^* b$ genau dann, wenn b von a aus in CFG erreichbar ist.

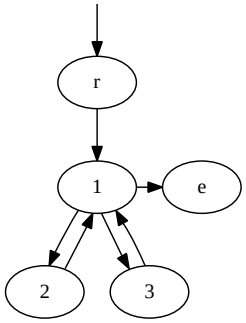


Abbildung 2.3: Ein Kontrollflussgraph

Weiterhin gilt $a \rightarrow^+ b$ genau dann, wenn es einen Weg von a nach b in CFG gibt, dessen Länge mindestens 1 ist.

Lemma 2.1: $\rightarrow^+$ *und* $\rightarrow^*$ *sind transitiv.*

$\rightarrow^*$ *ist reflexiv.*

Beweis. Sei $a \rightarrow^* b$ und $b \rightarrow^* c$ für beliebige Knoten a, b und c. Dann gibt es einen Weg von a nach b und einen Weg von b nach c. Diese Wege lassen sich zu einem Weg von a nach c zusammensetzen und es folgt $a \rightarrow^* c$.

Analoges gilt für $\rightarrow^+$. Da hier außerdem die Wege jeweils die Mindestlänge 1 haben, hat auch der zusammengesetzte Weg mindestens die Länge 1.

Für jeden Knoten a gibt es einen Weg (a) von a nach a. Somit gilt $a \rightarrow^* a$. □

2.3.1 Dominatoren und Postdominatoren

Aus den Bereichen der Kontrollflussanalyse und der Programmoptimierung ist bekannt, dass es in einem Kontrollflussgraphen bestimmte Knoten geben kann, die auf jedem Weg vom Startknoten r zu einem Knoten n bzw. von n zum Endknoten e durchlaufen werden[ASU97]. Diese werden als Dominatoren bzw. Postdominatoren bezeichnet.

Definition 2.9: Seien a und b Knoten des Kontrollflussgraphen $CFG\ \langle N, E, r, e\rangle$.

- a *dominiert* b genau dann, wenn $a \neq b$ und es für jeden Weg $(v_1, \ldots, v_n)$ von r nach b ein $j \in \{2, \ldots, n-1\}$ gibt mit $v_j = a$. Wir schreiben $a\ dom\ b$ und bezeichnen die Knoten, die b dominieren als die *Dominatoren* von b.

- a *postdominiert* b genau dann, wenn $a \neq b$ und es für jeden Weg $(v_1, \ldots, v_n)$ von b nach e ein $j \in \{2, \ldots, n-2\}$ gibt mit $v_j = a$. Wir schreiben $a\ postdom\ b$ und bezeichnen die Knoten, die b postdominieren als die *Postdominatoren* von b.

Die Dominator- und Postdominator-Relationen sind transitiv, irreflexiv und somit asymmetrisch. Alle Knoten außer r werden von r dominiert und alle Knoten außer e werden von e postdominiert.

Satz 2.1: *Sei $CFG\ \langle N, E, r, e\rangle$ ein Kontrollflussgraph und a, b und c Knoten in CFG mit $a\ dom\ b$ und $b\ dom\ c$.*

Dann gibt es für jeden Weg $W = (v_1, \ldots, v_n)$ von a nach c ein $j \in \{2, \ldots, n-1\}$ mit $v_j = b$.

Beweis. Aufgrund der Asymmetrie von dom folgt aus der Voraussetzung, dass a nicht von b dominiert wird. Somit gibt es einen Weg $W' = (w_1, \ldots, w_m)$ von r nach a mit $w_j \neq b$ für alle $j \in \{2, \ldots, m-1\}$.

Sei $W = (w_{m+1}, \ldots, w_{m+n})$ ein Weg von a nach c, so dass $v_j \neq b$ für alle $j \in \{w_{m+2}, \ldots, w_{m+n-1}\}$.

Nach Definition 2.9 folgt $v_{m+1} = a \neq b$ aus $a\ dom\ b$. Somit ist $(w_1, \ldots, v_{m+n})$ ein Weg von r nach c mit $w_j \neq b$ für alle $j \in \{2, \ldots, m+n-1\}$.

Also kann c nicht von b dominiert werden, was der Voraussetzung widerspricht. □

Ein analoger Satz gilt für die Postdominator-Relation.

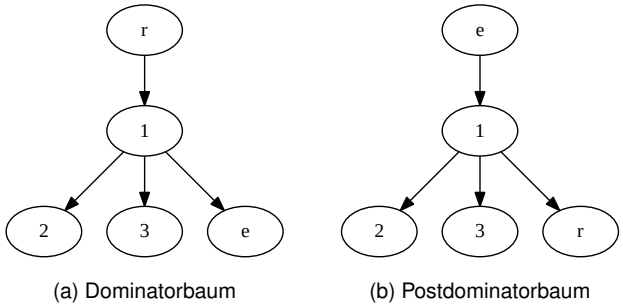
Satz 2.2: *Sei $CFG\ \langle N, E, r, e\rangle$ ein Kontrollflussgraph und a, b und c Knoten in CFG mit $c\ postdom\ b$ und $b\ postdom\ a$.*

Dann gibt es für jeden Weg $W = (v_1, \ldots, v_n)$ von a nach c ein $j \in \{2, \ldots, n-1\}$ mit $v_j = b$.

Die Relationen lassen sich zudem als Baum mit Wurzel r bzw. e darstellen[ASU97]. In Abbildung 2.4 sind Dominator- und Postdominatorbaum für den Kontrollflussgraphen aus Abbildung 2.3 dargestellt.

Definition 2.10: Seien a und b Knoten des Kontrollflussgraphen $CFG\ \langle N, E, r, e\rangle$.

- a ist genau dann der *direkte Dominator* von b, wenn b von a dominiert wird und $\forall c : c\ dom\ b \Rightarrow c\ dom\ a$. Wir schreiben $a = idom\,(b)$.
- a ist genau dann der *direkte Postdominator* von b, wenn b von a postdominiert wird und $\forall c : c\ postdom\ b \Rightarrow c\ postdom\ a$. Wir schreiben $a = ipostdom\,(b)$.



(a) Dominatorbaum (b) Postdominatorbaum

Abbildung 2.4: Dominator- und Postdominatorbaum für Abbildung 2.3

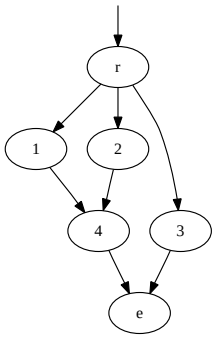


Abbildung 2.5: Kontrollflussgraph mit mehrdeutigen Kontrollabhängigkeiten

Im Dominatorbaum ist $idom\,(a)$ der Vaterknoten von a. Entsprechend ist im Postdominatorbaum $ipostdom\,(a)$ der Vaterknoten von a.

2.3.2 Kontrollabhängigkeiten

Ähnlich den Dominatoren, die auf jedem Weg vom Startknoten zu einem Knoten n durchlaufen werden, gibt es Kanten, die durchlaufen werden müssen, um n zu erreichen. Betrachten wir etwa den Kontrollflussgraphen in Abbildung 2.5, so ist offensichtlich, dass zum Erreichen des Knotens 2 mindestens die Kante $r \rightarrow 2$ durchlaufen werden muss.

Allerdings sind diese Kanten nicht eindeutig bestimmt in dem Sinne, dass auf jedem Weg zum Knoten n eine bestimmte Kante durchlaufen wird. Dies ist gut am Knoten 4 zu erkennen, der entweder über die Kante $r \rightarrow 1$ oder die Kante $r \rightarrow 2$ zu erreichen ist, nicht jedoch über die Kante $r \rightarrow 3$.

Die Ursprungsknoten solcher Kanten werden als die Kontrollabhängigkeiten von n bezeichnet.

Definition 2.11: Sei $CFG\,\langle N, E, r, e\rangle$ ein Kontrollflussgraph und a und b Knoten in CFG. Dann ist b *kontrollabhängig* von a, wenn gilt:

- Es existiert ein Weg $(v_1, \ldots, v_n)$ von a nach b, so dass b jeden Knoten v_i mit $1 < i < n$ postdominiert, und
- a wird nicht von b postdominiert.

Mit anderen Worten: Es existiert ein Nachfolger von a, von dem aus b sicher erreicht wird, aber ebenso ein Weg von a nach e, der nicht über b läuft.

In Abbildung 2.5 etwa sind die Knoten 3 und 4 kontrollabhängig von Knoten r, aber Knoten 4 ist von keinem der Knoten 1 und 2 kontrollabhängig.

2.3.3 Schleifen

Programme enthalten regelmäßig Abschnitte, die mehrfach ausgeführt werden können. Diese Abschnitte bezeichnen wir als *Schleifen*.

Üblicherweise erzeugen Compiler für imperative und objektorientierte Sprachen nur sogenannte *natürliche Schleifen*[ASU97]. Eine natürliche Schleife ist erkennbar durch eine Rückwärtskante $w \to v$, bei der w von v dominiert wird.

Somit ist v in jedem Durchlauf enthalten sowie jeweils der erste Knoten des Schleifendurchlaufs, und wird als *Kopf* der Schleife bezeichnet. Entsprechend ist w *Fuß* der Schleife.

Als *Rumpf* der Schleife wird die Menge der Knoten bezeichnet, die v und genau jene Knoten enthält, von denen aus w erreicht werden kann, ohne v zu durchlaufen. Hiermit können wir Ein- und Ausgangskanten als jene Kanten definieren, bei denen genau ein Endknoten in und ein Endknoten außerhalb des Rumpfes liegt.

Nicht alle Schleifen sind jedoch natürliche Schleifen. Im Allgemeinen ist das Vorhandensein einer Schleife dadurch erkennbar, dass ein Knoten v existiert, der von sich selbst aus erreichbar ist.

Definierte man die Menge der Knoten im Rumpf einer Schleife um v als die Menge der Knoten, die auf einem Weg von v nach v liegen, so enthielte der Rumpf einer klassischen inneren Schleife auch die Knoten der sie umgebenden Schleife. Somit wären die eigentlichen Ein- und Ausgangskanten der inneren Schleife nicht erkennbar.

Wir verallgemeinern deshalb die Begriffe Schleifenfuß, Schleifenkopf, Schleifenrumpf sowie Ein- und Ausgangskante.

Definition 2.12: Sei $CFG\,\langle N, E, r, e\rangle$ ein Kontrollflussgraph.

Ein Knoten v ist genau dann *Kopf einer Schleife*, wenn es einen Knoten w gibt, so dass v auf einem schleifenfreien Weg von r nach w liegt und $w \to v$ gilt.

Wir bezeichnen w als einen *Fuß der Schleife um* v.

Sei v Kopf einer Schleife und F die Menge der Füße der Schleife um v. Ein Knoten u ist genau dann im *Rumpf der Schleife um* v enthalten, wenn es einen Weg $(v_1, \ldots, v_k)$ von v zu einem Fuß $w \in F$ gibt mit $v_i = u$ für ein $i \in \{1, \ldots, k\}$, so dass $v_j \notin F$ für alle $j \in \{2, \ldots, k-1\}$.

Eine Kante $(s, t) \in E$ ist genau dann *Eingangskante in die Schleife um* v, wenn t im Rumpf der Schleife um v enthalten ist, nicht aber s.

Eine Kante $(s, t) \in E$ ist genau dann *Ausgangskante aus der Schleife um* v, wenn s im Rumpf der Schleife um v enthalten ist, nicht aber t.

Die Menge der Schleifenköpfe in einem Programmflussgraphen erhalten wir durch eine Tiefensuche mit Backtracking, ausgehend vom Wurzelknoten. Dabei konstruieren wir sämtliche beim Wurzelknoten beginnenden schleifenfreien Wege des Programmflussgraphen. Haben wir einen schleifenfreien Weg $(v_1, \ldots, v_n)$ mit $n \geq 1$ und $v_1 = r$ konstruiert und gibt es für ein $i \in \{1, \ldots, n\}$ eine Kante $v_n \to v_i$, dann ist v_i Schleifenkopf.

2.3.4 Semantik von Kontrollflussgraphen

Der Kontrollflussgraph beschreibt eine Obermenge der möglichen Ausführungswege eines Programms. Die Semantik des Programms im Sinne einer Spezifikation (s. Abschnitt 2.2) ist durch ihn jedoch nicht definiert.

Wir beschreiben im Folgenden eine Erweiterung des Kontrollflussgraphen, den wir als *Programmflussgraphen* bezeichnen, sowie die operationelle Semantik von Programmflussgraphen in Form einer abstrakten Maschine. Durch Anwendung der abstrakten Maschine auf einen Programmflussgraphen kann die Spezifikation des beschriebenen Programms bestimmt werden.

Unser Ausführungsmodell basiert auf Ausführungszuständen, die insbesondere den aktuellen Speicherzustand beinhalten. Dieser Speicherzustand enthält vor Beginn der Ausführung die Programmeingabe und nach dem Abschluss der Ausführung die Programmausgabe.

Der Programmflussgraph wird beginnend mit dem Startknoten durchlaufen und der Speicherzustand wird durch Ausführung der Knoten des Graphen transformiert.

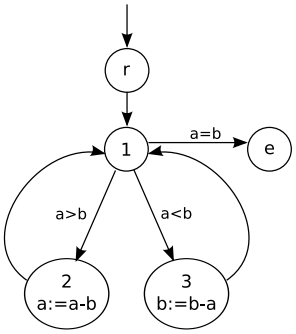


Abbildung 2.6: Ein Programmflussgraph

Die Semantik dieser Transformation beschreiben wir, indem wir jedem Knoten je eine Spezifikation zuordnen. Der Ausgabezustand jedes Knoten wird zum Eingabezustand des nachfolgenden Knotens in der Ausführungsfolge.

Unser Automat trifft Verzweigungsentscheidungen auf Grundlage des Ausgabezustands des Startknotens einer Kante. Die Entscheidungsvorschrift definieren wir, indem wir jeder Kante je eine Bedingung zuordnen, die genau für jene Zustände erfüllt ist, in denen die jeweilige Kante passiert werden darf.

Definition 2.13: Ein Kontrollflussgraph $CFG\,\langle N, E, r, e\rangle$ sowie zwei Funktionen $\mathcal{B}$ und $\mathcal{C}$ bilden genau dann einen *Programmflussgraphen* $PFG\,\langle N, E, r, e, \mathcal{B}, \mathcal{C}\rangle$, wenn $\mathcal{B}\,(n)$ für jeden Knoten n in N eine Spezifikation und $\mathcal{C}\,(a, b)$ für jede Kante $(a, b) \in E$ eine Bedingung ist.

Für einen Knoten n repräsentiert dabei $\mathcal{B}\,(n)$ den Rumpf („Body“) des Knotens und für eine Kante (a, b) bezeichnet $\mathcal{C}\,(v, w)$ die Verzweigungsbedingung („Condition“) für diese Kante.

Ein Beispiel für einen Programmflussgraphen ist in Abbildung 2.6 dargestellt. Er erweitert den Kontrollflussgraphen aus Abbildung 2.3.

Wir beschriften generell die Knoten mit Anweisungen, die ihre Spezifikation beschreiben, und die Kanten mit logischen Ausdrücken, die ihre Bedingungen definieren. Ist die Rumpfspezifikation eines Knotens die universelle Spezifikation 1 bzw. die Bedingung einer Kante die Relation $\top$, so lassen wir die Beschriftung weg. In Abbildung 2.7 sind die Funktionen $\mathcal{B}$ und $\mathcal{C}$ gemäß Definition 2.13 dargestellt.

Ein Programmaufruf in einem Knoten kann etwa als $\mathcal{S}_1; \mathcal{S}_{P'}; \mathcal{S}_2$ dargestellt werden, wobei P' das aufzurufende Programm ist. Die Spezifikationen $\mathcal{S}_1$ und $\mathcal{S}_2$ stellen die vor bzw. nach dem Programmaufruf auszuführenden Teile des Knotens dar.

$$
\begin{aligned}
\mathcal{B}(r) &= 1 \\
\mathcal{B}(1) &= 1 \\
\mathcal{B}(e) &= 1 \\
\mathcal{B}(2) &= \{((a,b),(a-b,b)) \,|\, a,b \in \mathbb{N}\} \\
\mathcal{B}(3) &= \{((a,b),(a,b-a)) \,|\, a,b \in \mathbb{N}\} \\
\mathcal{C}(r,1) &= \top \\
\mathcal{C}(1,e) &= \{((a,b),(a,b)) \,|\, a,b \in \mathbb{N} \wedge a = b\} \\
\mathcal{C}(1,2) &= \{((a,b),(a,b)) \,|\, a,b \in \mathbb{N} \wedge a > b\} \\
\mathcal{C}(1,3) &= \{((a,b),(a,b)) \,|\, a,b \in \mathbb{N} \wedge a \leq b\} \\
\mathcal{C}(2,1) &= \top \\
\mathcal{C}(3,1) &= \top
\end{aligned}
$$

Abbildung 2.7: Spezifikationen und Bedingungen zu Abbildung 2.6

Wir beschreiben nun einen Zustandsautomaten, auf dem Programmflussgraphen ausgeführt werden können. Als Zustandsübergänge definieren wir dabei die Ausführung eines Knotens und die Überquerung einer Kante. Wir unterscheiden also zwischen dem Zustand vor bzw. nach Ausführung eines Knotens und bezeichnen ersteres als α-Zustand und letzteres als β-Zustand.

Definition 2.14: Ein *Ausführungszustand* eines Programmflussgraphen PFG ist ein Tupel $S = \langle n, s, x \rangle$, wobei n ein Knoten des Programmflussgraphen und $s \in \{\alpha, \beta\}$ ist.

Wir bezeichnen x als den *Speicherzustand von* S.

Definition 2.15: Sei $PFG\,\langle N, E, r, e, \mathcal{B}, \mathcal{C} \rangle$ ein Programmflussgraph mit Kantenrelation $\rightarrow$.

Dann ist $\mapsto_{PFG}$ die *Übergangsrelation* für PFG. Sie ist die kleinste Relation, für die gilt:

$$
\begin{aligned}
\forall n, x, y : x\ \mathcal{B}(n)\ y &\quad\Leftrightarrow\quad \langle n, \alpha, x \rangle \mapsto_{PFG} \langle n, \beta, y \rangle \\
\forall n_1, n_2, x : n_1 \rightarrow n_2 \wedge x\ \mathcal{C}(n_1, n_2)\ x &\quad\Leftrightarrow\quad \langle n_1, \beta, x \rangle \mapsto_{PFG} \langle n_2, \alpha, x \rangle
\end{aligned}
$$

Wenn der Bezug eindeutig ist, lassen wir den Index PFG weg und schreiben kurz $\mapsto$.

Definition 2.16: Sei $PFG\,\langle N, E, r, e, \mathcal{B}, \mathcal{C}\rangle$ ein Programmflussgraph und sei $S = \langle n, s, x\rangle$ ein Zustand von PFG.

Wir bezeichnen S genau dann als *Initial- oder Startzustand*, wenn $n = r$.

Wir bezeichnen S genau dann als *(erfolgreichen) Endzustand*, wenn $n = e$.

Wir bezeichnen S genau dann als *Fehlerzustand*, wenn $n \neq e$ und es keinen Zustand S' gibt mit $S \mapsto S'$.

Definition 2.17: Sei $PFG\,\langle N, E, r, e, \mathcal{B}, \mathcal{C}\rangle$ ein Programmflussgraph mit der Übergangsrelation $\mapsto$.

Eine Folge von Zuständen $(S_1, \ldots, S_n)$ mit $n \geq 1$ ist genau dann eine *(endliche) Ausführungsfolge von S_1 nach S_n* in PFG, wenn gilt

$$\forall i \in \{1, \ldots, n-1\} : S_i \mapsto S_{i+1}$$

Dabei bezeichnen wir n als die *Länge* der Ausführungsfolge.

Wir nennen $(S_1, \ldots, S_n)$ genau dann eine *vollständige Ausführungsfolge* in PFG, wenn S_1 ein Initialzustand und S_n ein erfolgreicher Endzustand ist.

Aus dieser operationellen Semantik lässt sich nun die Spezifikation für ein solchermaßen beschriebenes Programm ableiten. Hierfür definieren wir zunächst die Spezifikation für einen Programmablauf beginnend bei einem beliebigen Knoten a und endend bei einem beliebigen Knoten b. Die Spezifikation des Programms entspricht dann dem Fall $a = r$ und $b = e$.

Definition 2.18: Sei $PFG\,\langle N, E, r, e, \mathcal{B}, \mathcal{C}\rangle$ ein Programmflussgraph und a und b Knoten in PFG.

Dann gilt für ein Paar (x, y) genau dann $x\ \mathcal{S}_{PFG[a,b]}\ y$, wenn es eine Ausführungsfolge $(S_1, \ldots, S_n)$ gibt mit $S_1 = \langle a, \alpha, x\rangle$, $S_n = \langle b, \beta, y\rangle$.

Die *Spezifikation von PFG* $\mathcal{S}_{PFG[r,e]}$ wird kurz als $\mathcal{S}_{PFG}$ bezeichnet.

In einem allgemeinen Programmflussgraph existiert zu einer Eingabe nicht notwendigerweise nur eine einzige mögliche Ausführungsfolge. Es ist möglich, dass die Bedingungen mehrerer abgehender Kanten für einen bestimmten Programmzustand erfüllt sein können. In diesem Fall ist der Programmflussgraph nichtdeterministisch.

Definition 2.19: Sei $PFG\,\langle N, E, r, e, \mathcal{B}, \mathcal{C}\rangle$ ein Programmflussgraph.

Wir bezeichnen PFG als *deterministisch im Knoten* a, wenn $\mathcal{B}(a)$ eindeutig ist und es für jedes x maximal einen Nachfolger b von a gibt für den $x\ \mathcal{C}(a,b)\ x$ gilt. Anderenfalls ist PFG *nichtdeterministisch im Knoten* a.

Ist ein Programmflussgraph in jedem seiner Knoten deterministisch, so wird er kurz als *deterministisch* bezeichnet. Anderenfalls ist PFG *nichtdeterministisch*.

Satz 2.3: *Sei* $PFG\,\langle N, E, r, e, \mathcal{B}, \mathcal{C}\rangle$ *ein deterministischer Programmflussgraph.*

Seien S_1, S_2 *und* S_2' *Zustände in* PFG *mit* $S_1 \mapsto S_2$ *und* $S_1 \mapsto S_2'$.

Dann gilt $S_2 = S_2'$.

Beweis. Sei $S_1 = \langle v, \alpha, x\rangle$ und $S_2 = \langle v, \beta, y\rangle$. Aufgrund von $S_1 \mapsto S_2'$ muss S_2' die Form $\langle v, \beta, y'\rangle$ haben. Aus der Eindeutigkeit von $\mathcal{B}(v)$ folgt $y = y'$ und somit $S_2 = S_2'$.

Sei alternativ $S_1 = \langle v, \beta, x\rangle$ und $S_2 = \langle w, \alpha, x\rangle$. Aufgrund von $S_1 \mapsto S_2'$ muss S_2' die Form $\langle w', \alpha, x\rangle$ haben. Da PFG deterministisch insbesondere im Knoten v ist, folgt $w' = w$ und somit $S_2 = S_2'$. □

Dies bedeutet insbesondere, dass in deterministischen Programmflussgraphen alle Ausführungsfolgen eindeutig durch ihre Länge und ihren Startzustand bestimmt sind.

Definition 2.20: Sei $F = (S_1, \ldots, S_k)$ eine Ausführungsfolge mit $S_i = \langle v_i, s_i, x_i\rangle$ für alle $i \in \{1, \ldots, k\}$.

Wir bezeichnen F als *schleifenfrei* genau dann, wenn es keine l und m, $1 \leq l < m \leq k$ gibt mit $v_l = v_m$, $s_l = \beta$ und $s_m = \alpha$.

Lemma 2.2: *Existiert eine Ausführungsfolge* $(S_1, \ldots, S_n)$ *mit* $S_1 = \langle a, \alpha, x\rangle$ *und* $S_n = \langle b, \beta, y\rangle$*, so gilt* $a \to^* b$.

Existiert eine Ausführungsfolge $(S_1, \ldots, S_n)$ *mit* $S_1 = \langle a, \beta, x\rangle$ *und* $S_n = \langle b, \alpha, y\rangle$*, so gilt* $a \to^+ b$.

Beweis. Wir beweisen die erste Behauptung durch Induktion über n. Der Beweis für die zweite Behauptung verläuft analog.

Nach Definition 2.15 ist n gerade. Sei zunächst $n = 2$. Dann gilt $a = b$ und somit auch $a \to^* b$.

Sei nun $n > 2$. Dann gilt $S_{n-2} = \langle c, \beta, y'\rangle$ und nach Induktionshypothese somit $a \to^* c$. Aufgrund von $S_{n-2} \mapsto S_{n-1}$ und $S_{n-1} \mapsto S_n$ gilt auch $c \to b$. Es folgt $a \to^* b$. □

2.3.5 Datenfluss

Bei der Datenflussanalyse wird festgestellt, wie die Variablenzuweisungen mit den Nutzungen (=Lesezugriffen) auf diese Variablen zusammenhängen. Wir bezeichnen wie in Abschnitt 2.1.3 die Zuweisungen als *Definitionen* und die Nutzungen der Variablen als ihre *c-uses* bzw. *p-uses*[RW82].

Wir unterscheiden hier noch zusätzlich in *lokale und globale c-uses*. Eine c-use u einer Variablen v ist global, wenn in dem Knoten, der u enthält, keine Definition von v enthalten ist, die vor u ausgeführt wird. Ansonsten ist die c-use lokal.

Lokale c-uses ignorieren wir für den Softwaretest, da sie unabhängig vom einzelnen Testfall nur das Ergebnis der im selben Knoten vorausgehenden Definition nutzen kann.

Zu den Definitionen zählen wir auch die Zuweisungen von Parameterwerten an die Parametervariablen, die unmittelbar vor dem Start des Programms stattfinden. Sonst wären entgegen der Intention diese vor ihrer ersten Nutzung im Programm undefiniert.

Definition 2.21: Sei P ein Programm und $PFG\langle N, E, r, e, \mathcal{B}, \mathcal{C}\rangle$ der Programmflussgraph von P.

Seien weiterhin

- V die Menge der Programmvariablen in P,
- die Funktion $D_P : V \to \mathcal{P}(N)$ die Zuordnung von Variablen zu Definitionen mit $D(v)$ der Menge von Knoten, die eine Definition von $v \in V$ enthalten,
- die Funktion $U_{P,c} : V \to \mathcal{P}(N)$ die Zuordnung von Variablen zu ihren c-uses mit $U_c(v)$ der Menge von Knoten, die eine globale c-use von $v \in V$ enthalten, und
- die Funktion $U_{P,p} : V \to \mathcal{P}(E)$ die Zuordnung von Variablen zu ihren p-uses mit $U_p(v)$ der Menge von Kanten, die eine p-use von $v \in V$ enthalten.

Wenn keine Verwechslung möglich ist, lassen wir den Index P auch weg und schreiben kurz D, U_c und U_p.

Definition 2.22: Sei P ein Programm und $PFG\langle N, E, r, e, \mathcal{B}, \mathcal{C}\rangle$ der Programmflussgraph von P. Sei v eine Programmvariable in P.

Wir bezeichnen einen Weg $W = (v_1, \ldots, v_n)$ als *definitionsfrei* bzgl. v, wenn es kein $j \in \{2, \ldots, n-1\}$ gibt mit $v_j \in D_P(v)$.

```
1 for i:=1 to n do
2         minidx:=i
3         for j:=i+1 to n do
4                 if a(j)<a(minidx) then minidx:=j
5         end
6         tmp:=a(minidx)
7         a(minidx):=a(i)
8         a(i):=tmp
9 end
```

Listing 2.1: Selection Sort

Wir bezeichnen eine Ausführungsfolge $F = (S_1, \ldots, S_n)$ als *definitionsfrei* bzgl. v, wenn es kein $j \in \{2, \ldots, n-2\}$ gibt mit $S_j = \langle n, \alpha, x \rangle$ und $S_{j+1} = \langle n, \beta, y \rangle$ für ein $n \in D_P(v)$.

Eine Ausführungsfolge ist also definitionsfrei bzgl. v, wenn außer ganz am Anfang oder ganz am Ende in ihr kein Knoten ausgeführt wird, der eine Definition der Variablen v enthält. Ein Knoten wird nur dann tatsächlich in der Folge ausgeführt, wenn die Folge einen Übergang vom zugehörigen α- zum β-Zustand enthält.

2.3.6 Ausführbarkeit von Wegen

Anhand der in Abschnitt 2.3.4 definierten Semantik können nun auch die Spezifikationen einzelner Wege abgeleitet und somit die Bedingungen bestimmt werden, die Ein- und Ausgabe erfüllen müssen, damit der jeweils betrachtete Weg zur Ausführung kommen kann.

Definition 2.23: Ein Weg $W = (v_1, \ldots, v_n)$ in einem Programmflussgraphen PFG ist genau dann *ausführbar*, wenn es eine Ausführungsfolge $(S_1, \ldots, S_{2n})$ in PFG gibt mit $S_{2i-1} = \langle v_i, \alpha, x_i \rangle$ und $S_{2i} = \langle v_i, \beta, x_{i+1} \rangle$ für alle $i \in \{1, \ldots, n\}$.

Es gibt Wege, die nicht ausführbar sind, wie wir anhand der Implementierung des Selection-Sort-Algorithmus in Listing 2.1 erkennen. Hier sind die Iterationen von innerer und äußerer Schleife miteinander verknüpft. Es ist nicht möglich, eine Eingabe zu finden, mit der die äußere Schleife mehr als einmal durchlaufen wird, die innere aber keinmal.

Mit Definition 2.23 wird jedem Weg eine möglicherweise leere Menge von Ausführungsfolgen zugeordnet. Diese Ausführungsfolgen unterscheiden sich nur in der Belegung der Speicherzustände x_i voneinander. Darunter sind auch der Eingabezustand x_1 und der Ausgabezustände x_{n+1}, wobei n die Länge des Weges ist.

Der Zusammenhang zwischen diesen x_i wird durch eine Konjunktion von Aussagen der Form $x_i \ \mathcal{B}\,(v_i)\ \ x_{i+1}$ und $x_i \ \mathcal{C}\,(v_{i-1}, v_i)\ \ x_i$ definiert. Diese Konjunktion bezeichnen wir als *path constraint*[FK96] oder *Wegconstraint*. Sie nimmt meist die Form eines Systems von Gleichungen und Ungleichungen in den Variablen des Programms an.

Ist das Wegconstraint für eine Lösung $(x_1, \ldots, x_{n+1})$ erfüllt, so existiert zur Eingabe x_1 eine Ausführungsfolge, die dem Weg $(v_1, \ldots, v_n)$ im Programm folgt und die Ausgabe x_{n+1} liefert.

2.4 CHR

Constraint Handling Rules (CHR)[Frü98] ist eine regelbasierte deklarative Sprache zur Definition von Constraintlösern.

Eingabe für ein CHR-Programm ist eine Konjunktion von Prädikaten – sog. *Constraints* –, die auf Grundlage der vom Programm durch Ableitungsregeln definierten *Constrainttheorie* transformiert und dabei vorzugsweise vereinfacht werden. Diese Konjunktion wird als *Ziel* bezeichnet.

Ein CHR-Programm besteht aus Regeln, die bedingte Implikationen oder Äquivalenzen für die zu bearbeitenden Constraints definieren.

Grundlage eines jeden CHR-Programms ist eine vordefinierte Constrainttheorie, die mindestens die Gleichheit von Termen sowie die Constraints `true` und `false` definieren muss, aber oft durch weitere vordefinierte Constraints erweitert wird.

Es wird zwischen *vordefinierten* und *nutzerdefinierten* Constraints unterschieden. Die Theorie der nutzerdefinierten Constraints wird durch das Programm definiert, die der vordefinierten Constraints wird als gegeben angenommen.

In Listing 2.2 ist ein Beispiel für ein CHR-Programm dargestellt. Es definiert eine Constrainttheorie einer Halbordnung leq.

Der Text vor dem `@` ist optional und wird als *Label* oder Name der Regel bezeichnet. Jede der Regeln besteht aus drei Teilen:

- Kopf `H`: eine Konjunktion von nutzerdefinierten Constraints

```
1 idempotence @ leq(A,B) \ leq(A,B) <=> true.
2 reflexivity @ leq(A,A) <=> true.
3 antisymmetry @ leq(A,B), leq(B,A) <=> A=B.
4 transitivity @ leq(A,B), leq(B,C) ==> leq(A,C).
```

Listing 2.2: CHR-Programm für $\leq$

- Guard `G`: eine Konjunktion von vordefinierten Constraints (optional)
- Rumpf `B`: eine Konjunktion von nutzer- und vordefinierten Constraints

Der Guard kann weggelassen werden, wenn er `true` ist.

Eine *Simplifikationsregel* hat die Form `H <=> G | B` bzw. `H <=> B` und eine *Propagationsregel* hat die Form `H ==> G | B` bzw. `H ==> B`.

Zusätzlich können noch sog. *Simpagationsregeln* definiert werden, die eine Kombination aus Simplifikations- und Propagationsregel darstellen und hauptsächlich für die automatische Ausführung von CHR-Programmen relevant sind. Sie haben die Form `H1 \ H2 <=> G | B` bzw. `H1 \ H2 <=> B` und sind logisch äquivalent zu `H1, H2 <=> G | H1, B` bzw. `H1, H2 <=> H1, B`.

Im klassischen Ansatz von CHR ist den Regeln eine logische Lesweise zugeordnet. Eine Simplifikationsregel `H <=> G | B` drückt eine Äquivalenz der Form $\forall \vec{x} : G \Rightarrow (H \Leftrightarrow \exists \vec{y} : B)$ aus, wobei $\vec{x}$ die freien Variablen aus G und H beinhaltet und $\vec{y}$ die freien Variablen aus B darstellt, die nicht in $\vec{x}$ enthalten sind.

Eine Propagationsregel `H ==> G | B` stellt eine Implikation der Form $\forall \vec{x} : G \Rightarrow (H \Rightarrow \exists \vec{y} : B)$ dar.

Wir sehen, dass das Beispielprogramm die folgenden Aussagen deklariert:

$$\forall a, b : a \leq b \wedge a \leq b \quad \Leftrightarrow \quad a \leq b \tag{2.1}$$

$$\forall a : a \leq a \quad \Leftrightarrow \quad \top \tag{2.2}$$

$$\forall a, b : a \leq b \wedge b \leq a \quad \Leftrightarrow \quad a = b \tag{2.3}$$

$$\forall a, b, c : a \leq b \wedge b \leq c \quad \Rightarrow \quad a \leq c \tag{2.4}$$

Aussage 2.1 ist eine Tautologie, da die Konjunktion $\wedge$ idempotent ist. Die operationelle Semantik definiert jedoch den Programmzustand als Multimenge von Constraints, so dass die Idempotenz der Konjunktion explizit modelliert werden muss.

Bei den anderen Aussagen erkennen wir, dass sie die Reflexivität, die Antisymmetrie und die Transitivität der Halbordnung $\leq$ darstellen. Insbesondere können wir Aussage 2.2 zu $\forall a : a \leq a$ vereinfachen.

Operationell werden die Regeln in einem Zustandsautomaten umgesetzt, der einem Termersetzungssystem ähnelt. Der Zustand dieses Automaten besteht vereinfacht betrachtet aus zwei Teilen:

- Einer Konjunktion von benutzerdefinierten Constraints, und
- einer Konjunktion von vordefinierten Constraints.

Dieser Automat kann in jedem Schritt eine von vier Operationen ausführen:

- Simplify: Anwendung einer Simplifikationsregel
- Propagate: Anwendung einer Propagationsregel
- Simpagate: Anwendung einer Simpagationsregel
- Solve: Lösung oder Vereinfachung der vordefinierten Constraints

Ein erfolgreicher Endzustand ist dann erreicht, wenn keine Operation mehr angewendet werden kann und die vordefinierten Constraints nicht äquivalent zu `false` sind. Ein erfolgloser Endzustand ist erreicht, wenn die vordefinierten Constraints äquivalent zu `false` sind.

Die Reihenfolge der Operationen, der Regeln und der Constraints, auf die sie angewendet werden, ist nicht definiert. Ist das entsprechende Termersetzungssystem jedoch konfluent, so lässt sich folgern, dass die vom CHR-Programm definierte Constrainttheorie vollständig und korrekt ist[Frü98]. Anschaulich ausgedrückt müssen die in einem konfluenten CHR-Programm auf unterschiedlichen Wegen abgeleiteten Ergebnisse zueinander äquivalent sein und können sich somit insbesondere nicht widersprechen. Aber auch nicht-konfluente CHR-Programme können vollständig und korrekt sein.

Ein CHR-Programm *terminiert* für einen Startzustand S, wenn es keine von S ausgehenden unendlichen Ableitungspfade gibt[Frü00].

2.4.1 Suche

CHR ist eine committed-choice-Sprache, d.h. während der Ausführung getroffene Entscheidungen sind unwiderruflich. Somit sind Backtracking oder Suche wie von Prolog bekannt dem Sprachkonzept von CHR fremd.

Die Spracherweiterung $CHR^{\vee}$ ermöglicht die Nutzung von Disjunktionen im Rumpf der Regeln und bietet insofern begrenzte Möglichkeiten der Suche[AS98]. Hier kann der Rumpf `B` einer CHR-Regel die Form `B1 ; B2 ; ...` annehmen, was als Disjunktion der Teilrümpfe `B1`, `B2`, und so weiter zu lesen ist. Durch weitergehende Modifikation der operationellen Semantik von $CHR^{\vee}$ können sogar unter-

schiedliche Suchstrategien wie depth-first-search und breadth-first-search implementiert werden[KSD06].

Allerdings ist die so mögliche Disjunktion von Constraints im Rumpf der Regeln keineswegs gleichwertig zur Suche in Prolog, die insbesondere die Suche über alternativ anwendbare Regeln ermöglicht. Dies ist in $CHR^{\vee}$ nur möglich, wenn die Köpfe der Regeln übereinstimmen.

Die verfeinerte operationelle Semantik von CHR[DSBH04] legt eine Priorität der Regeln für die Anwendung fest. Erweitert man diese verfeinerte Semantik auf $CHR^{\vee}$ und definiert ein in die Ausführungssemantik eingreifendes Meta-Constraint `remove_constraint/1`, so können die möglichen Regelanwendungen explizit gesammelt und ihre Rümpfe über eine Disjunktion verknüpft werden, so dass auch eine Suche über alternativ anwendbare Regeln möglich wird. Auf derselben Grundlage kann auch eine zufällige Wahl der anzuwendenden Regel implementiert werden[FPW02].

2.4.2 Probabilistic CHR

Probabilistic CHR (PCHR) ist eine Spracherweiterung für CHR, die die zufällige Auswahl der anzuwendenden Regelinstanz ermöglicht[FPW02].

Jeder Regel wird dazu ein Gewicht zugeordnet. In jedem Zustand werden alle anwendbaren Regelinstanzen gesammelt und deren Gewichte zusammengezählt. Die Wahrscheinlichkeit einer Regelinstanz, ausgeführt zu werden, entspricht genau dem Verhältnis zwischen dem Gewicht dieser Regel und der Gewichtsumme über alle anwendbaren Regelinstanzen.

Als Regelinstanz wird dabei eine Regel zusammen mit einer Belegung ihrer freien Variablen angesehen. Dies ist speziell dann relevant, wenn eine Regel mit mehreren unterschiedlichen Variablenbelegungen auf den aktuellen Zustand anwendbar ist.

So ist die Anwendungswahrscheinlichkeit einer bestimmten Regel nicht unbedingt über den gesamten Programmablauf hinweg konstant. Selbst dann, wenn einer Regel ein hohes Gewicht zugeordnet ist, kann sie mit geringerer Wahrscheinlichkeit gewählt werden als eine Regel mit niedrigerem Gewicht, sofern ausreichend viele Regelinstanzen der letzteren auf den Zustand anwendbar sind.

PCHR-Programme sind probabilistische Programme und erben deren besondere Terminierungseigenschaften. Ein PCHR-Programm *terminiert probabilistisch*, wenn die Wahrscheinlichkeit für eine unendliche Ableitung 0 ist. Insbesondere

```
1 walk(X,Y) <=>1: X\=Y | walk(X+1,Y).
2 walk(X,Y) <=>1: X\=Y | walk(X,Y+1).
3 walk(X,Y) <=>1: X=Y | true.
```

Listing 2.3: Gambler's Ruin

kann ein PCHR-Programm probabilistisch terminieren, selbst wenn die Terminierung für sein nicht-probabilistisches Pendant nicht garantiert ist.

Ein Beispiel für ein PCHR-Programm ist in Listing 2.3[FPW02] dargestellt. Die Regelgewichte werden hinter dem Regeloperator `<=>` oder `==>` notiert.

Sei `X` die Anzahl der gewonnenen Spiele und `Y` die Anzahl der verlorenen Spiele eines Glücksspielers. Der Spieler besitzt somit `X-Y` Münzen. Bei jedem Spiel kann der Spieler mit gleicher Wahrscheinlichkeit gewinnen oder verlieren und hat dann entsprechend eine Münze mehr oder eine weniger. Hat er keine Münzen mehr, gilt also `X=Y`, so kann er nicht mehr weiterspielen.

Das nicht-probabilistische Pendant dieses Programms könnte auf der Eingabe `walk(1,0)` unendlich lange laufen, wenn immer nur die erste oder die ersten beiden Regeln abwechselnd angewendet werden. Das PCHR-Programm terminiert jedoch probabilistisch auf derselben Eingabe. Der Spieler wird somit in jedem Fall nach endlicher Zeit alles verlieren, was er eingesetzt hat.

2.5 Static Single Assignment Form

Als Static Single Assignment Form[CFR91] (SSA-Form) wird eine spezielle Form des Programmcodes bezeichnet, in der zu jeder Variablen genau eine Zuweisung existiert. Ziel ist insbesondere eine vereinfachte Darstellung des Datenflusses in einem Programm.

Die wenigsten Programme liegen bereits in SSA-Form vor und müssen somit transformiert werden[CFR91]. Um das SSA-Kriterium zu erfüllen, werden sogenannte ϕ-Funktionen verwendet und die Variablen umbenannt.

Die ϕ-Funktionen dienen dazu, an den Stellen, die von mehreren verschiedenen Definitionen einer Variablen erreicht werden, eine Auswahl zu treffen.

Betrachten wir als Beispiel den Programmflussgraphen aus Abbildung 2.6. Das darin enthaltene Programm liegt nicht in SSA-Form vor, auch wenn jeweils nur eine Zuweisung an `a` und `b` sichtbar ist. Eine weitere findet aber vor dem Start

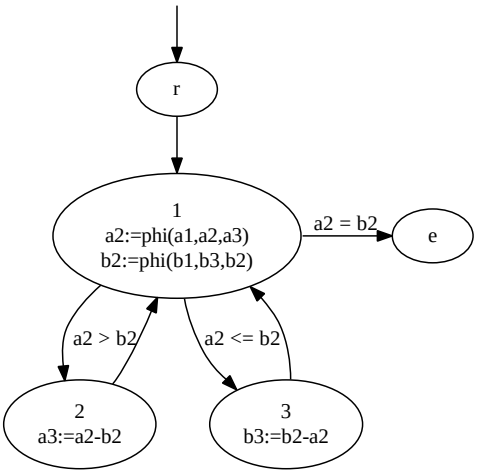


Abbildung 2.8: SSA-Form

des Programms statt, nämlich dann, wenn den Parametern $\mathtt{a}$ und $\mathtt{b}$ ihr Startwert zugewiesen wird.

In Abbildung 2.8 ist das Programm in SSA-Form dargestellt. Die Argumente der ϕ-Funktionen sind die Variablen, die an dieser Stelle zusammenkommen und ursprünglich dieselbe Variable darstellten, jeweils entsprechend der eingehenden Kante. Wird der Knoten r während der Ausführung über die Kante $2 \rightarrow 1$ betreten, so gilt $\phi(a_1, a_2, a_3) = a_3$ und $\phi(b_1, b_3, b_2) = b_2$, da entlang dieser Kante die Definitionen a_3 und b_2 sichtbar sind.

Da nun jeder Nutzung einer Variablen genau eine Zuweisung zugeordnet ist, vereinfacht sich insbesondere die Darstellung der Datenflussstrukturen. Aussagen über die Äquivalenz von Berechnungen im Programmcode sind ebenso leichter durch einen reinen Vergleich der Terme zu treffen.

