Pfadbedingungen in Abhängigkeitsgraphen und ihre Anwendung in der Softwaresicherheitstechnik

Torsten Robschink

Juli 2004

Dissertation zur Erlangung des Doktorgrades
in den Naturwissenschaften

Danksagung

Meinem Doktorvater Prof. Dr. Gregor Snelting möchte ich an erster Stelle danken. Seine langjährige wissenschaftliche und persönliche Unterstützung, sein Vertrauen und Rat haben es mir ermöglicht, diese Arbeit erfolgreich abzuschließen. Ich danke Prof. Dr. Andreas Zeller für seine sofortige Bereitschaft, als weiterer Gutachter zu fungieren.

Mein ehemaliger Kollege Jens Krinke hat durch seine Forschung und Promotion dazu beigetragen, dass diese Arbeit im Rahmen des VALSOFT-Projekts erst möglich wurde. Danke, Jens! Ebenso möchte ich Bernd Fischer danken, der mich zur universitären Forschung inspirierte.

Meinen Kollegen Mirko Streckenbach, Maximilian Störzer und Christian Hammer möchte ich für die vielen Anregungen, Diskussionen und Gespräche danken, die sicherlich auch diese Arbeit beeinflusst haben.

Die studentischen Leistungen sollen nicht unerwähnt bleiben. Elisabeth Angerer-Grubmüller hat den Constraint-Solver-Anschluss implementiert (9.1). Stephan Bösebeck, Thomas Schön und Daniel Gmach implementierten die neue grafische Visualisierung (9.2). Florian Tausch übersetzte die C++-Anwendung PalME in ein ANSI-C-Programm (8.2). Martin Grimme implementierte die XAssertion-Erweiterung (6.4) und Dominik Schestauber half bei der Vereinheitlichung der Zyklenzerlegungen (7.5).

Mein besonderer Dank geht an Szilvia Thaller, die immer an mich und diese Arbeit glaubte und meine Motivation in den letzten drei Jahren beflügelte. Meinen Eltern danke ich für die erfahrene Unterstützung und dafür, dass sie mir meine bisherige Karriere ermöglicht haben.

VALSOFT wurde vom Bundesministerium für Bildung und Forschung (FKZ 01 IS 513 C9) und von der Deutschen Forschungsgemeinschaft (FKZ Sn11/5-1 and Sn11/5-2) gefördert.

Abstract

Diese Arbeit präsentiert eine neue Methode zur Sicherheitsanalyse von Software im Bereich der Manipulationsprüfung und der Einhaltung von Informationsflüssen zwischen verschiedenen Sicherheitsniveaus. Program-Slicing und Constraint-Solving sind eigenständige Verfahren, die sowohl zur Abhängigkeitsbestimmung als auch zur Berechnung arithmetischer Eigenschaften verwendet werden. Die erstmalige Kombination dieser beiden Verfahren mittels Pfadbedingungen liefert nicht nur binäre Abhängigkeitsinformationen wie Slicing, sondern exakte notwendige Bedingungen über die Informationsflüsse zwischen zwei Programmpunkten. Constraint-Solver können domänenspezifisch diese Bedingungen für Eingabevariablen auswerten und somit zeugenähnliche Variablenbelegungen ermitteln, unter denen die diagnostizierten Informationsflüsse tatsächlich stattfinden.

Neben der Definition der Grundlagen von Abhängigkeitsgraphen und einfachen Pfadbedingungen werden neue Erweiterungen für kontextsensitive interprozedurale Pfadbedingungen gezeigt und die Integration von domänenspezifischen Verfahren für Arrayfelder und abstrakten Datentypen demonstriert.

Der Schwerpunkt der Arbeit liegt in der Realisierung von Pfadbedingungen für echte Programme in echten Programmiersprachen. Hierfür werden Verfahren vorgeschlagen, realisiert und empirisch untersucht, wie Pfadbedingungen für große Programme skalieren. Die zum Einsatz kommenden Techniken sind u.a. Intervallanalyse und Binäre Entscheidungsgraphen, mit denen die generelle exponentielle Komplexität von Pfadbedingungen beherrschbar wird.

Fallstudien für den Einsatz von Pfadbedingungen und die empirische Untersuchung mehrerer Verfahren zur Intervallanalyse zeigen, dass Pfadbedingungen für die praktische Programmanalyse und das Programmverstehen geeignet und empfehlenswert sind. Zur Veranschaulichung wird hierfür eine Manipulation sicherheitskritischer Berechnungen aufgedeckt und analysiert.

Inhaltsverzeichnis

1 Einleitung **1**
1.1 Die Entstehungsgeschichte der Arbeit 1
1.2 Informationsflusskontrolle 2
1.3 Program-Slicing 6
1.4 Pfadbedingungen 6
1.5 Aufbau der Arbeit 8

2 Programmabhängigkeitsgraphen **11**
2.1 Kontrollfluss 11
2.2 Dominatoren 12
2.3 Datenfluss 13
2.4 Kontrollabhängigkeit 14
2.5 Datenabhängigkeit 14
2.6 Programmabhängigkeitsgraph 15
2.7 Erweiterung für interprozedurale Programmabhängigkeitsgraphen 17
2.8 Beeinflussung und Program-Slicing 18

3 Vom Slicing zu Pfadbedingungen **21**
3.1 Die Grenzen von Program-Slicing 21
3.2 Pfadbedingungen als informativeres Program-Slicing 22
3.2.1 Eigenschaften von Pfadbedingungen 22
3.2.2 Das Obstwaagenbeispiel 23