3 Motivation

In der industriellen Praxis wird der Softwaretest bisher vorwiegend manuell ausgeführt. Durch den damit verbundenen Aufwand beschränkt sich die Industrie selbst bei hochkritischer Software deshalb auf Testvorgaben, die die Aussagekraft des Testergebnisses nur unzureichend sicherstellen.

Gesucht sind also Verfahren, mit denen die einzelnen Stufen des Softwaretests von Modul- bis Abnahmetest automatisiert werden können (s. Abschnitt 2.1.1). Mit solchen Verfahren wäre es möglich, die bisher durch den Aufwand bedingten Beschränkungen zu überwinden.

Für jede dieser Stufen müssen jeweils Testfallauswahl, Testausführung und Testauswertung unterstützt werden. Das Problem der Testfallauswahl wird dabei meist in die Auswahl der Testeingaben und die Zuordnung der erwarteten Ausgaben aufgeteilt.

Eines der Kernziele unserer Arbeit ist es, dem Anwender die einfache und intuitive Definition anwendungs- oder domänenspezifischer Testvorgaben zu ermöglichen. Statt einzelne Testfälle manuell abzuleiten, sollen vielmehr die wesentlichen Eigenschaften des Testfalls anhand von Eingabe, Kontrollfluss, Ausgabe oder gar Zwischenzuständen beschrieben werden. Genau wie bei den bereits bekannten Kriterien können solche Vorgaben auch aufgrund allgemeiner Regeln definiert werden, die dann jeweils automatisch auf den jeweils vorliegenden Prüfling angewendet werden.

In dieser Arbeit stellen wir ein Verfahren für die automatische Erzeugung von Testeingaben für den Modultest vor. Das Verfahren wählt die Eingaben auf Grundlage einer formalen Vorgabe sowie dem Programmflussgraphen des Prüflings.

Dieses Kapitel ist wie folgt aufgebaut: Wir konkretisieren zunächst unsere Kritik an der derzeitigen Praxis zur Nutzung von Testkriterien und zeigen auf, wie diese den Qualitätsanforderungen an den Softwaretest zuwiderläuft.

Dann geben wir einen kurzen Überblick über Ansätze und Methoden für die Testautomation im Allgemeinen und betrachten den Stand der Technik der automatischen Testdatenerzeugung für den Modultest im Speziellen.

Schließlich diskutieren wir einige wesentliche Beschränkungen der aktuell bekannten Verfahren und motivieren daraus die Anforderungen an das in dieser Arbeit beschriebene neue Verfahren.

3.1 Grenzen von Testkriterien

Einige der Standardkriterien aus Abschnitt 2.1.3 sind äußerst schwach[RW82]. Die Kriterien all-nodes, all-edges oder MC/DC etwa zwingen bei der Testfallauswahl nicht zur Unterscheidung zwischen einer Schleife und einer Konditionalanweisung.

Zudem wird die Unterscheidungskraft bei strukturbasierten Abdeckungskriterien durch die Struktur des Programms selbst limitiert. Wird etwa ein Sonderfall im Programm nicht behandelt, obwohl ein korrektes Programm ihn behandeln müsste, so können strukturbasierte Abdeckungskriterien auch keinen Testfall erzwingen, der den Sonderfall prüft.

Die vorgestellten Kriterien können somit lediglich als notwendige, nicht jedoch als hinreichende Bedingungen für geeignete Testfallmengen dienen, da sie weder die Erkennung der Fehler noch eine hinreichend geringe Irrtumswahrscheinlichkeit sicherstellen.

Für Luftfahrtsoftware, deren Fehlfunktion katastrophale Folgen haben kann, wird dennoch nur MC/DC als zu erfüllendes Kriterium festgelegt[Rad92]. Der vorherrschende manuelle Softwaretest in der industriellen Praxis wäre für schärfere Kriterien kaum bezahlbar. Als Konsequenz daraus beschränkt sich die Industrie selbst auf diese eigentlich unzureichenden Kriterien.

Im Vordergrund der industriellen Praxis steht also die Minimierung der Testfallmengen. Dies führt zu einem Konflikt mit den funktionalen Anforderungen für die Qualität von Testfallmengen (s. Abschnitt 2.1.2).

Um diese Umkehrung der Prioritäten zwischen den Anforderungen aufheben zu können, muss der Aufwand für Testfallauswahl, Testausführung und -auswertung reduziert werden. Dies ist nur durch Automation der bisher manuellen Abläufe möglich.

Ist der Aufwand reduziert, so können auch stärkere als die bisher im Einsatz befindlichen Kriterien definiert und umgesetzt werden. Um etwa Schleifenkonstrukte besser abzubilden, kann das Überspringen oder mindestens zweimalige Durchlaufen einer jeden Schleife gefordert werden[GGBC06].

Anwendungsspezifisch können Testeingaben und -fälle für individuell vorgegebene Abläufe am Programmflussgraphen definiert werden. Ähnlich wie bei den beste-

henden Kriterien können neue Vorgaben auch domänenspezifisch verallgemeinert und für jedes Programm neu instantiiert werden.

Schließlich können die Testfallmengen grundsätzlich vergrößert werden, indem zum Beispiel beim Kriterium all-nodes nicht nur einer sondern mehrere Testfälle für jeden Knoten gefordert werden. Die Qualität der Testfallmengen ist dann weniger abhängig von der Wahl des Kriteriums, da größere Testfallmengen tendenziell größere Chancen haben, die funktionalen Anforderungen aus Abschnitt 2.1.2 an die Qualität von Testfallmengen zu erfüllen.

Zudem eröffnet sich durch die großen Testeingabemengen die Möglichkeit einer statistischen Auswertung der Beobachtungen beim Test, etwa auf Basis heuristischer Orakel[Ger07, GGBM07, GGB07].

3.2 Überblick

Beim Modultest existieren für die Auswahl von Testeingaben verschiedene Verfahren, die wir in den folgenden Abschnitten genauer betrachten und für die wir darlegen werden, dass sie in vielen Anwendungsfällen unzureichend sind.

Für die automatische Ausführung der selektierten Testfälle beim Modultest kann eines der vielen verfügbaren Rahmenwerke wie zum Beispiel JUnit für Java[BG98] verwendet werden.

Beim Modultest besteht die Schnittstelle meist in Prozedur- oder Funktionsaufrufen mit typisierten Parameterlisten, die einfach aus dem Programmcode zu extrahieren sind. Bei Integrations-, System- und Abnahmetest hingegen tauchen auch heterogene und weit komplexere Schnittstellen auf, die nicht so einfach automatisch analysierbar sind, etwa bei anwendungsspezifischen Protokollen oder graphischen Oberflächen.

Testfälle können hier zum Beispiel anhand formaler Modelle automatisch abgeleitet werden[HHSS08, WRQC08]. Solche Modelle sind in zunehmendem Maße ohnehin Teil der Analyse und Entwicklung.

Auch hier werden constraintbasierte Methoden zur Testfallbestimmung benötigt, etwa bei Modellen auf Basis von endlichen Zustandsautomaten[WRQC08]. Die operationelle Semantik solcher Modelle zeigt große Ähnlichkeiten mit der operationellen Semantik von Programmflussgraphen, wie in dieser Arbeit definiert. Es besteht also Grund zur Annahme, dass Fortschritte in dieser Richtung beim Modultest auch auf den modellbasierten Test für die anderen Testdisziplinen übertragen werden können.

Für die automatische Testausführung werden domänen- oder anwendungsspezifische Testtreiber benötigt. Im Bereich graphischer Oberflächen etwa existieren kommerzielle Werkzeuge wie WinRunner, mit denen solche Testtreiber entwickelt werden können.

Die Prüfung der Ausgaben kann durch sogenannte Orakel[Hof01] erfolgen. Ein Orakel ist ein Programm, das entweder zu einer Eingabe die erwartete Ausgabe liefert oder anhand einer Eingabe und der beobachteten Ausgabe bestimmt, ob letztere korrekt ist. Letztere Form kommt insbesondere dann zum Einsatz, wenn zu einer Eingabe mehr als eine korrekte Ausgabe möglich ist.

Orakel müssen nicht immer vollständig sein, sondern können auch Wahrscheinlichkeiten für die Korrektheit statt einer reinen korrekt/nicht-korrekt-Entscheidung liefern.

So können zum Beispiel statistische Analysen der Ausgaben[MG04], Symmetrien in der Spezifikation[Got03] oder Referenzimplementierungen und ausführbare Modelle[Arj04] als Basis genutzt werden. Grundlage für einfach zu implementierende Orakel kann auch der von einer Variablen bei der Ausführung minimal oder maximal angenommene Wert oder das Auftreten von Laufzeitfehlern oder Zeitüberschreitungen sein[GGB07].

Zur Auswertung gehört neben der Prüfung der Ausgaben auf Korrektheit auch die Messung der Codeabdeckung. Dies geschieht über die Instrumentierung des Programmcodes oder der Hardware, auf der die Tests ausgeführt werden. Alternativ kann die Ausführung der Tests auch auf einem Modell simuliert und damit die Abdeckung auf der jeweiligen Abstraktionsstufe bestimmt werden.

Es sind außerdem bereits mehrere Werkzeuge zur Messung der Codeabdeckung nach verschiedenen Kriterien durch Instrumentierung im Einsatz[Hof09, GGB07, Sch08].

Für die Messung der spezifikationsbasierten Abdeckung ist auch im automatischen Softwaretest eine manuelle Zuordnung von Testfällen zu den einzelnen Anforderungen in der Spezifikation notwendig.

Die spezifikationsbasierte Abdeckung kann jedoch auch gemessen werden, indem jeder Anforderung eine formale Testanforderung zugeordnet wird. Aus dieser können dann Testfälle oder -eingaben in der benötigten Menge automatisch erzeugt werden. Alternativ kann umgekehrt für jeden erzeugten Testfall – zum Beispiel beim Zufallstest – anhand der Testvorgaben automatisch festgestellt werden, welcher Anforderung er zuzuordnen ist. Mit dieser Methode ist der manuelle Zuord-

nungsaufwand nicht von der Zahl der Testfälle sondern lediglich von der Zahl der Anforderungen abhängig.

3.3 Stand der Wissenschaft

Die bekannten Methoden der Testdatenerzeugung können grob in zufällige und zielgerichtete Methoden unterteilt werden. Bei den zielgerichteten Methoden wird das Testziel oder -kriterium zur Einschränkung der Kandidatenmenge beziehungsweise für eine geführte Suche herangezogen, bei den zufälligen jedoch nicht.

Der Zufallstest dient meist als Vergleichsgrundlage, da er eine untere Schranke für die Menge verwendeter Information über die Struktur von Prüfling und Testkriterium darstellt. Ein Verfahren, das komplizierter ist und eine höhere Laufzeit- oder Speicherkomplexität als der Zufallstest aufweist, ist nur dann als sinnvoll für den Einsatz anzusehen, wenn es eine Trefferrate aufweist, die in entsprechendem Maße höher ist.

3.3.1 Zufallstest

Die zufällige Erzeugung von Testeingaben[Ham94] ist wohl das am einfachsten zu implementierende Verfahren. Hierfür wird üblicherweise die Schnittstelle des Prüflings – etwa die Parameterdeklaration einer Funktion – analysiert und daraus Zufallsgeneratoren für die benötigten Datentypen konstruiert[GF93].

In Listing 3.1 ist dargestellt, wie beispielsweise Testdaten für eine Funktion mit zwei vorzeichenlosen Ganzzahlparametern erzeugt werden können. Die Parameter werden zusammen mit dem Ergebnis zur weiteren Verarbeitung aufgezeichnet.

Aufgrund der zufälligen Auswahl der Testeingaben können bei ausreichend großen Testsätzen aufgrund statistischer Analysen Aussagen über die Qualität der getesteten Software gemacht werden, z.B. auf Basis einer Restfehlerschätzung.

Allerdings kann die Wahrscheinlichkeit für die Erfüllung eines vorgegebenen Testziels verschwindend gering werden, wenn die Menge der geeigneten Eingaben relativ zur gesamten Eingabemenge klein ist.

Im Listing 3.1 wäre ein solcher Spezialfall z.B. der Grenzfall der Schleife, `a==b`. Die Wahrscheinlichkeit, zufällig zwei Elemente aus einer Menge der Größe n auszuwählen, so dass sie gleich sind, ist $\frac{1}{n}$. Entsprechend ist $1 - \left(1 - \frac{1}{n}\right)^k$ die Wahrscheinlichkeit, nach k Versuchen mindestens ein Paar gleicher Elemente gewählt

```
1  unsigned int ggt(unsigned int a, unsigned int b) {
2          while (a!=b) {
3                  if (a<b) b = b - a;
4                  else a = a - b;
5          }
6          return a;
7  }
8
9  void testggt() {
10         unsigned int c, a, b, result;
11         for (c=0;c<1000;++c) {
12                 a = random_unsigned_int();
13                 b = random_unsigned_int();
14                 result = ggt(a,b);
15                 printf("ggt(%d,%d)=%d\n",a,b,result);
16         }
17 }
```

Listing 3.1: Zufällige Testdatenerzeugung

zu haben. Um also bei 32-Bit-Werten zu 90% sicher sein zu können, dass mindestens eine solche Eingabe erzeugt wurde, sind im Schnitt fast 10 Milliarden Versuche notwendig.

Ein ähnliches Problem entsteht, wenn die Deklaration der Parametertypen zu ungenau ist. In den üblichen Programmiersprachen kann die Eingabemenge einer Prozedur oder Funktion nur als kartesisches Produkt der Wertemengen der einzelnen Parameter definiert werden. Abhängigkeiten unter den Parametern, etwa $i + j < 100$ für zwei natürliche Zahlen i und j sind nicht als Teil der Parameterdeklaration beschreibbar. Ebenso bieten manche Programmiersprachen wie Java oder C nur sehr einfache Basistypen an, die nicht weiter eingeschränkt werden können.

Sind nur Eingaben im Bereich von 0 bis 7 zulässig, so ist in C der Datentyp der Wahl `unsigned char`. Dieser lässt jedoch 256 verschiedene Werte zu und ist somit um Faktor 32 überdimensioniert. Das Resultat beim Zufallstest ist eine nicht zweckmäßige Überbetonung der unzulässigen Eingaben gegenüber den zulässigen Eingaben im Verhältnis 31 zu 1.

Beim adaptiven Zufallstest[CHK07] wird für jede Testeingabe ein Element aus einer Menge von zufällig gewählten Kandidaten ausgewählt. Die Auswahl wird so getroffen, dass die Distanz der Testeingaben voneinander entsprechend einer vorgegebenen Distanz maximiert wird. Dadurch soll vermieden werden, dass mehrere nah beieinanderliegende Testfälle denselben Fehler aufdecken und damit überflüssigen Testaufwand verursachen.

Hierdurch können die zuvor genannten Probleme geringer Trefferwahrscheinlichkeiten jedoch nicht beseitigt werden, da die Testeingaben auch hier rein zufällig ausgewählt werden.

3.3.2 Zielgerichtete Verfahren

Bei vielen der bekannten Verfahren wird zunächst ein Weg vom Einsprung- zu einem Endpunkt des Programms ausgewählt und dann eine Eingabe gesucht, die zur Ausführung dieses Weges führt.

Eine Möglichkeit ist, zunächst aus dem Programmflussgraphen einen Weg entsprechend der Vorgaben zu konstruieren. Aus diesem Weg wird dann das Wegconstraint bestimmt und dieses gelöst. Dieses Vorgehen schlagen in verschiedenen Variationen DeMillo und Offutt[DO91], Offutt et al[OJP97] sowie Denise et al[DGG04] vor. Insbesondere können dabei Verfahren aus dem Bereich der regulären Sprachen zur Beschreibung der nach dem Testkriterium interessanten Wege verwendet werden[Oud07].

Der Nachteil dieser Ansätze ist allerdings, dass die Ausführbarkeit des gewählten Weges erst dann erkannt wird, wenn er bereits vollständig konstruiert ist. Dann muss ein neuer Weg erzeugt und auf seine Ausführbarkeit geprüft werden.

Alternativ kann zunächst die Eingabe ausgewählt und das Programm mit dieser Eingabe ausgeführt werden, wie Korel[Kor90] sowie Godefroid et al[GKS05] vorschlagen. Bereits während der Ausführung wird der ausgeführte Weg beobachtet und die Ausführung abgebrochen, wenn erkennbar ist, dass der Weg das Testkriterium nicht erfüllen wird, etwa weil ein bestimmter Knoten gar nicht mehr erreicht werden kann.

Der Weg bis zum Abbruchpunkt kann als ausführbar vorausgesetzt werden. Für ihn oder einen Teil davon wird deshalb sein Wegconstraint erstellt und um die Bedingung für die gewünschte Abzweigung ergänzt. So wird eine neue Eingabe erzeugt, mit der das Programm erneut ausgeführt wird.

Bei diesem Verfahren werden nicht-ausführbare Wege zum Teil vermieden, da die Konstruktion des Weges ja durch die Ausführung des Programms selbst gesteuert wird.

Über weite Teile der Ausführung kann die Auswahl des Weges aber nicht beeinflusst werden, da sie hauptsächlich von der Wahl der ersten und aller weiteren Eingaben abhängt. Aus den Teilwegen wird der Weg sequentiell zusammengesetzt und es ist nicht klar, ob überhaupt ein ausführbarer Weg vom erreichten Knoten zum Zielknoten existiert.

Das Verfahren ist deshalb vor allem zur schnellen Erreichung einer möglichst hohen Abdeckung schleifenfreier Wege geeignet[God07].

Ebenso kann beginnend beim Startknoten vollständig sequentiell vorgegangen werden, indem jeweils ein passender Nachfolger gesucht wird. Dieses Verfahren wenden Nguyen und Deville[ND01] an. Bei jedem Schritt wird die Bedingung für die durchlaufene Kante dem bereits bestehenden Wegconstraint hinzugefügt. Wird ein Widerspruch erkannt, so ist der bis zu diesem Punkt konstruierte Weg bereits nicht ausführbar. Somit wird der Suchzweig abgebrochen und die Suche bei einer früheren Alternative fortgesetzt.

Bei der Auswahl des Nachfolgers können Präferenzen wie etwa die Kontrollabhängigkeiten des zu erreichenden Knoten oder die möglichst frühe Beendigung einer Schleife berücksichtigt werden.

Bei diesem Verfahren können früh im Ablauf Entscheidungen getroffen werden, für die sich erst spät herausstellt, dass sie zur Testvorgabe – zum Beispiel dem Erreichen eines bestimmten Knotens – im Widerspruch stehen. Dann muss ein Teil dieser Entscheidungen durch Backtracking rückgängig gemacht und vom neuen Aufsetzpunkt ein neuer Weg konstruiert werden.

Die Präferenz bzgl. einer frühen Beendigung von Schleifen ist optional und im Zusammenhang mit der Schwäche mancher Abdeckungskriterien bzgl. iterativen Kontrollflusskonstrukten äußerst kritisch zu sehen (s. Abschnitt 3.1).

Das Testwerkzeug InKA von Gotlieb et al[GBR98] implementiert ein Verfahren, in dem der Weg als Verkettung mehrerer Teilwege konstruiert wird. Die Eingabe ist in diesem Fall nicht ein Programmflussgraph, sondern das Programm in einer `while-if`-Sprache unter Verwendung einer speziellen SSA-Form.

Die einzelnen Konstrukte werden als Meta-Constraints dargestellt, deren logische Lesform exakt der Semantik dieser Konstrukte entspricht. Schleifen können über entsprechende Regeln abgerollt und Entscheidungen getroffen werden. Konstruk-

te, die im Programm sequentiell aufeinander folgen, können in dieser Constraintdarstellung nebeneinander betrachtet und bearbeitet werden.

Gegenüber den anderen Verfahren besitzt dieses den Vorteil, dass nicht streng sequentiell vorgegangen, sondern der sequentiellen Ausführung vorgegriffen wird. Insbesondere können so Entscheidungen auch am Beginn des gewählten Ausführungsweges frühzeitig erkannt werden, die zum Testziel im Widerspruch stehen.

Der Nachteil besteht allerdings in der Notwendigkeit einer speziellen Form für das zu testende Programm. Diese ist zwar ebenso wie die Programmflussgraphen turing-vollständig und somit äquivalent in letztere zu übersetzen. Um jedoch die Vorteile des Verfahrens zu erhalten, muss nicht nur die Ausführungssemantik sondern auch die Struktur des Programms – Schleifen und Konditionalanweisungen – sinnvoll in die `while-if`-Sprache übersetzt werden. Die reine Emulation, wie sie meist im Rahmen des konstruktiven Äquivalenzbeweises von Sprachen angewendet wird, ist nicht ausreichend.

Schließlich können auch genetische Algorithmen eingesetzt werden, wie von Pargas et al[PHP99] untersucht. Das Ziel ist die Abdeckung einer bestimmten Anweisung. Die Populationen bestehen dabei aus Mengen von Testeingaben. Als Fitnessfunktion wird die Anzahl der Kontrollabhängigkeiten der Zielanweisung verwendet, die durch die Testeingaben erfüllt wird. Durch Mutation und Rekombination werden jeweils neue Populationen erzeugt um die Anzahl der erfüllten Kontrollabhängigkeiten zu maximieren und so eine Testeingabe zu finden, durch die die Anweisung erreicht wird.

Dieses Verfahren ist spezifisch auf die Anweisungsabdeckung ausgelegt und damit wenig flexibel.

Pargas et al zeigen für einige Beispiele einen höheren Durchsatz als für den Zufallstest. Darunter ist auch eines, das zum Erreichen eines bestimmten Knotens im Programm die Gleichheit aller drei Parameter erfordert. Dieser Fall wird vermutlich lediglich durch eine Besonderheit der Rekombinationsmethode abgedeckt, durch die mehrere Parameter bei Rekombination eines Individuums mit sich selbst denselben Wert annehmen können.

Es ist fraglich, ob das Verfahren auch bei komplexeren Bedingungen noch besser ist als der Zufallstest, etwa beim Ausdruck `a*a*a+b==c` mit Parametern `a`, `b` und `c`.

Wie beim Zufallstest kann es sein, dass die Ursprungspopulation keine Eingabe enthält, die das Ziel erreicht. Eine passende Eingabe kann also nur dann entstehen, wenn die Individuen der Population nicht nur vermischt sondern individuell

mutiert und rekombiniert werden, so dass völlig neue Eingaben entstehen. Es erscheint unsicher, dass zwei gute Eingaben bei der Rekombination eine erheblich höhere Chance haben sollen, das Ziel zu erreichen als eine einfach zufällig gewählte Eingabe.

3.3.3 Lösung der Wegconstraints

Lösungen für die Wegconstraints werden in den in der Literatur dokumentierten Verfahren entweder mit heuristischen oder constraintbasierten Verfahren abgeleitet. Bei einigen der Verfahren muss der Constraintlöser den schrittweisen Aufbau des Wegconstraint sowie die Erkennung von Widersprüchen bei noch unvollständigen Wegconstraints unterstützen.

Bei den heuristischen Verfahren werden die Bedingungen in eine Bewertungsfunktion transformiert, die von den Eingabeparametern abhängt[Kor90, PHP99]. Der Wert der Bewertungsfunktion wird dann mit numerischen Verfahren durch Modifikation der Eingabeparameter maximiert oder minimiert.

Eine Ungleichung der Form $a > b$ wird etwa in $F(a, b) = a - b$ transformiert und diese wird dann maximiert, um eine Lösung der Ungleichung $F(a, b) > 0$ zu finden. Kann der Funktionswert nicht mehr weiter maximiert werden und ist die Ungleichung dennoch nicht erfüllt, so wird ein Widerspruch in den Gleichungen angenommen. Dabei ist zu beachten, dass lokale Maxima zur fehlerhaften Annahme eines Widerspruchs führen können.

Bei constraintbasierten Verfahren[DO91, GBR98, ND01, DGG04, GKS05] werden die Bedingungen als von den Eingabeparametern oder internen Variablen abhängige Prädikate dargestellt. Über bestimmte Regeln werden die Bedingungen umgeformt und vereinfacht.

Idealerweise steht am Ende dieser Transformation eine gelöste Form, also eine Liste von Äquivalenzen $v_1 = c_1, v_2 = c_2, \ldots$ mit Variablen v_i und Konstanten c_i. Dies ist allerdings selten der Fall, da Wegconstraints meist unterbestimmt und somit mehrere Lösungen möglich sind.

Stattdessen werden oft Finite-Domain-Constraintlöser (FD-Solver) eingesetzt. Jeder der Variablen wird dort ein endlicher Wertebereich (finite domain) zugeordnet, entweder als Intervall oder als endliche Liste von Werten.

Diese Wertebereiche werden schrittweise auf Basis der Constraints reduziert bis ein Fixpunkt erreicht ist. Dann werden die Wertebereiche künstlich eingeschränkt, z.B. durch ein Bisektionsverfahren oder aber durch zufällige Auswahl eines Werts

aus dem Wertebereich. Dies wird solange wiederholt, bis alle Variablen auf einen Wert beschränkt sind.

Ein Widerspruch ist dann erkannt, wenn einer der Wertebereiche auf die leere Menge reduziert wurde. In diesem Fall muss durch Backtracking eine andere Alternative für die Belegung probiert werden.

3.4 Diskussion

Bei genauerer Betrachtung des Stand der Technik fallen einige Unzulänglichkeiten der verfügbaren Verfahren im Sinne eines industriellen Einsatzes auf.

Wie wir an der Problematik der geringen Trefferwahrscheinlichkeiten beim Zufallstest sehen, muss das zu erfüllende Testkriterium oder die entsprechende Testvorgabe explizit zur Reduktion des Suchraums verwendet werden. Ansonsten ist die pro erzeugter Testeingabe aufzubringende Zeit zu hoch und damit der Durchsatz des Verfahrens zu gering für den praktischen Einsatz.

Somit verbleiben als mögliche Lösungen lediglich die zielgerichteten Verfahren von DeMillo und Offutt, Offutt et al, Denise et al, Korel, Godefroid et al, Nguyen und Deville sowie Gotlieb et al. Auch genetische Algorithmen, wie von Pargas et al vorgeschlagen, sind nicht einsetzbar, da sie den Suchraum nicht beschränken und ihre Trefferquote für die allgemeine Anwendung als zu niedrig angenommen werden muss.

3.4.1 Klassische und individuelle Testkriterien

Für den industriellen Einsatz benötigen wir zunächst ein Verfahren, das die Auswahl von Testdaten für die gängigen Testkriterien wie in Abschnitt 2.1.3 definiert unterstützt.

Unsere Vision von der domänen- und anwendungsspezifischen Teststrategie als Basis des automatisierten Softwaretest erfordert darüber hinaus auch die Unterstützung individueller Testvorgaben. Die Menge möglicher Testfälle soll dabei über Beschränkungen von Ein- und Ausgaben sowie des Ausführungsweges definierbar sein.

Hierfür sind nur die Verfahren von DeMillo und Offutt, Offutt et al, Denise et al, Nguyen und Deville oder Gotlieb et al verwendbar. Nur sie wählen einen Weg entsprechend der Vorgaben und konstruieren daraus ein Wegconstraint.

Die Verfahren von Korel sowie Godefroid et al bieten nicht genügend Einflussmöglichkeiten auf den Weg, um verschiedene Kriterien durch Beschränkung der Wegmenge sinnvoll zu unterstützen.

Das Werkzeug InKA von Gotlieb et al unterstützt explizit die Knoten- und Kantenabdeckung. Dafür müssen Constraints hinzugefügt und gegebenenfalls dynamisch neu bestimmt werden, die die Kontrollabhängigkeiten der zu erreichenden Knoten und Kanten darstellen.

Da die klassischen Kriterien anhand des Kontrollflussgraphen definiert sind, müssen sie zunächst in die `while-if`-Form transformiert werden, wofür gegebenenfalls auch eine Umformung des Prüflings notwendig ist.

Die verbleibenden Verfahren von DeMillo und Offutt, Offutt et al, Denise et al sowie Nguyen und Deville erlauben die explizite Selektion eines Weges im Kontrollflussgraphen.

Hier können die klassischen Kriterien direkt anhand des Kontrollflussgraphen angewendet werden. Auch weitergehende Kriterien sind entsprechend darstellbar.

3.4.2 Verarbeitung von Programmflussgraphen

Programme können in einer Vielfalt von Formen vorliegen. Als Beispiele seien `while-if`-Sprachen, funktionale Darstellungen in Haskell oder logische Programme in Prolog genannt. Alle diese Sprachen sind – so sie turingvollständig sind – äquivalent ineinander übersetzbar.

Bei der Übersetzung von Programmen in Maschinencode wird genau diese Äquivalenz ausgenutzt. Als interne Repräsentation nutzen Übersetzer von imperativen und objektorientierten Sprachen dabei meist einen Basisblockgraphen[ASU97], der direkt in einen Programmflussgraphen überführt werden kann.

In der Industrie ist auch oft der Abdeckungsnachweis auf Ebene der Maschinensprache gefordert[Rad92], so dass auch die Erzeugung von Testdaten aus Assemblercode unterstützt werden muss. Basisblockgraphen sind ebenso aus Maschinencode einfach zu konstruieren.

Die nach den vorigen Überlegungen noch verbleibenden Verfahren von DeMillo und Offutt, Offutt et al, Denise et al sowie Nguyen und Deville verwenden alle die Darstellung des Prüflings als Programmflussgraph.

Lediglich das Werkzeug InKA von Gotlieb et al bildet eine Ausnahme, da es den Prüfling als `while-`if-Programm erwartet. Die Erkennung nicht-ausführbarer Wege hängt hier stark davon ab, wie der Prüfling strukturiert ist.

```
1  node:=r
2  while node <> e do
3      if node=r then
4          i:=1
5          found:=false
6          node:=1
7      else if node=1 then
8          if i>n then node:=5 else node:=2 end if
9      else if node=2 then ...
10     else if node=3 then ...
11     else if node=4 then ...
12     else if node=5 then
13         if found=true then node:=6 else node:=e end if
14     else if node=6 then
15         print('item was found')
16         node:=e
17     end if
18 done
```

Listing 3.2: Lineare Suche als while-if-Programm

Auch ein Programmflussgraph kann natürlich in ein `while-if`-Programm übersetzt werden. Der Übersetzungsansatz, wie er aus konstruktiven Äquivalenzbeweisen bekannt und in Listing 3.2 für den Graphen aus Abbildung 5.2 dargestellt ist, ist jedoch nicht ausreichend. Der Kontrollfluss wird zwar emuliert, aber wesentliche Zusammenhänge sind nicht mehr ohne Weiteres erkennbar.

Es bedarf einigen Aufwands zum Beispiel durch abstrakte Interpretation, um festzustellen, dass `node` den Wert `3` nicht mehr annehmen kann, sobald er den Wert `5` zum ersten Mal angenommen hat. Aus dem Kontrollflussgraphen heraus ist diese Information hingegen mit den Grundlagen der Graphentheorie einfach abzuleiten.

Sprachen wie JAVA[GJSB05] oder C bieten Anweisungen wie `break`, `continue` oder gar `goto`, deren Übersetzung zum Beispiel in ein `while-if`-Programm ohne Verlust der Kontrollflussinformation ein kompliziertes Verfahren erfordert.

Somit besitzt das Verfahren von Gotlieb et al hier einen erheblichen Nachteil. Die Verfahren von DeMillo und Offutt, Offutt et al, Denise et al sowie Nguyen und Deville hingegen können direkt auf Progammflussgraphen angewendet werden.

3.4.3 Behandlung nicht ausführbarer Wege

In Abschnitt 2.3.6 haben wir bereits festgestellt, dass es in einem Programmflussgraphen Wege geben kann, die für keine der möglichen Eingaben ausgeführt werden. Dies gilt ebenso für andere turingvollständige Darstellungformen von Programmen.

Wir werden zeigen, dass diese nicht-ausführbaren Wege bereits bei einfachen Beispielen einen erheblichen Anteil an der Menge der Wege im Programmflussgraphen ausmachen.

Ein Verfahren zur Auswahl von Testeingaben, das auf Basis des Programmcodes arbeitet, muss dieser Tatsache Rechnung tragen und solche nicht-ausführbaren Wege vermeiden. Ansonsten kann der Durchsatz des Verfahrens in nicht vernachlässigbarem Umfang reduziert werden.

Somit verbleibt lediglich das Verfahren von Nguyen und Deville. Das Verfahren von Gotlieb et al unterstützt ebenso die Vermeidung von nicht-ausführbaren Wegen, nicht aber die Darstellung des Prüflings als Programmflussgraph.

Denise et al und Korel wählen den Weg aus, bevor sie das Wegconstraint konstruieren, und erkennen nicht-ausführbare Wege zu spät.

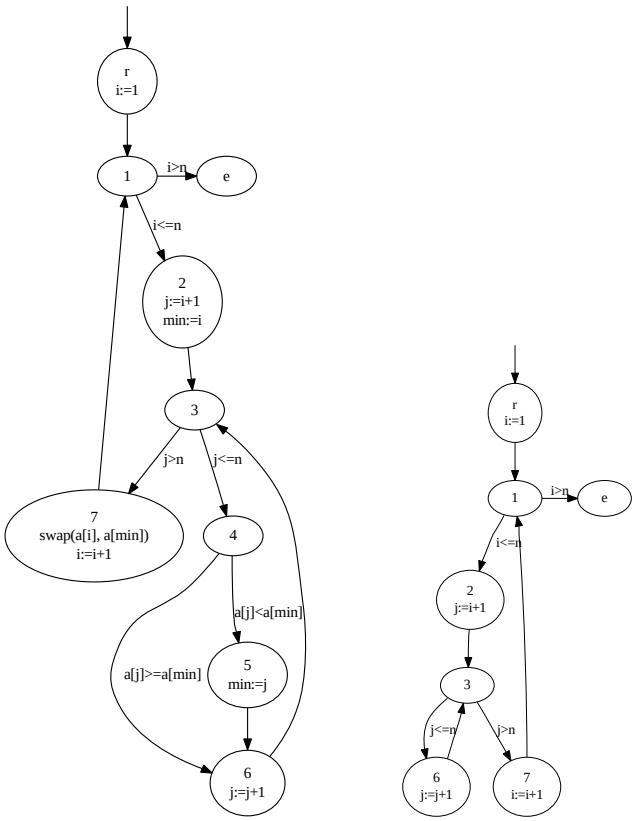
Wir werden dies anhand des Selection-Sort-Algorithmus aus Listing 2.1 deutlich machen, dessen Programmflussgraph in Abbildung 3.1a dargestellt ist. In Abbildung 3.1b wurden die Kante $3 \rightarrow 4$ durch eine Kante $3 \rightarrow 6$ ersetzt und die Knoten 4 und 5 sowie die `swap`-Anweisung zum Vertauschen der Werte von `a[i]` und `a[min]` aus Knoten 7 entfernt. Dadurch vereinfachen sich die folgenden Berechnungen.

Wir zeigen nun, dass bereits in einfachen Fällen der Anteil der nicht-ausführbaren Wege nicht mehr vernachlässigbar ist und somit reines Durchprobieren nicht effizient sein kann.

Die Menge der konstruierbaren Wege ist ebenso unendlich wie die der ausführbaren Wege. Deshalb untersuchen wir die Entwicklung der Anzahl der Wege mit deren maximaler Länge L.

Selection-Sort hat eine Komplexität von $O\left(n^2\right)$, d.h. bei einer Eingabe der Länge n wird ein Weg einer Länge $l\left(n\right) = O\left(n^2\right)$ abgelaufen. Dabei ist jedem n eindeutig ein Weg zugeordnet. Somit gibt es n ausführbare Wege, deren Länge maximal $l\left(n\right)$ ist. Die Zahl der Wege kürzer als L ist also $O\left(\sqrt{L}\right)$.

Betrachten wir nun die Menge der konstruierbaren Wege im Kontrollflussgraphen. Sei u die Anzahl der Durchläufe durch die äußere Schleife $1 \rightarrow 2 \rightarrow 3 \rightarrow^{+} 7$ und



(a) Original (b) Vereinfacht

Abbildung 3.1: Selection Sort

v die Gesamtzahl der Durchläufe durch die innere Schleife $3 \rightarrow 6$. Dann ist die Länge des zugehörigen Weges $l\,(u, v) = 2 + 4u + 2v$.

Jeder der v Durchläufe der inneren Schleife kann frei einem der Durchläufe der äußeren Schleife zugeordnet werden. Dafür gibt es jeweils u^v Möglichkeiten. Der Maximalwert von v für $u = 2$ wächst asymptotisch mindestens linear mit L. Somit wächst allein schon die Zahl der Möglichkeiten mit zwei Durchläufen der äußeren Schleife mindestens mit 2^L. Erst recht wächst die Gesamtzahl der konstruierbaren Wege mindestens exponentiell mit L.

Wir zeigen nun noch, dass die Vereinfachung des inneren Schleifenrumpfs diese Abschätzung nicht beeinträchtigt und betrachten dafür den Originalgraphen. Die Zahl der Durchläufe durch Knoten 4 ist $k = \frac{n(n-1)}{2}$ für ein Array der Länge n. Ignorieren wir die Bedingungen an den Kanten $4 \rightarrow 5$ und $4 \rightarrow 6$, so haben wir 2^k Möglichkeiten, für jeden Durchlauf des Knotens 4 einen Nachfolger zu wählen. Die Bedingungen an den genannten Kanten sind aber nur von der Sortierung des Arrays abhängig. Es existieren $n!$ Permutationen für ein Array dieser Länge und es gilt $2^k > n!$ für $n \geq 3$. Also können gar nicht alle möglichen Durchläufe durch die Knoten 4, 5 und 6 ausführbar sein. Dies gilt erst recht, wenn wir auch die anderen Kantenbedingungen ignorieren.

Also wächst der Quotient der Zahlen nicht-ausführbarer und konstruierbarer Wege exponentiell mit der maximalen Länge L und ist somit nicht vernachlässigbar. Die Ursache hierfür ist die gegenseitige Abhängigkeit von innerer und äußerer Schleife über den Iterationszähler `i`.

Bei verschachtelten Schleifen liegen solche Abhängigkeiten über Iterationsvariablen nahe. Ein ähnliches Problem besteht bei Schleifen, deren Iterationsanzahl fest oder beschränkt ist[Gou05].

Da die Verfahren von DeMillo und Offutt, Offutt et al sowie Denise et al zunächst den Weg auswählen und erst dann das Wegconstraint konstruieren, können diese nicht-ausführbare Wege nicht vermeiden. Lediglich die Verfahren von Nguyen und Deville sowie Gotlieb et al unterstützen die Vermeidung nicht-ausführbarer Wege grundsätzlich.

3.4.4 Vorwärts- und rückwärtsgerichtete Einschränkungen

Da die Menge nicht-ausführbarer Wege nicht vernachlässigbar ist, muss das gesuchte Verfahren eine Strategie zur Vermeidung dieser Wege implementieren. Die Existenz nicht-ausführbarer Wege manifestiert sich bei der Suche nach einer Aus-

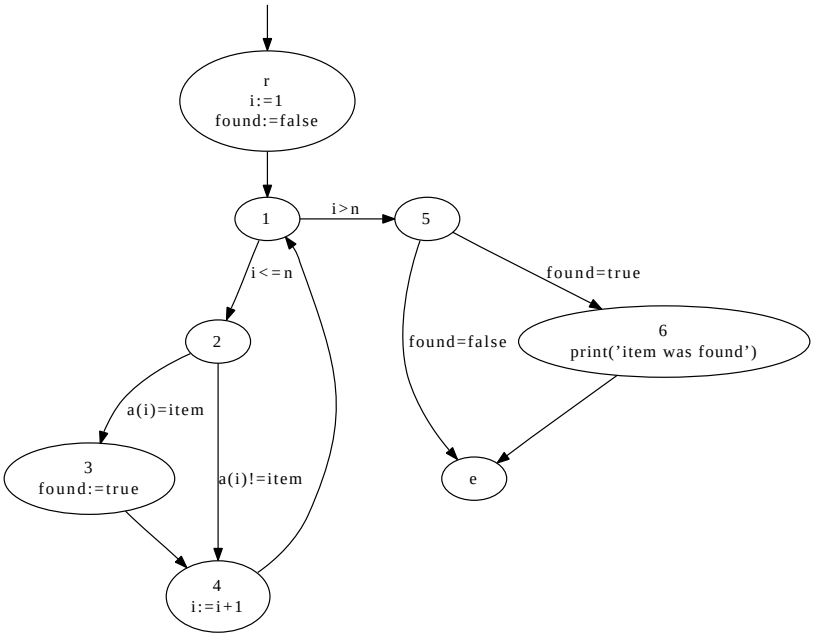


Abbildung 3.2: Lineare Suche

führungsfolge darin, dass Entscheidungen zum Beispiel über Verzweigungen getroffen werden, die in eine Sackgasse führen.

Betrachten wir als Beispiel den Programmflussgraphen einer linearen Suche in Abbildung 3.2. Nehmen wir an, wir suchen nach einer Ausführungsfolge vom Programmstart über die Kante $5 \rightarrow e$ zum Programmende. Wir konstruieren die Ausführungsfolge sequentiell, beginnend beim Startzustand $\langle r, \alpha, x \rangle$. Mit jeden Schritt treffen wir eine Entscheidung, welcher der Kanten wir folgen.

Dabei kann es sein, dass wir den Knoten 2 erreichen und uns entscheiden, der Kante $2 \rightarrow 3$ zu folgen. Nachdem wir Knoten 3 das erste Mal ausgeführt haben, gilt `found=true` unverändert bis wir Knoten 5 erreichen.

Dort erkennen wir, dass wir mit `found=true` die Kante $5 \rightarrow e$ gar nicht durchlaufen können. Durch Backtracking machen wir nun sukzessive Entscheidungen rückgängig und probieren die Alternativen. Möglicherweise haben wir den Knoten 3 aber mehrfach durchlaufen und es muss jede dieser Entscheidungen rückgängig gemacht werden.

Bei naiver Vorgehensweise oder komplexeren Ausdrücken als denen im Beispiel erkennen wir unter Umständen nicht, dass wir beim nächsten Versuch Knoten 3 grundsätzlich meiden müssen und stoßen deshalb erneut auf das Problem.

Konstruieren wir die Ausführungsfolge rückwärts beim Programmende beginnend, so durchlaufen wir automatisch gemäß Vorgabe die Kante $5 \rightarrow e$ und wissen so, dass `found=false` gelten muss.

Erreichen wir beim Rückwärtsdurchlauf den Knoten 4, so wird die Entscheidung für den Vorgänger 3 sofort zum Widerspruch führen. Das Backtracking umfasst nur einen Schritt und die Alternative führt uns direkt am Knoten 3 vorbei.

Umgekehrt führt die Vorgabe, Knoten 3 zu durchlaufen, zu keinen besonderen Problemen bei einem vorwärtsgerichteten Durchlauf. Erreichen wir Knoten 1 und haben Knoten 3 noch nicht durchlaufen, so wird Knoten 5 dadurch als Nachfolger ausgeschlossen, dass von ihm aus Knoten 3 nicht erreichbar ist.

Erreichen wir schließlich Knoten 5, nachdem wir Knoten 3 durchlaufen haben, so ist die Entscheidung zwischen den Kanten $5 \rightarrow e$ und $5 \rightarrow 6$ mit höchstens einem einzelnen Backtrackingschritt zu klären.

Problematisch wäre hier hingegen ein Rückwärtsdurchlauf.

Machen wir Vorgaben über zu durchlaufene Knoten oder Kanten, so können unsere Wahlmöglichkeiten sowohl in als auch entgegen der normalen Ausführungsrichtung des Programms eingeschränkt werden. Wir sprechen deshalb von in Ausführungsrichtung gesehen vorwärtsgerichteten und rückwärtsgerichteten Einschränkungen.

Das gesuchte Verfahren muss diesen Beschränkungen Rechnung tragen und darf somit durch die Ausführungsreihenfolge des Programms nicht in der Analysereihenfolge eingeschränkt sein.

Diese Vorgaben erfüllt nur das Verfahren von Gotlieb et al. Nguyen und Deville gehen rein sequentiell vor, beginnend beim Startknoten, und können somit rückwärtsgerichtete Beschränkungen nicht berücksichtigen.

3.5 Schlussfolgerungen

Nach den vorigen Überlegungen erkennen wir, dass keines der Verfahren die Anforderungen der industriellen Anwendung erfüllt. Dies ist auch in Tabelle 3.1 nochmals dargestellt. Lediglich die Verfahren von Nguyen und Deville sowie von Gotlieb et al kommen diesen Anforderungen besonders nahe.

Der Nutzen des Verfahrens von Gotlieb et al ist dadurch beschränkt, dass der typischerweise als Programmflussgraph vorliegende Prüfling zunächst in ein `while-if`-Programm transformiert werden muss.