4 Einfache Pfadbedingungen **25**
4.1 Beeinflussung und Pfadbedingungen 25
4.2 Static-Single-Assignment-Form (SSA) 27
4.3 Grundlegende Formeln für Pfadbedingungen 28
4.3.1 Ausführungsbedingungen 29
4.3.2 Pfadbedingungen 30
4.3.3 Behandlung von Zyklen auf einzelnen Pfaden 31
4.3.4 Pfadbedingungen für SSA-Form 33
4.4 Starke und schwache Pfadbedingungen 34

4.5 Pfadbedingungen am Beispiel „mergesort“ 35

5 Komplexe Pfadbedingungen 41
5.1 Slicing und „realisierbare Knoten“ 41
5.2 Interprozedurale Pfadbedingungen 42
5.2.1 Parameterbindung an Aufrufkontexten 42
5.2.2 Funktionsaufrufe als Zerlegungspunkte 42
5.2.3 Summarybedingungen 44
5.2.4 Parametereingangskanten und Summarybedingungen . 45
5.2.5 Summarybedingungen im Falle von Rekursion 46
5.2.6 Abstiegsbedingungen 46
5.2.7 Aufstiegsbedingungen 47
5.2.8 Behandlung von Funktionsaufrufkanten 48
5.2.9 Behandlung von Funktionsrückgabewerten 49
5.3 Schleifenkontexte und widersprüchliche Pfadbedingungen . . 50

6 Pfadbedingungen für spezielle Datenstrukturen 55
6.1 Pfadbedingungen mit Datenabhängigkeitsinformationen . . . 55
6.2 Pfadbedingungen für Arrays 56
6.2.1 Array-Bedingungen am Beispiel 59
6.3 Pfadbedingungen für Abstrakte Datentypen 60
6.3.1 ADT-Bedingungen am Beispiel 62
6.4 Pfadbedingungen für Assertions 64
6.4.1 XAssertions am Beispiel 67

7 Pfadbedingungen für reale Programme 73
7.1 Anforderungen realer Programme 73
7.2 Binäre Entscheidungsgraphen 77
7.2.1 Redundante Terme 77
7.2.2 Pfadbedingungen und boolesche Algebra 78
7.2.3 Transformation von Prädikaten zu BDD-Knoten . . . 79
7.2.4 BDD vs. DNF 81
7.2.5 Empirische Untersuchung 82
7.3 Dominatoren 83
7.3.1 Bewertung 84
7.4 Zyklen 86
7.4.1 Lineare Pfadbedingungen im azyklischen PDG 87
7.4.2 Intraprozedurale oder interprozedurale Zyklen? 88
7.4.3 Empirische Untersuchung 89
7.5 Intervallanalyse 92
7.5.1 Eigenschaften von ineinander geschachtelten Zyklen . 94
7.5.2 Faktorisierungspunkte geschachtelter Zyklen 94
7.5.3 Zerlegung nach Steensgaard 96
7.5.4 Zerlegung nach Sreedhar, Gao, Lee 97

7.5.5 Zerlegung nach Havlak 99
7.5.6 Zerlegung nach Ramalingam 99
7.5.7 Vier Zyklenzerlegungen am Beispiel „looptest“ 99
7.5.8 Intervallanalyse, Spieletheorie und Komplexität im Detail 103
7.5.9 Zyklenzerlegung und Caching-Effekt am Beispiel . . . 106
7.5.10 Empirische Untersuchung 109
7.6 Pfadauswertungsbegrenzung 120
7.6.1 Empirische Untersuchung 121
7.7 Fazit 132

8 Fallstudien 135
8.1 Wackeltisch 135
8.1.1 Informationsfluss von der Kamera zur Motorsteuerung 136
8.1.2 Pfadbedingungen reduzieren Program-Slices 139
8.1.3 Entdeckung einer Sicherheitsverletzung 139
8.2 Elektronische Geldbörse 143
8.2.1 Pfadbedingung PalME 1 146
8.2.2 Pfadbedingung PalME 2 147
8.2.3 Pfadbedingung PalME 3 148
8.2.4 Pfadbedingung PalME 4 148
8.3 Fazit 149

9 VALSOFT 153
9.1 Constraint-Solving 153
9.1.1 Reduce/Redlog 154
9.1.2 ECLiPSe 155
9.1.3 Constraint-Solving in VALSOFT 155
9.1.4 Fazit 156
9.2 Visualisierung von PDGs mittels Overlaying 157
9.3 Der Pfadbedingungsgenerator „Solver“ 160
9.4 Symbolischer Ablauf einer Softwareanalyse 162

10 Verwandte Arbeiten 163

11 Zusammenfassung 167
11.1 Pfadbedingungen und ihre Anwendung in der Softwaresicherheitstechnik 167
11.2 Ausblick 168

A Weitere BDDs 171
A.1 BDDs zur Fallstudie „Wackeltisch“ 171
A.2 BDDs zur Fallstudie „PalME“ 174

B Zyklengröße und Zerlegungstiefe aller Fallstudien 177

Literaturverzeichnis **211**

Abbildungsverzeichnis

1.1 Softwaresicherheitsprüfung: Vom geeichten Gewicht zur komplexen Softwareanalyse . . . 1
1.2 Beispielprogramm mit Programmabhängigkeitsgraph . . . 7

2.1 Der Kontrollflussgraph mit zugehörendem Programm . . . 12
2.2 Dominatoren und Post-Dominatoren . . . 13
2.3 Kontrollabhängigkeiten . . . 14
2.4 Datenabhängigkeiten . . . 15
2.5 Der Programmabhängigkeitsgraph . . . 16