Methode	**PFG**	**Kriterien**		**Vermeidung**	
		klass.	**allg.**	**vorw.**	**rückw.**
DeMillo und Offutt	ja	ja		nein	
Offutt et al					
Denise et al					
Korel	ja	teilweise	nein	ja	nein
Godefroid et al					
Nguyen und Deville	ja	ja		ja	nein
Gotlieb et al	nein	teilweise	nein	ja	

Tabelle 3.1: Vergleich der zielgerichteten Verfahren

Nguyen und Deville können hingegen direkt den Programmflussgraphen verarbeiten. Anders als bei Gotlieb et al kann dieses Verfahren jedoch nicht mit rückwärtsgerichteten Bedingungen umgehen.

Weder Nguyen und Deville noch Gotlieb et al unterstützen direkt und ohne Umformung des Prüflings die klassischen Kriterien sowie allgemeine Testvorgaben.

Wir schließen daraus, dass bis jetzt kein für den industriellen Einsatz geeignetes Verfahren existiert, auch wenn unter den bekannten Ansätzen die gerade genannten dem Ziel am nächsten kommen.

Wir stellen in dieser Arbeit ein Verfahren vor, das für einen Prüfling in Form eines Programmflussgraphen für alle gängigen Testkriterien Testdaten erzeugen kann. Dafür sind keine Umformungen des Programms notwendig. Das Verfahren vermeidet nicht-ausführbare Wege und trägt vorwärts- und rückwärtsgerichteten Beschränkungen Rechnung. Wir werden außerdem aufzeigen, wie für unser Verfahren allgemeine Testvorgaben anhand von Ein- und Ausgabebeschränkungen sowie Bedingungen für die zulässigen Ausführungswege formuliert werden können.

4 Formalisierung von Testkriterien

In diesem Kapitel werden wir zunächst eine formale Darstellung der Testkriterien aus Abschnitt 2.1.3 auf Grundlage von Ausführungsfolgen von Programmflussgraphen vorstellen.

Diese Darstellung wird auf Systeme einiger weniger generischer Grundrelationen zurückgeführt, für die dann mit der Methode aus Kapitel 5 Lösungen gefunden werden können.

Weiterhin führen wir einige Werkzeuge zur formalen Konstruktion weitergehender Testkriterien auf Basis der Transformation und Modifikation von Programmflussgraphen ein.

Sowohl bekannte als auch neue Kriterien können so deutlich klarer und flexibler formuliert werden als es die bisherigen Verfahren ermöglichen.

Die Abdeckungskriterien all-nodes und all-edges können ebenso wie alle datenflussbasierten Kriterien bis auf all-du-paths ohne Umwege über Kontrollabhängigkeiten oder Graphtransformationen dargestellt werden.

Für die Darstellung des Kriteriums all-du-paths kann ein geringfügig veränderter Programmflussgraph genutzt werden. Das Verfahren hierzu stellen wir in Abschnitt 4.7 vor.

In den meisten Fällen ist die formalisierte Darstellung des Ziels direkt aus dem Testkriterium ableitbar und umgekehrt das Kriterium aus dem formalen Ziel einfach ablesbar.

4.1 Grundlagen formaler Testkriterien

Betrachten wir die Kriterien aus Abschnitt 2.1.3, so fällt auf, dass sie alle aus Aussagen der Form „in der Testfallmenge existiert mindestens ein Testfall t, so dass $P(t)$ erfüllt ist“ zusammengesetzt sind, wobei P ein Prädikat auf Testfällen ist.

Im Fall von *all-nodes* etwa existiert für jeden Knoten n ein Prädikat $Q_{n,PFG}(t)$, das genau dann erfüllt ist, wenn bei der Ausführung des Testfalls t der Knoten n durchlaufen wird.

Analog ist *all-edges* als Konjunktion von Prädikaten $Q_{(v,w),PFG}(t)$ definiert, die genau dann erfüllt sind, wenn bei Ausführung von t die Kante (v, w) durchlaufen wird.

Ebenso lässt sich der Durchlauf eines definitionsfreien Weges als Prädikat beschreiben und somit die entsprechenden datenflussorientierten Kriterien aus Abschnitt 2.1.3 definieren.

Definition 4.1: Sei P ein Programm und $\mathcal{T}_P$ die Menge möglicher Testfälle für P.

Ein *Testkriterium* Γ_P ist eine Untermenge von $\mathcal{P}(T_P)$.

Wir sagen, eine Testfallmenge $\mathcal{T} \subseteq \mathcal{T}_P$ erfüllt Γ_P genau dann, wenn $\mathcal{T} \in \Gamma_P$, und schreiben $\Gamma_P(\mathcal{T})$.

Aufgrund der zuvor dargelegten Beobachtungen nehmen wir im Folgenden an, dass die für uns relevanten Kriterien als Konjunktion von Aussagen der Form $\exists t \in \mathcal{T} : P(t)$ definiert sind. Dies beschränkt die Allgemeinheit unserer Betrachtungen, nicht jedoch die Relevanz für den Einsatz in der Praxis.

Testkriterien können auch unerfüllbar sein. Dies ist etwa der Fall für *all-nodes*, wenn ein Knoten n des Programms niemals ausgeführt werden kann. In diesem Fall wird das zugehörige Prädikat $Q_{n,PFG}$ durch keinen Testfall und somit das Kriterium auch durch keine Testfallmenge erfüllt.

Definition 4.2: Sei P ein Programm und $\mathcal{T}_P$ die Menge aller möglichen Testfälle für P. Sei Γ_P ein Testkriterium für P.

Das Kriterium Γ_P ist *erfüllbar*, wenn $\Gamma_P \neq \emptyset$.

Satz 4.1: *Die Erfüllbarkeit eines Kriteriums Γ ist nicht entscheidbar.*

Beweis. Sei P ein Programm.

Sei x eine Eingabe für P und Γ_x ein Testkriterium für P, so dass $\Gamma_x(\mathcal{T})$ für eine Testfallmenge $\mathcal{T}$ genau dann erfüllt ist, wenn P auf der Eingabe x terminiert.

Somit ist Γ_x genau dann erfüllbar, wenn P auf x terminiert. Da das allgemeine Halteproblem nicht entscheidbar ist, ist auch die Erfüllbarkeit von Γ_x nicht entscheidbar. □

4.1.1 Testfallsuche

Bei der Testfallsuche wird üblicherweise festgestellt, welche Prädikate des Testkriteriums noch nicht durch die bereits vorhandene – zu Beginn leere – Testfallmenge

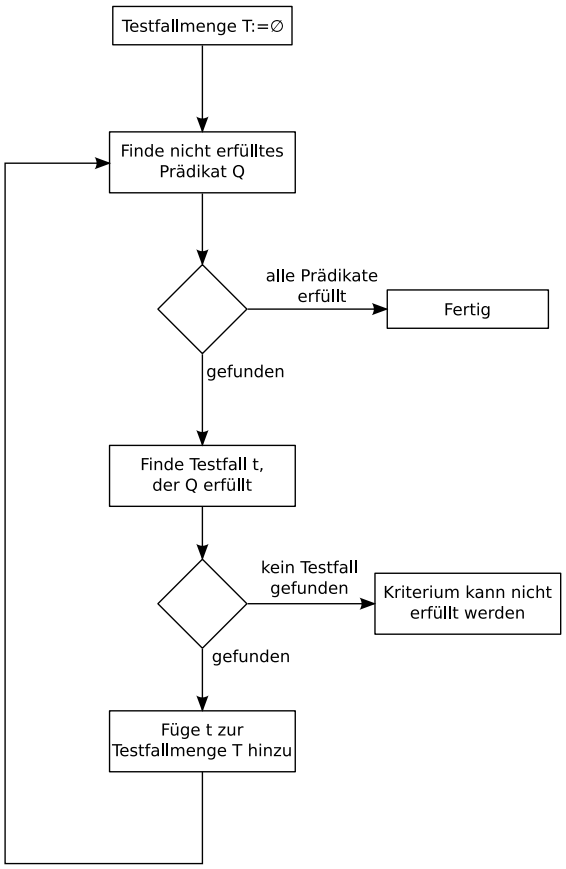


Abbildung 4.1: Testfallsuche

erfüllt sind. Ein solches nicht erfülltes Prädikat kann etwa einen noch nicht abgedeckten Knoten im Programmflussgraphen repräsentieren. Dann wird ein Testfall gesucht, der möglichst viele dieser Prädikate erfüllt. Alternativ wird eines der Prädikate ausgesucht und ein Testfall konstruiert, der dieses Prädikat erfüllt. Dieser Ablauf ist in Abbildung 4.1 dargestellt.

Nehmen wir also an, dass ein Programm P zu testen und dabei das Kriterium Γ zu erfüllen ist. Wir gehen davon aus, dass $\mathcal{T}_i$ die Kandidatenmenge bereits gefundener Testfälle ist. Wir wählen nun ein Prädikat Q von Γ, das noch nicht erfüllt ist. Existiert kein solches, so erfüllt $\mathcal{T}_i$ das Kriterium und die Suche ist abgeschlossen.

Sonst suchen wir einen Testfall t so, dass $Q\,(t)$ gilt. Existiert kein solcher Testfall, so ist Γ unerfüllbar. Ansonsten bestimmen wir die nächste Kandidatenmenge $\mathcal{T}_{i+1} = \mathcal{T}_i \cup \{t\}$ und fahren mit dieser fort.

Bei jedem Schritt reduzieren wir die Menge der noch nicht erfüllten Prädikate um mindestens eines. Somit terminiert der Algorithmus wenn Γ durch eine endliche Menge von Prädikaten definiert ist, vorausgesetzt, für jedes der Prädikate kann

entweder die Nicht-Existenz oder durch ein Beispiel die Existenz eines passenden Testfalls in endlicher Zeit nachgewiesen werden.

4.2 Rückführung auf Ausführungsfolgen

Wir werden einige der Kriterien aus Abschnitt 2.1.3 anhand der Ausführungssemantik aus Abschnitt 2.3.4 formal definieren. Die Testfälle haben in dieser Betrachtung die Form $t = (x, y)$ wobei x die Eingabe und y die erwartete Ausgabe ist.

Im nächsten Abschnitt werden wir zeigen, wie die Prädikate dieser Kriterien durch Verkettung von Spezifikationen ggf. modifizierter Programmflussgraphen dargestellt werden können.

Somit wird die Suche nach einer Eingabe x, die zusammen mit der erwarteten Ausgabe y das jeweilige Prädikat erfüllt, auf einen allgemeinen Fall reduziert, so dass nur ein allgemeiner Algorithmus zur Lösung der entsprechenden Aufgabe benötigt wird.

Definition 4.3: Sei P ein Programm, $PFG\,\langle N, E, r, e, \mathcal{B}, \mathcal{C}\rangle$ der Programmflussgraph von P.

Für jeden Testfall $t = (x, y)$ für P und jeden Knoten n gelte genau dann $Q_{n,PFG}\,(t)$, wenn es eine Ausführungsfolge $(S_1, \ldots, S_k)$ in PFG und ein z gibt, so dass $S_1 = \langle r, \alpha, x\rangle$, $S_k = \langle e, \beta, z\rangle$ und es ein $j \in \{1, \ldots, k-1\}$ gibt mit $S_j = \langle n, \alpha, x'\rangle$ und $S_{j+1} = \langle n, \beta, z'\rangle$.

Dann erfüllt eine Menge $\mathcal{T}$ von Testfällen für P genau dann das Kriterium *all-nodes*, wenn für jeden Knoten $n \in N$ ein Testfall $t \in \mathcal{T}$ existiert, so dass $Q_{n,PFG}\,(t)$ erfüllt ist.

Zu beachten ist hier, dass für die Abdeckung des Knotens unerheblich ist, ob die erwartete Ausgabe des Programms – hier y – mit der tatsächlichen Ausgabe – hier z – übereinstimmt. Das Programm muss jedoch für die Eingabe x mindestens eine vollständige Ausführungsfolge haben.

Bei der Formulierung des Kriteriums muss zwangsläufig die Frage nach der Determiniertheit des Programms gestellt werden. In einem konservativen Ansatz könnten wir auch fordern, dass nicht nur eine, sondern jede vollständige Ausführungsfolge mit der Eingabe x den Knoten n durchlaufen müsste.

Aufgrund dieses Unterschieds erfordern nicht-deterministische Programme ohnehin eine besondere Teststrategie. Für sie reicht es erst recht nicht aus, statische

Kriterien wie *all-nodes* oder *all-edges* zu betrachten, sondern es muss auch untersucht werden, inwiefern die getesteten Alternativen des nicht-deterministischen Verhaltens repräsentativ sind.

Zudem vereinfacht die liberale Formulierung wie in Definition 4.3 die theoretische Grundlage sowie die Implementierung des Selektionsverfahren für die Testeingaben. Wir verbleiben deshalb beim liberalen Ansatz.

Ähnlich können wir auch das Kriterium *all-edges* definieren.

Definition 4.4: Sei P ein Programm und $PFG\,\langle N, E, r, e, \mathcal{B}, \mathcal{C}\rangle$ der Programmflussgraph von P.

Für jeden Testfall t für P mit Eingabe x und jede Kante (v, w) gelte genau dann $Q_{(v,w),PFG}\,(t)$, wenn es eine Ausführungsfolge $(S_1, \ldots, S_k)$ in PFG und ein z gibt, so dass $S_1 = \langle r, \alpha, x\rangle$, $S_k = \langle e, \beta, z\rangle$ und es ein $j \in \{1, \ldots, k-2\}$ gibt mit $S_j = \langle v, \beta, x'\rangle$ und $S_{j+1} = \langle w, \alpha, x'\rangle$.

Dann erfüllt eine Menge $\mathcal{T}$ von Testfällen genau dann das Kriterium *all-edges*, wenn für jede Kante $(v, w) \in E$ ein Testfall $t \in \mathcal{T}$ existiert, so dass $Q_{(v,w),PFG}\,(t)$ erfüllt ist.

Die datenflussorientierten Kriterien basieren im Wesentlichen auf definitionsfreien Wegen bzgl. einer Variablen v von einer Definition v zu einer Nutzung u. Eine solche Nutzung kann entweder in einem Knoten (*c-use*) oder in einer Kantenbedingung vorkommen (*p-use*).

Wir definieren Prädikate für einen definitionsfreien Weg bzgl. einer Variablen v von einer Definition zu einem Knoten u bzw. zu einer Kante (u, w).

Definition 4.5: Sei P ein Programm und $PFG\,\langle N, E, r, e, \mathcal{B}, \mathcal{C}\rangle$ der Programmflussgraph von P.

Dann definieren wir zwei Prädikate $F_{v,d,u}$ und $F_{v,d,(u,w)}$ wie folgt.

Sei $t = (x, y)$ ein Testfall für P, $v \in V$ eine Variable und $d \in D\,(v)$ eine Definition von v.

Für jede globale c-use $u \in U_c\,(v)$ gilt genau dann $F_{v,d,u}\,(t)$, wenn es ein z und eine Ausführungsfolge $(S_1, \ldots, S_k)$ in PFG gibt mit $S_1 = \langle r, \alpha, x\rangle$ und $S_k = \langle e, \beta, z\rangle$, die eine bzgl. v definitionsfreie Teilfolge $(S_n, \ldots, S_m)$ enthält mit $S_n = \langle d, \beta, x'\rangle$ und $S_m = \langle u, \alpha, z'\rangle$.

Für jede p-use $(u, w) \in U_p\,(v)$ gilt genau dann $F_{v,d,(u,w)}$, wenn es ein z und eine Ausführungsfolge $(S_1, \ldots, S_k)$ in PFG gibt mit $S_1 = \langle r, \alpha, x\rangle$ und $S_k = \langle e, \beta, z\rangle$,

die eine bzgl. v definitionsfreie Teilfolge $(S_m, \ldots, S_n)$ mit $S_m = \langle d, \beta, x' \rangle$, $S_{n-1} = \langle u, \beta, y' \rangle$ und $S_n = \langle w, \alpha, y' \rangle$ enthält.

Auf dieser Grundlage definieren wir exemplarisch das Kriterium all-p-uses.

Definition 4.6: Sei P ein Programm und $PFG\,\langle N, E, r, e, \mathcal{B}, \mathcal{C} \rangle$ der Programmflussgraph von P.

Dann erfüllt eine Menge $\mathcal{T}$ von Testfällen genau dann das Kriterium *all-p-uses*, wenn für jede Variable v, jede Definition d von v und jeden von d erreichten *p-use* (u, w) von v ein Testfall $t \in \mathcal{T}$ existiert, so dass $F_{v,d,(u,w)}\,(t)$ erfüllt ist.

4.3 Vereinfachung von *all-edges*

Betrachten wir zunächst erneut das Kriterium *all-edges* aus Definition 4.4. Gefordert ist eine Ausführungsfolge, die vom Startknoten zum Knoten v, über die Kante (v, w) und dann vom Knoten w zum Endknoten führt. Es liegt nahe, diese Ausführungsfolge aus zwei Teilfolgen zusammenzusetzen und somit die Spezifikationen der Wege von r nach v und von w nach e zur Definition des Prädikats zu Hilfe zu nehmen.

Satz 4.2: *Sei P ein Programm, $PFG\,\langle N, E, r, e, \mathcal{B}, \mathcal{C} \rangle$ sein Programmflussgraph und (v, w) eine Kante in PFG.*

Dann ist für einen Testfall $t = (x, y)$ genau dann $Q_{(v,w),PFG}$ erfüllt, wenn gilt

$$\exists z : x \ \left(\mathcal{S}_{PFG[r,v]}; \mathcal{C}\,(v, w)\,; \mathcal{S}_{PFG[w,e]}\right)\ z$$

Beweis. Die rechte Seite der Behauptung ist äquivalent zu

$$\exists u, z : x\ \mathcal{S}_{PFG[r,v]}\ u \wedge u\ \mathcal{C}\,(v, w)\ \ u \wedge u\ \mathcal{S}_{PFG[w,e]}\ z$$

Nach Definition 2.18 sind die Teile der Konjunktion jeweils äquivalent zur Existenz einzelner Ausführungsfolgen. Diese können kombiniert werden zu einer Ausführungsfolge $(S_1, \ldots, S_k)$ mit $S_1 = \langle r, \alpha, x \rangle$ und $S_k = \langle e, \beta, z \rangle$, für die es ein $j \in \{2, \ldots, k-2\}$ gibt mit $S_j = \langle v, \beta, u \rangle$ und $S_{j+1} = \langle w, \alpha, u \rangle$.

Dies ist laut Definition 4.4 äquivalent zu $Q_{(v,w),PFG}\,(t)$. □

4.4 Vereinfachung von *all-nodes*

Die Prädikate $Q_{n,PFG}$ zum Kriterium *all-nodes* aus Definition 4.3 können nicht solchermaßen direkt auf Definition 2.18 zurückgeführt werden.

Während wir beim *all-edges*-Kriterium die Ausführungsfolge am Übergang von einem β-Zustand zu einem α-Zustand aufspalten konnten, müssen wir hier an einem Übergang von α- zu β-Zustand spalten, um unsere zwei Teilwege von r nach n und von n nach e analog darstellen zu können.

Wir erhalten dann zwei Teilfolgen, die von einem α- zu einem α-Zustand bzw. von einem β- zu einem β-Zustand verlaufen. Hierfür definieren wir eine weitere Relation auf Basis der Ausführungsfolgen.

Definition 4.7: Sei $PFG\,\langle N, E, r, e, \mathcal{B}, \mathcal{C}\rangle$ ein Programmflussgraph und seien a und b zwei Knoten in PFG.

Dann gilt für ein Paar (x, y) genau dann $x\ \mathcal{I}_{PFG[a,b]}\ y$, wenn es eine Ausführungsfolge $(S_1, \ldots, S_n)$ gibt mit $S_1 = \langle a, \beta, x\rangle$, $S_n = \langle b, \alpha, y\rangle$.

Diese Relation definiert die Beziehung zwischen der Ausgabe ihres Start- und der Eingabe ihres Endknotens. Aus diesem Grund bezeichnen wir sie als die *innere Spezifikation*.

Die Relation $\mathcal{S}_{PFG[a,b]}$ kann unter Einsatz der Konkatenation mit $\mathcal{B}(a)$ bzw. $\mathcal{B}(b)$ auf $\mathcal{I}_{PFG[a,b]}$ zurückgeführt werden (s. Satz 5.1).

Somit wird die folgende Formulierung von *all-nodes* möglich.

Satz 4.3: *Sei P ein Programm, $PFG\,\langle N, E, r, e, \mathcal{B}, \mathcal{C}\rangle$ sein Programmflussgraph und n ein Knoten in PFG.*

Sei außerdem $t = (x, y)$ ein Testfall für P.

Für $n \in \{r, e\}$ ist $Q_{n,PFG}(t)$ genau dann erfüllt, wenn es ein z gibt so, dass $x\ \mathcal{S}\,[r, e]\ z$ gilt.

Für $n \notin \{r, e\}$ ist $Q_{n,PFG}(t)$ genau dann erfüllt, wenn es ein z gibt so, dass $x\ \left(\mathcal{S}_{PFG[r,n]}\mathcal{I}_{PFG[n,e]}\mathcal{B}(e)\right)\ z$.

Beweis. Im ersten Fall ist der rechte Teil äquivalent zur Existenz einer Ausführungsfolge $(S_1, \ldots, S_k)$ mit $S_1 = \langle r, \alpha, x\rangle$ und $S_k = \langle e, \beta, z\rangle$. Es existiert ein $j \in \{1, \ldots, k-1\}$ mit $S_j = \langle n, \alpha, x'\rangle$ und $S_{j+1} = \langle n, \beta, z'\rangle$, nämlich $j = 1$ für $n = r$ und $j = k - 1$ für $n = e$.

Im zweiten Fall ist der rechte Teil äquivalent zur Existenz einer Ausführungsfolge $(S_,1 \ldots, S_k)$ mit $S_1 = \langle r, \alpha, x\rangle$ und $S_k = \langle e, \beta, z\rangle$, in der ein $j \in \{1, \ldots, k-1\}$ existiert mit $S_j = \langle n, \alpha, x'\rangle$ und $S_{j+1} = \langle n, \beta, z'\rangle$.

Die Behauptungen folgen mit Definition 4.3. □

4.5 Definitionsfreie Wege

Die bisher betrachteten Kriterien fordern, dass eine bestimmte Kante oder ein bestimmter Knoten im Weg enthalten sein muss. Diese Kriterien formulieren wir, indem wir die Ausführungsfolge in den Teil vor Durchlauf der Kante bzw. des Knotens und den Teil danach aufteilen und dies mit Hilfe der zur Verfügung stehenden Relationen allgemein darstellen.

Die datenflussorientierten Kriterien basieren auf definitionsfreien Wegen. Im Unterschied zu *all-nodes* und *all-edges* wird also gefordert, dass bestimmte Elemente des Programmflussgraphen in einem Teil des Weges nicht durchlaufen werden.

Sei P das zu testende Programm und $PFG\,\langle N, E, r, e, \mathcal{B}, \mathcal{C}\rangle$ sein Programmflussgraph. Betrachten wir in P die Programmvariable v, einen Knoten d mit einer Definition von v sowie eine Nutzung u von v.

Gesucht ist eine vollständige Ausführungsfolge in PFG, die eine definitionsfreie Teilfolge beginnend bei d und endend bei u enthält.

Um diese Anforderungen darstellen zu können benötigen wir eine neue Strategie. Zunächst teilen wir die Ausführungsfolge in drei Teile auf:

1. eine Teilfolge, die vom Programmstart bis d führt,
2. eine definitionsfreie Teilfolge von d bis u, und
3. eine Teilfolge von u zum Programmende.

Für den ersten und den letzten Teil sind keine besonderen Anforderungen gegeben, außer ihren jeweiligen Start- und Endpunkten. Sie können also wie üblich durch $\mathcal{S}_{PFG[r,d]}$ bzw. $\mathcal{S}_{PFG[u,e]}$ dargestellt werden.

Für den mittleren Teil müssen wir allerdings die Definitionsfreiheit sicherstellen. Dafür erweitern wir die Definition von $\mathcal{I}_{PFG[v,d]}$ durch die Angabe einer Menge A von Knoten, die gemieden werden sollen, und schreiben $\mathcal{I}_{PFG[v,d]\setminus A}$. Mit $A = \emptyset$ ist dies äquivalent zu $\mathcal{I}_{PFG[v,d]}$.

Für den mittleren Teil der gesuchten Ausführungsfolge wählen wir $A = D\,(v)$ und schließen damit die Knoten mit Definitionen von v aus.

Definition 4.8: Sei $PFG\,\langle N, E, r, e, \mathcal{B}, \mathcal{C}\rangle$ ein Programmflussgraph und seien a und b zwei Knoten in PFG. Sei $A \subseteq N$.

Dann gilt für ein Paar (x, y) genau dann $x\ \mathcal{I}_{PFG[a,b]\setminus A}\ y$, wenn es eine Ausführungsfolge $(S_1, \ldots, S_k)$ gibt mit $S_1 = \langle a, \beta, x\rangle$, $S_k = \langle b, \alpha, y\rangle$, so dass kein $j \in \{2, \ldots, k-1\}$ existiert mit $S_j = \langle n, s, z\rangle$ und $n \in A$.

Satz 4.4: *Sei P ein Programm, $PFG\,\langle N, E, r, e, \mathcal{B}, \mathcal{C}\rangle$ sein Programmflussgraph, v eine Variable in P und $d \in D_P(v)$.*

Sei $t = (x, y)$ ein Testfall für P mit der Eingabe x und der erwarteten Ausgabe y.

Dann gilt für jede globale c-use u von v genau dann $F_{v,d,u}(t)$, wenn

$$\exists z : x \ \left(\mathcal{S}_{PFG[r,d]}\mathcal{I}_{PFG[d,u]\setminus D(v)}\mathcal{S}_{PFG[u,e]}\right)\ z$$

Weiterhin gilt für jede p-use (u, w) *von v genau dann $F_{v,d,(u,w)}(t)$, wenn es ein z gibt mit*

- $d = r$ *und* $x \ \left(\mathcal{B}(r)\,\mathcal{C}(r, u)\,\mathcal{S}_{PFG[u,e]}\right)\ z$*, oder*
- $x \ \left(\mathcal{S}_{PFG[r,d]}\mathcal{I}_{PFG[d,u]\setminus D(v)}\mathcal{B}(u)\,\mathcal{C}(u, w)\,\mathcal{S}_{PFG[w,e]}\right)\ z$*.*

Beweis. Sei jeweils $(S_1, \ldots, S_k)$ mit $S_i = \langle v_i, s_i, x_i\rangle$ die Ausführungsfolge, zu deren Existenz die rechte Seite der Behauptung äquivalent ist.

Betrachten wir zunächst den Fall für die globalen c-uses. Nach Definition 4.8 gibt es ein m und ein n mit $1 < m < n < k$, $S_m = \langle d, \beta, x'\rangle$ und $S_n = \langle u, \alpha, y'\rangle$, so dass kein $j \in \{m, \ldots, n\}$ existiert mit $v_j \in D(v)$. Dies ist äquivalent zur Definition 4.5.

Betrachten wir nun die Fälle für die p-uses. Nach Definition 4.8 gibt es ein m und ein n mit $1 < m < n < k - 1$, $S_m = \langle d, \beta, x'\rangle$ und $S_n = \langle u, \alpha, y'\rangle$, so dass kein $j \in \{m, \ldots, n\}$ existiert mit $v_j \in D(v)$. Außerdem gilt $S_{n+1} = \langle w, \alpha, x_{n+1}\rangle$. Für $m = 2$ ist dies äquivalent zum ersten Fall, für $m > 2$ zum zweiten Fall. □

4.6 Allgemeinere Testvorgaben

Wie wir bereits in Abschnitt 3.1 erläutert haben, sind die üblichen strukturbasierten Testkriterien als Testvorgaben für die Qualität der Testfallmenge unzureichend.

Mit den bisher definierten Relationen können wir Testanforderungen anhand von zu durchlaufenen Knoten und Kanten sowie deren Reihenfolge festlegen. Wir können außerdem in einzelnen Abschnitten des zu durchlaufenen Weges bestimmte Knoten ausschließen.

Darüber hinausgehende Vorgaben können wir durch Manipulation des Programmflussgraphen definieren. So können wir zum Beispiel eine Schleife abrollen und damit mit jeder Iteration der neuen Schleife zwei Iterationen der alten Schleife ausführen. Entfernen wir die Kante, mit der die Schleife nach der ersten Iteration beendet werden könnte, so kann die Zahl der Iterationen nur gerade sein.

Entsprechend können wir einen Programmflussgraphen konstruieren, in dem nur eine ungerade Zahl von Iterationen möglich ist und haben somit zwei Klassen von Wegen definiert. Dies ist mit den bisher definierten Relationen nicht möglich.

Durch diese Konstruktionen können wir es außerdem vermeiden, weitere Relationen für spezielle Testkriterien einzuführen und damit das eigentliche Lösungsverfahren zu verkomplizieren.

Allgemein betrachtet können wir den Programmflussgraphen oder bereits das Ursprungsprogramm selbst durch Äquivalenztransformationen[Par90] zunächst in eine Form bringen, in der wir an einzelnen Stellen durch die Entfernung oder Einschränkung von Alternativen das gewünschte Testprädikat definieren können. Alternativ können wir unser Ziel auch durch direkte Manipulation des Programmflussgraphen erreichen.

Bezeichnen wir den Originalgraphen mit PFG und den transformierten Graphen mit PFG', so muss durch die Transformation die Äquivalenz dieser beiden Aussagen sichergestellt werden:

- Es existiert eine Ausführungsfolge $(S'_1, \ldots, S'_l)$ in PFG' mit $S'_1 = \langle r', \alpha, x\rangle$ und $S'_l = \langle e', \beta, y\rangle$.
- Es existiert eine Ausführungsfolge $(S_1, \ldots, S_k)$ in PFG mit $S_1 = \langle r, \alpha, x\rangle$ und $S_l = \langle e, \beta, y\rangle$, die die gewünschte Eigenschaft hat.

Der Beweis hierfür gliedert sich meist in drei Teile. Zunächst kann mit Hilfe von Lemma 4.2 aus $(S'_1, \ldots, S'_l)$ die Folge $(S_1, \ldots, S_k)$ konstruiert werden, wenn eine Knotensubstitution gemäß Definition 4.9 vorliegt. Dies ist immer dann der Fall, wenn bei der Transformation nur Knoten dupliziert und Knoten oder Kanten entfernt werden.

Dann ist zu zeigen, dass die transformierte Ausführungsfolge die Eigenschaften erfüllt. Damit ist eine Richtung der Äquivalenz bewiesen.

Die andere Richtung kann meist konstruktiv bewiesen werden, indem eine Ausführungsfolge in PFG mit den gewünschten Eigenschaften in eine entsprechende Ausführungsfolge in PFG' übertragen wird.

Wir zeigen dieses Vorgehen in Abschnitt 4.7 anhand der Konstruktion schleifenfreier Wege. Zunächst führen wir jedoch die zugrundeliegenden Lemmata ein.

Definition 4.9: Es seien $PFG\,\langle N, E, r, e, \mathcal{B}, \mathcal{C}\rangle$ und $PFG'\,\langle N', E', r', e', \mathcal{B}', \mathcal{C}'\rangle$ Programmflussgraphen.

Eine *Knotensubstitution von PFG' auf PFG* ist eine Funktion $\phi : N' \mapsto N$, für die gilt:

- Für jeden Knoten $n \in N'$ gilt $\mathcal{B}'(n) = \mathcal{B}(\phi(n))$.
- Für jede Kante $(v, w) \in E'$ gilt $(\phi(v), \phi(w)) \in E$.
- Für jede Kante $(v, w) \in E'$ gilt $\mathcal{C}'(v, w) = \mathcal{C}(\phi(v), \phi(w))$.

Lemma 4.1: *Seien $PFG\langle N, E, r, e, \mathcal{B}, \mathcal{C}\rangle$ und $PFG'\langle N', E', r', e', \mathcal{B}', \mathcal{C}'\rangle$ Kontrollflussgraphen und ϕ eine Knotensubstitution von PFG' auf PFG.*

Sei $(v'_1, \ldots, v'_k)$ ein Weg in PFG'.

Dann ist $(v_1, \ldots, v_k)$ mit $v_i := \phi(v'_i)$ für alle $i \in \{1, \ldots, k\}$ ein Weg in PFG.

*Wir bezeichnen $(v_1, \ldots, v_k)$ als die ϕ-*Transformierte *von $(v'_1, \ldots, v'_k)$.*

Beweis. Die v_i sind nach der Definition von ϕ Knoten in PFG.

Außerdem gilt $(v'_i, v'_{i+1}) \in E'$ für jedes $i \in \{1, \ldots, k-1\}$ und somit nach Definition 4.9 auch $(v_i, v_{i+1}) \in E$.

Somit ist $(v_1, \ldots, v_k)$ ein Weg in PFG. □

Lemma 4.2: *Seien $PFG\langle N, E, r, e, \mathcal{B}, \mathcal{C}\rangle$ und $PFG'\langle N', E', r', e', \mathcal{B}', \mathcal{C}'\rangle$ Kontrollflussgraphen und ϕ eine Knotensubstitution von PFG' auf PFG.*

Sei $F' = (S'_1, \ldots, S'_k)$ eine Ausführungsfolge in PFG' mit $S'_i = \langle v'_i, s_i, x_i\rangle$ für alle $i \in \{1, \ldots, k\}$.

Dann ist $F = (S_1, \ldots, S_k)$ mit $S_i := \langle v_i, s_i, x_i\rangle$ und $v_i = \phi(v'_i)$ für alle $i \in \{1, \ldots, k\}$ Ausführungsfolge in PFG.

*Wir bezeichnen F als die ϕ-*Transformierte *von F'.*

Beweis. Zunächst sind alle S_i auch Zustände in PFG, da $v_i \in N$ für alle $i \in \{1, \ldots, k\}$.

Wir zeigen nun für ein beliebiges $j \in \{1, \ldots, k-1\}$, dass $S_j \mapsto_{PFG} S_{j+1}$ gilt. Es gilt $S'_j \mapsto_{PFG'} S'_{j+1}$.

$s_j = \alpha$ Es gilt $v'_j = v'_{j+1}$ und $x_j \; \mathcal{B}'(v'_j) \; x_{j+1}$.

Somit gilt auch $v_j = v_j$ und $x_j \; \mathcal{B}(v_j) \; x_{j+1}$.

Es folgt direkt $S_j \mapsto_{PFG} S_{j+1}$.

$s_j = \beta$ Es gilt $(v'_j, v'_{j+1}) \in E'$, $x_j = x_{j+1}$ und $x_j \; \mathcal{C}'(v'_j, v'_{j+1}) \; x_j$.

Nach Definition 4.9 ist $(v_j, v_{j+1}) \in E$ und $\mathcal{C}(v_j, v_{j+1}) = \mathcal{C}'(v'_j, v'_{j+1})$.

Es folgt direkt $S_j \mapsto_{PFG} s_{j+1}$.

Somit ist $(S_1, \ldots, S_k)$ Ausführungsfolge in PFG. □

4.7 Schleifenfreie Wege

Das Kriterium *all-du-paths* aus Abschnitt 2.1.3 fordert, dass der Weg von der Definition zur Nutzung nicht nur definitionsfrei, sondern auch schleifenfrei ist.

Obwohl die Beschränkung auch durch Einführung einer speziellen Relation analog zu $\mathcal{I}_{PFG[a,b]\setminus A}$ darstellbar ist, wollen wir dieses Kriterium als Beispiel für die allgemeineren Testkriterien aus Abschnitt 4.6 verwenden. Wir konstruieren also aus dem Originalprogrammflussgraphen PFG einen weiteren Programmflussgraphen PFG', dessen Struktur die Beschränkung auf die schleifenfreien Wege repräsentiert. Die Konstruktion und den Beweis werden wir hier lediglich skizzieren.

Zur Veranschaulichung betrachten wir den Graphen in Abbildung 4.2a, der eine Schleife zwischen den Knoten 2 und 3 enthält. Entfernen wir die Kante $2 \to 3$ oder $3 \to 2$, so ist der Graph zwar schleifenfrei, aber ebenso sind die schleifenfreien Wege $(1, 2, 3, 4)$ bzw. $(1, 3, 2, 4)$ nicht mehr möglich.

Für die Konstruktion muss ein Startknoten a und ein Endknoten b gegeben sein. Diese sind bei der Formulierung der Testvorgaben bekannt, etwa die Knoten, in denen jeweils die betroffene Definition bzw. Nutzung der betrachteten Variablen enthalten ist.

Die Menge der schleifenfreien Wege von a nach b ist endlich und kann somit in endlicher Zeit bestimmt werden. Die Laufzeit für die Bestimmung ist $O\left(e^n\right)$, wobei n die Menge der Knoten im Graphen ist. Bei typischen von Compilern erzeugten Programmflussgraphen ist die Laufzeitkomplexität allerdings deutlich geringer.

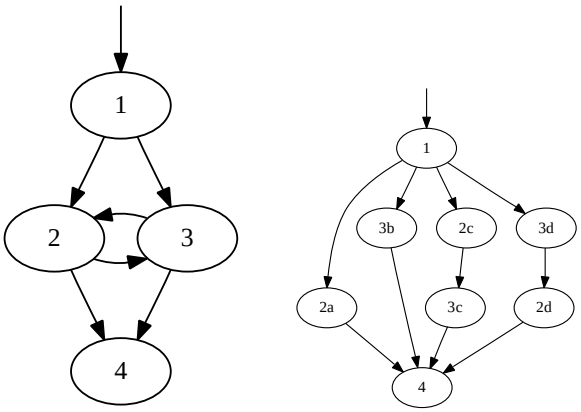
Im vorliegenden Beispiel sind $(1, 2, 4)$, $(1, 3, 4)$, $(1, 2, 3, 4)$ und $(1, 3, 2, 4)$ die schleifenfreien Wege von 1 nach 4.

Unser neu zu konstruierender Graph enthält zunächst nur die Knoten a und b. Für jeden der schleifenfreien Wege legen wir nun jeweils neue Kopien der durchlaufenen Knoten und Kanten an und fügen sie dem neuen Graphen hinzu. Lediglich a und b werden direkt übernommen, statt sie zu kopieren.

Für unser Beispiel entsteht ein Graph wie in Abbildung 4.2b illustriert.

Beim Kopieren der Knoten und Kanten werden auch Knoteninhalt und Kantenbedingungen kopiert. Aus dieser Konstruktion ergibt sich eine Knotensubstition gemäß Definition 4.9 und die Transformierten der Ausführungsfolgen aus PFG' sind nach Lemma 4.2 Ausführungsfolgen in PFG.

Aus der Konstruktion erhalten wir außerdem eine bijektive Abbildung der Wege in PFG' auf die schleifenfreien Wege in PFG, da die Knoten der Wege in PFG' bis



(a) PFG (b) PFG'

Abbildung 4.2: Konstruktion der Schleifenfreiheitsbedingung als Graph

auf a und b zueinander disjunkt sind. Dabei ist einem Weg in PFG' immer seine Transformierte zugeordnet. Damit können wir zeigen, dass die Transformierten der Wege in PFG' genau die schleifenfreien Wege von a nach b in PFG sind. Dies lässt sich auch auf die Ausführungsfolgen in PFG' und ihre Transformierten übertragen.

Diese Konstruktion ist weder bzgl. der Laufzeit- noch der Speicherkomplexität optimal. Liegen etwa nur natürliche Schleifen vor, so reicht es, die Rückwärtskanten aus dem Programmflussgraphen zu entfernen.

Selbst für den allgemeineren Fall müssten nur jene Teile des Graphen modifiziert werden, die überhaupt Schleifen enthalten. Ist etwa ein Knoten a von sich selbst aus erreichbar, so wird für jeden Nachfolger b von a, von dem aus a nicht erreicht werden kann, die obige Konstruktion angewendet. Die Kante von a nach b wird entfernt und die durch die Konstruktion erhaltenen zusätzlichen Knoten und Kanten werden mit dem Originalgraphen vereinigt.

5 Konstruktion von Wegen und Eingaben

In Kapitel 4 haben wir gezeigt, wie übliche und allgemeine Testkriterien als Kombination einzelner Prädikate dargestellt werden können.

Diese Prädikate haben wir auf Konjunktion von Aussagen der folgenden Grundformen zurückgeführt (jeweils für feste Knoten a und b und eine Knotenmenge A):

- $x \; \mathcal{S}_{PFG[a,b]} \; y$,
- $x \; \mathcal{I}_{PFG[a,b]} \; y$,
- $x \; \mathcal{I}_{PFG[a,b]\setminus A} \; y$,
- $x \; \mathcal{B}\,(a) \; y$, und
- $x \; \mathcal{C}\,(a,b) \; x$.

Die Konjunktionen sind dabei bezüglich der freien Variablen existenzquantifiziert.

Wir stellen nun die theoretischen Grundlagen für ein Verfahren vor, das diese Aussagensysteme auf Konjunktionen von Aussagen der Form $x \; \mathcal{B}\,(a) \; y$ und $x \; \mathcal{C}\,(a,b) \; x$ zurückführt.

Dabei wird effektiv ein Weg durch den Programmflussgraphen konstruiert, wie auch bei den bereits bekannten Verfahren (s. Abschnitt 3.3.2). Allerdings ist nicht der Weg von Interesse, sondern nur seine Eigenschaft, ausführbar zu sein.

Der Weg ist deshalb nicht Teil der Ausgabe des Verfahrens, sondern vielmehr eben jenes Aussagensystem, das für die Programmeingabe erfüllt sein muss, damit der Weg zur Ausführung kommen kann.

Da dennoch implizit ein Weg ausgewählt wird, ist die Ausgabe des Verfahrens nicht äquivalent zu seiner Eingabe, sondern stellt nur eine Untermenge der möglichen Lösungen dar.

Das Verfahren beinhaltet deshalb einen Suchprozess. Die Suche in einem Zweig des Suchbaums wird abgebrochen, sobald erkennbar ist, dass das bis dahin konstruierte Aussagensystem keine Lösung hat und die zugehörige Menge von Wegen somit nicht ausführbar sein kann.

Um diesen Suchabbruch zu vermeiden bzw. seine Auswirkungen zu verringern, wird eine Methode zur Wegvorhersage verwendet, die in ihrer Auswirkung der

Methode von Gotlieb et al[GBR98] ähnlich ist. Während jedoch bei Gotlieb et al die Information für die Vorhersage bereits in der Eingabe in Form der `while-if`-Struktur explizit vorliegen muss, leiten wir diese Information direkt aus dem Programmflussgraphen ab. Speziell nutzen wir hierfür die Postdominator- und Dominatorrelationen.

Wir beweisen zunächst, dass sich Aussagen der Form $x \ \mathcal{S}_{PFG[a,b]} \ y$ unmittelbar auf die Relationen $\mathcal{B}(a)$, $\mathcal{B}(b)$ sowie $\mathcal{I}_{PFG[a,b]}$ zurückführen lassen.

Damit reduzieren wir die Menge der zu betrachtenden Formen auf die Relationen $\mathcal{B}(n)$, $\mathcal{C}(v,w)$, $\mathcal{I}_{PFG[a,b]\setminus A}$ und $\mathcal{I}_{PFG[a,b]}$.

Dann zeigen wir, wie Aussagen der Form $x \ \mathcal{I}_{PFG[a,b]\setminus A} \ y$ und $x \ \mathcal{I}_{PFG[a,b]} \ y$ durch rekursive Suche auf $\mathcal{B}(n)$ und $\mathcal{C}(v,w)$ reduziert werden können.

Dies verdeutlicht, dass sich die Reihenfolge der Ableitung eines ausführbaren Weges und der zugehörigen Wegconstraints nicht an der Ausführungsreihenfolge des Programms orientieren muss.

Danach führen wir Vorhersageableitungen für Kontroll- und Datenfluss ein. Der Vorhersagecharakter dieser Ableitungen besteht darin, dass durch sie keine neuen Suchzweige eingeführt werden, sondern sie vielmehr zusätzliche allgemeingültige Information ableiten.

Wie wir in Abschnitt 7.2 zeigen, ist die rekursive Suche nicht für die Implementierung geeignet. Deshalb spezialisieren wir am Schluss dieses Kapitels noch die rekursive Suche zu einer linearen Rekursion, die Vorwärts- oder Rückwärtsschritte im Programmablauf repräsentiert.

5.1 Umformung der Spezifikation

Wir zeigen, dass $\mathcal{S}_{PFG[a,b]}$ für beliebige Knoten a und b auf $\mathcal{I}_{PFG[a,b]}$, $\mathcal{B}(a)$ und $\mathcal{B}(b)$ zurückzuführen ist.