3.1 Obstwaagensteuerung . . . 24

4.1 (a) Einzelner Pfad, (b) Einzelner Pfad mit Zyklen . . . 31
4.2 Der Quelltext zu „mergesort“ . . . 36
4.3 Der Startknoten 177 in `main` . . . 37
4.4 Der Aufruf von `mergesort` in `main` . . . 37
4.5 Die rekursiven Aufrufe in `mergesort` . . . 38
4.6 Der Aufruf von `merge` in `mergesort` . . . 38
4.7 Der Zielknoten 90 in `merge` . . . 39

5.1 Symbolisierter Funktionsaufruf mit Summarykanten . . . 43
5.2 Zyklische Abhängigkeiten zwischen Parametern . . . 45
5.3 Funktionen, die 2^a berechnen, `pow2` (Schleife), `rpow2` (Rekursion), `nrpow2` (externe Schleife) . . . 52

6.1 Unterschiedliche Arrayrealisierungen in Programmabhängigkeitsgraphen, (a) Standard, (b) speziell für Pfadbedingungen 57
6.2 Komplexes Beispiel für präzise Array-Bedingungen . . . 59
6.3 Stärkere Pfadbedingungen mittels ADT-Bedingungen . . . 63
6.4 Formale Spezifikation in Pfadbedingungen . . . 65
6.5 Der Klassiker . . . 67
6.6 Der Klassiker mit XAssertions . . . 70
6.7 Steuerungssoftware mit XAssertions angereichert . . . 71

7.1 Ein durchgehendes Beispiel („looptest“) 77
7.2 BDD mit Literalordnung $x_1 < x_3 < x_2 < x_4$ 80
7.3 BDD mit Literalordnung $x_1 < x_2 < x_3 < x_4$ 80
7.4 Der BDD zum Programm „looptest“ 82
7.5 Mögliche Positionen für intraprozedurale Dominatoren (links) und interprozedurale Dominatoren (rechts) 85
7.6 Sharing von Funktionen 89
7.7 Programmabhängigkeitsgraph und Zyklus zu „looptest“ . . . 90
7.8 Hierarchie geschachtelter Zyklen 95
7.9 Zyklenhierarchie nach Sreedhar, Gao und Lee 101
7.10 Zyklenhierarchie nach Steensgaard 101
7.11 Zyklenhierarchie nach Havlak 102
7.12 Zyklenhierarchie nach Ramalingam 102
7.13 Pfadbedingung mit Zyklenzerlegung und Caching-Effekt . . . 106

8.1 Der „Wackeltisch“ 135
8.2 Zwei Monome der Pfadbedingung von der Kamera zur Schrittmotorsteuerung 137
8.3 Drei Monome der Pfadbedingung von der Tastatureingabe zur Schrittmotorsteuerung 141
8.4 Quelltextauszug Wackeltisch (Hauptfunktion) 142
8.5 Die elektronische Geldbörse „PalME“ 143
8.6 Pfadbedingung PalME 1 147
8.7 Pfadbedingung PalME 2 147
8.8 Pfadbedingung PalME 3 (Auszug) 148
8.9 Pfadbedingung PalME 4 (Auszug) 149
8.10 Pfadbedingung für PalME 1 in Rohform 151

9.1 Darstellung von „PalME“ mittels grafischer Repräsentation . 158
9.2 Darstellung von „PalME“ mittels Overlaying-Techniken . . . 159
9.3 Ablaufplan einer Softwareanalyse mittels VALSOFT 162

A.1 Der BDD zur Fallstudie „Wackeltisch 1“ (siehe Abschnitt 8.1.1, Seite 136) 172
A.2 Der BDD zur Fallstudie „Wackeltisch 2“ (siehe Abschnitt 8.1.3, Seite 139) 173
A.3 Der BDD zur Fallstudie „PalME 1“ (siehe Abbildung 8.6, Seite 147) 174
A.4 Der BDD zur Fallstudie „PalME 2“ (siehe Abbildung 8.7, Seite 147) 175
A.5 Der BDD zur Fallstudie „PalME 3“ (siehe Abbildung 8.8, Seite 148) 175
A.6 Der BDD zur Fallstudie „PalME 4“ (siehe Abbildung 8.9, Seite 149) 176

Tabellenverzeichnis

7.1 Die Testsuite 74
7.2 Untersuchte Chops der Testsuite 75
7.3 Performancevergleich mit/ohne BDD und unter Berücksichtigung von Zyklen (Starke Zusammenhangskomponenten) . . 91
7.4 Zerlegung nach Sreedhar, Gao und Lee 110
7.5 Zerlegung nach Steensgaard 111
7.6 Zerlegung nach Havlak 112
7.7 Zerlegung nach Ramalingam 113
7.8 Performancevergleich der Zyklenzerlegungsverfahren 114
7.9 Pfadbedingung mit Sreedhar/Gao/Lee-Zerlegung 116
7.10 Pfadbedingung mit Steensgaard-Zerlegung 117
7.11 Pfadbedingung mit Havlak-Zerlegung 118
7.12 Pfadbedingung mit Ramalingam-Zerlegung 119
7.13 Pfadauswertungsbegrenzung mit Sreedhar/Gao/Lee-Zerlegung 123
7.13 Pfadauswertungsbegrenzung mit Sreedhar/Gao/Lee-Zerlegung 124
7.13 Pfadauswertungsbegrenzung mit Sreedhar/Gao/Lee-Zerlegung 125
7.14 Pfadauswertungsbegrenzung mit Steensgaard-Zerlegung . . . 125
7.14 Pfadauswertungsbegrenzung mit Steensgaard-Zerlegung . . . 126
7.14 Pfadauswertungsbegrenzung mit Steensgaard-Zerlegung . . . 127
7.15 Pfadauswertungsbegrenzung mit Havlak-Zerlegung 127
7.15 Pfadauswertungsbegrenzung mit Havlak-Zerlegung 128
7.15 Pfadauswertungsbegrenzung mit Havlak-Zerlegung 129
7.15 Pfadauswertungsbegrenzung mit Havlak-Zerlegung 130
7.16 Pfadauswertungsbegrenzung mit Ramalingam-Zerlegung . . . 130
7.16 Pfadauswertungsbegrenzung mit Ramalingam-Zerlegung . . . 131
7.16 Pfadauswertungsbegrenzung mit Ramalingam-Zerlegung . . . 132