Die Ableitung basiert auf der Erkenntnis, dass

- $\mathcal{S}_{PFG[a,b]}$ immer Ausführungsfolgen repräsentiert, die vor der Ausführung von beginnen a und nach der Ausführung von b enden, während
- $\mathcal{I}_{PFG[a,b]}$ immer Ausführungsfolgen repräsentiert, die nach der Ausführung von a beginnen und vor der Ausführung von b enden.

Um $\mathcal{I}_{PFG[a,b]}$ jeweils zu $\mathcal{S}_{PFG[a,b]}$ zu vervollständigen, müssen wir also lediglich die Spezifikationen der Knoten a bzw. b voranstellen bzw. hinten anhängen.

Der einzige Spezialfall ist $a = b$, da dort für $\mathcal{S}_{PFG[a,b]}$ Ausführungsfolgen der Länge 2 möglich sind und für diese Ausführungsfolgen insbesondere keine zu $\mathcal{I}_{PFG[a,b]}$ gehörige Teilsequenz existiert.

Satz 5.1: *Sei $PFG\,\langle N, E, r, e, \mathcal{B}, \mathcal{C}\rangle$ ein Programmflussgraph.*

Seien a und b Knoten in PFG und x und y beliebig.

Für $a \neq b$ gilt

$$x\ \mathcal{S}_{PFG[a,b]}\ y \Leftrightarrow x\ \left(\mathcal{B}\,(a)\,\mathcal{I}_{PFG[a,b]}\mathcal{B}\,(b)\right)\ y$$

Weiterhin gilt

$$x\ \mathcal{S}_{PFG[a,a]}\ y \Leftrightarrow x\ \left(\mathcal{B}\,(a) \cup \mathcal{B}\,(a)\,\mathcal{I}_{PFG[a,a]}\mathcal{B}\,(a)\right)\ y$$

Beweis. Wir betrachten beide Behauptungen gleichzeitig.

$\Rightarrow$ Es gilt $x\ \mathcal{S}_{PFG[a,b]}\ y$. Hiervon sind die linken Seiten der beiden Behauptungen jeweils ein Spezialfall, alternativ mit $a = b$ oder $a \neq b$.

Es folgt die Existenz einer Ausführungsfolge $(S_1, \ldots, S_k)$ mit $S_1 = \langle a, \alpha, x\rangle$ und $S_k = \langle b, \beta, y\rangle$.

Sei $k > 2$ und somit $(S_2, \ldots, S_{k-1})$ Ausführungsfolge entsprechend Definition 4.7. Sei $S_2 = \langle a, \beta, y'\rangle$ und $S_{k-1} = \langle b, \alpha, x'\rangle$. Dann gilt $y'\ \mathcal{I}_{PFG[a,b]}\ x'$.

Aufgrund von $S_1 \mapsto S_2$ und $S_{k-1} \mapsto S_k$ gilt außerdem jeweils $x\ \mathcal{B}\,(a)\ y'$ und $x'\ \mathcal{B}\,(b)\ y$. Für $a \neq b$ ist $k > 2$ immer erfüllt und somit gilt die rechte Seite der ersten Behauptung.

Für die zweite Behauptung verbleibt noch der Fall $k = 2$. Hieraus folgt explizit $a = b$.

Dann gilt $S_1 \mapsto S_k$ und somit $x\ \mathcal{B}\,(a)\ y$. Somit ist auch die rechte Seite der zweiten Behauptung für jedes k erfüllt.

$\Leftarrow$ Wir betrachten zunächst den Fall $x\ \left(\mathcal{B}\,(a)\,\mathcal{I}_{PFG[a,b]}\mathcal{B}\,(b)\right)\ y$. Dies ist für $a \neq b$ die rechte Seite der ersten Behauptung und für $a = b$ ein Fall der rechten Seite der zweiten Behauptung.

Es existiert somit eine Ausführungsfolge $(S_1, \ldots, S_k)$ mit $S_1 = \langle a, \beta, y'\rangle$ und $S_k = \langle b, \alpha, x'\rangle$, so dass $x\ \mathcal{B}\,(a)\ y'$ und $x'\ \mathcal{B}\,(b)\ y$.

Somit ist auch $(S_0, \ldots, S_{k+1})$ Ausführungsfolge in PFG mit $S_0 = \langle a, \alpha, x\rangle$ und $S_{k+1} = \langle b, \beta, y\rangle$ und somit gilt $x\ \mathcal{S}_{PFG[a,b]}\ y$ für beide Behauptungen.

Es verbleibt $x\ \mathcal{B}\,(a)\ y$ als zweiter Fall der zweiten Behauptung. Es folgt, dass (S_1, S_2) mit $S_1 = \langle a, \alpha, x\rangle$ und $S_2 = \langle a, \beta, y\rangle$ Ausführungsfolge in PFG ist und somit ebenfalls $x\ \mathcal{S}_{PFG[a,b]}\ y$ mit $a = b$ gilt.

□

5.2 Rekursiver Ansatz

Wir haben somit nun noch $\mathcal{I}_{PFG[a,b]\setminus A}$ sinnvoll auf $\mathcal{B}(n)$ und $\mathcal{C}(v,w)$ zurückzuführen.

Wir stellen hierfür einen rekursiven Ansatz vor, der die Grundlage für einen großen Teil der weiteren Theorie bildet. Der Ansatz basiert auf der Erkenntnis, dass jeder Weg von a nach b an jedem Zwischenknoten n in zwei Wege von a nach n bzw. von n nach b aufgespalten werden kann.

Dies übertragen wir auf die Ausführungsfolgen für $\mathcal{I}_{PFG[a,b]\setminus A}$. Diese Ausführungsfolgen führen immer von einem β- zu einem α-Zustand.

Bei einer Länge gleich 2 handelt es sich um den Übergang von einem Knoten a zu seinem Nachfolger b, dem im Wegconstraint die Kantenbedingung $\mathcal{C}(a,b)$ entspricht.

Bei einer Länge größer als 2 können wir jede solche Ausführungsfolge zwischen einem α- und einem β-Zustand aufteilen und erhalten zwei Teilsequenzen, die wiederum Ausführungsfolgen zu einer inneren Spezifikation sind. Verbindungsglied ist in diesem Fall die Knotenspezifikation $\mathcal{B}(n)$.

Als Teilungsknoten kommen nur jene Knoten in Frage, die auf einem Weg von a nach b liegen, der nicht über ausgeschlossene Knoten führt. Dabei ist es ausreichend, nur eine Untermenge dieser Knoten zu betrachten, solange sichergestellt ist, dass jeder passende Weg von a nach b über mindestens einen dieser Kandidaten verläuft.

Als Wahlmenge kommen etwa die direkten Nachfolger von a in Frage, da jeder Weg von a nach b über mindestens einen der Nachfolger von a verläuft (s. Satz 5.6). Analog reicht ein einziger Spaltungsknoten als Kandidat aus, sofern jeder Weg von a nach b über diesen einzelnen Knoten führt (s. Satz 5.4).

Die Kandidatenmenge wird in Satz 5.2 durch die Menge M repräsentiert, wovon durch den Ausschluss der Knoten in A nur noch die Knoten der Teilmenge $M \setminus A$ tatsächlich anwendbar sind.

Satz 5.2: *Sei $PFG\langle N, E, r, e, \mathcal{B}, \mathcal{C}\rangle$ ein Programmflussgraph und seien a und b zwei Knoten in PFG. Seien weiterhin A und M Mengen von Knoten.*

Für jeden Weg $(v_1, \ldots, v_l)$ von a nach b mit $l > 2$ gebe es mindestens ein $i \in \{2, \ldots, l-1\}$ mit $v_i \in M \cup A$.

Dann gilt für ein Paar (x, y) genau dann $x\ \mathcal{I}_{PFG[a,b]\setminus A}\ y$, wenn

- *$a \to b$ und $x\ \mathcal{C}(a,b)\ y$, oder*

- *es ein* $n \in M \setminus A$ *gibt mit*
  - $a \rightarrow^+ n$,
  - $n \rightarrow^+ b$, *und*
  - $x \ \left(\mathcal{I}_{PFG[a,n]\setminus A} \mathcal{B}(n) \mathcal{I}_{PFG[n,b]\setminus A}\right) \ y$.

Den ersten Fall bezeichnen wir als *Basisfall*, den zweiten als *Rekursionsfall*.

Beweis. $\Rightarrow$ Es gelte $x \ \mathcal{I}_{PFG[a,b]\setminus A} \ y$.

Somit existiert eine Ausführungsfolge $(S_1, \ldots, S_k)$ mit $S_1 = \langle a, \beta, x\rangle$ und $S_k = \langle b, \alpha, y\rangle$ und es gibt kein $j \in \{2, \ldots, k-1\}$ mit $S_j = \langle m, s, z\rangle$ und $m \in A$.

Wir unterscheiden nach den Fällen $k = 2$ und $k > 2$.

$k = 2$ Aus $S_1 \mapsto S_2$ folgt $a \rightarrow b$ und $x \ \mathcal{C}(a, b) \ y$.

$k > 2$ Nach Definition 2.15 gibt es einen Weg $(v_1, \ldots, v_{l+1})$ mit $v_{l+1} = b$, $k = 2l$ und $S_{2j-1} = \langle v_j, \beta, x_j\rangle$ für alle $j \in \{1, \ldots, l\}$.

Wir wählen ein $i \in \{2, \ldots, l-1\}$ so, dass $v_i \in M \setminus A$. Nach der Voraussetzung existiert ein solches i, denn $(v_1, \ldots, v_{l+1})$ ist ein Weg von a nach b.

Es gilt $S_{2i} = \langle v_i, \alpha, x'\rangle$ und $S_{2i+1} = \langle v_i, \beta, y'\rangle$. Außerdem sind die Teilfolgen $(S_1, \ldots, S_{2i})$ und $(S_{2i+1}, \ldots, S_k)$ Ausführungsfolgen in PFG.

Damit gelten $x \ \mathcal{I}_{PFG[a,n]\setminus A} \ x'$, $x' \ \mathcal{B}(n) \ y'$ und $y' \ \mathcal{I}_{PFG[n,b]\setminus A} \ y$ für ein $n \in M \setminus A$. Ebenso gilt $a \rightarrow^+ n$ und $n \rightarrow +b$.

$\Leftarrow$ Wir unterscheiden nach den beiden Fällen.

Fall 1 Sei $S_1 = \langle a, \beta, x\rangle$ und $S_2 = \langle b, \alpha, y\rangle$. Aus der Voraussetzung folgt, dass (S_1, S_2) Ausführungsfolge in PFG ist. Mit $k = 2$ folgt $x \ \mathcal{I}_{PFG[a,b]\setminus A} \ y$.

Fall 2 Nach Annahme existiert eine Ausführungsfolge $(S_1, \ldots, S_k)$ mit einem $i \in \{2, \ldots, k-1\}$, so dass $S_1 = \langle a, \beta, x\rangle$, $S_i = \langle n, \alpha, x'\rangle$, $S_{i+1} = \langle n, \beta, y'\rangle$ und $S_k = \langle b, \alpha, y\rangle$ gelten.

Nach Definition 4.8 gibt es kein $j \in \{2, \ldots, i-1\} \cup \{i+2, \ldots, k-1\}$ mit $S_j = \langle v, s, z\rangle$ für ein $v \in A$.

Somit ist $(S_1, \ldots, S_k)$ Ausführungsfolge in PFG und aufgrund $n \notin A$ existiert auch kein $j \in \{2, \ldots, k-1\}$ mit $S_j = \langle v, s, z\rangle$ für ein $v \in A$.

□

Die Konstruktion von $\mathcal{I}_{PFG[a,b]}$ folgt als Spezialfall aus Satz 5.2 mit $A = \emptyset$.

Satz 5.3: *Sei* $PFG\,\langle N, E, r, e, \mathcal{B}, \mathcal{C}\rangle$ *ein Programmflussgraph und seien* a *und* b *zwei Knoten in* PFG. *Sei* M *eine Menge von Knoten.*

Für jeden Weg $(v_1, \ldots, v_l)$ *von* a *nach* b *mit* $l > 2$ *gebe es mindestens ein* $i \in \{2, \ldots, l-1\}$ *mit* $v_i \in M$*.*

Dann gilt für ein Paar (x, y) *genau dann* $x \; \mathcal{I}_{PFG[a,b]} \; y$*, wenn*

- $a \to b$ *und* $x \; \mathcal{C}\,(a, b) \; y$*, oder*
- *es ein* $n \in M$ *gibt mit* $a \to^+ n$*,* $n \to^+ b$ *und* $x \; \left(\mathcal{I}_{PFG[a,n]} \mathcal{B}\,(n)\, \mathcal{I}_{PFG[n,b]}\right) \; y$*.*

Beweis. Direkt aus Satz 5.2 mit $A = \emptyset$. □

5.3 Wegvorhersage

Aufgrund der Informationen aus den Dominator- und Postdominator-Relationen (s. Abschnitt 2.3.1) ist es möglich, den Verlauf von Programmabläufen teilweise vorherzusagen.

Der erste rekursive Ansatz (Abschnitt 5.2) sieht vor, dass ein Programmablauf von einem Knoten a zu einem Knoten b an einem Zwischenknoten n aufgeteilt werden kann, wenn $a \to^+ n$ und $n \to^+ b$ erfüllt sind. Unter diesen Zwischenknoten sind auch jene Knoten, für die $a \; dom \; n$ und $n \; dom \; b$ gilt, da nach Satz 2.1 nicht nur ein, sondern jeder Weg von a nach b über diese Knoten n läuft. Somit findet bei Teilung an einem solchen Knoten eine teilweise Vorhersage des Ablaufs statt. Insbesondere ist eine solche Teilung eine Äquivalenztransformation, erfordert also keine Suchschritte.

Satz 5.4: *Sei* $PFG\,\langle N, E, r, e, \mathcal{B}, \mathcal{C}\rangle$ *ein Programmflussgraph und sei* $A \subseteq N$*. Seien weiterhin* a *und* b *Knoten in* PFG*.*

Es sei n *ein Knoten in* PFG*, so dass es für jeden Weg* $(v_1, \ldots, v_k)$ *von* a *nach* b *ein* $j \in \{2, \ldots, k-1\}$ *gibt mit* $v_j = n$*.*

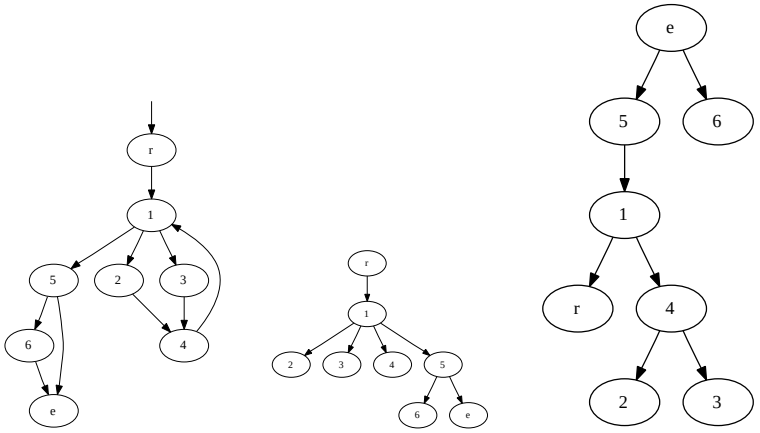
Ist $n \notin A$*, so gilt*

$$\forall x, y : x \; \mathcal{I}_{PFG[a,b]\setminus A} \; y \Leftrightarrow x \; \left(\mathcal{I}_{PFG[a,n]\setminus A}; \mathcal{B}\,(n)\,; \mathcal{I}_{PFG[n,b]\setminus A}\right) \; y$$

Ist $n \in A$*, so gilt*

$$\forall x, y : \neg \left(x \; \mathcal{I}_{PFG[a,b]\setminus A} \; y\right)$$

Beweis. Die Behauptungen folgen mit $M = \{n\}$ direkt aus Satz 5.2. Da jeder Weg von a nach b nach Voraussetzung länger als 1 sein muss, gilt $a \not\to b$. Beim zweiten Teil der Behauptung gilt außerdem $M \subseteq A$. □



(a) Kontrollflussgraph (b) Dominatorbaum (c) Postdominatorbaum

Abbildung 5.1: Kontrollflussgraph, Dominator- und Postdominatorbaum

Korollar 5.1: *Insbesondere gilt*

$$\forall x, y : x \; \mathcal{I}_{PFG[a,b]\setminus A} \; y \Leftrightarrow x \; \left(\mathcal{I}_{PFG[a,n]\setminus A}; \mathcal{B}\left(n\right); \mathcal{I}_{PFG[n,b]\setminus A}\right) \; y$$

wenn $n \notin A$ *sowie* $a \; dom \; n$ *und* $n \; dom \; b$ *oder* $b \; postdom \; n$ *und* $n \; postdom \; a$*.*

Beweis. Direkt mit den Sätzen 2.1 bzw. 2.2. □

Zu beachten ist, dass zusätzlich zu den in Korollar 5.1 ausgewählten Knoten n weitere Knoten existieren können, die auf jedem Weg von a nach b durchlaufen werden.

Außerdem ist die Transformation nach Korollar 5.1 nicht äquivalent zu einer, die entweder nur die Dominator- oder nur die Postdominatorrelation verwendet.

Dies können wir am Beispiel aus Abbildung 5.1 erkennen. Für $x \; \mathcal{I}_{PFG[2,e]} \; y$ sind nach der Postdominatorrelation Teilungen bei den Knoten 4, 1 und 5 möglich, nach der Dominatorrelation jedoch keine. Umgekehrt sind für $x \; \mathcal{I}_{PFG[1,e]} \; y$ bei der Dominatorrelation Teilungen beim Knoten 5 möglich, nach der Postdominatorrelation jedoch keine.

5.4 Datenflussvorhersage

Wir beschreiben nun eine Erweiterung, bei der statische Informationen über allgemeine Beziehungen zwischen Programmzuständen ausgenutzt werden können,

um Teilziele über noch unbekannte Partien des Programmablaufs hinweg zu verbinden.

Betrachten wir zur Motivation ein Zwischenergebnis für $u \; \mathcal{I}_{PFG[a,d]} \; z$ nach zwei Teilungen:

$$u \, \mathcal{I}_{PFG[a,b]} \; v \wedge v \; \mathcal{B}\left(b\right) \; w \wedge w \, \mathcal{I}_{PFG[b,c]} \; x \wedge x \; \mathcal{B}\left(c\right) \; y \wedge y \, \mathcal{I}_{PFG[c,d]} \; z$$

Die Zustände v und y sind wohl durch $v \; \left(\mathcal{B}\left(b\right); \mathcal{I}_{PFG[b,c]}; \mathcal{B}\left(c\right)\right) \; y$ verbunden, aber insbesondere der Teil $\mathcal{I}_{PFG[b,c]}$ ist uns noch nicht bekannt.

Es gibt aber in den meisten Programmen Knotenpaare, zwischen denen bestimmte Aspekte des Speicherzustands unverändert bleiben, etwa weil alle Wege zwischen den beiden Knoten definitionsfrei bzgl. einer bestimmten Variablen sind.

Somit können wir zumindest Teile der Beziehung zwischen v und y auch ohne vollständige Kenntnis von $\mathcal{I}_{PFG[b,c]}$ beschreiben. Die so gewonnene Information können wir nutzen, um einige mögliche Rechenschritte für die weitere Lösung auszuschließen und somit den Suchraum einzuschränken.

Betrachten wir als Beispiel eine bestimmte Variable v in einem Programm. Wir könnten nun interessiert daran sein, ob der Wert von v vor der Ausführung unseres Programms in jedem Fall kleiner ist als der Wert von v nach der Ausführung des Programms.

Sind x und y Speicherzustände dieses Programms, so können wir die Tatsache, dass der Wert von v in x kleiner oder gleich dem Wert von v in y ist durch $x \preceq y$ ausdrücken. Dabei ist $\preceq$ also eine Relation zwischen Speicherzuständen.

Wird der Wert von v etwa in jedem Ausführungsschritt erhöht, so wird $x \preceq y$ für alle aufeinanderfolgenden Speicherzustände x und y gelten. Daraus können wir folgern, dass $x \preceq y$ auch immer dann gilt, wenn x Eingabe- und y Ausgabespeicherzustand unseres Programms ist.

Wir leiten nun ein Kriterium her, mit dem wir Vorhersagen dieser Art für alle Ausführungsfolgen von einem Zustand $\langle a, \beta, x\rangle$ zu einem Zustand $\langle b, \alpha, y\rangle$ machen können.

Satz 5.5: *Sei $PFG\,\langle N, E, r, e, \mathcal{B}, \mathcal{C}\rangle$ ein Programmflussgraph und $A \subseteq N$. Sei $\preceq$ eine transitive und reflexive Relation, und M eine Menge von Knoten in PFG, so dass $\mathcal{B}\left(n\right) \subseteq \preceq$ für jeden Knoten n in M gilt.*

Seien weiterhin a und b zwei Knoten in PFG.

Für jeden Weg $(v_1, \ldots, v_l)$ von a nach b gebe es entweder mindestens ein $j \in \{2, \ldots, l-1\}$ mit $v_j \in A$ oder es gelte $v_j \in M$ für jedes $j \in \{2, \ldots, l-1\}$.

Dann gilt $\mathcal{I}_{CFG[a,b]\setminus A} \subseteq \preceq$.

Wir können stattdessen auch schreiben

$$\forall x, y : x \; \mathcal{I}_{CFG[a,b]\setminus A} \; y \Rightarrow x \preceq y$$

Beweis. Es gelte $x \; \mathcal{I}_{CFG[a,b]\setminus A} \; y$.

Somit existiert eine Ausführungsfolge $(S_1, \ldots, S_k)$ in PFG mit $S_1 = \langle a, \beta, x\rangle$ und $S_k = \langle b, \alpha, y\rangle$, so dass es kein $j \in \{2, \ldots, k-1\}$ gibt mit $S_j = \langle v, s, x'\rangle$ und $s \in A$.

Wir beweisen die Behauptung durch Induktion über k. Nach Voraussetzung ist k gerade.

$k = 2$ Es gilt $S_1 \mapsto S_k$ und somit $x \; \mathcal{C}\,(a, b) \; y$. Da $\mathcal{C}\,(a, b)$ eine Bedingung nach Definition 2.3 ist, gilt auch $x = y$ und somit mit der Reflexivität von $\preceq$ auch $x \preceq y$.

$k > 2$ Aus Definition 2.15 lässt sich folgern, dass es einen Weg $(v_1, \ldots, v_{l+1})$ gibt mit $v_{l+1} = b$, $k = 2l$ und $S_{2j-1} = \langle v_j, \beta, x_j\rangle$ für alle $j \in \{1, \ldots, l\}$.

Wir wählen ein $i \in \{2, \ldots, l-1\}$ so, dass $v_i \in M \setminus A$. Nach der Voraussetzung existiert ein solches i, denn $(v_1, \ldots, v_{l+1})$ ist ein Weg von a nach b.

Es gilt $S_{2i} = \langle v_i, \alpha, x'\rangle$ und $S_{2i+1} = \langle v_i, \beta, y'\rangle$. Außerdem sind die Teilfolgen $(S_1, \ldots, S_{2i})$ und $(S_{2i+1}, \ldots, S_k)$ Ausführungsfolgen in PFG.

Damit gelten $x \; \mathcal{I}_{PFG[a,n]\setminus A} \; x'$, $x' \; \mathcal{B}\,(n) \; y'$ und $y' \; \mathcal{I}_{PFG[n,b]\setminus A} \; y$ für ein $n \in M \setminus A$.

Nach der Induktionshypothese gilt also $x \preceq x'$ und $y' \preceq y$. Aus $n \in M$ folgt außerdem $x' \preceq y'$. Mit der Transitivität von $\preceq$ schließen wir also $x \preceq y$.

□

5.4.1 Variablenvorhersage

Bisher haben wir Eingaben und Ausgaben als abstrakte Objekte betrachtet, ohne deren Struktur zu definieren. Üblicherweise handelt es sich jedoch um Tupel von Variablenwerten oder Speicherzustände, die in einzelne Speicherstellen zerlegt werden können.

Definition 5.1: Sei P ein Programm und v eine Variable in P. Sei x ein Speicherzustand in einer Ausführung von P.

Dann ist $x.v$ der *Wert von* v *in* x.

So ergibt sich für jede dieser Speicherstellen eine natürliche Äquivalenzrelation, in der zwei Zustände äquivalent sind, wenn die Werte in den jeweiligen Speicherstellen in den zwei Zuständen gleich sind.

Diese Information können wir zum Beispiel durch Datenflussanalyse oder abstrakte Interpretation[ASU97] erhalten.

Definition 5.2: Sei P ein Programm und v eine Variable in P. Seien x und y Speicherzustände in einer Ausführung von P.

Wir bezeichnen x und y als *äquivalent bezüglich* v genau dann, wenn $x.v = y.v$ gilt, und schreiben $x \equiv_v y$.

Die Eigenschaften der Identitätsrelation übertragen sich auf $\equiv_v$ und somit ist $\equiv_v$ insbesondere transitiv und reflexiv.

Korollar 5.2 (zu Satz 5.5)**:** *Sei P ein Programm und $PFG\ \langle N, E, r, e, \mathcal{B}, \mathcal{C}\rangle$ ein Programmflussgraph für P und A eine Menge von Knoten.*

Sei v eine Variable in P und seien a und b zwei Knoten in CFG.

Für jeden Weg $(v_1, \ldots, v_l)$ von a nach b mit $v_i \notin A$ für alle $i \in \{2, \ldots, l-1\}$ gelte $x\ \mathcal{B}\,(v_i)\ y \Rightarrow x \equiv_v y$ für alle $i \in \{2, \ldots, l-1\}$.

Dann gilt:

$$\forall x, y : x\ \mathcal{I}_{CFG[a,b]\setminus A}\ y \Rightarrow x \equiv_v y$$

Beweis. Direkt mit der Transitivität und Reflexivität von $\equiv_v$ und Satz 5.5. □

Hiermit ist eine Vorhersage des Datenflusses über bisher unbekannte Abschnitte der Ausführungsfolge möglich.

5.4.2 Weitergehende Vorhersagen

In Listing 5.1 ist ein Algorithmus für eine lineare Suche nach einem Element `v` in einem Feld `a` der Länge `n` dargestellt. Die Tatsache, ob das Element im Feld enthalten ist, wird in der booleschen Variablen `found` festgehalten. Ihr Wert ist genau dann `true`, wenn das Element gefunden wurde.

Der zugehörige Programmflussgraph ist in Abbildung 5.2 zu sehen. Von Interesse ist hier der Aspekt, dass Knoten 6 nur dann erreicht werden kann, wenn vorher Knoten 3 durchlaufen wurde, da ansonsten `found<>true` gilt.

```
1 found:=false
2 for i:= 1 to n do
3         if a(i)=v then
4                 found:=true
5         end
6 end
7 if found=true then
8         ...
9 endif
```

Listing 5.1: Lineare Suche

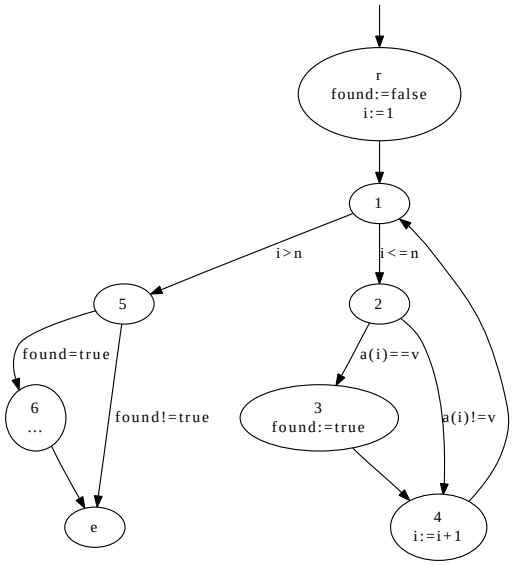


Abbildung 5.2: Programmflussgraph Lineare Suche

Wir können hier die Relation $\preceq$ zwischen Speicherzuständen definieren. Es gilt $x \preceq y$ genau dann, wenn $x.found \Rightarrow y.found$.

Im Programmflussgraphen in Abbildung 5.2 gilt $\forall x, y : x \; \mathcal{B}(n) \; y \Rightarrow x \preceq y$ für alle Knoten n außer r. Der Knoten r ist deshalb ausgeschlossen, weil $x.found$ vor Ausführung des Startknoten noch nicht eindeutig definiert ist.

Somit gilt für jedes b mit $3 \rightarrow^+ b$ und jede Knotenmenge A:

$$\forall x, y : x \; \mathcal{I}_{CFG[3,b]\setminus A} \; y \Rightarrow x \preceq y$$

Da für $x' \; \mathcal{B}(3) \; x$ auch $x.found = true$ gilt, können wir somit ableiten, dass `found` nach erstmaligem Erreichen des Knoten 3 in jedem Fall bis zum Ende der Schleife `true` bleibt. Dies ermöglicht die weitere Bearbeitung von Abläufen, die auf der Information basieren, ob das Element im Feld gefunden wurde, ohne die Schleife vollständig weiterzubearbeiten.

Üblicherweise wird besonders die Vorhersage der Wahrheitswerte der im Programm vorkommenden booleschen Ausdrücke von Interesse sein.

Im allgemeinen ist dies jedoch ein typisches Erfüllbarkeitsproblem der Constraintprogrammierung. Leider ist die Vollständigkeit und Korrektheit bzgl. Erfüllbarkeit oft nur für einfache Teilsprachen gegeben[FA03].

Es ist jedoch anzunehmen, dass mit abstrakter Interpretation und Datenflussanalyse viele, wenn auch nicht alle Fälle erkannt werden können, in denen eine Vorhersage möglich ist.

5.5 Iterativer Ansatz

Statt der rekursiven Ableitung können wir auch ein iteratives Verfahren verwenden. Ein solches iteratives Verfahren hat mehrere Vorteile gegenüber einem rekursiven Verfahren, insbesondere bezüglich der Terminierungseigenschaften. Details hierzu werden in Kapitel 7 diskutiert.

Satz 5.6: *Sei $PFG\,\langle N, E, r, e, \mathcal{B}, \mathcal{C}\rangle$ ein Programmflussgraph und seien a und b zwei Knoten in N mit $a \neq b$. Sei außerdem A eine Menge von Knoten.*

Dann gilt für alle Paare (x, y) genau dann $x \; \mathcal{I}_{PFG[a,b]\setminus A} \; y$, wenn

- $a \rightarrow b$ *und* $x \; \mathcal{C}(a, b) \; y$ *gilt, oder*
- *es ein* $n \in N \setminus A$ *gibt, so dass* $x \; \left(\mathcal{C}(a, n)\,\mathcal{B}(n)\,\mathcal{I}_{PFG[n,b]\setminus A}\right) \; y$ *sowie* $a \rightarrow n$ *und* $n \rightarrow^+ b$ *gelten.*

Wir bezeichnen den ersten Fall erneut als Basisfall, den zweiten Fall als *Vorwärtsschritt*.

Beweis. Mit Satz 5.2 und M der Menge der Nachfolger von a gilt die linke Seite der Behauptung genau dann, wenn

- $a \to b$ und $x \, \mathcal{C}\,(a, b) \; y$ gilt, oder
- es ein $m \in M \setminus A$ gibt, so dass $x \; \left(\mathcal{I}_{PFG[a,m]\setminus A} \mathcal{B}\,(m)\, \mathcal{I}_{PFG[m,b]\setminus A}\right) \; y$ sowie $a \to^+ m$ und $m \to^+ b$ gelten.

Wir zeigen die Äquivalenz zur rechten Seite der Behauptung. Für den ersten Fall ist nichts mehr zu beweisen.

Betrachten wir also die Äquivalenz im zweiten Fall.

$\Leftarrow$ Es gilt nach Annahme $x \, \mathcal{C}\,(a, n) \; x$. Aufgrund von $a \to n$ folgt $x \, \mathcal{I}_{PFG[a,n]\setminus A} \; x$ und damit auch die Behauptung.

$\Rightarrow$ Nach Voraussetzung gibt es eine Ausführungsfolge $(S_1, \ldots, S_k)$ mit $S_1 = \langle a, \beta, x\rangle$ und $S_k = \langle m, \alpha, x'\rangle$. Mit $S_1 \mapsto S_2$ folgt $S_2 = \langle n, \alpha, x\rangle$ mit $a \to n$ und $x \, \mathcal{C}\,(a, n) \; x$.

Nach Definition 4.8 ist außerdem $n \notin A$.

□

Es ist allerdings auch möglich, die innere Spezifikation entgegen der normalen Ausführungsrichtung zu entwickeln, wie Satz 5.7 zeigt.

Satz 5.7: *Sei $PFG\,\langle N, E, r, e, \mathcal{B}, \mathcal{C}\rangle$ ein Programmflussgraph und seien a und b zwei Knoten in N mit $a \neq b$. Sei außerdem $A \subseteq N$ eine Menge von Knoten.*

Dann gilt für alle Paare (x, y) genau dann $x \, \mathcal{I}_{PFG[a,b]\setminus A} \; y$, wenn

- $a \to b$ *und* $x \, \mathcal{C}\,(a, b) \; y$ *gilt, oder*
- *es ein* $n \in N \setminus A$ *gibt, so dass* $x \; \left(\mathcal{I}_{PFG[a,n]\setminus A} \mathcal{B}\,(n)\, \mathcal{C}\,(n, b)\right) \; y$ *sowie* $a \to^+ n$ *und* $n \to b$ *gelten.*

Beweis. Analog zu Satz 5.6. □

Hier wird der erste Fall erneut als Basisfall, der zweite als *Rückwärtsschritt* bezeichnet.

6 Implementierung

Zur Evaluierung des Verfahrens wurde ein Testfallgenerator für eine Untermenge der Programmiersprache JAVA[GJSB05] entwickelt. Die Sprachuntermenge beschränkt sich auf imperative Anteile von JAVA:

- Statische Methoden: `public static int ggt(...) {...}`
- Basistypen: `boolean`, `byte`, `char`, `short`, `int`, `long`
- Einfache arithmetische Ausdrücke: `+`, `-`, `*`, `/`
- Konditionalausdrücke: `(...?...:...)`
- Direkte Zuweisung: `=`, `+=`, `-=`, `*=`, `/=`
- Vergleichsoperatoren: `==`, `!=`, `>`, `>=`, `<=`, `<`
- Logische Verknüpfungen: `&&`, `&`, `||`, `|`, `!`
- Kontrollkonstrukte: `if`/`else`, `while`, `do`, `for`, `break`, `continue`
- Arrays als Parameter: `public static void sort(int[] a) {...}`
- Zusicherungen: `assert(...)`

Die objektorientierten Anteile – insbesondere Objektinstantiierung, dynamische Bindung und Vererbung – wurden ausgeblendet, um die Komplexität des Prototypen zu reduzieren und so schneller zu Ergebnissen gelangen zu können.

Prinzipiell ist das Verfahren allerdings auch auf objektorientierte Sprachen anwendbar. Hierfür müssen zusätzliche Constraints eingeführt werden, um Aussagen über die Typen von Objekten formulieren zu können. Gegebenenfalls sind spezielle Constraints für den Zugriff auf Objektattribute nötig. Überladene Methoden können als alternative Implementierungen ein- und derselben Methode mit unterschiedlichen Beschränkungen für den Objekttyp dargestellt werden.

Der JAVA-Code wird über eine Zwischensprache in Prolog-Code übersetzt, der den Programmflussgraphen repräsentiert. Zu diesem Prolog-Code wird mit Hilfe des in Prolog eingebetteten CHR-Compilers das CHR-Programm aus Abschnitt 6.1 gebunden, das das Verfahren auf Basis der Erkenntnisse in Kapitel 5 implementiert. Mit diesem Gesamtprogramm können Testdaten für die JAVA-Funktion erzeugt werden. Der Gesamtablauf ist in Abbildung 6.1 dargestellt.

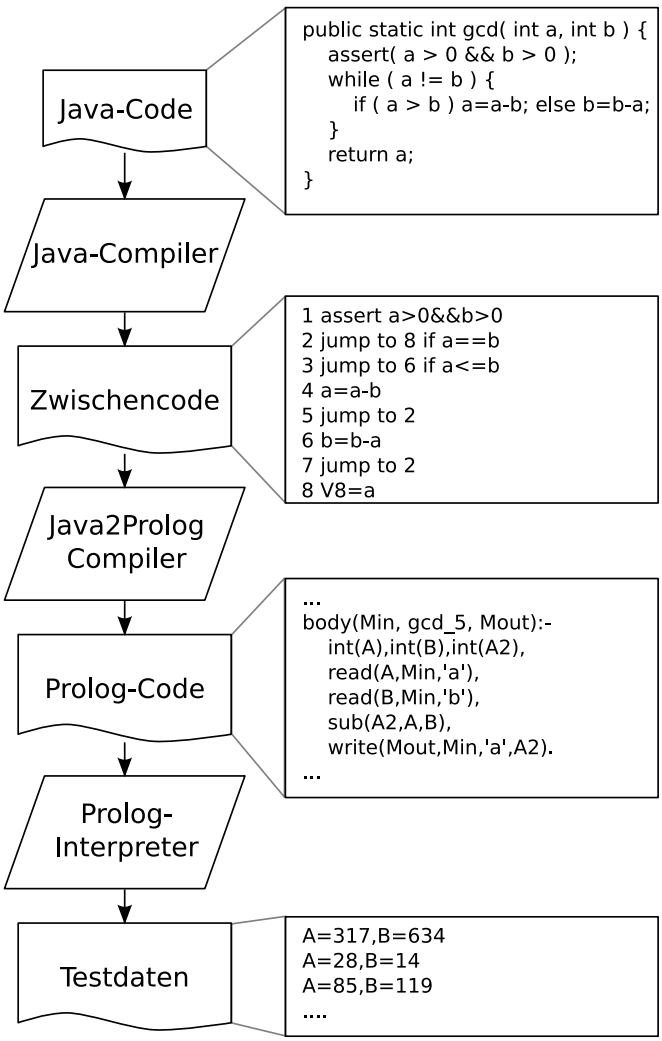


Abbildung 6.1: Flussgraph des Testfallgenerators

Das implementierte Verfahren verwendet einen in CHR definierten erweiterten Finite-Domain Constraintlöser zur Darstellung der Spezifikationen und Bedingungen aus dem Programmflussgraphen. Dieser Constraintlöser kann Constraints wie z.B. `A=B+C`, `A<B` und ähnliche verarbeiten, und ist dafür optimiert, Widersprüche möglichst effizient zu erkennen.

Als Prolog-Platform wurde SWI Prolog und für die CHR-Programme der integrierte CHR-Compiler[SD04] verwendet.

6.1 Formulierung in CHR

Mit $CHR^\vee$ können die wesentlichen Sätze der vorangegangenen Abschnitte ohne größere Umwege in ein Programm umgesetzt werden. Das $CHR^\vee$-Programm in Listing 6.1 beschreibt die Constraints `spec(X,V,W,Y)`, `ispec(V,X,Y,W)` und `ispec(V,X,Y,W,A)`, die jeweils den Aussagen $X\ \mathcal{S}_{PFG[V,W]}\ Y$, $X\ \mathcal{I}_{PFG[V,W]}\ Y$ und $X\ \mathcal{I}_{PFG[V,W]\setminus}\ Y$ entsprechen.

Dabei sind V und W Knoten, X und Y Speicherzustände und A eine Menge von auszuschließenden Knoten. In der Implementierung ist A als sortierte Liste mit einer entsprechenden Einfügeoperation implementiert. Wir verwenden den Ausdruck `A+L` für die Menge, die aus der Vereinigung von `A` und der Menge der Elemente in der Liste `L` entsteht.

Wir verwenden außerdem den Ausdruck `X.Var`, um die Belegung der Programmvariablen `Var` im Programmzustand `X` zu repräsentierten.

Grundlage sind zusätzlich zur Termgleichheit $=$ die vordefinierten Constraints aus Tabelle 6.1. Die Constraints `body(X,N,Y)` und `cond(X,A,B)` können auch als nutzerdefinierte Constraints eingeführt werden.

Dies ist notwendig, wenn der Programmflussgraph Unterprogrammaufrufe enthält. Sonst würde speziell bei rekursiven Programmen eine Schleife in der Hierarchie der CHR-Constraintlöser entstehen, da die vordefinierten Constraints `body` und `cond` von den nutzerdefinierten Constraints `ispec` und `spec` abhängig wären.

Die logische Lesweise dieser Regeln entspricht direkt den Sätzen 5.1, 5.2 und 5.4 sowie dem Korollar 5.2. Die Regel `ispec` repräsentiert den Sonderfall $\mathcal{I}_{PFG[v,w]} = \mathcal{I}_{PFG[v,w]\setminus\emptyset}$. Somit ist bereits bewiesen, dass die von diesem Programm gelieferten Lösungen korrekt sind und jede mögliche Lösung für das Eingabeconstraint auch abgeleitet werden kann.

Das vordefinierte Constraint `onallways(V,N,W)` definieren wir in der Implementierung ausschließlich über die Sätze 2.2 und 2.1. Die Ergebnisse des Programms

Constraint	Bedeutung
`body(X,V,Y)`	$X\ \mathcal{B}(V)\ Y$
`cond(X,V,W)`	$X\ \mathcal{C}(V,W)\ X$
`edge(V,W)`	$V \rightarrow W$
`reachable(V,W)`	$V \rightarrow^{+} W$
`onallways(V,N,W)`	N liegt auf jedem Weg von V nach W (Voraussetzung nach Satz 5.4).
`ispvar(Var)`	Var ist eine Programmvariable.
`varconst(V,Var,W)`	In allen Knoten, die auf einem Weg von V nach W liegen, bleibt der Wert der Programmvariablen `Var` unverändert(s. Korollar 5.2).
`member(E,A)`	Der Term `E` ist Element der Menge `A`.

Tabelle 6.1: Vordefinierte Constraints

```
1  spec @ spec(X,V,W,Y) <=>
2      (V=W, body(X,V,Y));
3      (body(X,V,XI), ispec(V,XI,YI,W), body(YI,W,Y)).
4
5  ispec @ ispec(V,X,Y,W) <=> ispec(V,X,Y,W,[]).
6
7  ispec_fail @ ispec(V,X,Y,W,A) <=>
8          onallways(V,N,W), member(N,A) | false.
9
10 predict_way @ ispec(V,X,Y,W,A) <=>
11     onallways(V,N,W), \+(member(N,A)) |
12
13 predict_var @ ispec(V,X,Y,W) ==>
14     ispvar(Var), varconst(V,Var,W) | X.Var=Y.Var .
15
16 search_recursive @ ispec(V,X,Y,W,A) <=>
17     (edge(V,W), cond(X,V,W), X=Y);
18     (reachable(V,N), reachable(N,W), \+(member(N,A)),
19      ispec(V,X,XN,N,A), body(XN,N,YN), ispec(N,YN,Y,W,A)).
```

Listing 6.1: CHR-Formulierung des Verfahrens

predict_way hängen natürlich davon ab, in welcher Reihenfolge die Knoten N, die auf jedem Weg von V nach W liegen, durchprobiert werden.

Es erscheint sinnvoll, die Reihenfolge der Regelanwendungen so zu bestimmen, dass zunächst die Regeln predict_var und predict_way angewendet werden, solange sie anwendbar sind. So wird die Einführung von Suchverzweigungen vermieden.

Die Anwendung der verfeinerten operationellen Semantik von CHR[DSBH04] ist jedoch nicht zu empfehlen, da dann insbesondere die beiden Teile der Regel search_recursive immer in derselben Reihenfolge betrachtet werden. Eine solche sequentielle Vorgehensweise ist bei diesem Verfahren gerade nicht wünschenswert.

6.1.1 Auflösung von Nichtdeterminismen

Der Kernteil der Suche ist in Regel search_recursive enthalten. Allerdings sind auch in predict_way mehrere Alternativen möglich, da für den Spaltungsknoten N mehrere Knoten in Frage kommen können. Diese Alternativen unterscheiden sich in der Reihenfolge der Spaltungsknoten, nicht jedoch im Ergebnis bei erschöpfender Anwendung der Regel.

Dazu betrachten wir das Constraint ispec(V,X,Y,W,A) in einem CHR-Zustand und nehmen an, dass es zwei unterschiedliche Knoten N1 und N2 nicht in A gibt so, dass onallways(V,N1,W) und onallways(V,N2,W) gelten. Somit gibt es zwei CHR-Nachfolgezustände, von denen einer den Weg von V nach W über N1 und der andere über N2 repräsentiert.

Somit muss entweder jeder Weg von N1 nach W über N2 oder jeder Weg von N2 nach W über N1 führen. Sonst könnten wir einen Weg konstruieren, der von V nach W führt und nur genau einen der Knoten N1 und N2 durchläuft, was der Annahme widerspricht.

Entsprechend führt jeder Weg von V nach N1 über N2 oder jeder Weg von V nach N2 über N1.

Nehmen wir an, dass onallways(N1,N2,W) und onallways(V,N1,N2) gelten. Somit lässt sich wie in Abbildung 6.2 dargestellt der Teilweg von V nach N2 am Knoten N1 bzw. der Teilweg von N1 nach W am Knoten N2 aufteilen. Beide Zustände repräsentieren nun einen Weg von A nach B, der zunächst über N1 und dann über N2 verläuft. Sie sind somit äquivalent.

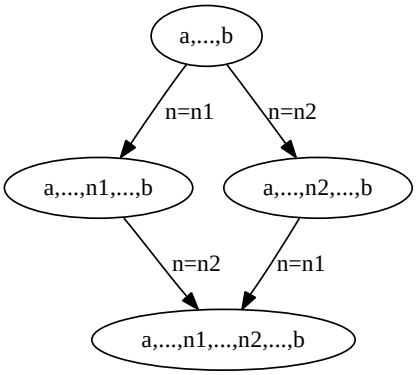


Abbildung 6.2: Konfluenz von `predict_way`

```
1 search_forward @ ispec(V,X,Y,W,A) <=>
2     (edge(V,W), cond(X,V,W,Y));
3     (edge(V,N), reachable(N,W), \+(member(N,A)),
4      cond(X,V,N), body(X,N,YN), ispec(N,YN,Y,W,A)).
5
6 search_backward @ ispec(V,X,Y,W,A) <=>
7     (edge(V,W), cond(X,V,W,Y));
8     (reachable(V,N), edge(N,W), \+(member(N,A)),
9      ispec(V,X,XN,W,A)), body(XN,N,Y), cond(Y,N,W).
```

Listing 6.2: Iterative Suchregeln

Analog argumentieren wir für den alternativen Fall `onallways(N2,N1,W)` und `onallways(V,N2,N1)`.

Bei Nutzung der Dominatoren- und Postdominatoren-Relationen zur Definition von `onallways` können wir einen Teil des Nichtdeterminismus entfernen, indem wir $idom$ und $ipostdom$ einsetzen.

So gibt es zu jedem Paar (a, b) maximal ein n, das $a\ idom\ n \wedge n\ dom\ b$ erfüllt. Entsprechend existiert zu einem solchen Paar auch maximal ein n, das $n\ ipostdom\ a \wedge b\ postdom\ n$ erfüllt.

Die Regel `search_recursive` kann entsprechend der Sätze 5.6 und 5.7 durch die Regeln `search_forward` und/oder `search_backward` aus Listing 6.2 ersetzt werden.

6.1.2 Beispiel

Betrachten wir zum Beispiel den Programmflussgraphen aus Abbildung 2.6 und das Ziel `body(X,r,Z1), ispec(r,Z1,Z2,2), spec(Z2,2,e,Y).` Wir verwenden dabei die nicht-deterministische operationelle Semantik, bei der wir zu jedem Zeitpunkt eine beliebige der anwendbaren Regeln auswählen können. Bei Disjunktionen im Rumpf wählen wir direkt eine der Alternativen.

$$body\,(X, r, Z_1)\,, ispec\,(r, Z_1, Z_2, 2, Z_2)\,, \underline{spec\,(Z_2, 2, e, Y)}$$
$$\downarrow \text{spec}$$
$$body\,(X, r, Z_1)\,, \underline{ispec\,(r, Z_1, Z_2, 2)}$$
$$body\,(Z_2, 2, Z_3)\,, \underline{ispec\,(2, Z_3, Z_4, e)}, body\,(Z_4, e, Y)$$
$$\downarrow \text{ispec(2x)}$$
$$body\,(X, r, Z_1)\,, ispec\,(r, Z_1, Z_2, 2, \emptyset)\,,$$
$$body\,(Z_2, 2, Z_3)\,, \underline{ispec\,(2, Z_3, Z_4, e, \emptyset)}, body\,(Z_4, e, Y)$$
$$\downarrow \text{predict_way}$$
$$body\,(X, r, Z_1)\,, \underline{ispec\,(r, Z_1, Z_2, 2, \emptyset)},$$
$$body\,(Z_2, 2, Z_3)\,, ispec\,(2, Z_3, Z_5, 1, \emptyset)\,, body\,(Z_5, 1, Z_6)\,, ispec\,(1, Z_6, Z_4, e, \emptyset)\,,$$
$$body\,(Z_4, e, Y)$$
$$\downarrow \text{predict_way}$$
$$body\,(X, r, Z_1)\,, \underline{ispec\,(r, Z_1, Z_7, 1, \emptyset)}, body\,(Z_7, 1, Z_8)\,, \underline{ispec\,(1, Z_8, Z_2, 2, \emptyset)},$$
$$body\,(Z_2, 2, Z_3)\,, \underline{ispec\,(2, Z_3, Z_5, 1, \emptyset)}, body\,(Z_5, 1, Z_6)\,, \underline{ispec\,(1, Z_6, Z_4, e, \emptyset)},$$
$$body\,(Z_4, e, Y)$$
$$\downarrow \text{search_recursive(4x)}$$
$$body\,(X, r, Z_1)\,, \underline{cond\,(Z_1, r, 1)}, body\,(Z_1, 1, Z_2)\,, \underline{cond\,(Z_2, 1, 2)},$$
$$body\,(Z_2, 2, Z_3)\,, \underline{cond\,(Z_3, 2, 1)}, body\,(Z_3, 1, Z_4)\,, \underline{cond\,(Z_4, 1, e)},$$
$$body\,(Z_4, e, Y)$$

Zunächst werden die `spec`- und `ispec`-Constraints in die Normalform von `ispec` mit Menge ausgeschlossener Knoten transformiert.

Auf Grundlage der Dominator- und Postdominatorrelationen können wir in den nächsten zwei Schritten vorhersagen, dass auf dem Weg von r nach 2 und auf dem Weg von 2 nach e jeweils einmal der Knoten 1 durchlaufen wird.

Zuletzt schließen wir durch den Basisfall der Regel `search_recursive` alle offenen Teilwege ab.

6.1.3 Probabilistische Erweiterung

Eine systematische Auflistung eines Teils der möglichen Lösungen wäre mit dem obigen CHR-Programm machbar, ist aber im Sinne des Softwaretest nicht wünschenswert. Die Auswahl der Testeingaben soll nur von den Vorgaben an die Testeingaben selbst, nicht jedoch direkt vom Verfahren zur Testauswahl oder den Details seiner Implementierung direkt abhängig sein.

Weiterhin würde durch die Bearbeitung der `ispec`-Constraints in fester Reihenfolge die Wegsuche entgegen der Anforderungen wieder rein sequentiell ausgeführt.

Wir schlagen deshalb eine probabilistische Erweiterung des obigen Programms vor. Probabilistisches CHR[FPW02] bietet hier aber keine akzeptable Semantik, da dort nicht den einzelnen Regeln sondern vielmehr ihren anwendbaren Instanzen ein fixes Gewicht zugeordnet wird.

Je mehr mögliche Spaltungsknoten `N` für ein bestimmtes `ispec`-Constraint existieren, umso stärker überwiegt die Wahrscheinlichkeit, dass auf dieses Constraint der Rekursionsfall statt dem Basisfall angewendet wird.

Weiterhin ist für ein `ispec`-Constraint mit vielen möglichen Spaltungsknoten die Wahrscheinlichkeit, ausgewählt zu werden, größer als für ein `ispec`-Constraint mit wenigen möglichen Spaltungsknoten.

Die Wahrscheinlichkeit dafür, den Basisfall überhaupt anzuwenden, fällt außerdem mit der Zahl der verfügbaren `ispec`-Constraints, auf die der Rekursionsfall anzuwenden ist.

Somit ist die Anwendungswahrscheinlichkeit vom aktuellen Zustand abhängig. Insbesondere hat das Verfahren die Tendenz, die Zahl der `ispec`-Constraints mit ansteigender Wahrscheinlichkeit ständig zu vergrößern, was die Bevorzugung sehr langer Wege und gegebenenfalls ein Nichtterminieren zur Folge hat. Außerdem wird Vorhersage des probabilistischen Verhaltens etwa bezüglich der mittleren Länge des gewählten Weges deutlich erschwert.

Für ein Verfahren mit explizit gleichverteilten bzw. konstanten Auswahlwahrscheinlichkeiten hingegen ergeben sich bessere Möglichkeiten der statischen Analyse und somit auch der Steuerung der mittleren Weglängen. Eine detaillierte Analyse dieser Eigenschaften ist in den Abschnitten 7.2 und 7.3 zu finden.

Die Wahrscheinlichkeit der Auswahl eines bestimmten `ispec`-Constraints soll deshalb zu jedem Zeitpunkt während der Ausführung genauso groß sein wie für alle seine Geschwister. Ebenso soll die Wahrscheinlichkeit p für die Anwendung des

Rekursionsfalls nicht abhängig von der Zahl der `ispec`-Constraints sein, auf die der Rekursionsfall anwendbar wäre.

Wir haben hierfür eine eigene probabilistische operationelle Semantik eingeführt und dafür die Grundidee der für probabilistisches CHR vorgeschlagenen Implementierung übernommen. Diese basiert auf der Emulation der Einzelschritte von probabilistischem CHR auf Basis der verfeinerten operationellen Semantik von CHR. In unserer Umsetzung folgt das Programm damit für jeden Schritt dem in Abbildung 6.3 dargestellten Ablauf.

In der iterativen Fassung des Verfahrens werden Vorwärts- oder Rückwärtsschritte statt des Rekursionsfalls eingesetzt und statt eines Spaltungsknotens ein Vorgänger bzw. Nachfolger ausgewählt.

Untersuchungen der Längen der erzeugten Wege zeigten außerdem die Erfordernis einer Erweiterung des iterativen Algorithmus, da sonst innere Schleifen nur mit sehr geringen Iterationszahlen unabhängig von p durchlaufen würden.

So muss eine Schleifenausgangs- bzw. -eingangskante mit Wahrscheinlichkeit $1-p$ und jede andere Kante mit Wahrscheinlichkeit p durchlaufen werden. Dies wird erreicht, indem bei einem Schritt die Nachfolger oder Vorgänger, die im Kontext der Anfrage einer Ausgangs- bzw. Eingangskante entsprechen, mit $1-p$ gewichtet werden, alle anderen mit p.

Weiterhin kann bei der Wegvorhersage innerhalb einer Schleife die Schleife in beiden neu eingeführten Teilwegen mehrfach durchlaufen werden. Dies macht sich besonders bei verschachtelten Schleifen ungünstig bemerkbar. Als Gegenmaßnahme wird bei der Wegvorhersage der Spaltungsknoten der Vermeidungsmenge eines der beiden neuen Teilwege hinzugefügt, um so die Schleifendurchläufe auf den anderen Teilweg zu beschränken.

Diese Phänomene werden in Abschnitt 7.3 ausführlich dargelegt und diskutiert.

6.2 Speicherzugriffe, Arrays und Referenzen

Einige der bekannten Verfahren repräsentieren den Zustand explizit als Tupel der aktuellen Werte der Programmvariablen[Kor90, GBR98].

Die Array-SSA-Form[KS98, Bra01] ermöglicht die Darstellung von Array-Zugriffen. Eine Zuweisung der Form `a[i]=v` wird hierbei zu `a1=arraywrite(a,i,v)`, wobei `a1` das neue Array repräsentiert, und ein Zugriff der Form `v=a[i]` wird zu `v=arrayread(a,i)`. Es werden also nicht die Definitionen und Nutzungen einzelner Array-Elemente sondern nur die des Arrays als Ganzes verfolgt.

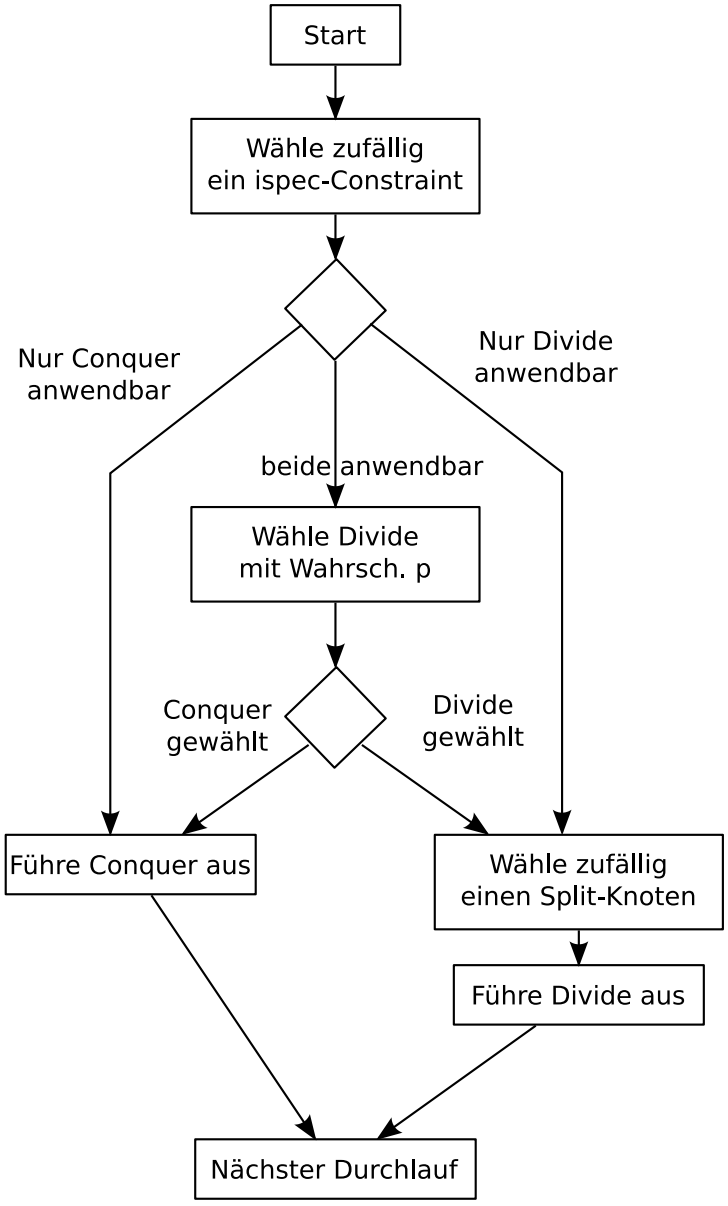


Abbildung 6.3: Zufallsverfahren

```
1 write(M1,_,A1,V1) \ read(M1,A2,V2) <=>
2         eqref(A1,A2) | V1=V2.
3 write(M1,M0,A1,_) \ read(M1,A2,V) <=>
4         neqref(A1,A2) | read(M0,A2,V).
5 write(M1,M0,A1,V1) \ write(M2,M0,A2,V2) <=>
6         eqref(A1,A2) | M1=M2, V1=V2, A1=A2.
7 write(_,M0,A1,_), write(_,M0,A2,_) <=>
8         A1\=A2 | false.
```

Listing 6.3: Speichermodell in CHR

Zudem kann die SSA-Form durch spezielle ϕ-Funktionen erweitert werden, die Referenzen auflösen[GDB05b]. Hierfür muss jedoch bereits zur Übersetzungszeit die Menge der Variablen bekannt sein, die eine Zeigervariable referenzieren kann, und diese Menge muss endlich sein, ebenso wie die Menge der Zeigervariablen selbst.

Die Menge der Variablen ist allerdings nicht bekannt, wenn die Länge von Feldern zur Übersetzungszeit nicht bekannt oder zumindest beschränkt ist und Elemente von Feldern auch referenziert werden können oder andere Variablen referenzieren können.

Da in JAVA auch Arrays als Objekte repräsentiert und insbesondere mehrdimensionale Arrays durch Arrays von Referenzen dargestellt werden, muss ein erweitertes Verfahren angewendet werden.

Für den Prototypen haben wir deshalb ein Speichermodell entwickelt, das näher am tatsächlichen Speicherzugriff ist als diese Darstellungsformen. Ein Speicherzustand wird dabei als Abbildung von Speicheradressen zu Speicherinhalten interpretiert, wobei die Speicherinhalte beliebige Prolog-Terme sind. Zwei Adressen zeigen dabei genau auf dieselbe Speicherstelle, wenn die Adressen gleich sind.

Das Constraint `read(V,M,A)` ist genau dann erfüllt, wenn an der Adresse `A` im Speicherzustand `M` der Inhalt `V` steht. Ein Schreibzugriff wird durch das Constraint `write(M1,M0,A,V)` dargestellt. Der Speicherzustand `M1` entsteht dabei aus dem Speicherzustand `M0` durch Schreiben des Inhalts `V` an die Adresse `A`.

Eine einfache Implementierung der zugrundeliegenden Constrainttheorie ist in Listing 6.3 dargestellt.

Die Bedingung `eqref(A1,A2)` ist dann für zwei Adressen `A1` und `A2` erfüllt, wenn die beiden Adressen gleich sind. Dies ist nicht äquivalent zur Termgleichheit. So gilt

auch dann `eqref(A1,A2)`, wenn zum Beispiel die Constraints `A1=B+C, A2=B+C` im Constraintstore liegen.

Entsprechend ist die Bedingung `neqref(A1,A2)` dann erfüllt, wenn die Adressen `A1` und `A2` nicht gleich sind. Analog zur obigen Überlegung ist dies nicht äquivalent zur Termungleichheit.

Dieses Modell stellt den gesamten Speicher als Array dar, wobei die Indices beliebige Terme sein dürfen. Im Prototypen kommt ausschließlich dieses Modell zum Einsatz, wobei nicht ausgeschlossen ist, zur Optimierung der Bearbeitung einfacherer Strukturen die bereits bekannten Ansätze komplementär einzusetzen.

6.3 Optimierter Finite-Domain-Löser

Wir haben einen Finite-Domain-Constraintlöser für die Lösung der Wegconstraints implementiert. Dieser ist besonders auf die Erkennung von Widersprüchen optimiert.

Wie bei Finite-Domain Constraintlösern üblich[FA03] wird jeder Variablen x ein beschränkter Wertebereich (Domain) R zugewiesen der durch ein Intervall beschrieben wird. Dies stellen wir durch das Constraint $x \in [x_l; x_u]$ dar.

Bei der Propagation wird dieser Wertebereich immer weiter eingeschränkt, bis entweder ein Fixpunkt erreicht wurde oder einer der Wertebereiche auf die leere Menge reduziert wurde. Im letzteren Fall wurde ein Widerspruch erkannt und das Constraintproblem besitzt keine Lösung.

Konsistenzbedingung ist dabei die lokale Kantenkonsistenz (arc-consistency). Ein Constraint $c(v_1, \ldots, v_n)$ ist dann kantenkonsistent, wenn für jede Variable v_i jeder Wert $x_i \in R_i$ Bestandteil mindestens einer Lösung $c(x_1, \ldots, x_n)$ ist[FA03].

Um die Kantenkonsistenz eines Constraints zu erreichen, werden diejenigen Werte aus dem Wertebereich gestrichen, die nicht zu einer Lösung des Constraints selbst beitragen. Da sich die Konsistenz hier auf einzelne Constraints bezieht, müssen die einzelnen Werte aus den Wertebereichen allerdings nicht unbedingt zu einer Gesamtlösung des Constraintsystems beitragen.

Dies ist jedoch nur bedingt ausreichend um z.B. Widersprüche wie $a < b \wedge b < a$ zu erkennen. Die Propagation wird abwechselnd den Wertebereich von a und b um einen Schritt verkleinern, bis einer der Wertebereiche leer ist. Gilt zu Beginn etwa $a, b \in \{-2^{31}, \ldots, 2^{31} - 1\}$, so werden 2^{33} Schritte benötigt bis der Widerspruch erkannt wird. Dabei ist $a < b \wedge b < a$ aufgrund der Asymmetrie von $<$ auch ohne die Wertebereichsreduktion als Widerspruch zu erkennen.

Der optimierte Finite-Domain-Constraintlöser im Prototypen enthält deshalb neben den klassischen Regeln zur Wertebereichsreduktion auch Regeln, die den Axiomen der jeweiligen Relationen entsprechen. So sind für Addition, Subtraktion, Multiplikation und Division insbesondere die Gruppenaxiome und für $\leq$ und $<$ die entsprechenden Relationsaxiome für Transitivität, Reflexivität/Irreflexivität und Antisymmetrie/Asymmetrie umgesetzt.

Der Constraintlöser unterstützt die folgenden Constraints:

- $x \in [x_l; x_u]$
- $z = \max(x, y)$, $z = \min(x, y)$
- $x = y \Leftrightarrow b$, $x \leq y \Leftrightarrow b$
- $x < y$, $x \leq y$, $x \neq y$, $x = y$
- $x = -y$
- $z = x + y$
- $z = x * y$

Die bedingten Formen $x = y \Leftrightarrow b$ und $x \leq y \Leftrightarrow b$ sowie die Bildung von Maximum und Minimum werden zur Bildung zusammengesetzter logischer Ausdrücke verwendet, unter der Annahme, dass die Wahrheitswerte `false` und `true` als `0` bzw. `1` abgebildet werden. Die Bedingung $x = y \vee x = z$ wird z.B. wie folgt umgesetzt:

$$(x = y \Leftrightarrow b_1) \wedge (x = z \Leftrightarrow b_2) \wedge 1 = \max(b_1, b_2) \wedge b_1, b_2 \in \{0, 1\}$$

Ebenso lassen sich Subtraktion und Division auch zu Addition und Multiplikation umformen, wobei bei letzterem auf die Nullstellen zu achten ist.

Nach Erreichen des Fixpunktes der Propagation haben einige Variablen oft noch mehrelementige Wertebereiche. Nicht alle Elemente dieser Wertebereiche sind notwendigerweise Teil einer Gesamtlösung. Den Variablen wird beim *Labelling* ein eindeutiger Wert aus ihrem Wertebereich zugeordnet[FA03].

Im Prototypen wird hierfür ein einfaches Bisektionsverfahren eingesetzt. Der Wertebereich einer Variablen wird an einer zufälligen Stelle geteilt und auf einen der beiden Teile eingeschränkt. Darauf folgt eine erneute Propagation mit dem neuen Wertebereich, wodurch die Wertebereiche aller Variablen zusätzlich eingeschränkt werden. Bei einem Widerspruch wird der jeweils andere Teil probiert, ansonsten wird die Bisektion bei einer anderen Variablen fortgesetzt, bis alle Variablen eindeutige Werte haben oder aber die Möglichkeiten erschöpft sind.

6.4 Übersetzung in Prolog-Code

Unser Compiler für die zuvor definierte Teilmenge von JAVA basiert auf dem quelloffenen und frei verfügbaren Lexer- und Parser-Generator antlr[Par09] und einer dafür bereitgestellten Grammatik für JAVA5. Im Compiler wird der JAVA-Code der einzelnen statischen Methoden in einen Zwischencode übersetzt, dessen Anweisungen in Zuweisungs- und Sprunganweisungen eingeteilt werden können.

Die Zuweisungen haben die Form `V=E` oder `V[I]=E`, wobei `V` und `I` Variablen sind und `E` ein Ausdruck ist. Die Sprache der Ausdrücke beinhaltet Konstanten, Variablen sowie Lesezugriffe auf Arrayelemente und ist unter Negation, Addition, Subtraktion, Multiplikation und Division abgeschlossen. JAVA-Ausdrücke mit Seiteneffekten – etwa mit konditionalen Ausdrücken – werden bei der Übersetzung entsprechend aufgeteilt.

Die Sprunganweisungen haben die Form `jump to P` bzw. `jump to P if C`, wobei `P` eine Anweisungsadresse und `C` eine Bedingung ist. Die Sprache der Bedingungen beinhaltet Zugriffe auf Variablen vom Typ `boolean` sowie Vergleiche zwischen Ausdrücken und ist unter Bildung von logischer Negation, Konjunktion und Disjunktion abgeschlossen.

Aus diesem Zwischencode wird dann ein Basisblockgraph erzeugt[ASU97]. Für jeden Knoten in diesem Graphen wird eine Prolog-Klausel für das Prädikat `body/3` erzeugt. Entsprechend wird für jede Kante im Graphen eine Prolog-Klausel für `cond/3` generiert.

7 Eigenschaften

Wir untersuchen nun die Eigenschaften des vorgestellten Verfahrens. Von besonderem Interesse sind die Qualität der erzeugten Testfälle, das Terminierungsverhalten sowie die Laufzeit.

Dazu untersuchen wir diese Eigenschaften jeweils zunächst auf theoretischer Ebene auf Basis eines Modells und validieren sie dann anhand von Messungen mit Beispielen in Monte-Carlo-Experimenten bzw. am Prototypen.

Wir vergleichen dabei insbesondere die Eigenschaften der verschiedenen Ausprägungen des Verfahrens. Da einzelne Ausprägungen des neuen Verfahrens ganz oder teilweise bereits bestehenden Verfahren entsprechen, können wir durch den Vergleich der Ausprägungen miteinander auch in Teilen den Vergleich mit diesen bekannten Verfahren durchführen.

Da in der Literatur nur Angaben über einzelne Durchläufe, nicht aber Untersuchungen der asymptotischen Laufzeitkomplexität zu finden sind, ist ein direkter Vergleich nicht möglich. Referenzimplementierungen stehen entweder nicht zur Verfügung oder sind nicht mit unserer Implementierung vergleichbar, wie wir im Folgenden ausführen.

Das Verfahren von Nguyen und Deville[ND01] entspricht direkt der Variante, die nur Vorwärtsschritte und keine der Vorhersageregeln verwendet. Die Kontrollabhängigkeiten, die Nguyen und Deville nutzen, um vorgegebene Knoten oder Kanten zu erreichen, werden bei unserem Verfahren nicht benötigt, da wir explizit nach zwei Teilwegen suchen und Spaltungsknoten n, über die es keinen Weg zum Ziel gibt, durch die Bedingung $a \rightarrow^+ n \wedge n \rightarrow^+ b$ ausgeschlossen werden.

Das Verfahren von Gotlieb et al[GBR98] entspricht keiner unserer Varianten vollständig. Teilaspekte werden jedoch abgebildet, etwa die gleichzeitige Verarbeitung von sequentiellen Programmkonstrukten wie auch das Vorwärtsabrollen der Schleifen.

Die gleichzeitige Verarbeitung sequentieller Programmkonstrukte wird in unserem Verfahren durch die Wegvorhersage bewerkstelligt. Die Spaltungsknoten entsprechen bei den üblichen Compilerschemen von `while`- und `if`-Anweisungen jenen Stellen, an denen auch Gotlieb et al die Programmkonstrukte voneinander trennen.

Schleifen werden bei Gotlieb et al grundsätzlich vorwärts abgerollt, also in zwei Teile aufgeteilt, deren einer Teil genau einen Schleifendurchlauf und der andere Teil alle weiteren Schleifendurchläufe repräsentiert. Dies bilden wir durch die Wegvorhersage mit Knotenvermeidung ab.

Spaltungsknoten ist der Schleifenkopf, der bei `while`-Schleifen gleichzeitig Dominator und Postdominator der Rumpfknoten ist. Bestimmen wir, dass dieser Knoten im ersten der bei der Spaltung neu entstehenden Teilwege nicht mehr durchlaufen werden darf, so entspricht dieser Teilweg genau einem Schleifendurchlauf. Im zweiten Teil können dann weitere Schleifendurchläufe folgen.

Die Verknüpfung der logischen Variablen, die die Werte der Programmvariablen in verschiedenen Programmzuständen repräsentieren, geschieht bei unserem Verfahren nur durch die Constraints `body`, `cond` sowie durch die Datenflussvorhersage. In der Implementierung werden jedoch nicht alle Fälle erkannt, in denen die Voraussetzungen für eine Vorhersage vorliegen und in denen das Verfahren von Gotlieb et al eine Gleichheit der Variablenwerte erkennt. Dies sehen wir am Selection-Sort-Beispiel in Abschnitt 7.5.2.

Die existierende kommerzielle Implementierung InKA sowie die in der Literatur angegebenen Laufzeiten eignen sich nicht für einen direkten Vergleich, da sie auf anderer Hardware und einem anderen Constraintlöser basieren. Eine Referenzimplementierung auf Basis unseres Constraintlösers scheiterte an ungenauen Angaben in der zur Verfügung stehenden Literatur[GBR98, GBR00, Got01, GDB05a, GDB05b]. So wird nicht beschrieben, wie die zur Erreichung der Knotenabdeckung verwendete dynamische Neuberechnung von Kontrollabhängigkeiten in Schleifen umgesetzt wird. Ebenso ist erwähnt, dass während der Widerspruchserkennung einige Constraints deaktiviert werden, jedoch nicht, welche.

Wir können deshalb im Vergleich nicht alle Aspekte des Verfahrens von Gotlieb et al berücksichtigen.

7.1 Testfallqualität

Die Qualität einer Testfallmenge bestimmt sich primär aus ihrer Eignung, möglichst viele Fehler in einem Programm mit möglichst wenigen Testfällen aufzudecken. Werden durch die Testfallmenge keine Fehler aufgedeckt, so soll sie sekundär die Annahme der Fehlerfreiheit mit möglichst geringer Irrtumswahrscheinlichkeit rechtfertigen. Beim vorgestellten Verfahren kommen für die Beeinflussung der Qualität mehrere Faktoren in Frage.

Die Testeingaben werden primär nach den Vorgaben des Anwenders ausgewählt. Entsprechend hängt die Qualität der erzeugten Testfallmengen auch stark von diesen Vorgaben ab (s. Abschnitt 3.1).

Wird ein Programmflussgraph verwendet, der das Verhalten des Programms nur ungenau abbildet, so liefert das Verfahren möglicherweise Eingaben, die entsprechend dem Programmflussgraph zwar die Vorgaben erfüllen, nicht aber im tatsächlichen Prüfling.

Bei der Implementierung des Constraintslösers ist insbesondere auf die korrekte Semantik der arithmetischen Operationen zu achten. Die Rechenoperationen sind in den meisten Programmiersprachen – abgesehen von einigen funktionalen oder logischen Sprachen sowie einigen Scriptsprachen – als Zweierkomplementoperationen mit Über- bzw. Unterlauf ausgeführt. Bildet der Constraintlöser diese Semantik nicht korrekt ab, so werden per Definition keine Testfälle konstruiert, die die wichtigen Überlauffehler aufdecken können.

Die Testfallqualität kann auch durch die Auswahl der Wege beeinflusst werden. Werden tendenziell solche Wege häufiger gewählt, die nicht zur Fehleridentifikation beitragen, so wird die Qualität vermindert. Umgekehrt kann die Qualität durch bevorzugte Wahl fehlerauslösender Wege verbessert werde.

Das Verfahren wählt die Wege jedoch primär zufällig aufgrund der lokalen Struktur des Programmflussgraphen. Insbesondere stehen dem Verfahren keine Informationen darüber zur Verfügung, wie sich Fehler manifestieren und welche Eingaben zu ihrer Identifikation führen können. Besondere Eigenschaften des Programms werden nur verwendet, um nicht-ausführbare Wege auszuschließen.

Aus der in Abschnitt 6.1.3 eingeführten probabilistischen Semantik lässt sich ableiten, dass das Verfahren etwa nicht alle Wege mit gleicher Wahrscheinlichkeit wählt. Unabhängig von der Wahl des Prozessparameters p ist davon auszugehen, dass kürzere Wege den längeren bevorzugt werden. Die Verteilung ändert sich abhängig von p lediglich quantitativ, nicht aber qualitativ.

Dem kann der Anwender begegnen, indem zwischen mehreren Läufen des Verfahrens p jeweils variiert wird. Wir analysieren deshalb in Abschnitt 7.3 den Zusammenhang zwischen p und der mittleren Weglänge sowie deren Streuung für verschiedene Beispielprogramme. Dabei zeigen wir, dass auch bei komplexeren Beispielen eine Erhöhung von p eine Erhöhung von mittlerer Weglänge sowie deren Streuung bewirkt. Diese Regel kann auch automatisiert angewendet werden, indem zum Beispiel mehrere äquidistante Werte von p bei aufeinanderfolgenden Ausführungen des Verfahrens verwendet werden.

Der Anteil der fehleraufdeckenden Testfälle in der Ausgabe des Verfahrens hängt also – abgesehen von den vorgegebenen Testkriterien – im Wesentlichen von der Struktur des Programms selbst ab. Dies gilt in ähnlicher Weise auch für den klassischen Zufallstest, bei dem die Verteilung nur von der Struktur des Eingaberaums abhängig ist. Zwar ergibt sich durch die Zufallswahl in der Wegemenge statt dem Eingaberaum eine andere Verteilung der Testeingaben. Eine generelle Bevorzugung oder Benachteiligung fehleraufdeckender Testfälle ist jedoch nicht zu erwarten.

7.2 Terminierung

Das Terminierungsverhalten des Verfahrens wird durch seine Eingabe bestimmt, bestehend aus den Parametern für die Zufallsauswahl, dem Programmflussgraphen des zu testenden Programms und der Anfrage in Form eines oder mehrerer Spezifikationsconstraints.

Es ist von vornherein festzustellen, dass unser CHR-Programm keinesfalls für jede mögliche Eingabe terminieren kann, da sonst das allgemeine Halteproblem lösbar wäre, wie bereits in Satz 4.1 bewiesen wurde.

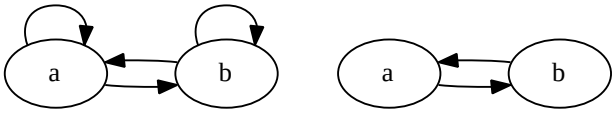
Wir untersuchen zunächst das Terminierungsverhalten der probabilistischen Erweiterung unter der optimistischen Annahme, dass jeder Weg tatsächlich ausführbar ist. Dabei zeigen wir, dass die rekursive Variante lediglich in trivialen Fällen ein akzeptables Terminierungsverhalten zeigt, während sich die iterative Variante als deutlich robuster in dieser Hinsicht herausstellt.

Dazu betrachten wir in erster Näherung einen besonders vorteilhaften Programmflussgraphen und modellieren das Zufallsverhalten für diesen Graphen. Daraufhin bestimmen wir das Verhalten für kompliziertere Beispiele mit Hilfe von Monte-Carlo-Simulationen.

In der Folge diskutieren wir die Terminierung des Verfahrens im allgemeinen Fall, in dem ein nicht zu vernachlässigender Anteil der Wege nicht ausführbar ist. Wir erläutern dabei notwendige Kriterien für die Terminierung.

7.2.1 Theoretisches Zufallsverhalten

Für das Zufallsverfahren beschreiben wir die Terminierung durch die Terminierungswahrscheinlichkeit, also die Wahrscheinlichkeit in endlicher Zeit eine Ausführungsfolge zu konstruieren. Die Tatsache, eine endliche Ausführungsfolge konstru-



(a) vollständig (b) partiell

Abbildung 7.1: Beispielgraphen mit 2 Knoten, vollständig und partiell verbunden

iert zu haben, impliziert dabei eine endliche Schrittzahl, weshalb es ausreichend ist, die Wahrscheinlichkeit für eine endliche Ausführungsfolge zu betrachten.

Die Zufallsvariable ist in diesem Fall die Länge des erzeugten Weges L und somit ist die Terminierungswahrscheinlichkeit gegeben durch Gleichung 7.1, wobei $P(L = l)$ die Wahrscheinlichkeit ist, einen Weg der Länge l zu erzeugen.

Sowohl $P(L < \infty)$ als auch $P(L = l)$ hängen vom Verfahrensparameter p ab, den wir zur besseren Lesbarkeit nicht explizit aufführen.

$$P(L < \infty) \quad = \quad \sum_{l=1}^{\infty} P(L = l) \tag{7.1}$$

Wir betrachten zunächst einen vollständig verbundenen Graphen. Für jedes Paar von Knoten a und b gilt in diesem Graphen $a \rightarrow b$. Insbesondere ist auch jeder Knoten direkt mit sich selbst verbunden. Für einen Graphen mit 2 Knoten ist dies in Abbildung 7.1a exemplarisch dargestellt. Dieser Graph ist kein Kontrollflussgraph, da in ihm $e \rightarrow r$ gilt. Dies ist für die folgenden Betrachtungen jedoch nicht relevant.

Durch seine Konstruktion stellen wir vielmehr sicher, dass auf jede Anfrage sowohl Basis- als auch Rekursionsfall, Vorwärts- und Rückwärtsschritt anwendbar sind. Weiterhin ist die Menge der Antworten für eine Anfrage isomorph zur Menge der Antworten für jede andere Anfrage, so dass wir die konkreten Start- und Endknoten der Anfrage vernachlässigen können.

Wir gehen von einer Anfrage bestehend aus einem einzelnen `ispec/5`-Constraint aus. Für das Divide&Conquer-Verfahren ist $P(L = l)$ durch Gleichung 7.2 beschrieben. Beim iterativen Verfahren handelt es sich um einen Bernoulli-Prozess, und die Wahrscheinlichkeit $P(L = l)$ hierfür ist durch Gleichung 7.3 definiert.

$$P_{Div}(L = l) \quad = \quad \begin{cases} 1 - p & \text{für} \quad l = 1 \\ p \sum_{i=1}^{l-1} P(L = i) P(L = l - i) & \text{für} \quad l > 1 \end{cases} \tag{7.2}$$

$$P_{Step}(L = l) \quad = \quad \begin{cases} 1 - p & \text{für} \quad l = 1 \\ pP(L = l - 1) & \text{für} \quad l > 1 \end{cases} \tag{7.3}$$

Für beide Varianten ist die Wahrscheinlichkeit, aus einer einzelnen Anfrage einen Weg der Länge 1 zu erzeugen, jeweils $1-p$, da dies die Wahrscheinlichkeit für die Anwendung des Basisfalls auf eine offene Anfrage ist.

Im Fall des Divide&Conquer-Verfahrens entstehen längere Wege nur durch Aufteilung durch den Rekursionsfall mit Wahrscheinlichkeit p. Ein Weg der Länge l kommt nur dann zustande, wenn die beiden neuen Anfragen Wege der Länge i und $l-i$ erzeugen.

Im iterativen Verfahren wird mit Wahrscheinlichkeit p eine Anfrage durch eine neue ersetzt. Die Länge l kommt dann zustande, indem aus der neuen Anfrage ein Weg der Länge $l-1$ erzeugt wird.

Aus diesen und ähnlichen rekursiven Gleichungen für andere Graphen und Verfahrensvarianten wollen wir jeweils die Terminierungswahrscheinlichkeit und in Abschnitt 7.3 die erwartete Weglänge sowie ihre Varianz bestimmen. Als Ansatz hierfür verwenden wir die Potenzreihe aus Gleichung 7.4. Für sie gilt insbesondere $F(1) = P(L < \infty)$.

$$F(q) = \sum_{l=1}^{\infty} P(L=l)\, q^l \tag{7.4}$$

Für die iterative Variante ergibt sich $F_{Step}(q)$ entsprechend Gleichung 7.5. Die Terminierungswahrscheinlichkeit ist demnach 1 für $p \in [0;1)$. Für $p = 1$ ermitteln wir die Terminierungswahrscheinlichkeit direkt aus Gleichung 7.3 und erhalten so Gleichung 7.6.

$$\begin{aligned} F_{Step}(q) &= \sum_{l=1}^{\infty} P_{Step}(L=l)\, q^l \\ &= (1-p)\, q + \sum_{l=1}^{\infty} P_{Step}(L=l+1)\, q^{l+1} \\ &= (1-p)\, q + pq \sum_{l=1}^{\infty} P_{Step}(L=l)\, q^l \\ &= (1-p)\, q + pq F_{Step}(q) \\ &= \frac{(1-p)\, q}{1-pq} \end{aligned} \tag{7.5}$$

$$P_{Step}(L<\infty) = \begin{cases} 1 & \text{für} \quad p \in [0;1) \\ 0 & \text{für} \quad p = 1 \end{cases} \tag{7.6}$$

Für das Divide&Conquer-Verfahren nutzen wir die Cauchy-Faltung, um die Summe der Produkte in $P_{Div}(L=l)$ für $l > 1$ aufzulösen und erhalten somit Gleichung 7.8.

$$
\begin{aligned}
F_{Div}(q) &= (1-p)\,q + p\sum_{l=1}^{\infty}\sum_{i=1}^{l} P_{Div}(L=i)\,P_{Div}(L=l+1-i)\,q^{l+1} && (7.7)\\
&= (1-p)\,q + p\left(\sum_{l=1}^{\infty} P_{Div}(L=l)\,q^{l}\right)^{2}\\
&= (1-p)\,q + p\,(F_{Div}(q))^{2}\\
F_{Div}(q) &= \frac{1 \pm \sqrt{1-4qp\,(1-p)}}{2p} && (7.8)
\end{aligned}
$$

Mit dem Ansatz $f(z) = \sqrt{1-z}$ sowie $F_{Div}(q) = \frac{1 \pm f(4qp(1-p))}{2p}$ erhalten wir nach wenigen Rechenschritten die Potenzreihenentwicklung in Gleichung 7.9.

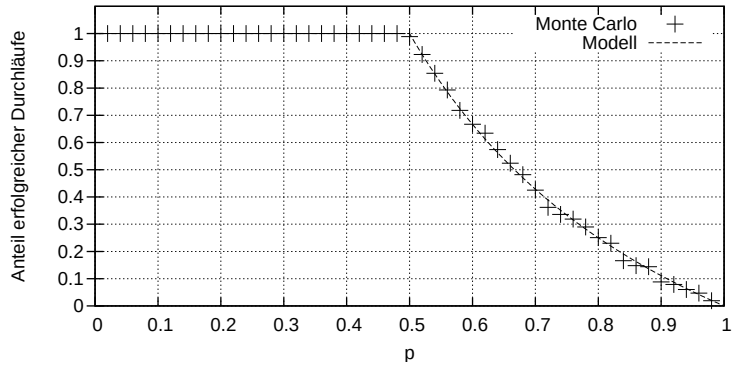
$$
\begin{aligned}
F_{Div} &= \frac{1}{2p}\left(1 \pm \sum_{l=0}^{\infty} k_l\,(4p\,(1-p))^{l}\,q^{l}\right) && (7.9)\\
k_0 &= \frac{1}{2} && (7.10)\\
k_{l+1} &= \frac{2l-1}{2}k_l && (7.11)
\end{aligned}
$$

In Gleichung 7.7 sind die Koeffizienten zu q^l nicht negativ. Auf Gleichung 7.9 trifft dies für $p \in (0;1)$ nur auf die Alternative mit negativem Vorzeichen vor der Wurzel zu. Somit kann nur Gleichung 7.12 Lösung von Gleichung 7.7 für $p \in (0;1)$ sein.

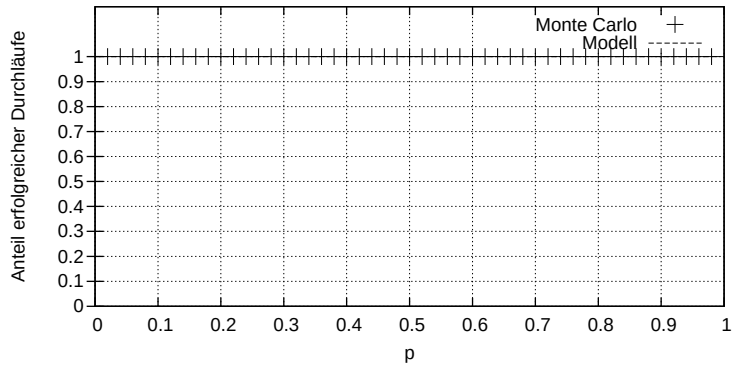
Für $p = 0$ und $p = 1$ entnehmen wir die Terminierungswahrscheinlichkeit direkt Gleichung 7.2 und erhalten Gleichung 7.13.

$$
\begin{aligned}
F_{Div}(q) &= \frac{1 - \sqrt{1-4qp\,(1-p)}}{2p} && (7.12)\\
P_{Div}(L<\infty) &= \frac{1-|1-2p|}{2p} && (7.13)\\
&= \begin{cases} 1 & \text{für} \quad p \in \left[0;\frac{1}{2}\right] \\ \frac{1-p}{p} & \text{für} \quad p \in \left(\frac{1}{2};1\right] \end{cases} && (7.14)
\end{aligned}
$$

Wir erkennen, dass die Terminierungswahrscheinlichkeit für $p > \frac{1}{2}$ abfällt und für $p \to 1$ gegen 0 geht. Während beim iterativen Verfahren ständig nur eine Anfrage offen ist und somit die Berechnung jederzeit mit Wahrscheinlichkeit $1-p$ zum Abschluss gebracht werden kann, müssen beim Divide&Conquer-Verfahren dafür alle zuvor eröffneten Teilanfragen abgeschlossen werden. Für $p > \frac{1}{2}$ wird im Mittel häufiger eine neue Anfrage durch den Rekursionsfall erzeugt als eine alte Anfrage durch den Basisfall abgeschlossen. Somit ist es möglich, dass das Verfahren in diesem Fall nicht terminiert.