9.1 Komponenten zur Pfadbedingungsberechnung 160

Kapitel 1

Einleitung

1.1 Die Entstehungsgeschichte der Arbeit

Abbildung 1.1: Softwaresicherheitsprüfung: Vom geeichten Gewicht zur komplexen Softwareanalyse

Die Physikalisch-Technische Bundesanstalt (PTB) in Braunschweig ist das nationale Metrologie-Institut mit wissenschaftlich-technischen Dienstleistungsaufgaben. Sie betreibt Grundlagenforschung und Entwicklung in Bereichen zur Bestimmung von Fundamental- und Naturkonstanten. Weiterhin arbeitet sie an der Darstellung, Bewahrung und Weitergabe der gesetzlichen Einheiten des SI[1], der Sicherheitstechnik, Dienstleistung und Messtechnik für den gesetzlich geregelten Bereich und die Industrie. Über Kooperationen mit Forschungsinstituten und Universitäten sorgt sie zudem für einen

[1]Standard for Use of the International System of Units

Technologie-Transfer, über den die Vorarbeiten und Grundlagen dieser Arbeit geschaffen wurden.

In Deutschland müssen technische Systeme wie Mess- und Steuerungsgeräte einer Bauartzulassung unterzogen werden [Bun97]. Die Zulassungsprüfung erstreckt sich auf die Einhaltung der Anforderungen des Eichgesetzes und der Eichordnung. Bei der Zulassungsprüfung wird untersucht, ob die Geräte die Anforderungen an Messrichtigkeit und Zuverlässigkeit für die vorgesehene Dauer der Eichgültigkeit einhalten werden. Wichtige Prüfkriterien sind dabei die Beständigkeit der Messgeräte gegenüber Umgebungseinflüssen, aber insbesondere auch der Schutz gegen *Verfälschung von Messwerten* durch Bedienungsfehler oder Manipulation. Dies gilt auch für die *Software* rechnergesteuerter Geräte. Zusatzeinrichtungen und Schnittstellen an Messgeräten werden u.a. auf Rückwirkungsfreiheit untersucht, so dass keine unzulässige Beeinflussung der Messwerte des eichpflichtigen Gerätes möglich ist.

Steuerungssoftware wird in Waagen, Zählern, z.B. in der Verkehrsüberwachung oder der Lebensmittelindustrie eingesetzt. Um die Bedeutung und Notwendigkeit vertrauenswürdiger und damit manipulationsfreier Software zu veranschaulichen, sind einige reale Systeme aufgeführt, die bei Fehlfunktion erhebliche wirtschaftliche Schäden oder Personenschäden verursachen können: Tankzapfsäulen, Obstwaagen im Supermarkt, Kraftfahrzeuge (Airbag-Controller), Flugzeug-Leitsysteme, medizinisches Gerät (Tomographen), Geldautomaten, Überwachungssysteme, Energieerzeuger (Kraftwerks-Controller), Betriebssysteme, Sicherheitssoftware (Firewalls und Emailprogramme).

All diese Systeme besitzen sicherheitskritische (ggf. auch eichwürdige) und nicht-sicherheitskritische Bereiche. Das Ziel dieser Arbeit ist u.a., Verfahren vorzustellen, mit denen die Ursachen für unerwünschte Informationsflüsse zwischen nicht-kritischen und kritischen Bereichen ermittelt werden können.

1.2 Informationsflusskontrolle

Der vorherige Abschnitt zeigt, dass sehr viele Aufgaben des täglichen Lebens durch technische Systeme übernommen wurden. Die Informationstechnik hat alle gesellschaftlichen Bereiche erfasst. Daher ist es besonders wichtig, dass wir den Systemen, die für uns arbeiten, auch vertrauen können. Der beste Ansatz hierfür sind integrierte Sicherheitsfunktionen sowohl in der Soft- als auch in der Hardware, da Prüfungen meistens erst nach einem registrierten Auftreten von Fehlfunktionen durchgeführt werden.

Software-Prozesse und entsprechende Prozessmanagementsysteme als The-

men der 90er Jahre, mit denen formalisierte Darstellungen aller Verfahrensschritte der Prozesse geschaffen und die Implementierung verwaltet werden können, helfen hierbei wenig. Die Integrität der Systeme kann z.B. durch Prozesse wenig verbessert werden, wenn die Integritätsverletzung aufgrund von Manipulationen verursacht wird, die aus der Granularität der Prozessbeschreibungen herausfallen. Typisch wäre hierbei z.B. die mutwillige Manipulation eines spezifizierten Algorithmus, so dass dieser unter bestimmten Bedingungen ein nicht-spezifiziertes Verhalten aufweist und für dieses Fehlverhalten keine Testfälle vorliegen.

Ebensowenig können das amerikanische Capability Maturity Model (CMM,[PCCW93]) und die deutsche ISO/DIN9000-Norm [Bal98] Integritätsverletzungen der oben beschriebenen Art aufdecken, da beide im Hinblick auf besseres Prozessmanagement ausgerichtet sind und somit nur die Gesamtheit der Entwicklung, wie Abschätzbarkeit, Reproduzierbarkeit und Entwicklungsqualität optimieren. Ein Verhalten außerhalb der Spezifikation kann hierbei nur innerhalb der integrierten Qualitätssicherung erfasst werden und dies nur dann, wenn geeignete Testfälle vorliegen oder Code-Review-Prozesse unerwünschtes Softwareverhalten offenlegen.

Generell ist gegenseitiges Code-Review vom Preis-/Leistungsverhältnis eines der besten Verfahren, um Software fehlerfreier, wartungsfreundlicher und robuster zu machen. Falsches bzw. unerwünschtes Verhalten kann so durch Kontrolle und Gegenkontrolle in vielen Fällen vermieden werden. Zusammenfassend steht nach [SL02] Code-Review für:

- mehr Fehler in kürzerer Zeit zu finden,
- jungen Entwicklern durch erfahrenen Entwicklern zu helfen,
- häufige und immer wiederkehrende Fehler zu vermeiden,
- den Informationsaustausch im Team und damit auch die Problemlösungen zu verbessern und
- deutlich wartungsfreundlicheren Code zu realisieren.

Allerdings steigen mit der Softwarekomplexität auch die Anforderungen an die Reviewer, so dass softwaretechnische Hilfssysteme unvermeidlich werden.

In besonders sicherheitskritischen Bereichen, wie der Luft- und Raumfahrt existieren für zentrale Berechnungen mehrfach redundante Systeme, deren Algorithmen von unabhängigen Entwicklungsteams nach einer präzisen Spezifikation realisiert werden. Durch diese Unabhängigkeit können zwar potentielle Integrationsverletzungen eines Systems nicht verhindert werden, jedoch kann durch die Mehrheitsabstimmung der Effekt der möglicherweise angestrebten Manipulation nicht eintreten. Aufgrund der mehrfachen Kosten (Softwareentwicklung und Hardware) werden redundante Systeme nur in

kostenintensiven und kritischen Systemen wie Flugzeugen und Space Shuttles eingesetzt, um mögliche Schäden auf ein Mindestmaß zu reduzieren.

Anders sieht dies mit nicht-kritischen Systemen wie z.B. Tankzapfsäulen und elektronischen Waagen aus, die bei falscher Berechnung keinen kritischen Schaden, sondern (nur) volkswirtschaftliche Schäden verursachen können. Aufgrund einer hohen Anzahl an installierten Geräten können diese Schäden jedoch immens werden.

Das Vertrauen in die Informationstechnik kann nur dann gewährleistet werden, wenn Verlässlichkeit in Bezug auf Anwendungen, insbesondere auch bei der Sicherheit der verarbeiteten Daten, gewährleistet ist. Hierfür sind einheitliche Sicherheitskriterien notwendig, wie diese z.B. vom Bundesamt für Sicherheit in der Informationstechnik (BSI) und der PTB aufgestellt und in einer durch ein Akkreditierungsverfahren autorisierten Zertifizierungsstelle [BSI03b] überprüft werden. Diese Überprüfung erfolgt derzeit bei Software durch manuelles Code-Review und ist daher sehr zeit-, kosten- und fehlerintensiv.

Die Zertifizierung der Systeme erfolgt nach international anerkannten Kriterien [BSI03a], z.B. Trusted Computer Security Evaluation Criteria (Orange Book–TCSEC, 1983), IT-Sicherheitskriterien (ITS, 1989), Information Technology Security Evaluation Criteria (ITSEC, 1991), Common Criteria Version 2.0 (CC, 1998) und Common Evaluation Methodology for Information Technology Security, Teil 2 (CEM, 1999). Diese Standards sind für die Wirtschaft und die Verbraucher von großer Bedeutung, da sie internationale Vergleichsmöglichkeiten der Produkte bzgl. einer „Sicherheitsqualität" bieten.

Die Grundlage dieser Standards ist die *Informationsflusskontrolle* (Information Flow Control [Com99]). Definiert über sog. Schutzprofile spezifiziert sie Anforderungen an bestimmte Informationsflüsse innerhalb eines IT-Systems, um die Informationsflüsse für Benutzer besser zu schützen. Dabei werden den etablierten Schutzkonzepten, wie Zugriffs- und Übertragungsschutz, neue Aspekte wie Vertraulichkeit, Integrität und Authentizität hinzugefügt.

Einige klassische *Sicherheitsmodelle* nach [Hun03] sind:

- Bell/La Padula [BL73, BL76]: Das Modell beschreibt eine Zugriffskontrolle und besteht aus Objekten (Dokumenten) und Subjekten (Verwender), von denen ein Subjekt auf ein Quellobjekt lesenden und auf ein Zielobjekt schreibenden Zugriff haben kann, wenn das Zielobjekt mindestens denselben Sicherheitsgrad wie das Quellobjekt hat. Informationsflüsse von Objekten mit höheren Sicherheitsgraden zu Objekten mit niedrigeren Sicherheitsgraden sind in diesem Modell ausgeschlossen. Subjekte, die selbst einen Sicherheitsgrad haben müssen, wird der lesende und schreibende Zugriff auf Objekte mit höherem Si-

cherheitsgrad verweigert. Dieses Modell war das erste mathematische Modell einer mehrschichtigen Sicherheitspolitik und diente als Grundlage der nachfolgenden Modelle.