(a) Divide&Conquer



(b) Iteratives Verfahren

Abbildung 7.2: Erfolgreiche Durchläufe, vollständig verbundener Graph

7.2.2 Monte-Carlo-Experimente

Wir validieren nun unsere Modelle durch Monte-Carlo-Experimente. Dafür wurde jeweils für mehrere in gleichen Abständen gewählte Werte von p im Intervall $(0; 1)$ eine große Anzahl von Durchläufen simuliert. Die Durchläufe waren auf eine vorgegebene Schrittzahl limitiert, um nichtterminierende Ausführungen näherungsweise zu erkennen. Für jedes p wurde der Anteil der nicht-abgebrochenen Durchläufe aufgezeichnet.

In Abbildung 7.2 sind die Ergebnisse der Simulation für den Graphen aus Abbildung 7.1a dargestellt. Die Abweichungen zwischen dem Modell und den Ergebnissen der Monte-Carlo-Simulation sind darauf zurückzuführen, dass letztere lediglich eine Näherung des Modells darstellt. Aufgrund dieser Ergebnisse dürfen wir unser Modell als validiert ansehen.

Konstruieren wir nun aus dem vollständig verbundenen Graphen einen partiell verbundenen Graphen, indem wir die Verbindungen der Knoten zu sich selbst entfernen. Aufgrund der fehlenden Kanten steht der Basisfall hier nicht immer zur

Verfügung. Dadurch wird die steigende Tendenz der Anfragenzahl gegenüber dem vollständig verbundenen Graphen verstärkt.

Wir betrachten hier den partiell verbundenen Graphen mit 2 Knoten und bestimmen erneut die theoretische Terminierungswahrscheinlichkeit, diesmal getrennt für Wege zwischen unterschiedlichen und für Wege zwischen gleichen Knoten. Auch hier sei erwähnt, dass es sich nicht um einen Kontrollflussgraphen handelt, da $e \to r$ gilt. Wie auch beim vollständig verbundenen Graphen ist dies für unsere Untersuchungen irrelevant.

Wir bezeichnen mit $F_{ab}(q)$ die Funktion entsprechend Gleichung 7.4 für eine Anfrage nach einem Weg von einen Knoten a zu einem Knoten b.

Wir nutzen die Symmetrie des Graphen und erkennen, dass $F_{ab}(q) = F_{cd}(q)$ genau dann, wenn $a = b \Leftrightarrow c = d$. Wir bestimmen deshalb nur $F_{ab}(q)$ und $F_{aa}(q)$ für zwei Knoten a und b mit $a \neq b$.

$$F_{aa}(q) = \frac{F_{aa}(q)^2 + F_{ab}(q)^2}{2} \tag{7.15}$$

$$F_{ab}(q) = (1-p)\,q + pF_{aa}(q)\,F_{ab}(q) \tag{7.16}$$

Da dies kubische Gleichungen in $F_{aa}(q)$ sind, bestimmen wir p und F_{ab} abhängig von F_{aa}, statt direkt nach F_{aa} und F_{ab} aufzulösen. Wir erhalten die Terminierungswahrscheinlichkeit erneut für $q = 1$.

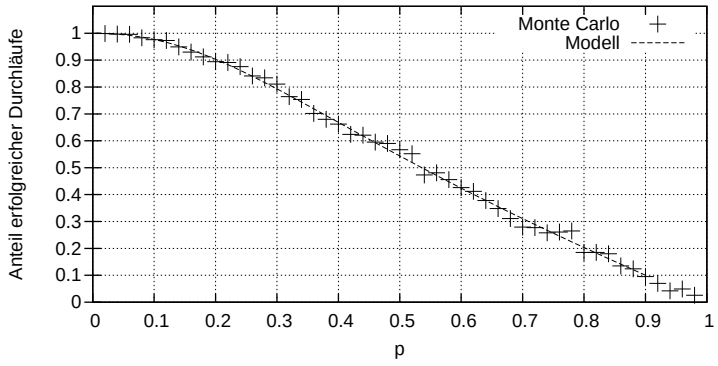
$$F_{ab}(1) = \pm\sqrt{2F_{aa}(1) - F_{aa}(1)^2} \tag{7.17}$$

$$p = \frac{1 - F_{ab}(1)}{1 - F_{aa}(1)\,F_{ab}(1)} \tag{7.18}$$

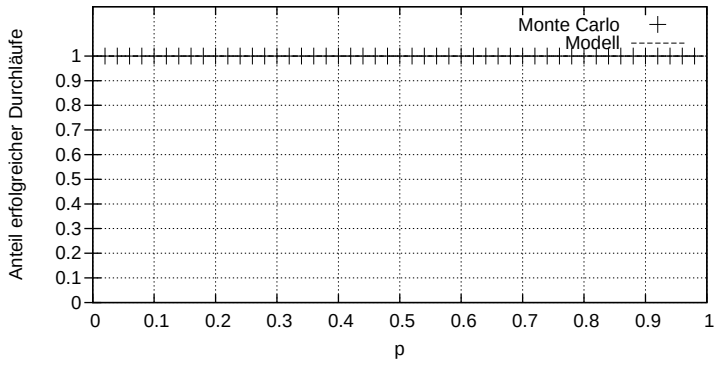
Da $F_{ab}(q)$ nicht negativ sein kann, kommt nur die Lösung mit positivem Vorzeichen in Frage. Der Nenner von Gleichung 7.18 wird nur 0, wenn $F_{aa}(1)\,F_{ab}(1) = 1$ gilt. Da $F_{aa}(1)$ und damit auch $F_{ab}(1)$ auf das Intervall $[0;1]$ beschränkt ist, kommt als Lösung nur $F_{aa}(1) = F_{ab}(1) = 1$ in Frage, was auch Gleichung 7.17 erfüllt.

Nach Gleichung 7.15 ist $F_{ab}(1) = 1$ für $p = 0$. Da sowohl $F_{ab}(1)$ wie auch p streng monoton in F_{aa} im Intervall $[0;1]$ sind, muss $p = 0$ die einzige Lösung für $F_{aa}(1) = F_{ab}(1) = 1$ sein.

Abbildung 7.3 zeigt die Ergebnisse der Monte-Carlo-Experimente für den partiell verbundenen Graphen mit zwei Knoten zusammen mit den Vorhersagen der jeweiligen theoretischen Modelle. Im Experiment wurde jeweils nach einem Weg



(a) Divide&Conquer



(b) Iteratives Verfahren

Abbildung 7.3: Erfolgreiche Durchläufe, partiell verbundener Graph

zwischen zwei verschiedenen Knoten gesucht. Wir beobachten, dass die Terminierungswahrscheinlichkeit bereits für $p < \frac{1}{2}$ schnell unter 1 sinkt. Dies ist eine deutliche Verschlechterung gegenüber dem vollständig verbundenen Graphen.

Für das iterative Verfahren hingegen zeigen sowohl Theorie als auch Monte-Carlo-Experiment durchgängig eine Terminierungswahrscheinlichkeit von 1.

Je größer die Zahl n der Knoten im Graphen ist, umso geringer wird der Anteil $\frac{1}{n}$ der Anfragen, für die der Basisfall nicht zur Verfügung steht. Dieser Anteil ist jedoch bei realistischen Programmflussgraphen deutlich höher, da ihre Kantenzahl meist linear statt quadratisch mit der Knotenzahl steigt. Entsprechend ist in der Praxis ein deutlich schlechteres Terminierungsverhalten zu erwarten.

Bei den weiteren Monte-Carlo-Experimenten mit dem iterativen Verfahren zur Validierung der mittleren Weglängen (s. Abschnitt 7.3) waren hingegen keine wesentlichen Abbrüche zu beobachten.

Lediglich bei Einbeziehung der Wegvorhersage ohne Schleifenkorrektur war ein Anstieg der Abbruchraten zu verzeichnen. Die Abbrüche wurden jedoch größten-

teils durch den starken Anstieg der mittleren Weglänge und der zugehörigen Streuung verursacht, wodurch die Schrittgrenze häufiger überschritten wurde.

Bei der Wegvorhersage ist die Anzahl der auf eine Anfrage nacheinander anwendbaren Spaltungsoperationen beschränkt durch die Anzahl der Knoten im Graphen. Sie kann sogar 0 sein, wenn keine Vorhersage für die Anfrage möglich ist. Im Unterschied dazu ist die Anzahl der Spaltungen beim reinen Divide&Conquer-Verfahren nicht limitiert sondern wird rein zufällig gesteuert.

Deshalb ist davon auszugehen, dass auch mit Wegvorhersage eine Terminierung in jedem Fall beim iterativen Verfahren sichergestellt ist.

7.2.3 Allgemeine Terminierung

Die Ausführungen in den vorangegangenen Abschnitten betreffen lediglich die Terminierung des Wegauswahlverfahrens unter der Annahme, dass alle Wege ausführbar sind. Wie wir bereits in Abschnitt 3.4.3 gezeigt haben, ist diese Annahme meist nicht zulässig. Wir müssen also noch die Terminierung für den allgemeineren Fall analysieren.

Ein CHR-Programm terminiert genau dann für eine bestimmte Eingabe, wenn es ausgehend von dieser Eingabe keine unendlichen Ableitungen gibt. Unser CHR-Programm kann keinesfalls für jede Eingabe terminieren. Sonst könnte es Lösungen für Aussagen der Form $x = (x_1, \ldots, x_n) \wedge x \ \mathcal{S}_{PFG} \ y$ finden und damit das Halteproblem entscheiden.

Eine unendliche CHR-Ableitung existiert insbesondere, wenn es im Programmflussgraphen unendliche Ausführungsfolgen gibt. Eine unendliche Ausführungsfolge ist eine unendliche Folge von Zuständen $(S_1, \ldots)$, so dass $S_i \mapsto_{PFG} S_{i+1}$ für alle $i \geq 1$ gilt. Einer solchen Ausführungsfolge kann insbesondere eine unendliche Folge von Anwendungen der Schrittregel zugeordnet werden.

Unendliche CHR-Ableitungen existieren aber auch dann, wenn der Programmflussgraph eine Schleife enhält, was in der Praxis in den meisten Programmen der Fall sein dürfte. Auch in diesem Fall lässt sich die Schrittregel unendlich wiederholt anwenden, vorausgesetzt, es werden dabei keine Widersprüche im entstehenden Wegconstraint erkannt.

Die Erkennung eines Widerspruchs setzt jedoch voraus, dass die CHR-Operation *Solve* angewendet wird. Nach der klassischen nichtdeterministischen operationellen Semantik von CHR[Frü98] sind jedoch auch Ableitungen zulässig, in denen Solve niemals angewendet wird.

Zu beachten ist auch, dass feste Vorgaben für die Reihenfolge der Regelanwendung dazu führen können, dass zum Beispiel bei einer Schleife niemals die Ausgangskanten ausgewählt werden. Die verfeinerte operationelle Semantik von CHR gibt eine solche Reihenfolge vor[DSBH04].

Demnach terminiert unser CHR-Programm nach der klassischen nichtdeterministischen operationellen Semantik grundsätzlich nicht auf Programmflussgraphen mit Schleifen.

Genauere Aussagen können wir für eine probabilistische Implementierung treffen. Ein probabilistisches CHR-Programm terminiert probabilistisch, wenn die Wahrscheinlichkeit für eine unendliche Ableitung 0 ist[FPW02].

Notwendige Voraussetzung hierfür ist, dass von jedem CHR-Zustand, der vom Startzustand erreichbar ist, ein Endzustand erreicht werden kann. Wir zeigen, dass dies nur dann der Fall ist, wenn die Länge der Ausführungsfolgen im Programmflussgraphen nicht beschränkt ist, die Solve-Operation mit nicht-verschwindender Wahrscheinlichkeit bei jedem Schritt angewendet wird und Widersprüche vollständig erkennt.

Diese Eigenschaft ist nicht hinreichend für die probabilistische Terminierung. Dies zeigen die Untersuchungen in Abschnitt 7.2.1: Das Divide&Conquer-Verfahren terminiert selbst dann nicht mit Wahrscheinlichkeit 1, wenn diese Voraussetzung wie im vollständig verbundenen Graphen gegeben sind.

Zu jedem CHR-Zustand gibt es zunächst eine endliche Zahl direkter Nachfolgezustände, die durch Anwendung verschiedener Instanzen der zutreffenden Regeln erreicht werden können. Nehmen wir sinnvollerweise an, dass jede dieser Regelinstanzen mit nicht-verschwindender Wahrscheinlichkeit angewendet wird. Somit fällt die Wahrscheinlichkeit, eine bestimmte CHR-Ableitung zu wählen, mit ihrer Länge gegen 0 ab, ist aber für jede endliche Länge größer als 0.

Gäbe es nun einen vom Startzustand erreichbaren Zustand, von dem aus nur unendliche Ableitungen möglich sind, so wäre die Wahrscheinlichkeit, diesen Zustand zu erreichen, ebenfalls nicht 0. Dann aber würde das Programm nicht probabilistisch terminieren.

In unserem CHR-Programm repräsentieren die Zustände Teile von Ausführungsfolgen. Keine der Regeln im Programm ist dabei in der Lage, diese Ausführungsfolgen zu verkürzen.

Jedem CHR-Zustand kann also eine Länge zugeschrieben werden, die jede von ihm ausgehend konstruierte Ausführungsfolge mindestens hat. Ist die einem CHR-Zustand zugeordnete Mindestlänge größer als die maximale Länge der Ausfüh-

rungsfolgen im Programm, so kann von diesem Zustand zumindest kein erfolgreicher Endzustand erreicht werden.

Da keine Ausführungsfolge der entsprechenden Mindestlänge existiert, muss der CHR-Zustand bereits einenWiderspruch enthalten. Nur wenn die Solve-Operation mit nicht-verschwindender Wahrscheinlichkeit bei jedem Schritt angewendet wird und Widersprüche vollständig erkennt, wird mit Wahrscheinlichkeit 1 nach endlicher Schrittzahl ein erfolgloser Endzustand erreicht.

In den bekannten Implementierungen von CHR wird die Solve-Operation immer dann angewendet, wenn ein neues vordefiniertes Constraint eingefügt wird. Insbesondere wird sie also sofort dann angewendet, wenn Constraints eingefügt werden, die im Widerspruch zum den bereits vorhandenen Constraints stehen.

Allerdings kann nicht für alle Constrainttheorien vollständig entschieden werden, ob eine endliche Menge von Constraints widersprüchlich ist[FA03].

Erkennt der Constraintlöser den Widerspruch, so hängt die Terminierung von der Wahl des nächsten Suchzweiges ab. Dabei kann es zu Endlosschleifen kommen, wie wir bei dem Beispiel in Abschnitt 7.5.3 zeigen.

Interessanterweise ist also insbesondere die Terminierung des Verfahrens für jene Programme kritisch, die garantiert terminieren, da die Länge ihrer Ausführungsfolgen beschränkt ist.

Schlussendlich ist eine Terminierung allerdings nur dann zu garantieren, wenn ein unvollständiges oder undefiniertes Ergebnis akzeptiert wird. Dann kann eine Limitierung der Schrittzahl oder der Laufzeit eingesetzt werden, um den Lauf nach endlicher Zeit abzubrechen.

7.2.4 Schlussfolgerungen

Das Divide&Conquer-Verfahren zeigt bereits dann ein unerwünschtes Terminierungsverhalten, wenn wir davon ausgehen, dass alle konstruierbaren Wege auch ausführbar sind. Während wir dem für den vollständig verbundenen Graphen noch durch Wahl von p im Intervall $\left[0; \frac{1}{2}\right)$ ausweichen können, ist die Termininierungswahrscheinlichkeit beim partiell verbundenen Graphen für jedes $p > 0$ ungleich 1. Für realistische Programmflussgraphen ist mit einem ebenso inakzeptablen Terminierungsverhalten zu rechnen.

Das iterative Verfahren jedoch terminiert für die betrachteten Fälle für jedes $p \in [0; 1)$ mit der Wahrscheinlichkeit 1.

Aus diesem Grund werden wir uns auch bei den folgenden Untersuchungen auf das iterative Verfahren und seine Erweiterung durch die Wegvorhersage konzentrieren.

Weiterhin haben wir das Terminierungsverhalten für den allgemeinen Fall untersucht, in dem auch nicht-ausführbare Wege möglich sind. Dabei haben wir festgestellt, dass das Verfahren nicht allgemein terminieren kann und insbesondere solche Programmflussgraphen kritisch sind, deren maximale Länge ausführbarer Wege beschränkt ist. Letzteres Ergebnis widerspricht der Intuition, da aufgrund der endlichen Menge ausführbarer Wege eine garantierte Terminierung zu erwarten wäre.

Wir haben bei der Untersuchung festgestellt, dass dem Constraintlöser für die Wegconstraints besondere Bedeutung zukommt, da die Terminierung auch wesentlich von dessen Fähigkeit abhängt, Widersprüche zu erkennen.

7.3 Weglängen

Wir untersuchen nun die Verteilung der Weglängen in Abhängigkeit vom Prozessparameter p und dem vorgegebenen Programmflussgraphen. Wir betrachten dafür jeweils Erwartungswert und Varianz der Weglänge. Wieder nehmen wir vereinfachend an, dass alle Wege ausführbar seien.

Zunächst nutzen wir theoretische Modelle zur Ableitung der betrachteten Parameter und ergänzen dies durch die Ergebnisse von Monte-Carlo-Experimenten mit weiteren Graphen.

Wir präsentieren insbesondere einige Anomalien, die ursprünglich bei Tests mit den Beispielen in Kapitel 7.5 entdeckt wurden. Die Effekte waren nicht als direkte Konsequenz der vereinfachenden Annahmen der theoretischen Modelle erkennbar, im Nachhinein jedoch mit Hilfe weiterer Modelle theoretisch erklärbar. Dadurch konnten ihre Ursache erkannt und durch Modifikationen des Verfahrens korrigiert werden.

7.3.1 Theoretisches Verhalten

Bereits in Abschnitt 7.2.1 haben wir die Funktion $F(q)$ gemäß Gleichung 7.4 für einen vollständig verbundenen Graphen hergeleitet.

Für die Ermittlung der erwarteten Weglänge und ihrer Varianz nutzen wir die Gleichungen 7.19 und 7.20. Für $q = 1$ erhalten wir jeweils den Erwartungswert $E(L)$ bzw. $E\left(L^2\right)$. Die Varianz erhalten wir durch Gleichung 7.21.

$$q\frac{d}{dq}F(q) = \sum_{l=1}^{\infty} lP(L=l)\,q^l \tag{7.19}$$

$$q\frac{d}{dq}q\frac{d}{dq}F(q) = \sum_{l=1}^{\infty} l^2P(L=l)\,q^l \tag{7.20}$$

$$V(L) = E\left(\left(L - E(L)\right)^2\right)$$

$$V(L) = E\left(L^2\right) + E(L)^2\left(P(L<\infty) - 2\right) \tag{7.21}$$

Mit Gleichung 7.5 erhalten wir die erwartete Weglänge in Gleichung 7.22 und ihre Varianz in Gleichung 7.23.

$$E(L) = \frac{1}{1-p} \tag{7.22}$$

$$V(L) = \frac{p}{(1-p)^2} \tag{7.23}$$

Für den partiell verbundenen Graphen aus Abbildung 7.1b bestimmen wir die Funktionen $F_{v,w}(q) = \sum_{l=1}^{\infty} P_{v,w}(L=l)\,q^l$, wobei $P_{v,w}(L=l)$ die Wahrscheinlichkeit ist, auf eine Anfrage nach einem Weg von v nach w einen solchen der Länge l zu erzeugen. Da die Wege von a nach b zu den Wegen von b nach a isomorph sind, müssen wir nur einen der beiden Fälle betrachten und erhalten Gleichung 7.26.

$$F_{a,b}(q) = (1-p)\,q + pqF_{b,b}(q) \tag{7.24}$$

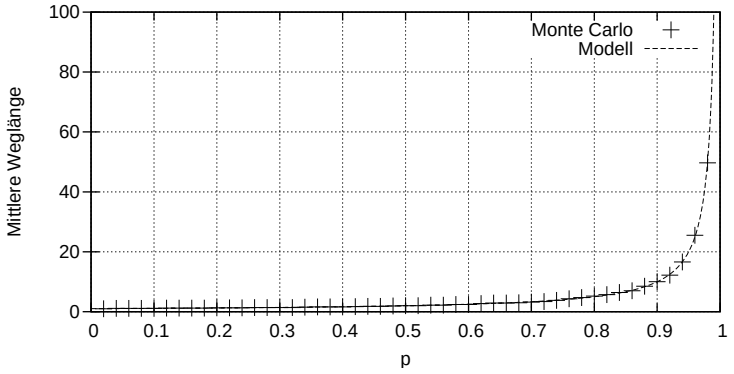
$$F_{b,b}(q) = qF_{a,b} \tag{7.25}$$

$$F_{a,b}(q) = \frac{(1-p)\,q}{1-pq^2} \tag{7.26}$$

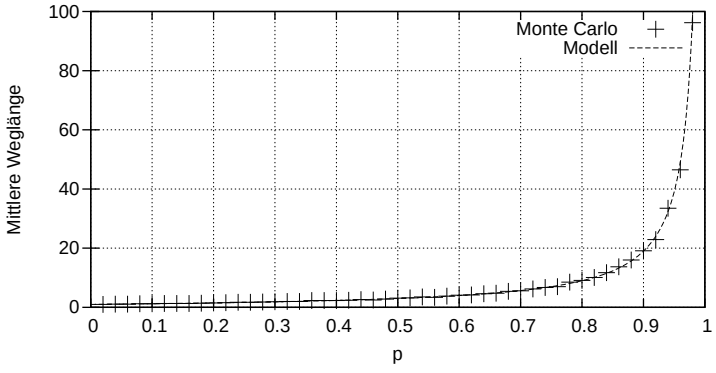
Für die erwartete Weglänge und ihre Varianz erhalten wir somit die Gleichungen 7.27 und 7.28. Die Vorhersagen des Modells und die Ergebnisse des zugehörigen Monte-Carlo-Experiments werden in Abbildung 7.4 gezeigt.

$$E(L) = \frac{1+p}{1-p} \tag{7.27}$$

$$V(L) = \frac{4p}{(1-p)^2} \tag{7.28}$$



(a) vollständiger Graph



(b) partieller Graph

Abbildung 7.4: Mittlere Weglänge – Monte-Carlo-Ergebnisse

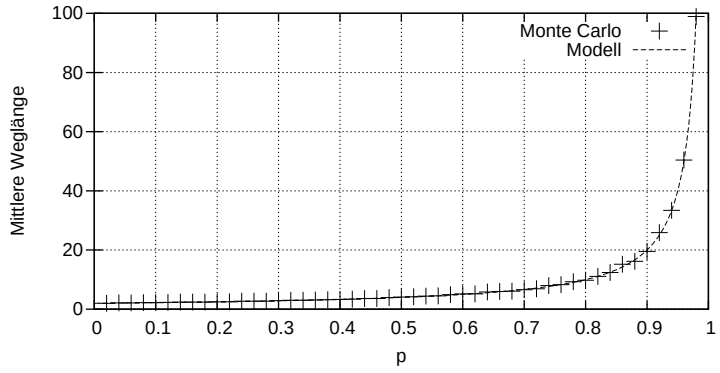


Abbildung 7.5: Mittlere Weglänge ggT

7.3.2 ggT-Algorithmus

Abbildung 7.5 zeigt die mittlere Weglänge des iterativen Verfahrens für eine Implementierung des ggT-Algorithmus. Der zugehörige Programmflussgraph ist in Abbildung 2.6 zu sehen. Bei der Messung war der Anteil der erfolgreichen Durchläufe durchgängig 1.

Betrachten wir das Wahrscheinlichkeitsmodell für diesen Graphen:

$$F_{r,e}(q) = qF_{1,e}(q) \tag{7.29}$$

$$F_{1,e}(q) = (1-p)\,q + pq\frac{F_{2,e}(q) + F_{3,e}(q)}{2} \tag{7.30}$$

$$F_{2,e}(q) = qF_{1,e}(q) \tag{7.31}$$

$$F_{3,e}(q) = qF_{1,e}(q) \tag{7.32}$$

Hieraus ergibt sich

$$F_{r,e}(q) = \frac{p(1-p)\,q^2}{1-pq^2} \tag{7.33}$$

$$P_{r,e}(L<\infty) = 1 \tag{7.34}$$

$$E_{r,e}(L) = \frac{2}{1-p} \tag{7.35}$$

Dieses Ergebnis unterscheidet sich qualitativ nur unwesentlich vom Ergebnis für den vollständigen und den partiell verbundenen Graphen. Der Vorfaktor 2 ist durch die Mindestlänge eines Schleifendurchlaufs bedingt.

7.3.3 Schleifen

In Abbildung 7.6 ist der Kontrollflussgraph eines Programms mit zwei ineinander verschachtelten natürlichen Schleifen zu sehen. Die äußere Schleife durchläuft die Knoten 1, 2 und 4 und die innere Schleife durchläuft die Knoten 2 und 3.

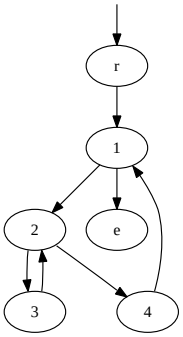


Abbildung 7.6: Verschachtelte Schleifen

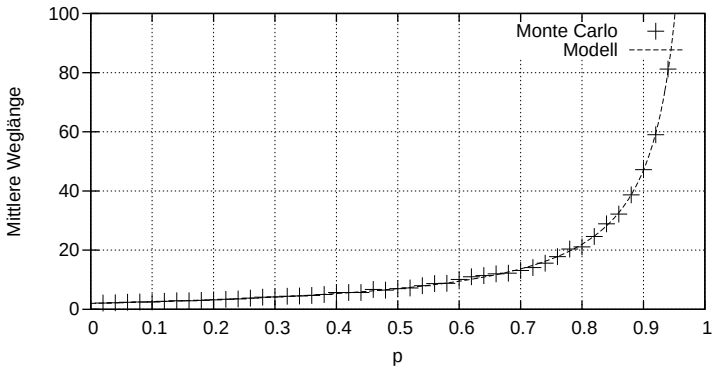


Abbildung 7.7: Mittlere Weglänge verschachtelte Schleifen

Die Ergebnisse des Monte-Carlo-Experiments sind in Abbildung 7.7 dargestellt.

Betrachten wir erneut das theoretische Modell:

$$F_{r,e}(q) = qF_{1,e}(q) \tag{7.36}$$

$$F_{1,e}(q) = (1-p)\,q + pqF_{2,e}(q) \tag{7.37}$$

$$F_{2,e}(q) = q\frac{F_{3,e}(q) + F_{4,e}(q)}{2} \tag{7.38}$$

$$F_{3,e}(q) = qF_{2,e}(q) \tag{7.39}$$

$$F_{4,e}(q) = qF_{1,e}(q) \tag{7.40}$$

Hieraus ergibt sich die erwartete Weglänge $E(L) = \frac{5}{1-p} - 3$. Der deutlich stärkere Anstieg der mittleren Weglänge im Vergleich zum optimistischen Modell ist also erneut durch die Mindestschleifenlänge zu erklären.

Diese mittlere Weglänge verteilt sich allerdings ungleichmäßig auf innere und äußere Schleife. Dies liegt daran, dass die Schleife um die Knoten 2 und 3 nicht durch den Basisfall abgeschlossen wird. Vielmehr wird auf dem Weg von 2 nach e einer

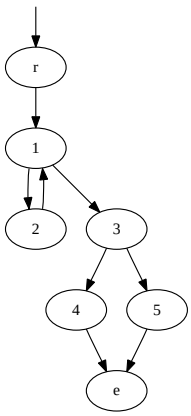


Abbildung 7.8: Schleife mit Verzweigung

der möglichen Nachfolger ausgewählt und zwar jeweils mit Wahrscheinlichkeit $\frac{1}{2}$. Die innere Schleife wird gewissermaßen so ausgeführt, als sei $p = \frac{1}{2}$. Somit ist die mittlere Weglänge der inneren Schleife 3, unabhängig vom gewählten p.

Ein ähnliches Phänomen beobachten wir beim Kontrollflussgraphen aus Abbildung 7.8. Es handelt sich um eine Schleife mit einer anschließenden Verzweigung. Auch hier wird im iterativen Verfahren ohne Wegvorhersage die Schleife nicht durch den Basis- sondern durch den Rekursionsfall abgeschlossen. Die daraus entstehende mittlere Weglänge ist 6, erneut unabhängig vom gewählten p.

7.3.4 Wegvorhersage

Beim Einsatz der Wegvorhersage werden neue Teilziele eingeführt, jedoch anders als beim Divide&Conquer-Verfahren in deterministischer Weise. Durch diese Aufteilung in Teilziele erhöht sich auch bei der Wegvorhersage die Anzahl der Möglichkeiten, einen Weg einer bestimmten Länge zu erzeugen.

Abbildung 7.3.4 zeigt die Ergebnisse einer Monte-Carlo-Simulation des iterativen Verfahrens mit Wegvorhersage zusammen mit der theoretischen Vorhersage aus dem iterativen Modell ohne Wegvorhersage. Wieder liegt der ggT-Algorithmus aus Abbildung 2.6 zugrunde. Wir beobachten einen deutlichen Anstieg der mittleren Weglänge gegenüber der beim iterativen Verfahren erwarteten Länge.

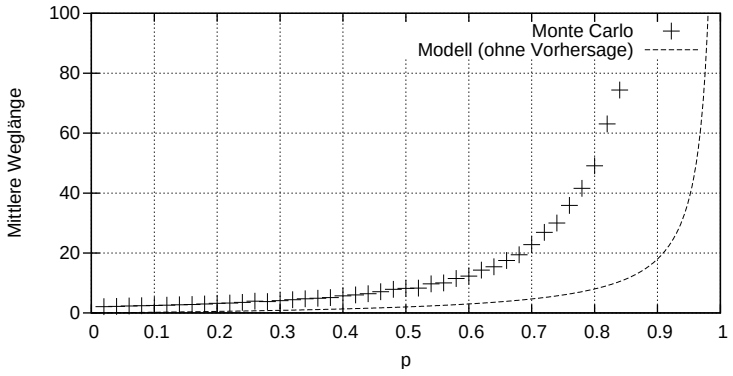


Abbildung 7.9: Monte-Carlo-Ergebnisse ggT mit Wegvorhersage – Weglänge

Wir konstruieren die Funktion $F_{r,e}(q)$ und berücksichtigen dabei die Auswirkungen der Wegvorhersage:

$$F_{r,e}(q) = F_{r,1}(q) F_{1,e}(q) \tag{7.41}$$

$$F_{r,1}(q) = (1-p) q + pq \frac{F_{2,1}(q) + F_{3,1}(q)}{2} \tag{7.42}$$

$$F_{2,1}(q) = (1-p) q + pq F_{1,1}(q) \tag{7.43}$$

$$F_{3,1}(q) = (1-p) q + pq F_{1,1}(q) \tag{7.44}$$

$$F_{1,1}(q) = q \frac{F_{2,1}(q) + F_{3,1}(q)}{2} \tag{7.45}$$

$$F_{1,e}(q) = (1-p) q + pq \frac{F_{2,e}(q) + F_{3,e}(q)}{2} \tag{7.46}$$

$$F_{2,e}(q) = F_{2,1}(q) F_{1,e}(q) \tag{7.47}$$

$$F_{3,e}(q) = F_{3,1}(q) F_{1,e}(q) \tag{7.48}$$

Für die Terminierungswahrscheinlichkeit erhalten wir 1 für $p < 1$. Die erwartete Weglänge und ihre Varianz sind durch die Gleichungen 7.49 und 7.50 gegeben.

$$E(L) = \frac{2}{(1-p)^2} - p \tag{7.49}$$

$$V(L) = \frac{4p(2-p)}{(1-p)^4} - 3p - p^2 \tag{7.50}$$

Wir sehen, dass sowohl die mittlere Weglänge als auch die Varianz deutlich stärker ansteigen als ohne Wegvorhersage. Die Abweichung kommt durch die Aufteilung der Anfrage nach einem Weg von r nach e in zwei Teilanfragen von r nach 1 und von 1 nach e zustande. In beiden Teilen kann die Schleife um Knoten 1 mehrfach durchlaufen werden.

Wie wir an Gleichung 7.47 sehen können, werden außerdem bei jedem Durchlauf der Schleife im Teilziel von 1 nach e zwei neue Teilziele erzeugt, eines für einen

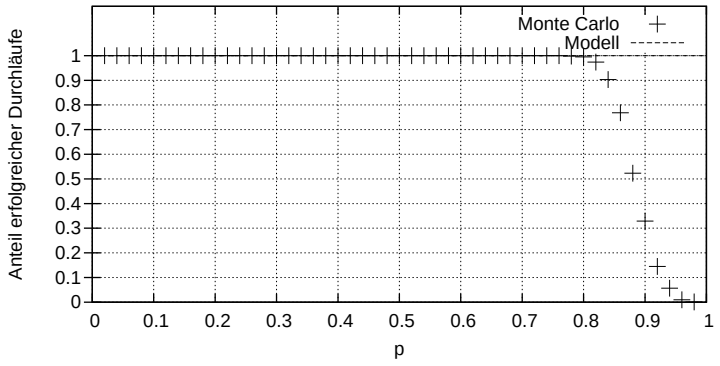


Abbildung 7.10: Iterativ mit Wegvorhersage, verschachtelte Schleifen – Anteil erfolgreicher Ableitungen

Weg von 2 bzw. 3 nach 1 und eines erneut für einen Weg von 1 nach e. In beiden neuen Teilzielen kann erneut die Schleife mehrfach durchlaufen werden.

Mit zunehmender Schachtelungstiefe von Schleifen im Programmflussgraph verstärkt sich dieser Effekt erheblich. Zwar ist die theoretische Terminierungswahrscheinlichkeit sowohl beim ggT-Beispiel als auch bei den verschachtelten Schleifen aus Abbildung 7.6 immer 1.

In der Praxis jedoch wirkt sich der Anstieg der mittleren Weglänge und die damit einhergehende stärkere Streuung der Weglängen deutlich auf den Anteil der aufgrund Zeitüberschreitungen abgebrochenen Ableitungen aus. In Abbildung 7.10 sind diese Auswirkungen für das Beispiel der verschachtelten Schleife aus Abbildung 7.6 deutlich zu sehen.

7.3.5 Schlussfolgerungen

Die Experimente haben gezeigt, dass sich die Ergebnisse des vollständig verbundenen Graphen durchaus auf realistische Kontrollflussgraphen übertragen lässt. In einigen Fällen, die die Einbettung von Schleifen in Programme betreffen, haben die Experimente jedoch unerwünschte Abweichungen von diesem Modell.

Die Modelle zeigen, dass das Verfahren tendenziell kürzere Wege längeren bevorzugt. Dies ist im Sinne einer repräsentativen Auswahl von Testeingaben problematisch. Allerdings besteht ein einfacher Zusammenhang zwischen Erwartungswert und Streuung der Weglänge einerseits und dem Verfahrensparameter p andererseits. Somit kann dieses Ungleichgewicht der Weglängen durch Variation von p im Intervall $(0; 1)$ praktisch ausgeglichen werden.

Der unerwünschte Anstieg der Streuung und des Mittelwerts der Weglänge kann vermieden werden, indem bei der Wegvorhersage in einem Teil des vorhergesagten Weges der mehrmalige Durchlauf des Spaltungsknotens untersagt wird.

Die unzureichende Kontrolle über die Anzahl der Durchläufe von Schleifen, die nicht durch den Basisfall beendet werden, ist zu beheben, indem zwischen den möglichen Nachfolger- oder Vorgängerknoten unterschiedlich gewichtet wird.

Dazu werden die Kandidaten danach sortiert, ob bei ihrer Wahl eine Schleifenaus- bzw. -eingangskante durchlaufen würde oder nicht. Sind sowohl Kandidaten vorhanden, durch die die Schleife beendet wird, als auch solche, durch die sie fortgesetzt wird, so soll mit Wahrscheinlichkeit p einer der letzteren und mit Wahrscheinlichkeit $1 - p$ einer der ersteren gewählt werden.

Dann unterliegt auch der Abschluss einer Schleife im Allgemeinen der Kontrolle durch p.

7.4 Analyse der laufzeitbestimmenden Faktoren

Wir analysieren die Laufzeitkomplexität einer Implementierung des Verfahrens mit optimierten Datenstrukturen. Die dabei ermittelte Komplexität ist aus zwei Gründen nur bedingt mit der des Prototypen vergleichbar.

Erstens entscheidet der CHR-Compiler bei der Übersetzung des Prototypen, welche Datenstrukturen für die Verwaltung der Constraints eingesetzt werden. Diese Datenstrukturen können sich je nach Compiler und je nach Compileroptionen unterscheiden. Das Verfahren soll allgemein untersucht und bewertet werden, unabhängig von der gewählten Implementierungssprache und dem verwendeten Compiler.

Zweitens müssen wir in der theoretischen Analyse von jenem Laufzeitaufwand abstrahieren, der für die Verarbeitung der `body`- und `cond`-Constraints benötigt wird. Diese hängt wesentlich vom verwendeten Constraintlöser ab und ist nicht allgemein zu beschreiben. Dieser Löser für die Basisconstraints ist ebenso nicht Teil des Beitrags dieser Arbeit und soll deshalb nicht Bestandteil der allgemeinen Überlegungen zur Laufzeitkomplexität sein.

Wir diskutieren deshalb lediglich den Zusatzaufwand, der dadurch entsteht, dass der Weg nicht direkt, sondern von unserem Verfahren erzeugt wird. Bezugspunkt ist also der Aufwand, der allein durch das Einfügen der `body/3`- und `cond/3`-Constraints entsteht.

Dieser Vergleichsaufwand ist sicherlich abhängig von der Reihenfolge, in der die Constraints eingespeist werden. Dies wird klar, wenn man eine Folge von Constraints $(x_1 = 1, x_2 = 1, x_i = x_{i-2} + x_{i-1}, \ldots, x_n = x_{n-2} + x_{n-1})$ betrachtet, durch die die Fibonacci-Folge berechnet wird. Werden sie in der oben gegebenen Reihenfolge eingespeist, in der auch ein Programm die Folge berechnen würde, so kann der Wert der x_i jeweils direkt berechnet werden und das gerade eingefügte Constraint $x_i = x_{i-2} + x_{i-1}$ sofort wieder entfernt werden.

Fügen wir die Constraints jedoch in umgekehrter Reihenfolge ein, so können die Constraints erst dann gelöst werden, wenn alle Constraints eingefügt wurden. Die Größe des Constraintstore wächst an und für jedes neu eingefügte Constraint wird zunächst nach möglichen Partnern für eine Lösung gesucht.

Der Aufwand für diese Suche wächst gegebenenfalls mit der Größe des Constraintstore, so dass wir im ersten Fall einen konstanten Suchaufwand, im zweiten Fall einen ansteigenden Suchaufwand haben.

Da insbesondere durch die Zufallselemente und die Vorhersageoperationen die Reihenfolge der Constraints verändert wird, ist dieser Vergleichswert also nicht für jede der Varianten unseres CHR-Programms gleich. Konkret beschreiben können wir ihn dennoch nicht, da er abhängig vom Constraintlöser sowie dem CHR-Interpreter oder -Compiler ist.

Dazu untersuchen wir den Aufwand für einen einzelnen Verfahrensschritt abhängig von der Ausprägung des Verfahrens. Wir zeigen, dass dieser Aufwand nur von der Größe des Programmflussgraphen abhängt, nicht jedoch von der Zahl der vorhergehenden Schritte.

Der gesamte Laufzeitbeitrag des Verfahrens als Summe der Laufzeiten der Einzelschritte wächst somit linear mit der Zahl der Schritte. Letztere ist insbesondere durch die Länge des erzeugten Weges, aber auch durch Zahl und Tiefe notwendiger Backtracking-Operationen bestimmt. Beides hängt von Struktur und Inhalt des Programmflussgraphen ab, die Länge zusätzlich noch vom Prozessparameter p.

Insbesondere ist also die Länge nicht Bestandteil der Eingabe und somit nicht vor dem Ende eines Durchlaufs bekannt. Sie kann lediglich quantitativ abgeschätzt werden, wie wir ja bereits in Abschnitt 7.3 gezeigt haben.

7.4.1 Datenstrukturen

Wir benötigen Datenstrukturen zur Darstellung des Programmflussgraphen und der Menge der noch nicht abgearbeiteten `ispec`-Constraints.

Als Datenstruktur für die Speicherung der `ispec`-Constraints wählen wir ein Feld $(c_i)_{1 \leq i \leq n}$, wobei n die Anzahl der Elemente ist, die durch Einfügen oder Entfernen von Elementen verändert werden kann. Wir werden später zufällig ein $i \in \{1, \ldots, n\}$ wählen und können dann in $O(1)$ auf das entsprechende Constraint zugreifen.

Um ein Element c_i zu entfernen, vertauschen wir c_i und c_n und verkleinern n um 1. Um ein Element einzufügen, muss zuerst der zur Verfügung stehende Speicherplatz vergrößert werden, wobei ggf. das gesamte Feld im Speicher umkopiert werden muss. Dann wird das neue Element als c_{n+1} eingefügt und n um 1 vergrößert.

Das Entfernen geschieht also in $O(1)$, während das Einfügen worst-case $O(n)$ aber amortisiert $O(1)$ benötigt, sofern wir bei jedem Einfügen den verfügbaren Speicherplatz um ein Vielfaches der aktuellen Größe vergrößern.

Weiterhin müssen wir Informationen über den Programmflussgraphen darstellen, die wir vorberechnen können. Dazu gehört die Kantenrelation, die Erreichbarkeitsrelation sowie die Dominanz- bzw. Postdominanzrelation und für jeden Knoten der jeweilige direkte Dominator bzw. Postdominator.

Ebenso benötigen wir für jede Variable die Liste der Definitionsknoten sowie die Liste der Knoten, in denen die Variable lebendig ist. Eine Variable v ist in einem Knoten n lebendig, wenn n auf einem definitionsfreien Weg von einer Definition von v zu einer Nutzung von v liegt.

Wir bezeichnen die Knoten des Graphen mit fortlaufenden Nummern, die wir als Feldindices verwenden können.

Da wir den Programmflussgraphen aus dem Basisblockgraphen konstruieren, hat keiner der Knoten mehr als zwei Nachfolger: Bei einem unbedingten Sprung existiert nur ein Nachfolger, bei einem bedingten Sprung existiert der Nachfolger, der erreicht wird, wenn der Sprung ausgeführt wird, und der Nachfolger, der bei Nichtausführung betreten wird. Somit können wir die Kantenrelation als einfaches Feld e_{ij} mit jeweils zwei Einträgen für jeden Knoten i darstellen. Dieses Feld kann auch die Information enthalten, ob $i \to j$ ein Schleifenausgang ist.

Für die Erreichbarkeitsrelation gibt es lediglich die Anzahl $\|N\|$ der Knoten im Graphen als obere Grenze. Wir ordnen deshalb jedem Knoten i ein Feld s_{ij} zu mit $1 \leq j \leq n_i \leq \|N\|$, wobei n_i die Anzahl der von i aus erreichbaren Knoten ist und s_{ij} einen von i aus erreichbaren Knoten bezeichnet. Wir nehmen an, dass die s_{ij} untereinander sortiert sind, also

$$\forall i \in \{1, \ldots, \|N\|\}, j, k \in \{1, \ldots, n_i\} : j \leq k \Leftrightarrow s_{ij} \leq s_{ik}$$

gilt. Ebenso speichern wir in p_{ij} jene Knoten, von denen aus i erreichbar ist.

Wir speichern zu jedem Knoten i seinen direkten Dominator $idom\,(i)$ bzw. Postdominator $ipostdom\,(i)$ in einem Feld. Die Dominator- und Postdominatorrelationen bilden also jeweils einen Baum, den wir durch Felder $dom_i.pre$ und $dom_i.post$ bzw. $postdom_i.pre$ und $postdom_i.post$ darstellen können. Wir können $dom_i.pre$ und $dom_i.post$ so wählen, dass $i\ dom\ j$ genau dann gilt, wenn $dom_i.pre < dom_j.pre$ und $dom_j.post < dom_i.pre$. Die Werte von $dom_i.pre$ erhalten wir durch fortlaufende Nummerierung der Knoten in einem Pre-Order-Durchlauf, die Werte von $dom_i.post$ in einem Post-Order-Durchlauf.