- Biba [Bib77]: Bei diesem Modell handelt es sich wie beim Bell/La Padula-Modell um eine Zugriffskontrolle mit Sicherheitsstufen. Der Schwerpunkt liegt in der Wahrung der Integrität der Objekte.

- Sandhu/Coyne/Feinstein/Youman [SCFY96]: Um die Zugriffsverwaltung zu vereinfachen, werden rollenbasierte Zugriffskontrollen eingeführt. Statt die Objekte und Subjekte einzeln zu betrachten, werden sie als Mengen aufgefasst und Relationen zwischen ihnen hergestellt.

- Goguen/Meseguer [JG84]: Nebem dem Bell/La Padula gehört das Goguen/Meseguer-Modell zu den am häufigsten zitierten Sicherheitsmodellen. Es basiert auf Nicht-Interferenz (engl. non-interference) und partitioniert ein System in eine endliche Menge von Sicherheitsdomänen. Über die Nicht-Interferenz wird verhindert, dass eine Domäne das beobachtbare Verhalten einer anderen Domäne beeinflussen kann. Über ein Nicht-Interferenz-Kriterium kann für ein System anschließend bestimmt werden, ob es in Bezug zu einer Nicht-Interferenz-Relation sicher ist.

Das Bundesamt für Sicherheit in der Informationstechnik veröffentlichte einen Leitfaden für die Erstellung und Prüfung formaler Sicherheitsmodelle [BSI03c], an dem sich Firmen und Behörden orientieren können. Die PTB arbeitet z.B. nach dem Goguen/Meseguer-Modell und fordert, dass zu zertifizierende Software, die technische Systeme steuert, in zwei Bereiche partitioniert wird:

- Kritische Bereiche werden durch sog. *Eichpfade* beschrieben, die den Informationsfluss von der Eingabeeinheit (z.B. Sensor, Datenfeld) zur Ausgabeeinheit (z.B. Display, Datenfeld) beschreiben.

- Nicht-kritische Bereiche, deren Informationsflüsse innerhalb der Partition nicht eingeschränkt werden.

Unter Nicht-Interferenz versteht man, dass Informationen auf den Eichpfaden nicht durch Fremdinformationen außerhalb der Eichpfade beeinflusst werden. Dies bedeutet, dass Daten und Algorithmen getrennter Bereiche keine Informationen untereinander austauschen.

Während der Entwicklung von Software können die oben genannten Kriterien zur Informationsflusskontrolle für die Qualitätssicherung verwendet werden, in sicherheitskritischen Bereichen sind sie im Rahmen der Zertifizierung fest vorgeschrieben. Bei bereits designten oder bestehenden Softwareapplikationen wurden diese Kriterien jedoch nicht unbedingt berücksichtigt, da

entweder der Einsatzzweck ein anderer war oder erst die Wiederverwendung von Komponenten die Einführung von Sicherheitskriterien notwendig machte. Um bei bestehenden Programmen die Erfüllung dieser Sicherheitskriterien zu gewährleisten und bei neu zu entwickelnden Programmen die Überprüfung zu erleichtern, muss *Programmanalyse* mit folgender Zielrichtung eingesetzt werden:

1. Es muss bestimmt werden, welche Programmteile andere beeinflussen.
2. Nach Feststellung, welche Beeinflussungen legitim sind und welche tatsächlich die Sicherheit der Eichpfade verletzen, muss bestimmt werden, wie die Beeinflussung erfolgt.

1.3 Program-Slicing

Für die Praxis bedeutet der Begriff „Beeinflussung“, dass eine Anweisung auf eine Berechnung einer anderen Anweisung einwirkt. Mittels *Program-Slicing* [Wei84, Wei79] lassen sich nun für eine Zielanweisung (Slice-Kriterium) alle Anweisungen bestimmen, die einen Einfluss auf das Ergebnis und die Erreichbarkeit des Slice-Kriteriums haben. Wenn Anweisung A von Anweisung B beeinflusst wird, dann ist A *abhängig* von B.

Diese Arbeit beruht auf der Beeinflussungserkennung und Ursachenanalyse einer graphischen Repräsentation der Programme, dem sog. Programmabhängigkeitsgraphen [FOW87] (PDG, engl. program dependence graph). Er besteht aus Knoten für die Anweisungen und Kanten für deren Abhängigkeiten.

Umfassende Details zu Program-Slicing und Programmabhängigkeitsgraphen finden sich in der Dissertation von Jens Krinke [Kri03], dessen Arbeit die Grundlage für diese Arbeit bildet. Aufgrund der Symbiose dieser beiden Arbeiten zu einem umfassenden Analysesystem für Program-Slicing und Pfadbedingungen wird auf Program-Slicing und PDGs nicht vertieft eingegangen.

1.4 Pfadbedingungen

Die Abbildung 1.2 zeigt ein einfaches Programm mit dem zugehörigem PDG. Gestrichelte Linien stellen Kontrollabhängigkeiten dar, wobei die Kantenquelle die Ausführung des Kantenziels regiert. Durchgezogene Linien stellen Datenabhängigkeitskanten dar, bei denen Kantenquellen Definitionstellen von Variablen sind und Kantenziele Verwendungen derselben Variablen, ohne dass die Variablen auf den Abhängigkeitspfaden umdefiniert werden.

```
sum = 0;
mul = 1;
a = 1;
b = read();
while (a <= b) {
  sum = sum + a;
  mul = mul * a;
  a = a + 1;
}
write(sum);
write(mul);
```

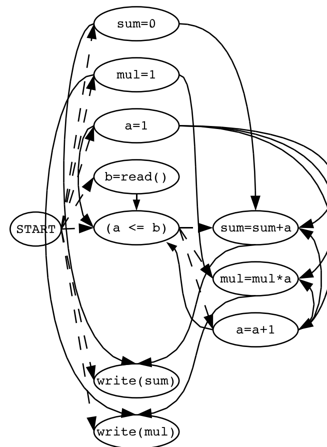


Abbildung 1.2: Beispielprogramm mit Programmabhängigkeitsgraph

Hat `read()` einen Einfluss auf `write(sum)`? Diese Frage kann mit Program-Slicing gelöst werden: die Antwort ist „ja“. Bei größeren Programmen, besonders wenn Funktionsaufrufe enthalten sind, reicht eine binäre Information nicht mehr aus. Zwischen den beiden Anweisungen gibt es Abhängigkeiten, da Pfade im PDG existieren. Interessant werden aber erst die genauen Umstände, unter denen Informationsflüsse zwischen zwei Programmpunkten auftreten. Dies berechnen *Pfadbedingungen*. Zwischen `read()` und `write(sum)` kann es nur dann einen Informationsfluss geben, wenn die Bedingung $a \leq \text{read}()$ erfüllt ist. Sollte $a > \text{read}()$ gelten, liefert Program-Slicing immer noch „ja“, jedoch wird es zur Laufzeit niemals einen Informationsfluss geben.

Diese gezeigte Pfadbedingung ist trivial, jedoch wurden hierfür alle Pfade zwischen den beiden Anweisungen betrachtet, was zu teilweiser exponentieller Komplexität führen kann. Im Gegensatz zu bereits bekannten Kontrollfluss-basierten Verfahren (siehe Kapitel 10) werden erstmalig präzise Bedingungen für interprozedurale Informationsflüsse (Daten- und Kontrollabhängigkeiten) berechnet und auf *reale* Programme angewendet. Es ist kein Verfahren bekannt, das ähnlich genaue Bedingungen liefert.

Präzise Bedingungen für reale Programme erfordern die Kombination verschiedener Techniken, um die exponentielle Komplexität zu beherrschen. Der Programmabhängigkeitsgraph wird hierzu partitioniert, um Pfade sinnvoll voneinander abzugrenzen (z.B. bei Funktionsaufrufen). Binäre Entscheidungsgraphen verwalten die prädikatenlogischen Bedingungen, und Verfahren aus der Spieletheorie reduzieren den Graphen auf notwendige Pfade. Zusammen mit einer Intervallanalyse, die Zyklen in eine Hierarchie von ineinander-geschachtelten Zyklen zerlegt, wird in Kombination mit einer guten Heuristik zur Pfadlängenauswertung eine Methode geliefert, die für

reale Programme in realen Programmiersprachen einsetzbar ist.

Die Intervallanalyse hat sich als Schlüssel zum Erfolg herausgestellt und wird umfassend durch verschiedene Realisierungen von Zyklenzerlegungen empirisch untersucht. Die Heuristik liefert zudem die Erkenntnis, dass ein Großteil von Pfaden lokal denselben Bedingungen zugeordnet ist, so dass eine Komplettauswertung in den meisten Fällen gar nicht notwendig ist, um die gesuchten Ursachen für Informationsflüsse zu ermitteln.

1.5 Aufbau der Arbeit

Die vorgelegte Arbeit gliedert sich in die folgenden Teile:

- In Kapitel 2 werden die Grundlagen von Abhängigkeiten, Programmabhängigkeitsgraphen und Program-Slicing wiederholt, da auf ihnen die Berechnung von Pfadbedingungen basiert.
- Kapitel 3 beleuchtet den Unterschied zwischen Program-Slicing und Pfadbedingungen. Dabei werden die Grenzen und Probleme der bisherigen Analysetechniken analysiert.
- Mit den formalen Grundlagen zu Pfadbedingungen als fundamentale Erweiterung zum Program-Slicing beschäftigt sich Kapitel 4.
- Kapitel 5 stellt neue Techniken zur Berechnung von interprozeduralen Pfadbedingungen vor.
- Neue Erweiterungen zu Pfadbedingungen für die besondere Behandlung von Arrays und abstrakten Datentypen werden in Kapitel 6 präsentiert. Zusätzlich wird ein Verfahren demonstriert, wie mittels Quelltextannotationen Pfadbedingungen noch präziser gemacht werden können.
- Kapitel 7 zeigt als neues Verfahren, wie durch den Einsatz von Binären Entscheidungsgraphen, einer hierarchischen Zerlegung von Zyklen und einer heuristischen Pfadauswertungsbegrenzung es möglich wird, trotz einer exponentiellen Berechnungskomplexität Pfadbedingungen für reale Programme zu berechnen. Eine umfangreiche empirische Untersuchung belegt die Ergebnisse.
- Mittels anschaulicher Fallstudien realer Programme wird in Kapitel 8 der Nutzen von Pfadbedingungen in der Softwaresicherheits-Analyse belegt.
- Mit der Realisierung des Pfadbedingungsgenerators innerhalb des VALSOFT-Frameworks beschäftigt sich Kapitel 9. Hier werden zu-

dem Constraint-Solving-Verfahren beschrieben und ein neues Visualisierungswerkzeug für Programmabhängigkeitsgraphen vorgestellt.