Wir müssen außerdem die Datenflussinformationen für die Variablenvorhersage speichern. Die Variablen seien ebenso wie die Knoten durchnummeriert. Zu jedem Knoten i speichern wir die Variablen d_{il}, die in i definiert werden. Dabei gelte $1 \leq l \leq v_i \leq \|V\|$, wobei V die Menge der Variablen im untersuchten Programm ist. Weiterhin seien die Variablen für jeden Knoten nach ihrem Index sortiert. Es gilt also

$$\forall i \in \{1, \ldots, \|N\|\}, j, k \in \{1, \ldots, v_i\} : j \leq k \Leftrightarrow d_{il} \leq d_{ik}$$

7.4.2 Laufzeit ohne Vorhersage

Wir setzen das Verfahren aus Abbildung 6.3 um und verwenden Vorwärts- oder Rückwärtsschritte statt dem Rekursionsfall.

Für die zufällige Auswahl eines `ispec/5`-Constraints benötigen wir üblicherweise $O\,(1)$, da sowohl die Auswahl einer Zufallszahl i mit $1 \leq i \leq n$ wie auch der Zugriff auf das entsprechende Element des Feldes c_i diese Laufzeit erfordern.

Die Prüfung, ob der Basisfall anwendbar ist, erfordert ebenfalls $O\,(1)$, da mit Hilfe von e_{aj} einfach bestimmt werden kann, ob $a \to b$ gilt.

Für einen Vorwärtsschritt erfordert die Sammlung der möglichen Nachfolgerknoten eine Laufzeit von $O\,(\log \|N\|)$. Für jeden der Nachfolger c von a – maximal zwei – wird geprüft, ob von ihm aus b erreichbar ist. Dies geschieht entweder über p_{bj} oder über s_{ck}, wobei durch die Sortierung maximal $\log \|N\|$ Schritte notwendig sind.

Für einen Rückwärtsschritt müssen alle Vorgänger von b darauf geprüft werden, ob sie von a aus erreichbar sind. Die erfordert eine Laufzeit von $O\,(\|N\|)$, da jeder Knoten nicht maximal 2 sondern maximal $\|N\|$ Vorgänger haben kann, die Listen der Vorgänger und der von a aus erreichbaren Knoten aber sortiert sind.

Die zufällige Entscheidung zwischen Basisfall und Schritt sowie die Auswahl eines der Nachfolgerknoten ist wieder in $O\,(1)$ möglich.

Der Basisfall wird ebenso wie ein Vorwärts- oder Rückwärtsschritt ohne Weg- und Datenflussvorhersage in $O(1)$ ausgeführt, da dies die Laufzeit für das Entfernen aus bzw. Ersetzen eines Eintrages im Constraintstore ist.

Die Laufzeit für einen einzelnen Durchlauf der Hauptschleife ist somit $O(\log \|N\|)$ nur mit Vorwärtsschritten und $O(\|N\|)$ mit Rückwärtsschritten. Die asymptotische Laufzeitkomplexität für gemischte Durchläufe ist ebenso $O(\|N\|)$.

7.4.3 Laufzeit mit Wegvorhersage

Bei der Wegvorhersage kommt für jedes durch einen Vorwärts- oder Rückwärtsschritt erzeugte `ispec/5`-Constraint Aufwand hinzu. Dank der Datenstrukturen für Dominator- bzw. Postdominatorbaum können wir in $O(1)$ prüfen, ob die Vorhersage anwendbar ist und erhalten dabei sofort den Spaltungsknoten.

Eines der neu entstandenen Constraints können wir sofort in den Constraintstore eintragen, da es nicht mehr weiter aufspaltbar ist. Das andere spalten wir entsprechend weiter auf, sofern weitere Spaltungsknoten zur Verfügung stehen. Dies geschieht maximal $h-2$ mal, wobei h die Höhe des Dominator- bzw. Postdominatorbaumes ist.

Für das Einfügen des neu entstandenen Constraints benötigen wir amortisiert $O(1)$, so dass für die Wegvorhersage amortisiert $O(h)$ pro Schritt benötigt wird.

Dem steht allerdings gegenüber, dass die durch die Spaltung eingefügten Knoten nicht noch durch einen Durchlauf der Hauptschleife eingefügt werden müssen, so dass wir dort maximal $h-2$ Schritte einsparen.

7.4.4 Laufzeit mit Variablenvorhersage

Bei jedem neuen `ispec/5`-Constraint bestimmen wir die Menge der in b lebendigen Variablen, die auf keinem Weg von a nach b verändert werden.

Dazu bestimmen wir zunächst die Menge der Knoten, die auf einem Weg von a nach b liegen in aufsteigender Reihenfolge des Knotenindex. Dies ist in einer Laufzeit von $O(\|N\|)$ möglich, da die s_{aj} und die p_{bk} bereits sortiert sind.

Wir bestimmen nun die Menge der Variablen, die in mindestens einem dieser Knoten definiert werden. Dies geschieht in einer Laufzeit von $O(\|N\| \, \|V\|)$, da die d_{il} ja bereits sortiert sind.

Für jede der in b lebendigen Variablen v prüfen wir nun, ob sie in dieser Menge der definierten Variablen enthalten ist. Da die lebendigen Variablen ebenso sortiert sind, können wir dies in einer Laufzeit von $O(\|V\|)$ bewerkstelligen.

Diese Informationen können auch vorberechnet werden. Der Speicherbedarf dafür ist $O\left(\|N\|^2 \|V\|\right)$, da jedes Paar (a, b) von Knoten berücksichtigt werden muss.

Für jede der Variablen, die zwischen a und b nicht geschrieben werden können, müssen wir vordefinierte Constraints hinzufügen, die die Werte dieser Variablen in `X` und `Y` gleichsetzen.

7.4.5 Laufzeit mit Knotenvermeidung

Für definitionsfreie Wege haben wir die Möglichkeit eingeführt, mit jedem `ispec`-Constraint eine Liste von Knoten mitzuführen, die nicht durchlaufen werden dürfen.

Dies beeinflusst das Programm an zwei Stellen: Bei den Schrittregeln und bei der Wegvorhersage. Hier darf kein Knoten als Spaltungsknoten, Vorgänger oder Nachfolger ausgewählt werden, der selbst in der Menge der zu vermeidenden Knoten liegt.

Führen wir die zu vermeidenden Knoten nach Knotenindex sortiert mit, so ist die Entscheidung, ob ein Knoten in dieser Menge enthalten ist, in $O(\log \|N\|)$ möglich. Das Erzeugen eines neuen Constraint mit einem zusätzlichen zu vermeidenden Knoten erfordert im schlimmsten Fall einen Aufwand von $O(\|N\|)$.

Bei der Wegvorhersage können wir erkennen, wenn der Spaltungsknoten in der Menge der zu vermeidenden Knoten enthalten ist und dementsprechend das Backtracking einleiten. Ist er nicht enthalten, so müssen wir jeweils für eines der neuen Constraints die Menge kopieren und den Spaltungsknoten als zu vermeidenden Knoten einfügen.

Bei der Auswahl eines Vorgängers bzw. Nachfolgers schließen wir jene aus, die in der Menge der zu vermeidenden Knoten enthalten sind. Verbleibt kein Kandidat, so kann kein Schritt ausgeführt werden. Ist auch der Basisfall nicht anwendbar, so wird das Backtracking eingeleitet.

Die Bestimmung der Kandidaten mit Knotenausschluss erfordert beim Vorwärtsschritt die Laufzeit $O(\log \|N\|)$, da für maximal zwei Knoten geprüft werden muss, ob sie in der Ausschlussmenge enthalten sind und von ihnen aus der Zielknoten erreichbar ist.

Beim Rückwärtsschritt können die Kandidaten in einer Laufzeit von $O(\|N\|)$ bestimmt werden, da alle beteiligten Knotenlisten sortiert sind.

7.4.6 Schlussfolgerungen

Aus den vorangegangenen Betrachtungen ergibt sich für jeden ohne Vorhersagen eingefügten Knoten eine Laufzeit von $O(\log \|N\|)$ ohne bzw. $O(\|N\|)$ mit Rückwärtsschritten.

Jeder durch Knotenvorhersage eingefügte Knoten erfordert hingegen $O(1)$. Mit der Variablenvorhersage kommt pro Knoten ein Aufwand von $O(\|N\| \, \|V\|)$ hinzu.

Bei Einsatz der Knotenvermeidung fällt für jeden vorhergesagten Knoten einen zusätzlicher Aufwand von $O(\|N\|)$, für jeden nicht vorhergesagten Knoten von $O(\log \|N\|)$ an.

Hinzu kommt jeweils der Aufwand für die Verarbeitung der vordefinierten Constraints sowie der durch Backtracking verursachte Aufwand. Dieser ist jedoch stark abhängig vom verwendeten Constraintlöser sowie der Struktur des Programmflussgraphen. Über diesen Aufwandsanteil kann nur schwer eine allgemeine Aussage getroffen werden.

7.5 Beispiele

Da die theoretische Erfassung der Eigenschaften nur schwer für den allgemeinen Fall möglich ist, betrachten wir nun einige praktische Beispiele.

Wir untersuchen die Unterschiede im Laufzeitverhalten zwischen den Verfahrensvarianten ohne und mit Weg- und Datenflussvorhersage, jeweils kombiniert mit vorwärtsgerichteter, rückwärtsgerichteter oder gemischter Durchlaufrichtung.

Die Wegvorhersage findet jeweils auf Grundlage der Dominatoren- und Postdominatorenrelation wie in Abschnitt 5.3 beschrieben statt.

Bei rückwärtsgerichtetem Durchlauf werden außer dem Basisfall nur Rückwärtsschritte, bei vorwärtsgerichtetem Durchlauf nur Vorwärtsschritte angewendet.

Bei gemischter Durchlaufrichtung werden jeweils alle möglichen Vorwärts- und Rückwärtsschritte für einen Teilweg gesammelt und entsprechend der Eigenschaften der durchlaufenen Kanten gewichtet zufällig ausgewählt.

Bei einem Vorwärtsschritt wird eine Schleifenausgangskante mit $1 - p$ gewichtet, alle anderen Kanten mit p gewichtet. Entsprechend werden bei einem Rückwärtsschritt Schleifeneingangskanten mit $1 - p$ und alle anderen Kanten mit p gewichtet.

Gemessen wurde jeweils die Laufzeit des Verfahrens bei unterschiedlichen Werten von p im Intervall $\left(\frac{1}{2}; 1\right)$ im Abstand von $\frac{1}{20}$ mit jeweils 100 Messpunkten für jeden

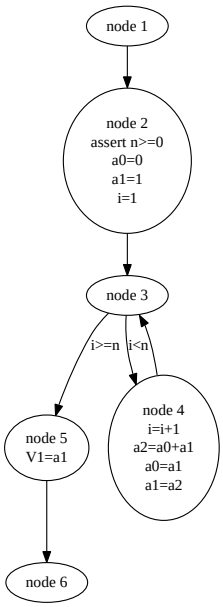


Abbildung 7.11: Berechnung der Fibonacci-Folge

der Werte von p. Das Intervall $\left(0; \frac{1}{2}\right]$ haben wir nicht untersucht, da dort nur geringe mittlere Weglängen zu erwarten sind.

Aufgenommen wurde die Laufzeit und der Anteil der erfolgreichen Durchläufe, wobei ein erfolgreicher Durchlauf dadurch definiert ist, dass er nicht länger als eine vorgegebene Zeitschranke benötigt hat. Bei Überschreiten der Zeitschranke wurde ein Durchlauf abgebrochen. Sowohl abgebrochene als auch erfolgreiche Durchläufe gelten als Messpunkte.

Zur Laufzeit eines Durchlaufs wird dabei nicht die Zeit gezählt, die zur Vorberechnung der Daten aus dem Programmflussgraphen benötigt wird. Für diese Berechnungen sind die Laufzeitkomplexitäten guter Algorithmen bekannt[CFR91, LT79].

7.5.1 Berechnung der Fibonacci-Folge

Der Algorithmus zur Berechnung der Fibonacci-Folge in Abbildung 7.11 zeichnet sich dadurch aus, dass jeweils genau ein Weg zu jeder Eingabe n gehört. Insbesondere existieren keine nicht-ausführbaren Wege, weshalb kein Laufzeitvorteil durch die Früherkennung und Vermeidung solcher Wege aufgrund der Weg- und Datenflussvorhersage zu erwarten ist.

Abbildung 7.12 zeigt das Laufzeitverhalten einer Implementierung, die den Programmflussgraphen direkt auf dem Finite-Domain-Constraintlöser ausführt.

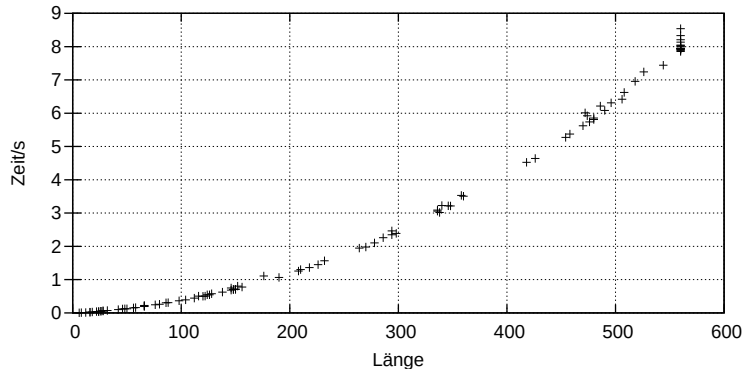


Abbildung 7.12: Fibonacci-Folge – Laufzeit der direkten Implementierung

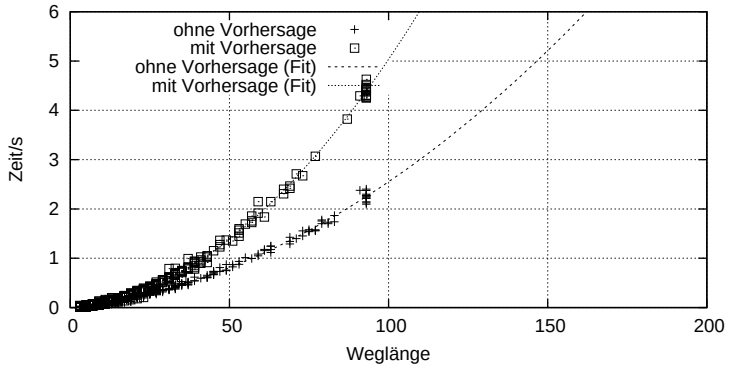


Abbildung 7.13: Fibonacci-Folge – Laufzeiten, vorwärts

Der quadratische Anteil der Laufzeitentwicklung ist darauf zurückzuführen, dass in jedem Schleifendurchlauf der Wert der Variable n mindestens einmal gelesen wird. Diese Variable wird aber nie im Programm geschrieben, so dass jedes Mal die gesamte Kette von Speicherzuständen bis zum Eingabezustand durchlaufen werden muss, um den Wert des Parameters zu finden. Die Länge dieser Kette wächst linear mit der Länge des Weges und somit ergibt sich ein quadratischer Gesamtbeitrag. Durch Caching im Constraintlöser kann dieser Anteil auf ein mit der Weglänge lineares Maß reduziert werden, auf Kosten eines höheren Speicheraufwands, ebenso durch Darstellung des Speicherzustands als Tupel der Variablen (s. Abschnitt 6.2).

Im Vergleich dazu zeigt Abbildung 7.13 die Laufzeiten des Verfahrens mit und ohne Vorhersage, jeweils mit einer durch Regression ermittelten Funktion $f(l) = al^2 + bl + c$ unterlegt. Die Anfrage $x \; \mathcal{S}_{PFG} \; y$, deren Laufzeit gemessen wurde, entspricht der simulierten Ausführung des Programms.

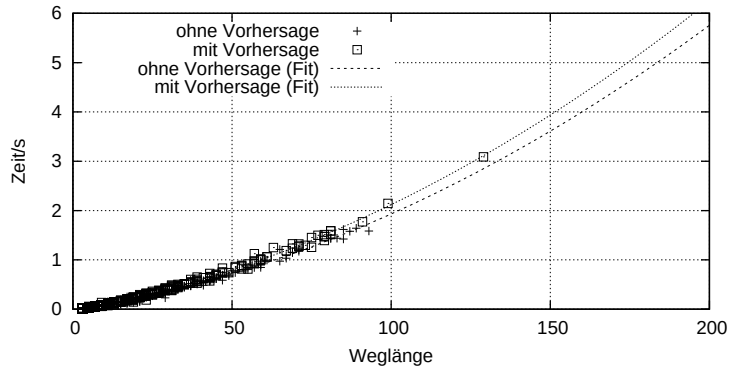


Abbildung 7.14: Fibonacci-Folge – Laufzeiten, rückwärts

Wir sehen, dass die Laufzeiten ohne Vorhersage deutlich langsamer ansteigen als mit Vorhersage. Durch die Vorhersage werden Teile der Fibonacci-Folge nicht in der vorgesehenen Reihenfolge berechnet, was zu den in Abschnitt 7.4 bereits angesprochenen Nachteilen führen kann. Hier kann auch die Variablenvorhersage nur wenig helfen, da für die Knoten in der Schleife lediglich die Variable `n` konstant ist.

Die verstärkte Streuung bei der jeweiligen im Diagramm maximalen Länge ist darauf zurückzuführen, dass bei dieser Länge die Werte der Folge den Wertebereich eines Java-`int` überschreiten, so dass es hier gehäuft zu Backtracking kommt. Mit Vorhersage können unabhängig voneinander mehrere Teilwege zusammengesetzt werden, die zusammen zur Überschreitung führen, für sich allein betrachtet jedoch nicht. Dies wird erst dann erkannt, wenn die einzelnen Wege abgeschlossen werden, so dass in diesem Fall die Abbruchentscheidung sogar erst später getroffen werden kann.

Hier könnte mit der Datenflussvorhersage teilweise Abhilfe geschaffen werden, wenn man erkennt, dass die Werte von `a0` und `a1` nicht kleiner werden.

Die Laufzeiten für die rückwärtsgerichtete Ausführung aus Abbildung 7.14 zeigen keine solch große Differenz. Die Kette der Speicherzustände wird hier von hinten nach vorne aufgebaut, so dass nicht bei jedem Schritt die vollständige Kette sondern jeweils nur der neu eingeführte Teil durchlaufen werden muss.

Dies wird größtenteils durch den höheren Aufwand für die umgekehrte Reihenfolge der Constraints entsprechend Abschnitt 7.4 kompensiert, so dass der Rückwärtsdurchlauf kein wesentlich anderes Laufzeitverhalten zeigt als der Vorwärtsdurchlauf, wie der direkte Vergleich in Abbildung 7.15 deutlich macht.

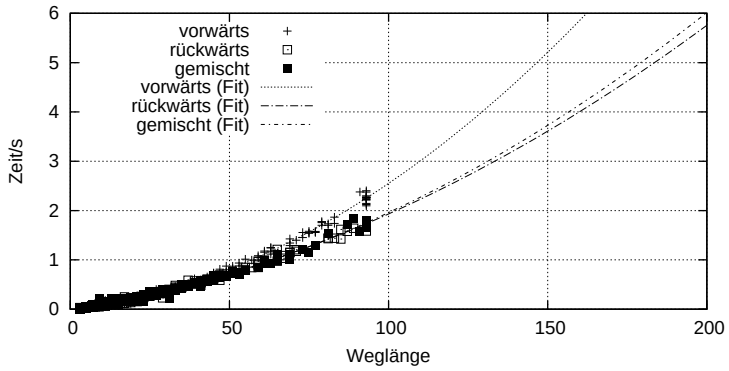


Abbildung 7.15: Fibonacci-Folge – Laufzeiten, ohne Vorhersage

Dort sind ebenso die Vergleichswerte für die Variante mit gleichzeitigem Vorwärts- und Rückwärtsdurchlauf ohne Vorhersage aufgetragen, die sich zwischen den Werten für den Vorwärts- und den Rückwärtsdurchlauf bewegen.

Beim Vergleich der Ergebnisse für den gemischten Durchlauf zeigt sich jedoch mit Vorhersage eine erheblich größere Streuung als ohne Vorhersage sowie eine deutlich geringere maximale Weglänge, wie in Abbildung 7.16 sichtbar.

Dieses Verhalten ist auf das Backtracking bei der Knotenvermeidung zurückzuführen. Bei der Wegvorhersage wird für einen der beiden neuen Teilwege festgelegt, dass auf ihm der Spaltungsknoten nicht durchlaufen werden darf.

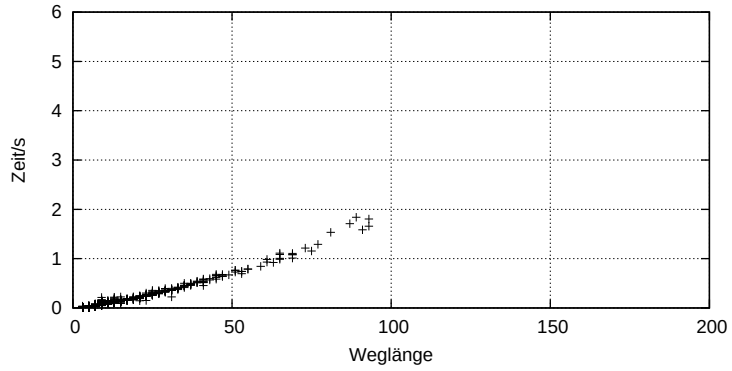
Der Ausschluss wird implementiert, indem die ausgeschlossenen Knoten nicht als Schrittkandidaten zugelassen werden. Bleibt kein Kandidat für einen Vorwärts- oder Rückwärtsschritt übrig und kann auch der Basisfall nicht angewendet werden, so wird das Backtracking ausgelöst.

Weiterhin wird mit Hilfe von Dominator- und Postdominatorrelation erkannt, wenn ein bestimmter Teilweg in jedem Fall über einen der zu vermeidenden Knoten führt. In diesem Fall wird auch hier das Backtracking ausgelöst.

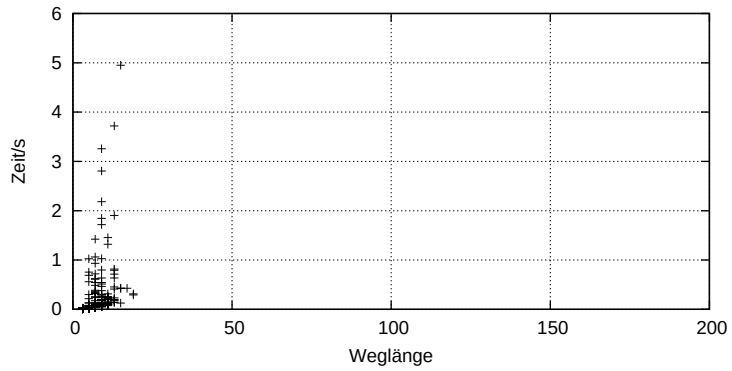
Betrachtet man die Teilweganforderungen nur nach ihrem jeweiligen Start- und Endknoten, so ist ihre Menge beschränkt durch das Kreuzprodukt $N \times N$ der Knotenmenge mit sich selbst.

Tatsächlich von einer Anfrage nach einem Weg von r nach e aufgrund der Ableitungsregeln erreichbar ist aber nur eine Teilmenge von $N \times N$, wie wir bereits bei der Konstruktion der Wahrscheinlichkeitsmodelle in Abschnitt 7.3 sehen konnten.

Im Fall des gemischten Durchlaufs ist diese Teilmenge deutlich größer als beim reinen Vorwärts- oder Rückwärtsdurchlauf. Zusammen mit der Wegvorhersage vergrößert sich die Menge der erreichbaren Teilanfragen weiter. Insbesondere sind



(a) ohne Vorhersage



(b) mit Vorhersage

Abbildung 7.16: Fibonacci-Folge – Laufzeiten, gemischter Durchlauf

mehr Teilanfragen enthalten, bei denen der Ausschluss von Schrittknoten nicht ausreichend ist, um die Knotenvermeidung zu implementieren.

Somit kommt öfter das Backtracking als letzte Möglichkeit zum Einsatz. Die große Streuung der Werte kommt dadurch zustande, dass sich die Laufzeit für die erfolglosen Suchzweige auf die Laufzeit des erfolgreichen Zweigs addiert, allerdings abhängig von der zufälligen Auswahl der Suchreihenfolge. Die aufsummierte Laufzeit überschreitet dabei mit ansteigender Weglänge häufiger das Zeitlimit (in diesem Fall 5 Sekunden), so dass nur äußerst kurze Wege erzeugt werden können. Dieses Verhalten ist für den Softwaretest unerwünscht, da ja auch längere Wege ausgeführt werden sollen.

7.5.2 Verschachtelte Schleife

Der Sortieralgorithmus Selection Sort wurde bereits in Abschnitt 3.4.3 als Beispiel für die Problematik nicht-ausführbarer Wege erwähnt. Wir untersuchen deshalb eine Implementierung des Algorithmus entsprechend Abbildung 7.17.

Die Ergebnisse der Messungen ohne Vorhersage sind in Abbildung 7.18 zu sehen. Erneut wurde die simulierte Ausführung entsprechend der Anfrage $x \; \mathcal{S}_{PFG} \; y$ gemessen. Für den vorwärtsgerichteten Durchlauf zeigt sich ein quadratischer Anstieg mit äußerst geringer Streuung. Die Streuung sowohl bzgl. der Länge als auch der Zeit ergibt sich aus der Wahlmöglichkeit zwischen den Kanten $6 \rightarrow 7$ und $6 \rightarrow 8$, die unterschiedliche Permutationen des Arrays a repräsentieren.

Dieses Verhalten beim vorwärtsgerichteten Durchlauf liegt darin begründet, dass beim Vorwärtsdurchlauf die Länge des Arrays a nach dem ersten Durchlauf der äußeren Schleife durch die Iterationszahl der inneren Schleife eindeutig festgelegt ist.

Wird bei einem weiteren Durchlauf die Ausgangskante $5 \rightarrow 9$ zu früh gewählt, führt dies unmittelbar zum Widerspruch und die Schleife wird mindestens ein weiteres Mal durchlaufen. Der Backtrackingaufwand hierfür ist als gering zu erwarten, da das Backtracking sich jeweils nur über einen kurzen Teilweg erstreckt.

Wie wir bereits aus Abschnitt 3.4.3 wissen, können auch bei den alternativen Kanten $6 \rightarrow 7$ und $6 \rightarrow 8$ Widersprüche durch die Auswahl entstehen. Das Backtracking für diesen Widerspruch erstreckt sich dann allerdings erneut nur auf eine einzige Kante. Der bzgl. der Laufzeit teure Zugriff auf das Arrayelement muss dann jedoch wiederholt werden, sofern der Compiler hier keine Optimierung vorsieht.

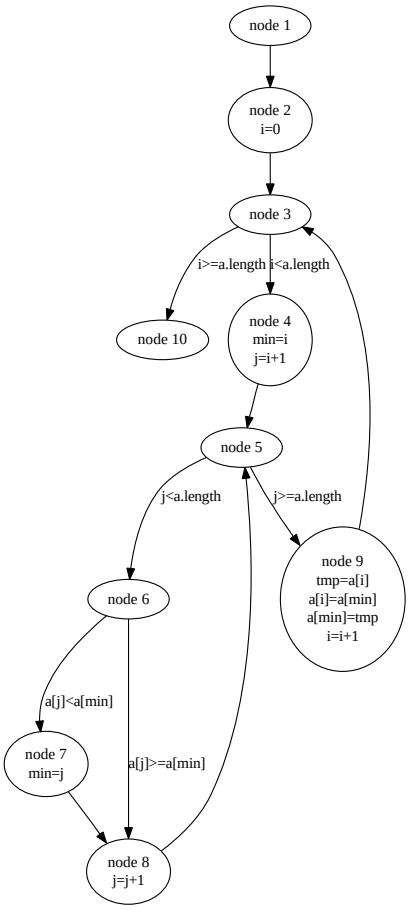


Abbildung 7.17: Selection Sort

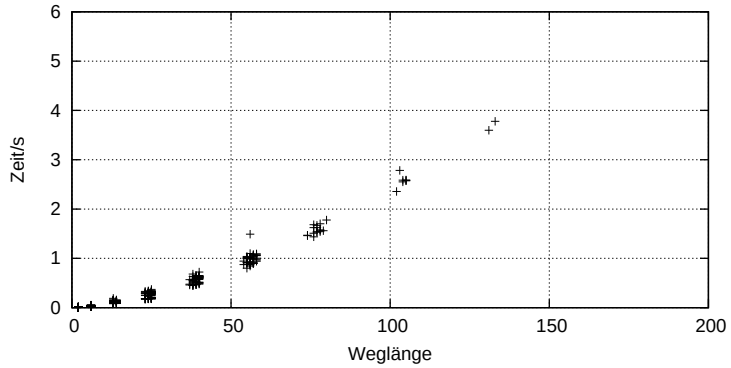


Abbildung 7.18: Selection Sort – Laufzeiten, vorwärts ohne Vorhersage

Im Gegensatz zur vorwärtsgerichteten Durchlaufrichtung existieren beim Rückwärtsdurchlauf scheinbar nur äußerst wenige erfolgreiche Durchläufe. Hier ist die Länge von `a` erst nach dem letzten Durchlauf der äußeren Schleife definiert.

Das Verfahren erkennt zwar einen Widerspruch, wenn die Anzahl der Durchläufe der inneren Schleife nicht mit der Anzahl der bisherigen äußeren Durchläufe übereinstimmt. Erkannt wird dieser Widerspruch allerdings erst, wenn die innere Schleife rückwärts über die Kante $4 \rightarrow 5$ verlassen wird.

Dann aber wurde die innere Schleife ggf. bereits mehrfach durchlaufen und jeder dieser Durchläufe muss einzeln durch Backtracking rückgängig gemacht werden. Mit jedem rückwärtigen Durchlauf der äußeren Schleife wird die Anzahl k der notwendigen Durchläufe der inneren Schleife größer. Somit fällt auch die Wahrscheinlichkeit, auf Anhieb die richtige Durchlaufzahl zu treffen proportional zu p^k.

Die Messungen mit Weg- und Variablenvorhersage zeigen sowohl für den rückwärtsgerichteten und den gemischten Durchlauf nur wenige Ausführungen innerhalb des Zeitlimits. Allerdings liefert auch der Vorwärtsdurchlauf hier nur wenige kurze Wege bei hoher Streuung der Laufzeiten. Durch die Wegvorhersage wird der Weg in Teilwege zerteilt, wodurch die Verbindung zwischen den verschiedenen Zuständen der Variablen `i` und `j` zeitweise verlorengeht, bis die jeweiligen Teilwege durch Anwendung des Basisfalls abgeschlossen werden.

Die Variablenvorhersage wie in Abschnitt 6.1 implementiert kann innerhalb der Schleife nicht erkennen, dass `i` über weite Strecken – etwa in der inneren Schleife – konstant ist. Es gilt schließlich $5 \rightarrow^+ 9$ und $9 \rightarrow^+ 9$. Deshalb kann auch die Wegvorhersage wie implementiert hier nicht helfen.

Bei der Wegvorhersage wird der Teilweg von 4 nach 3 an den Knoten 5 und 9 aufgeteilt. Aufgrund der Modifikation aus Abschnitt 7.3.5 ist es dabei auf dem Weg von 5 nach 9 untersagt, den Knoten 9 nochmals zu durchlaufen. Diese Information kann zusammen mit Satz 5.5 verwendet werden, um das Gleichbleiben des Wertes von `i` auf dem so eingeschränkten Weg von 5 nach 9 vorherzusagen.

Über den Teilweg von 6 nach 8 hinweg ist allerdings selbst unter Ausschluss von Knoten 8 nur schwer vorherzusagen, dass `j` unverändert bleibt. Schließlich existiert noch eine Zuweisung an `j` im Knoten 4, der auf mindestens einem Weg von 6 nach 8 liegt.

Zwar gibt es keinen Weg von 6 nach 8, der über 4, nicht aber ein weiteres Mal über 8 führt. Dies ist jedoch nicht allein mit der Dominator- und Postdominatorrelation zu erkennen.

Beim Verfahren von Gotlieb et al wird jedoch erkannt, dass der Wert von `i` beim Betreten und beim Verlassen der Schleife gleich ist. Diese Erkenntnis basiert auf den Informationen, die durch die Transformation in die SSA-Form in der Eingabe enthalten sind, und auf der Tatsache, dass im dortigen Verfahren explizit die einzelnen Durchläufe der Schleifen abgerollt werden. Aufgrund der Kapselung des Rumpfes in der `while-if`-Struktur ist dabei jeweils klar, dass Anweisungen, die außerhalb der Schleife liegen, während des Schleifendurchlaufs nicht ausgeführt werden können.

7.5.3 Vorgegebener Rückgabewert

Bei den vorangegangenen Beispielen war die Menge der zulässigen Wege lediglich durch deren Ausführbarkeit bestimmt. Wir werden die zulässigen Wege nun noch weiter einschränken, in dem wir einen Rückgabewert für eine Funktion vorgeben.

Die Funktion vergleicht zwei Zeichenketten miteinander und gibt bei Gleichheit den `boolean`-Wert `true` zurück und sonst `false`. Der Programmflussgraph der betrachteten Funktion ist in Abbildung 7.19 dargestellt.

Die Implementierung nutzt eine Variable namens `result`, um das Zwischenergebnis zu speichern und hat insofern Ähnlichkeit mit dem Beispiel zur linearen Suche aus Abschnitt 5.4.2.

Gesucht sind in der Messung Eingaben, die zum Rückgabewert `true` führen. Die Zeichenketten, die als Eingabe erzeugt werden, müssen also gleich sein. Dies ist bei zufälliger Erzeugung der Eingaben nur schwer zu erreichen.

Das Beispiel ist insbesondere deshalb so problematisch für rein sequentiell in Ausführungsrichtung vorgehende Verfahren, da in der Schleife mehrfach Knoten 9 durchlaufen werden kann, bevor der Endknoten erreicht und damit der Widerspruch zur Forderung `result=true` erkannt wird.

Allerdings ist auch nicht zu erwarten, dass die Variablenvorhersage hier weiterhelfen kann, da die Variable `result` zwar nach dem ersten Erreichen des Knoten 9 nicht mehr verändert wird, aber dennoch nicht jeder Weg von 9 nach 5 definitionsfrei bzgl. `result` ist.

Lediglich mit Hilfe der Datenflussvorhersage auf Basis abstrakter Interpretation, wie sie in Abschnitt 5.4.2 durchgeführt wurde, wäre hier ein Vorteil durch die Vorhersage zu erreichen. Im Prototypen wurde dies jedoch zur Beschränkung des Implementierungsaufwands nicht umgesetzt.

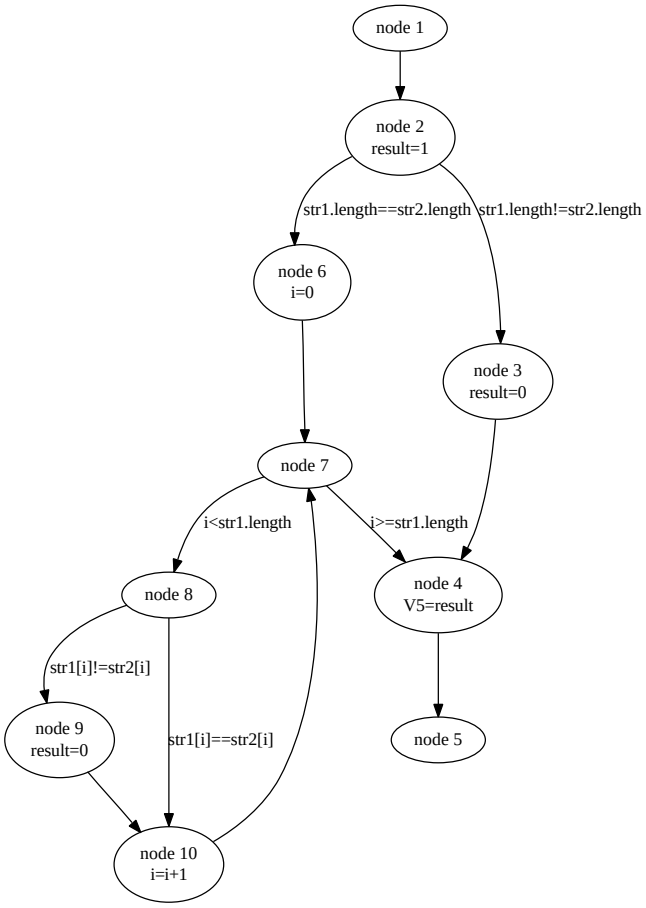


Abbildung 7.19: Vergleich von Zeichenketten

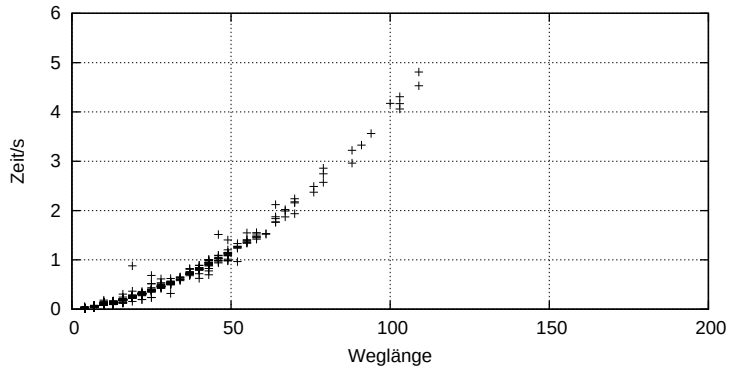


Abbildung 7.20: Vergleich von Zeichenketten – Laufzeiten, rückwärts ohne Vorhersage

Ohne Wegvorhersage sind vorwärtsgerichteter und gemischter Durchlauf nicht sehr erfolgreich. Wie in Abbildung 7.20 zu sehen, liefert allerdings der rückwärtsgerichtete Durchlauf eine hohe Zahl von Wegen bei geringer Streuung der Laufzeit. Das Zeitlimit von 5 Sekunden wird vollständig ausgenutzt.

Bei der rückwärtsgerichteten Ausführung kann ein Weg über den Knoten 4 zum Knoten 5 konstruiert werden. Dann stehen nur die möglichen Vorgänger 7 und 3 zur Verfügung. Da bei der schrittweisen Ausführung die Verbindung der Speicherzustände miteinander unmittelbar und auch ohne Variablenvorhersage erhalten bleibt, kann bei Erreichen des Knoten 3 direkt der Widerspruch zur Forderung `result==true` erkannt werden. Das Backtracking umfasst lediglich einen einzelnen Schritt und ist ohnehin nur in der Hälfte der Fälle notwendig.

Wurde der Knoten 7 durch rückwärtigen Durchlauf erreicht, so kommen die Vorgänger 6 und 10 in Frage. Bei Wahl des Knoten 10 sind dann die nächsten Schritte nach Knoten 9 oder nach Knoten 8 möglich. Auch hier kann direkt bei Betreten des Knoten 9 der Widerspruch erkannt werden, so dass das Backtracking auch hier nur wenig Laufzeit kostet. Wird schließlich Knoten 6 erreicht, so ist der Weg zum Startknoten eindeutig definiert und kann auch nicht mehr zu Widersprüchen führen.

Bei vorwärtsgerichteter Ausführung läuft das Verfahren vom Startknoten ausgehend andersherum ab. Der Durchlauf durch Knoten 3 führt erst bei Erreichen des Endknotens zum Widerspruch, so dass das Backtracking hier etwas mehr Laufzeit erfordert. Ebenso kann Knoten 9 mehrfach durchlaufen werden, bevor bei Erreichen von Knoten 5 der Widerspruch erkannt wird. Es ist also zu erwarten, dass auch hier der Backtrackingaufwand deutlich höher ist.

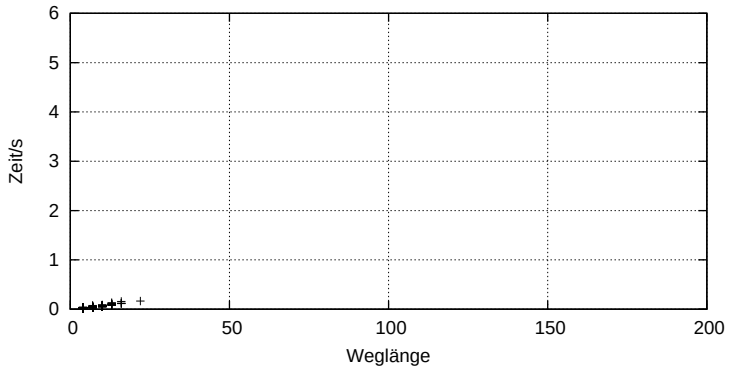


Abbildung 7.21: Vergleich von Zeichenketten – Laufzeit, vorwärts ohne Vorhersage

Betrachten wir jedoch die Laufzeit des vorwärtsgerichteten Durchlaufs ohne Vorhersage in Abbildung 7.21, so können wir nicht die für das Backtracking typische hohe Streuung erkennen. Vielmehr ist die Streuung äußerst gering und es wurden lediglich kurze Wege erzeugt. Zudem liegt keiner der Messpunkte oberhalb 150 Millisekunden.

Wenn im vorwärtsgerichteten Durchlauf der Knoten 5 erreicht wird, nachdem Knoten 9 durchlaufen wurde, wird das Backtracking eingeleitet. Es wird also nach der letzten getroffenen Entscheidung gesucht, zu der noch offene Alternativen vorliegen. In diesem Fall ist die die Entscheidung, als Nachfolger zu Knoten 7 den Knoten 4 zu wählen und damit die Schleife zu verlassen. Die einzige Alternative hier ist also die Fortsetzung der Schleife über Knoten 8.

Dies führt zu einer Endlosschleife für jeden Suchdurchlauf, bei dem Knoten 9 mindestens einmal durchlaufen wurde. Diese Endlosschleife wird erst durch die Zeitlimitierung abgebrochen.

Je länger der bereits erzeugte Weg ist, umso größer ist die Wahrscheinlichkeit, dass der Knoten 9 bereits durchlaufen wurde, deshalb können nur besonders kurze Wege erzeugt werden.

Somit ist auch die geringe Erfolgsquote für den gemischten Durchlauf zu erklären, da hier die Kante $7 \rightarrow 4$ auch früh im Programmablauf rückwärts durchlaufen werden kann, womit diese Entscheidung nicht mehr unbedingt als letztes vor der Erkennung des Widerspruchs getroffen wird und die Endlosschleife vermieden werden kann. Dennoch kann durch entsprechend häufige Vorwärtsdurchläufe eine lange Kette beginnend beim Knoten 9 geschaffen werden, die bei Widerspruchserkennung mit entsprechend hohem Backtrackingaufwand rückgängig gemacht werden muss.

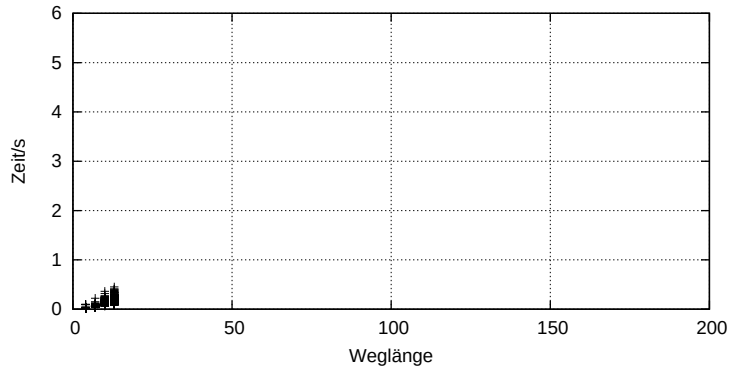


Abbildung 7.22: Vergleich von Zeichenketten – Laufzeiten, rückwärts mit Vorhersage

Mit Vorhersage ist kein wesentlich besseres Verhalten zu erzielen. Hier verschlechtert sich sogar die Erfolgsrate des rückwärtsgerichteten Durchlaufs, wie Abbildung 7.22 zeigt. Durch die Aufspaltung der Wege und den damit einhergehenden Verlust der Verbindung zwischen den Variablenzuständen kann hier selbst mit Hilfe des Rückwärtsdurchlaufs kein deutlich besseres Ergebnis erreicht werden.

Betrachten wir als alternative Implementierung noch den Programmflussgraphen aus Abbildung 7.23. Hier fehlt die Variable `result`. Stattdessen wird bei Erkennung einer Abweichung zwischen den Zeichenketten sofort eine Anweisung `return false` ausgeführt. Wird die Schleife vollständig durchlaufen, wird der Rückgabewert in Knoten 10 auf `true` festgelegt.

Der Rückgabewert wird in den Knoten 3, 8 und 10 festgelegt, von denen nur der direkte Weg zum Endknoten 4 offensteht. Wird also einer der Knoten 3 oder 8 betreten, kann der Widerspruch sofort erkannt werden.

Entsprechend liefern auch ohne Weg- und Variablenvorhersage alle Durchlaufvarianten gute Ergebnisse.

Die einzige Variable, die hier im Rahmen der Schleife verändert wird, ist der Laufindex `i`, so dass hier durch Variablenvorhersage zumindest die Laufzeit für die Simulation eines Lesezugriffs reduziert werden kann.

Der Laufzeitvergleich in Abbildung 7.25 für den Vorwärtsdurchlauf bestätigt die Erwartung, dass mit der Weg- und Variablenvorhersage eine geringere Laufzeit erreicht wird. Dies ist ebenso für den Rückwärtsdurchlauf zu beobachten. Lediglich beim gemischten Durchlauf tritt auch hier eine große Streuung aufgrund des Backtrackings zur Knotenvermeidung auf, so dass nur kurze Wege erzeugt werden können.

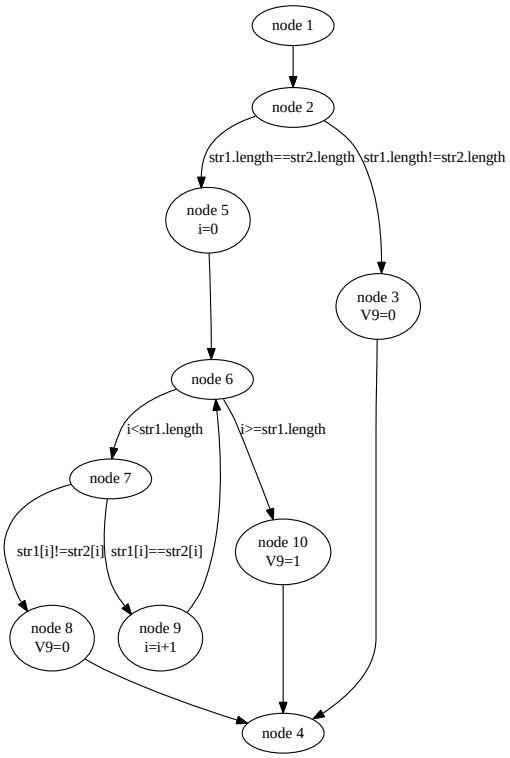


Abbildung 7.23: Vergleich von Zeichenketten – Variante 2

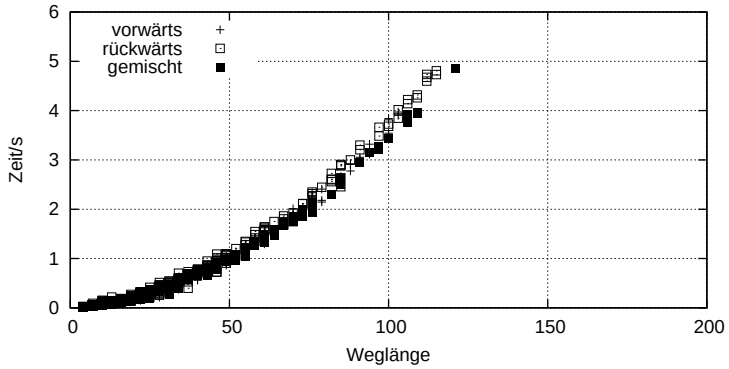


Abbildung 7.24: Vergleich von Zeichenketten 2 – Laufzeiten, ohne Vorhersage

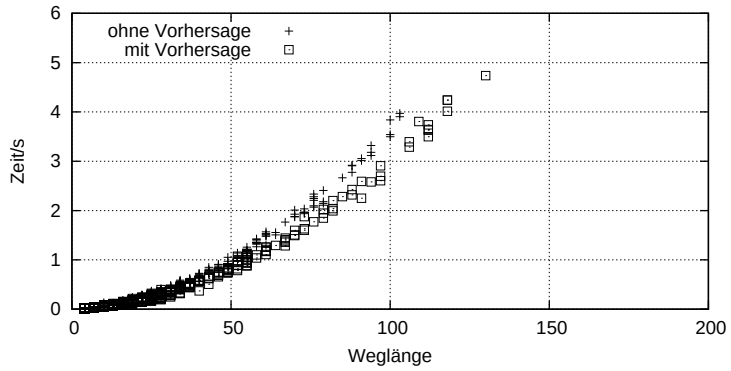


Abbildung 7.25: Vergleich von Zeichenketten 2 – Laufzeiten, vorwärts

7.5.4 Knotenabdeckung

Die Knotenabdeckung untersuchen wir anhand des Programmflussgraphen aus Abbildung 7.26. Er stellt einen Algorithmus zum Einfügen eines Elements in ein Array dar, wobei die leeren Felder durch den Wert `-1` markiert sind. Rückgabewert ist der Index des neuen Eintrags oder `a.length`, wenn kein freier Eintrag gefunden wurde.

Ziel ist die Abdeckung des Knoten 7, was dem Fall entspricht, in dem noch ein freies Feld gefunden wurde. Dieser Fall ist je nach Vorgehensweise mit zufälliger Eingabeerzeugung ggf. nur schwer zu erreichen. Die Anfrage ist also gemäß Abschnitt 4.4 $x \ \left(\mathcal{S}_{PFG[r,7]}\mathcal{I}_{PFG[7,e]}\mathcal{B}\left(e\right)\right) \ y$.

Sowohl bei vorwärts- als auch bei rückwärtsgerichtetem Durchlauf erwarten wir nur geringen Backtrackingaufwand. Der Knoten 6 darf entsprechend der Bedingung nur über die Kante $5 \rightarrow 6$ erreicht werden, da sonst `i >= a.length` gilt und somit der Widerspruch zur Bedingung der einzigen zu Knoten 7 führenden Kante hergestellt ist.

Bei Vorwärtsdurchlauf wird der Widerspruch bei Durchlaufen der Kante $3 \rightarrow 6$ nach einem weiteren Schritt erkannt. Die Möglichkeit, die Kante $3 \rightarrow 6$ zu durchlaufen besteht dabei für jede Schleifeniteration erneut.

Beim Rückwärtsdurchlauf wird zunächst von Knoten 7 aus Knoten 6 erreicht und dann besteht ein einziges Mal die Möglichkeit, die Kante $3 \rightarrow 6$ zu durchlaufen, was sofort zum Widerspruch und der Wahl der Alternative $5 \rightarrow 6$ führt. Daraufhin existiert keine weitere Möglichkeit, Backtracking zu verursachen.

Entsprechend zeigen die Ergebnisse der Laufzeitmessung in Abbildung 7.27 für alle Durchlaufrichtungen ohne Wegvorhersage ein akzeptables Verhalten des Verfahrens.

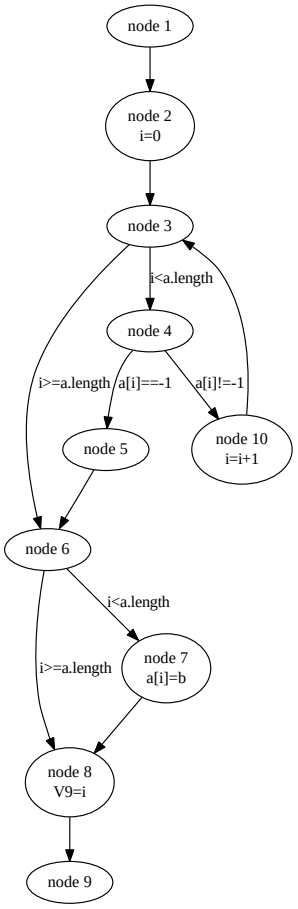


Abbildung 7.26: Einfügen in Array

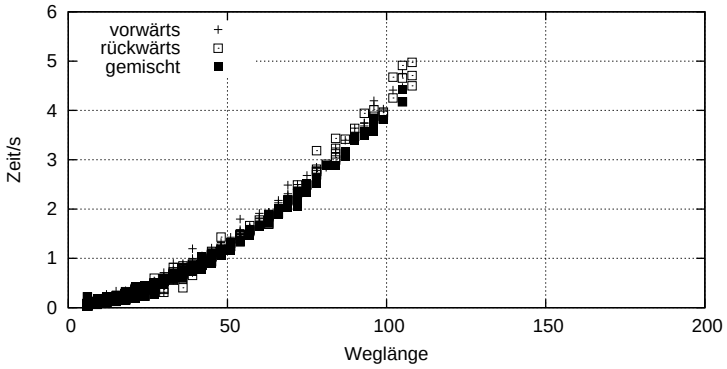


Abbildung 7.27: Einfügen in ein Array – Laufzeiten, ohne Vorhersage

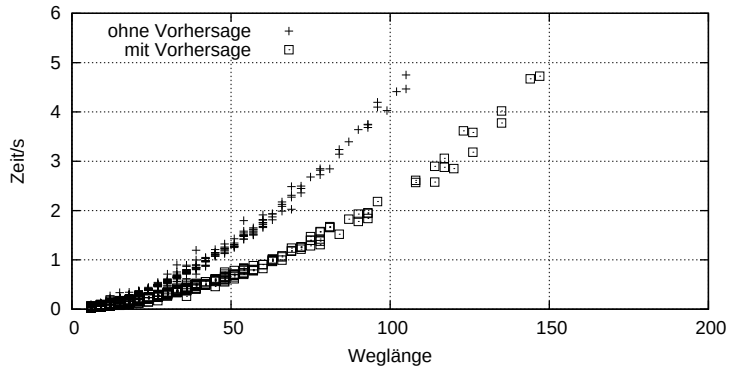


Abbildung 7.28: Einfügen in ein Array – Laufzeiten, vorwärts

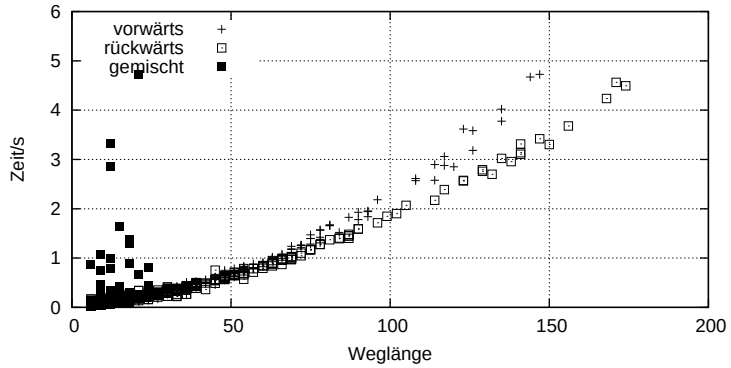


Abbildung 7.29: Einfügen in ein Array – Laufzeiten, mit Vorhersage

Für das Verhalten mit Weg- und Variablenvorhersage erwarten wir auch hier ein besseres Verhalten für vorwärts- und rückwärtsgerichteten Durchlauf als ohne Vorhersage, da auch hier `i` die einzige Variable ist, die in der Schleife beschrieben wird. Die Verbesserung der Laufzeit leiten wir erneut aus dem geringeren Aufwand für die Speicherzugriffe ab.

Die Messergebnisse aus Abbildung 7.28 bestätigen diese Erwartung und zeigen eine erheblich bessere Laufzeit für den Fall mit Vorhersage.

Wir erwarten auch mit gemischter Durchlaufrichtung bei Anwendung der Vorhersage schlechtere Ergebnisse als ohne Vorhersage, erneut aufgrund des erhöhten Backtrackinganteils für die Knotenvermeidung. Auch dies bestätigen die Messergebnisse, wie Abbildung 7.29 zeigt.

```
1  public static int triangle(int a, int b, int c) {
2          if (a<=0 || b<=0 || c<=0)
3                  return 4;
4          if (a==b && b==c) {
5                  return 1;
6          } else if (a==b || b==c || c==a) {
7                  return 2;
8          } else {
9                  return 3;
10         }
11 }
```

Listing 7.1: Dreiecksklassifizierung

7.5.5 Dreiecksklassifizierung

Abschließend betrachten wir noch ein bekanntes Beispiel[MBTS04]: Die Klassifizierung von Dreiecken.

Für diese Aufgabe sind jeweils drei Zahlen gegeben, die die Längen der drei Seiten eines Dreiecks darstellen sollen. Es soll eine Funktion implementiert werden, mit der festgestellt werden kann, ob es sich um ein gleichseitiges, ein gleichschenkliges oder ein gewöhnliches Dreieck handelt. Für ein gleichseitiges Dreieck soll der Wert 1, für ein gleichschenkliges der Wert 2 und für ein gewöhnliches Dreieck der Wert 3 zurückgegeben werden. Existiert kein Dreieck mit den angegebenen Seitenlängen, so soll der Rückgabewert 4 sein.

Eine fehlerhafte Implementierung dieser Funktion ist in Listing 7.1 zu sehen. Sie ist insofern fehlerhaft, als zum Beispiel die Angaben $a = 2$, $b = 2$ und $c = 5$ als gleichschenkliges Dreieck identifiziert werden, obwohl aufgrund von $a + b < c$ zu diesen Längen überhaupt kein Dreieck existiert.

Wir haben Testeingaben so erzeugt, dass jede `return`-Anweisung jeweils zweimal ausgeführt wird. Die Ergebnisse sind in Tabelle 7.1 aufgeführt. Wir geben zusätzlich an, ob jeweils tatsächlich ein Dreieck mit den vorgegebenen Seitenlängen existiert.

Wir sehen, dass die Testfälle im Rahmen der Vorgaben tatsächlich aus dem gesamten zulässigen Wertebereich gewählt wurden. Bis auf einen Fall wurden alle Dreiecke korrekt klassifiziert. Lediglich beim dritten Testfall wurde ein unzulässiges Dreieck als gleichschenklig eingestuft.

a	b	c	**Resultat**	**Dreieck?**
154346040	154346040	154346040	1	ja
590428349	590428349	590428349	1	ja
730601472	730601472	1798418546	2	nein
1186311676	1186311676	957380831	2	ja
867687259	1139600275	1306555234	3	ja
1254626770	1167588924	285575398	3	ja
0	1592549039	1752106985	4	nein
-626557546	-951287603	-603446218	4	nein

Tabelle 7.1: Testfälle für Dreiecksklassifizierung

Die Testfälle wären also dazu geeignet gewesen, den Fehler der nicht implementierten Abfrage zu finden. Dies ist jedoch nicht darauf zurückzuführen, dass das Verfahren dies gezielt herbeiführt, sondern vielmehr auf die zufällige Auswahl der tatsächlichen Eingabewerte im Rahmen dessen, was nach den Vorgaben zulässig ist. Bei einer ausreichend großen Anzahl von erzeugten Testeingaben ist also rein statistisch zu erwarten, dass mindestens ein solcher Fall erzeugt wird.

7.5.6 Schlussfolgerungen

Wir haben anhand der Beispiele gezeigt, dass keine der Varianten allgemeingültig optimal ist. Die Laufzeit und der Anteil erfolgreicher Durchläufe hängen wesentlich vom Aufbau des zu testenden Programms wie auch den Testvorgaben ab.

Deutlich wird dies insbesondere in Tabelle 7.2, wo wir für jedes der Beispiele die jeweils besten und schlechteste Varianten gegenüber gestellt haben. Bei der Beurteilung haben wir neben der Laufzeitentwicklung auch die Streuung und die damit verbundene Tendenz, nur kurze Wege zu erzeugen, einbezogen. Jene Variante, die jeweils von den beiden besten die vorteilhafteste ist, haben wir außerdem fett hervorgehoben. In einigen Fällen konnte aufgrund kurzer Wege und/oder hoher Streuung keine beste Variante ermittelt werden.

Wir können auch teilweise deutliche Laufzeiteinsparungen durch Verwendung einer passenden Variante erreichen. In Tabelle 7.5.6 sind jeweils die asymptotischen Einsparungen der besten Varianten im Vergleich zum Vorwärtsdurchlauf ohne Vorhersage aufgeführt. Wir haben jeweils ein quadratisches Fit der Laufzeiten in Abhängigkeit von der Weglänge bestimmt und die asymptotische Einsparung aus dem Quotienten der quadratischen Anteile ermittelt.

Beispiel	**beste**		**schlechteste**
	ohne Vorh.	**mit Vorh.**	
Fibonacci	**rückwärts**	rückwärts	gemischt mit Vorh.
Selection Sort	**vorwärts**	n/a	vorw. mit Vorh.
`strcmp` ohne `break`	**rückw.**	n/a	gemischt mit Vorh.
`strcmp` mit `break`	gemischt	**rückwärts**	gemischt mit Vorh.
Einfügen in Array	gemischt	**rückwärts**	vorw. mit Vorh.

Tabelle 7.2: Vergleich der Varianten

Beispiel	**asymptotische Einsparung**	
	ohne Vorhersage	**mit Vorhersage**
Fibonacci	ca. 49%	ca. 46%
Selection Sort	0%	n/a
`strcmp` ohne `break`	n/a	n/a
`strcmp` mit `break`	ca. 10%	ca. 28%
Einfügen in Array	ca. 7%	ca. 68%

Tabelle 7.3: Laufzeiteinsparung im Vergleich zu vorwärts ohne Vorhersage

Auch hier sind für einige Fälle aufgrund hoher Streuung und/oder kurzer Wege keine Angaben möglich. Beim Selection Sort liegt keine Einsparung vor, da der Vorwärtsdurchlauf ohne Vorhersage hier bereits die beste Strategie ist.

Die Datenflussvorhersage kann sogar in solchen Fällen Vorteile bringen, in denen sie nicht für die frühzeitige Erkennung von nicht-ausführbaren Wegen notwendig wäre. So können durch die Variablenvorhersage Speicherzugriffsoperationen vorweggenommen und somit Rechenaufwand eingespart werden.

Bei der Datenflussvorhersage sind in einigen Fällen noch Verbesserungen notwendig, um etwa in Variablen versteckte Kontrollflussinformation wie bei der linearen Suche und dem Vergleich von Zeichenketten zu extrahieren.

Die Laufzeit der Simulation durch das Verfahren ist erwartungsgemäß deutlich höher als die Laufzeit der realen Ausführung, so dass in vielen Fällen die zufällige Erzeugung von Eingaben effizienter sein dürfte. Lediglich in jenen Fällen, in denen auf diese Weise nicht alle Testanforderungen erfüllt werden können, ist es empfehlenswert, das Verfahren überhaupt einzusetzen.

Gotlieb et al[GBR98] führen keine qualitative Laufzeitanalyse durch und geben lediglich für einzelne Beispiele absolute Laufzeiten an. Bei einem Beispiel mit zwei verknüpften linearen Suchschleifen wird für einen einzelnen Testeingabevektor ei-

ne Laufzeit von ca. 1s benötigt. Der Wert basiert auf einer Ausführung auf einer Sparc 5 Workstation, deren Taktfrequenz nicht angegeben, aber der Architektur nach maximal 170MHz ist.

Unser Prototyp wurde auf einem Intel Core 2 Duo Prozessorsystem mit 1600MHz evaluiert. Er benötigt für dasselbe Beispiel bei einem Rückwärtsdurchlauf ohne Vorhersage zwischen 100 und 600ms, abhängig von der Weglänge. Der Leistungszuwachs ist auch aufgrund des Unterschieds zwischen RISC und CISC schwer abzuschätzen. Geht man von Faktor 10 aus, so müsste InKA auf dieser Architektur für dieses Beispiel ca. 100ms benötigen.

Anhand der vorliegenden Daten ist allerdings ein detaillierter und abschließender Vergleich mit dem Verfahren von Gotlieb et al nicht möglich.

8 Schlussfolgerungen und Ausblick

Mit dieser Arbeit liegt erstmals ein Rahmenwerk für ein neues Verfahren zur constraintbasierten Testdatenerzeugung vor, das die wesentlichen Anforderungen für die Anwendung in der industriellen Praxis erfüllt. Dieses Rahmenwerk stellt eine fundierte Grundlage für die weitere Entwicklung im Bereich der constraintbasierten Testdatenerzeugung dar.

Unser Verfahren wendet Strategien zur Vermeidung nicht-ausführbarer Wege an, berücksichtigt vorwärts- und rückwärtsgerichtete Beschränkungen und kann das zu testende Programm in Form eines Programmflussgraphen verarbeiten. Dadurch ist das Verfahren insbesondere direkt auf die von optimierenden Compilern erzeugten Basisblockgraphen, aber auch auf Code in Maschinensprache anwendbar.

Wir haben auch gezeigt, dass sich die klassischen Testkriterien direkt und intuitiv als Anfragen für das Verfahren formulieren lassen, ohne dass dafür eine Transformation des Programms notwendig ist.

Lediglich für die Erzeugung schleifenfreier Wege haben wir eine Transformation des Programmflussgraphen eingesetzt. Aber auch hier bietet das Rahmenwerk die notwendigen Grundlagen, um das Verfahren entsprechend zu erweitern, so dass diese Umformung entfallen kann.

Wir haben aufgezeigt, wie weitergehende Vorgaben für das Verfahren formuliert werden können. So kann eine große Bandbreite von Anforderungen durch einfache Umformungen des Programms abgedeckt werden. Auch hier bietet das Rahmenwerk die theoretischen Grundlagen für die Erweiterung des Verfahrens im Hinblick auf in Zukunft entwickelte wiederkehrende Konstrukte in den Testanforderungen.

Die Auswahl sowohl des Weges als auch der konkreten Eingabe aus den jeweiligen durch das Testkriterium eingeschränkten Kandidatenmengen geschieht zufällig. Die Bevorzugung kürzerer Wege gegenüber längeren kann durch automatische Variation von Verfahrensparametern im Wesentlichen ausgeglichen werden. Dadurch ist auch eine statistische Auswertung der Testergebnisse möglich.

Wir haben ausführlich die Eigenschaften des Verfahrens im Bezug auf Terminierung, Länge der erzeugten Wege und Laufzeit untersucht, sowohl auf theoretischer wie auch auf experimenteller Ebene.

Die Untersuchungen wurden durchgeführt in der Hoffnung, eine Strategie für ein Verfahren zu entdecken, die für alle oder möglichst viele Programme gleichermaßen geeignet ist. Grundlage war dabei die Variation der Vorgehensweise mit oder ohne Vorhersage sowie mit vorwärts- und rückwärtsgerichteten oder gemischten Durchläufen.

Wir haben gezeigt, dass das neue Rahmenwerk und das daraus entwickelte Verfahren in vielen Fällen wesentliche Vorteile gegenüber bereits bekannten Ansätzen bringt. Insbesondere haben wir anhand von Beispielen deutlich gemacht, dass die rein sequentielle Analyse in Ausführungsrichtung nicht ausreicht, um bestimmte nicht-pathologische Anwendungsfälle zu bearbeiten.

In den Ergebnissen unserer Messungen konnten wir klar die Auswirkungen des Backtracking erkennen, das eintritt, wenn ein Widerspruch und somit ein nicht-ausführbarer Weg erkannt wird. Dadurch erhöht sich die Laufzeit meist erheblich. Wir konnten also erstmals deutlich herausarbeiten, wie wichtig die frühzeitige Erkennung nicht-ausführbarer Wege für den Durchsatz eines Verfahrens ist.

Wir konnten aber auch erkennen, dass keine der vorgestellten Varianten allgemeingültig optimale Ergebnisse liefert. In einigen Fällen konnten nicht-ausführbare Wege besser durch vorwärtsgerichtete Schritte vermieden werden, während gerade diese Vorgehensweise in anderen Fällen zu erheblichen Nachteilen in der Laufzeit geführt hat. Ähnlich können auch Weg- und Datenflussvorhersage in einigen Fällen von Vorteil, in anderen aber explizit von Nachteil sein.

Ziel weiterer Arbeiten muss es also auch sein, herauszufinden, ob anhand einfach zu bestimmender Eigenschaften eines Programms eine geeignete Variation automatisch ausgewählt werden kann. Dazu ist es insbesondere notwendig, ein Profil der in der Praxis eingesetzten Programme und Programmstrukturen zu ermitteln und diesbezüglich das Verfahren zu optimieren.

Aufgrund der offensichtlichen Unterschiede bei Laufzeit- und Speicherkomplexität sowie der nur näherungsweise bestimmbaren statistischen Eigenschaften sollten constraintbasierte Verfahren dennoch nur komplementär zum Zufallstest eingesetzt werden.

Nur durch Zufallsgeneratoren können signifikante Testeingabemengen in akzeptabler Zeit mit den gewünschten statistischen Eigenschaften erzeugt werden. Constraintbasierte Verfahren können dort ergänzend wirken, wo der Zufallstest nur unzureichende Trefferraten erzielt.

In zukünftigen Arbeiten werden wir uns mit der Anwendung des Rahmenwerks auf die modellbasierte Testdatenerzeugung, unter anderem für kommunizierende

Prozesse, sowie der Verbesserung der Datenflussvorhersage im Zusammenspiel mit der Knotenvermeidung beschäftigen.

Das Verfahren soll in die bereits bestehenden Zufallstestwerkzeuge DARTT[GF93] für Ada, DCRTT für C und SmartG für Java als Ergänzung integriert werden.

Auf Grundlage dieser Werkzeuge wird das Verfahren für den industriellen Einsatz optimiert. Erkenntnisse über das Verhalten im Zusammenhang mit Programmen aus der industriellen Praxis sollen dann in die Entwicklung einer angepassten Strategie für die constraintbasierte Testdatenerzeugung im praktischen Umfeld einfließen.

Danksagung

Viele Personen haben mich bei meiner Forschung und dem Verfassen dieser Arbeit unterstützt. Ich möchte mich besonders bei meinem Betreuer Prof. Dr. Helmuth Partsch dafür bedanken, dass ich in einer offenen und inspirierenden Umgebung frei meinen Ideen und meiner Forschungsarbeit nachgehen konnte. Ich danke außerdem Prof. Dr. Franz Schweiggert, der sich als zweiter Gutachter für meine Arbeit zur Verfügung gestellt und mich durch Fürsprache und Antrieb unterstützt hat. Dank gilt auch meinem Vater Dr. Rainer Gerlich sowie Dr. habil. Johannes Mayer, Dr. Walter Guttmann, Dominik Gessenharter, Frank Reiser und vielen weiteren für wertvolle Kommentare und viele Stunden fruchtvoller Diskussionen. Schließlich möchte ich mich beim Vergabegremium der Universität Ulm für das Stipendium bedanken, das es mir ermöglicht hat, mich vollständig den Forschungen für diese Arbeit zu widmen.

Literaturverzeichnis

[Arj04] ARJONILLA, AUGUSTIN: *The last step towards full on-board software automatic development: the automatic test of code generated from models*. In: *Proceedings of DASIA 2004 - DAta Systems In Aerospace*. European Space Agency, 2004.

[AS98] ABDENNADHER, SLIM und HERIBERT SCHÜTZ: $CHR^{\vee}$*: A Flexible Query Language*. In: *Proceedings of Third International Conference on Flexible Query Answering Systems*, Band 1495 der Reihe *Lecture Notes in Artificial Intelligence*. Springer Verlag, Mai 1998.

[ASU97] AHO, ALFRED V., RAVI SETHI und JEFFREY D. ULLMANN: *Compilerbau*, Band 2. Addison-Wesley Publishing Company, 1997.

[BG98] BECK, KENT und ERICH GAMMA: *Test-infected: programmers love writing tests*. Java Report, 3(7):51–56, Juli 1998.

[Bra01] BRAND, SEBASTIAN: *Constraint Propagation in Presence of Arrays*. CoRR, cs.PL/0105024, 2001.

[CFR91] CYTRON, RON, JEANNE FERRANTE und BARRY K. ROSEN: *Efficiently Computing Static Single Assignment Form and the Control Dependence Graph*. ACM Transactions on Programming Languages and Systems, 13(4):451–490, 1991.

[CHK07] CHEN, T. Y., DE HAO HUANG und F.-C. KUO: *Adaptive random testing by balancing*. In: *RT '07: Proceedings of the 2nd international workshop on Random testing*, Seiten 2–9. ACM, 2007.

[DGG04] DENISE, A., M.-C. GAUDEL und S.-D. GOURAUD: *A generic method for statistical testing*. In: *Proceedings of the 15th IEEE International Symposium on Software Reliability Engineering (ISSRE)*, Seiten 25–34. IEEE, 2004.

[Dij72] DIJKSTRA, EDSGER W.: *Structured Programming*, Kapitel Notes on Structured Programming, Seiten 1–82. Academic Press Ltd., 1972.

[DO91] DEMILLO, RICHARD A. und A. JEFFERSON OFFUTT: *Constraint-Based Automatic Test Data Generation*. IEEE Trans. Softw. Eng., 17(9):900–910, 1991.

[DSBH04] DUCK, GREGORY J., PETER J. STUCKEY, MARIA GARCÍA DE LA BANDA und CHRISTIAN HOLZBAUR: *The refined operational semantics of Constraint Handling Rules*. In: *Lectures in Computer Science*, Band 3132, Seiten 90–104. Springer Verlag, 2004.

[FA03] FRÜHWIRTH, THOM und SLIM ABDENNADHER: *Essentials of Constraint Programming*. Springer Verlag, 2003.

[FK96] FERGUSON, ROGER und BOGDAN KOREL: *The chaining approach for software test data generation*. ACM Trans. Softw. Eng. Methodol., 5(1):63–86, 1996.

[FPB95] FREDERICK P. BROOKS, JR: *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley, 1995.

[FPW02] FRÜHWIRTH, THOM, ALESSANDRA DI PIERRO und HERBERT WIKLICKY: *Probabilistic constraint handling rules*. In: *WFLP 2002, 11th International Workshop on Functional and (Constraint) Logic Programming*, Band 76 der Reihe *Electronic Notes in Theoretical Computer Science*, Seiten 1–16. Elsevier, November 2002.

[Frü98] FRÜHWIRTH, THOM: *Theory and Practice of Constraint Handling Rules*. Journal of Logic Programming, Special Issue on Constraint Logic Programming, 37(1-3):95–138, Oktober 1998.

[Frü00] FRÜHWIRTH, THOM: *Termination of CHR Constraint Solvers*. In: APT, K.R., A.C. KAKAS, E. MONFROY und F. ROSSI (Herausgeber): *Papers from the Joint ERCIM/Compulog-Net Workshop*, Band 1865 der Reihe *Lecture Notes in Artificial Intelligence*. Springer Verlag, Mai 2000.

[GBR98] GOTLIEB, ARNAUD, BERNARD BOTELLA und MICHEL RUEHER: *Automatic test data generation using constraint solving techniques*. SIGSOFT Softw. Eng. Notes, 23(2):53–62, 1998.

[GBR00] GOTLIEB, ARNAUD, BERNARD BOTELLA und MICHEL RUEHER: *A CLP Framework for Computing Structural Test Data*. Lecture Notes in Computer Science, 1861:399–413, 2000.

[GDB05a] GOTLIEB, ARNAUD, TRISTAN DENMAT und BERNARD BOTELLA: *Constraint-based test data generation in the presence of stack-directed pointers*. In: *20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005)*, Seiten 313–316. ACM, 2005.

[GDB05b] GOTLIEB, ARNAUD, TRISTAN DENMAT und BERNARD BOTELLA: *Goal-Oriented Test Data Generation for Programs with Pointer Variables*. In:

29th Annual International Computer Software and Applications Conference (COMPSAC 2005), Seiten 449–454. IEEE, 2005.

[Ger07] GERLICH, RALF: *Integrierte Testkonzepte — Von der Theorie zur Praxis und zurück*. In: *Proceedings of the GI Software Engineering Conference 2007*, 113–122. Gesellschaft für Informatik e.V., 2007.

[GF93] GERLICH, RAINER und G. FERCHER: *A Random-Testing Environment for Ada Programs*. In: *Eurospace Symposium 'Ada in Aerospace'*, 1993.

[GGB07] GERLICH, RALF, RAINER GERLICH und THOMAS BOLL: *Random Testing: From the Classical Approach to a Global View and Full Test Automation*. In: *RT '07: Proceedings of the 2nd international workshop on Random testing*, Seiten 30–37. ACM, 2007.

[GGBC06] GERLICH, RALF, RAINER GERLICH, THOMAS BOLL und PHILIPPE CHEVALLEY: *Improving Test Automation by Deterministic Methods in Statistical Testing*. In: *Proceedings of DASIA 2006 - DAta Systems In Aerospace*. European Space Agency, 2006.

[GGBM07] GERLICH, RALF, RAINER GERLICH, THOMAS BOLL und JOHANNES MAIER: *Evaluation of Auto-Test Generation Strategies and Platforms*. In: *Proceedings of DASIA 2007 - DAta Systems In Aerospace*. European Space Agency, 2007.

[GJSB05] GOSLING, JAMES, BILL JOY, GUY STEELE und GILAD BRACHA: *Java™ Language Specification*. Prentice Hall, Dritte Auflage, 2005.

[GKS05] GODEFROID, PATRICE, NILS KLARLUND und KOUSHIK SEN: *DART: directed automated random testing*. In: *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, Seiten 213–223. ACM, 2005.

[God07] GODEFROID, PATRICE: *Random testing for security: blackbox vs. whitebox fuzzing*. In: *RT '07: Proceedings of the 2nd international workshop on Random testing*, Seite 1. ACM, 2007.

[Got01] GOTLIEB, ARNAUD: *INKA: An Automatic Software Test Data Generator*. In: *Proceedings of DASIA 2001 - DAta Systems In Aerospace*, Seiten 209–21, 2001.

[Got03] GOTLIEB, ARNAUD: *Exploiting Symmetries to Test Programs*. In: *Proceedings of the 14th International Symposium on Software Reliability Engineering (ISSRE)*, Seiten 365–374. IEEE, 2003.

[Gou05] GOURAUD, SANDRINE-DOMINIQUE: *AuGuSTe: a Tool for Statistical Testing — Experimental results*. Rapport de Recherche 1400, Laboratoire de Recherche en Informatique, Université Paris Sud, 2005.

[Ham94] HAMLET, RICHARD: *Random testing*. In: MARCINIAK, J. (Herausgeber): *Encyclopedia of Software Engineering*, Seiten 970–978. Wiley, 1994.

[HH98] HOARE, C.A.R. und JIFENG HE: *Unifying Theories of Programming*. Prentice Hall, 1998.

[HHSS08] HEISEL, MARITTA, DENIS HATEBUR, THOMAS SANTEN und DIRK SEIFEN: *Using UML Environment Models for Test Case Generation*. In: MAALEJ, WALID und BERND BRÜGGE (Herausgeber): *Software Engineering 2008 – Workshopband*, Band 122 der Reihe *LNI*, Seiten 399–406. GI, 2008.

[Hof01] HOFFMAN, DOUGLAS: *Using Oracles in Test Automation*. In: *Proceedings of the Pacific Northwest Software Quality Conference (PNSQC 2001)*, Seiten 90–117, 2001.

[Hof09] HOFFMANN, MARC R.: *EclEmma Webseite*, 2009. `http://www.eclemma.org/`, zuletzt abgerufen am 19.05.2009.

[HT90] HAMLET, RICHARD und ROSS TAYLOR: *Partition Testing does not inspire confidence*. IEEE Transactions on Software Engineering, 16(12):206–215, Dezember 1990.

[Kor90] KOREL, B.: *Automated Software Test Data Generation*. IEEE Trans. Softw. Eng., 16(8):870–879, 1990.

[KS98] KNOBE, KATHLEEN und VIVEK SARKAR: *Array SSA form and its use in parallelization*. In: *POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, Seiten 107–120. ACM, 1998.

[KSD06] KONINCK, LESLIE DE, TOM SCHRIJVERS und BART DEMOEN: *Search strategies in CHR(Prolog)*. In: *Proceedings of the Third Workshop on Constraint Handling Rules (CHR 2006)*, Seiten 109–123. Department of Computer Science, K.U.Leuven, 2006.

[LT79] LENGAUER, THOMAS und ROBERT ENDRE TARJAN: *A fast algorithm for finding dominators in a flowgraph*. ACM Trans. Program. Lang. Syst., 1(1):121–141, 1979.

[MBTS04] MYERS, GLENFORD J., TOM BADGETT, TODD M. THOMAS und COREY SANDLER: *The Art of Software Testing*. John Wiley & Sons, Inc., 2004.

[MG04] MAYER, JOHANNES und RALPH GUDERLEI: *Test Oracles using Statistical Methods*. In: *Testing of Component-Based Systems and Software Quality*, Seiten 179–189. Gesellschaft für Informatik e.V., 2004.

[ND01] NGUYEN, TRAN SY und YVES DEVILLE: *Automatic Test Data Generation for Programs with Integer and Float Variables*. In: *Proceedings of the 16th IEEE International Conference on Automated Software Engineering (ASE)*, Seiten 13–21. IEEE, 2001.

[OJP97] OFFUTT, A. JEFFERSON, ZHENYI JIN und JIE PAN: *The dynamic domain reduction procedure for test data generation*. Softw. Pract. Exper., 29(2):167–193, 1997.

[OLR+94] OFFUTT, JEFFERSON A., AMMEI LEE, GREGG ROTHERMEL, ROLAND UNTCH und CHRISTIAN ZAPF: *An Experimental Determination of Sufficient Mutation Operators*. Technischer Bericht ISSE-TR-94-100, George Mason University, 1994.

[ORZ93] OFFUTT, A. JEFFERSON, GREGG ROTHERMEL und CHRISTIAN ZAPF: *An experimental evaluation of selective mutation*. In: *ICSE '93: Proceedings of the 15th international conference on Software Engineering*, Seiten 100–107. IEEE Computer Society Press, 1993.

[Oud07] OUDINET, JOHAN: *Uniform Random Walks in Very Large Models*. In: *RT '07: Proceedings of the 2nd international workshop on Random testing*, Seiten 26–29. ACM Press, November 2007.

[Par90] PARTSCH, H.: *Specification and Transformation of Programs*. Texts and Monographs in Computer Science. Springer Verlag, 1990.

[Par09] PARR, TERENCE: *ANTLR Webseite*, 2009. `http://www.antlr.org/`, zuletzt abgerufen am 19.05.2009.

[PHP99] PARGAS, ROY P., MARY JEAN HARROLD und ROBERT PECK: *Test-Data Generation Using Genetic Algorithms*. Software Testing, Verification & Reliability, 9(4):263–282, 1999.

[Rad92] RADIO TECHNICAL COMMISSION FOR AERONAUTICS INC.: *DO-178B: Software Considerations in Airborne Systems and Equipment Certification*, Januar 1992.

[RW82] RAPPS, SANDRA und ELAINE J. WEYUKER: *Data flow analysis techniques for test data selection*. In: *ICSE '82: Proceedings of the 6th international conference on Software engineering*, Seiten 272–278. IEEE Computer Society Press, 1982.

[Sch08] SCHMIDBERGER, RAINER: *Glassboxtest zur Testsuite-Optimierung*. In: *Proceedings of the GI Software Engineering Conference 2008*, Seiten 407–414. Gesellschaft für Informatik e.V., 2008.

[SD04] SCHRIJVERS, TOM und BART DEMOEN: *The K.U.Leuven CHR System: Implementation and Application*. In: FRÜHWIRTH, THOM und MARC MEISTER (Herausgeber): *First Workshop on Constraint Handling Rules: Selected Contributions*, Seiten 1–5. Universität Ulm, 2004.

[WRQC08] WONG, ERIC W., ANDY RESTREPO, YU QI und BYOUNGJU CHOI: *An EFSM-based test generation for validation of SDL specifications*. In: *AST '08: Proceedings of the 3rd international workshop on Automation of software test*, Seiten 25–32. ACM, 2008.

[ZHM97] ZHU, HONG, PATRICK A. V. HALL und JOHN H. R. MAI: *Software unit test coverage and adequacy*. ACM Comput. Surv., 29(4):366–427, 1997.

Zusammenfassung

Die automatische Testdatenerzeugung dient zur Identifikation von Eingaben für ein zu testendes Programm, so dass bei der Ausführung dieses Programms mit den erzeugten Eingaben eine bestimmte Testvorgabe erfüllt wird. Bekannte Testvorgaben sind etwa die Abdeckung von bestimmten Anweisungen oder Verzweigungen im Programm.

Bei der constraintbasierten Testdatenerzeugung werden Constraintlösungsverfahren verwendet, um aus den Testvorgaben Eingaben für das zu testende Programm abzuleiten.

In dieser Arbeit untersuchen wir verschiedene Strategien zur Optimierung des Durchsatzes bei der constraintbasierten Testdatenerzeugung für den Modultest. Wir widmen uns dabei insbesondere dem Einfluss sogenannter nicht-ausführbarer Wege. Zu diesen Wegen im Programm existiert keine Eingabe, für die diese Wege ausgeführt werden.

Nicht-ausführbare Wege sind ein wesentliches Problem für die constraintbasierte Testdatenerzeugung, da durch sie der Durchsatz des Testdatengenerators erheblich reduziert werden kann. Es sind bereits verschiedene Optimierungsansätze bekannt, mit denen versucht wird, diese nicht-ausführbaren Wege zu erkennen und somit den Durchsatz zu erhöhen.

Ein Ergebnis unserer Arbeit ist ein verallgemeinertes theoretisches Rahmenwerk, das es uns ermöglicht, die verschiedenen Ansätze miteinander zu verschmelzen und in der Theorie auszubauen. Grundlage des Rahmenwerks ist eine operationelle Semantik eines erweiterten Kontrollflussgraphen, den wir als Programmflussgraphen bezeichnen.

Wir können unser Rahmenwerk neben dem Quellcode auch direkt auf Maschinencode oder den Zwischencode eines optimierenden Compilers anwenden, was nicht mit allen bekannten Ansätzen möglich, bei sicherheitskritischen Anwendungen aber notwendig ist.

Dieses Rahmenwerk beinhaltet eine Reihe einfacher Grundkonstrukte, mit denen alle gebräuchlichen Testkriterien formuliert, aber in das auch weitere und neue

Kriterien integriert werden können. Kein anderer Ansatz bietet alle diese Möglichkeiten.

Neu ist ebenfalls, dass Wege durch das Programm in Teilstücken entgegen der normalen Ausführungsrichtung konstruiert werden können. Dieses Vorgehen bietet in vielen Fällen Vorteile bei der Erkennung nicht-ausführbarer Wege.

Aus dem theoretischen Rahmenwerk leiten wir ein Verfahren zur constraintbasierten Testdatenerzeugung ab, das mehrere Strategien zur Vermeidung nicht-ausführbarer Wege unterstützt. Diese Strategien können einzeln und in Kombination angewendet werden, um den Durchsatz zu erhöhen.

Mit dem erweiterten Verfahren können verschiedene Strategien direkt miteinander verglichen werden, was bisher nicht möglich war. Wir untersuchen für die betrachteten Strategien theoretisch und anhand von Experimenten Laufzeit, Terminierungsverhalten sowie statistische Eigenschaften, die für die Testfallqualität von Bedeutung sind.

Viele dieser Strategien reduzieren messbar die Laufzeit gegenüber einem Vorgehen, das den Weg im Programm in einem Stück beginnend beim Startknoten konstruiert.

Wir identifizieren auch, in welchen Fällen sowohl die bekannten als auch die von uns neu eingeführten Strategien die Laufzeit sogar erhöhen können. Dabei handelt es sich keinesfalls um pathologische, sondern in der Praxis durchaus typische Beispiele.

Da weder eine einzelne der bekannten Strategien noch eine ihrer Kombinationen optimal ist, können unsere Erkenntnisse über die Stärken und Schwächen einer Strategie die Grundlage für die weitere Optimierung sein.

Diese Arbeit liefert somit sowohl aus theoretischer als auch aus praktischer Sicht neue Anstöße für den Bereich der constraintbasierten Testdatenerzeugung:

Erstens können mit den gewonnenen Erkenntnissen neue Strategien abgeleitet werden. Durch die Verallgemeinerung können das theoretische Rahmenwerk und das daraus abgeleitete Verfahren dafür die Grundlage bilden.

Zweitens überwindet unser Verfahren die bisherigen Einschränkungen in allen in der Praxis relevanten Fällen. Es integriert zum ersten Mal die bestehenden Strategien und macht sie für die unmittelbare Anwendung auf Programmflussgraphen verfügbar. Zusammen mit der Unterstützung einer Vielzahl praktisch relevanter Testkriterien vereinfacht das Verfahren damit den Einsatz der constraintbasierten Testdatenerzeugung im industriellen Umfeld erheblich und identifiziert Möglichkeiten zur Optimierung des Durchsatzes.