- Kapitel 10 und 11 gehen auf verwandte Arbeiten zur Softwareanalyse ein und fassen die Arbeit zusammen.
- Im Anhang A und B werden die Binären Entscheidungsgraphen aus Kapitel 8 gezeigt und Details zu den hierarchischen Zyklenzerlegungsverfahren aus Kapitel 7 gegeben.

Die hier beschriebene Arbeit ist implementiert und Bestandteil des Analyse-Systems VALSOFT[KS98]. Sie nutzt die grundlegenden Datenstrukturen und den integrierten Slicer. VALSOFT ist ein Analyse-Framework für ANSI-C, das eine Vielzahl von Techniken nutzt, um den vollen Sprachumfang, einschl. Seiteneffekten, Funktionen, Standard-Bibliotheken, Zeigern und unstrukturierten Kontrollfluss zu erfassen.

Kapitel 2

Programmabhängigkeitsgraphen

Pfadbedingungen werden über graphische Programmrepräsentationen, den Programmabhängigkeitsgraphen, berechnet. Dieses Kapitel stellt die Grundlagen der Kontrollflussanalyse, Datenflussanalyse und Programmabhängigkeitsgraphen dar, soweit sie für die Berechnung von Pfadbedingungen erforderlich sind. Für eine umfassende Beschreibung wird auf die Ausführungen in [Kri03] verwiesen, in denen die unterschiedlichen Ansätze zum Berechnen der obigen Verfahren und Graphen veranschaulicht werden.

2.1 Kontrollfluss

Mit der Kontrollflussanalyse ermittelt man den hierarchischen Fluss (Ablauf, Kontrolle) innerhalb eines Programms. Aufgrund der relativ komplexen Ablaufstrukturen von Programmen, besonders unstrukturierten sprungbehafteten, hat sich eine graphische Darstellung bewährt. Zum einen gibt es eine triviale Darstellung auf globaler Programmebene, den *Aufrufgraphen* (engl. call graph), der die hierarchische Darstellung der Funktionsaufrufe enthält; zum anderen gibt es den *Kontrollflussgraphen* (engl. control flow graph, CFG) auf Anweisungsebene. Für Compileroptimierungen werden auch mehrere Anweisungen zu sog. Basic-Blocks zusammengefasst. Der Kontrollflussgraph ist ein gerichteter Graph $G = \{N, E, n_s, n_e, \alpha\}$ mit N Knoten und E Kanten, einem Startknoten $n_s \in N$ und Endknoten $n_e \in N$. Jeder Knoten repräsentiert eine Anweisung, und jede Kante repräsentiert den Kontrollfluss zwischen zwei Knoten. Aufgrund unterschiedlicher logischer Auswertungen der Prädikatskonstrukte (in ANSI-C z.B. *if*, *while*, *switch*, ...) werden die Kanten entsprechend durch $\alpha : E \rightarrow \{\mathit{true}\ , \mathit{false}\ , \alpha\}$ attribuiert. Für ein

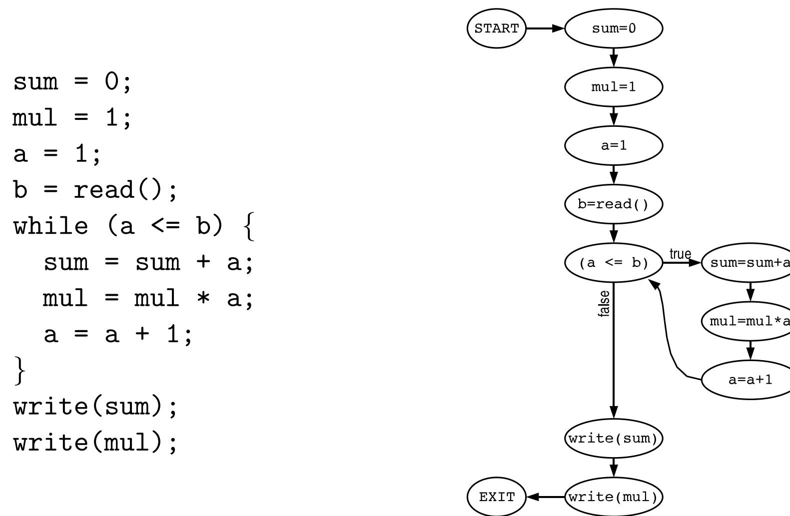


Abbildung 2.1: Der Kontrollflussgraph mit zugehörendem Programm

Beispielprogramm ist der CFG in Abb. 2.1 dargestellt.

2.2 Dominatoren

Programme haben die Eigenschaft, dass disjunkte Ausführungspfade den gemeinsamen Kontrollfluss spalten können, jedoch an späterer Stelle wieder zusammenlaufen lassen. Diese zusammenlaufenden Punkte nennt man *Dominatoren*. Seien $n, d \in N$ und d dominiert n, dann gilt für jeden möglichen Ausführungspfad vom Startknoten n_s nach n, dass d erreicht wird. d ist somit ein Dominator von n. In umgekehrter Richtung spricht man vom *Post-Dominator* d, wenn auf jedem möglichen Ausführungspfad von n zum Endknoten n_e der Knoten d erreicht wird. Dominatoren haben folgende Eigenschaften:

- Jeder Knoten dominiert sich selbst
- Startknoten dominieren alle untergeordneten Knoten
- Schleifeneintrittsknoten dominieren alle Knoten innerhalb der Schleife

Weiterhin existieren „direkte Dominatoren“ (engl. immediate dominators, *idom*), mit besonderen Eigenschaften für einen Knoten n:

- *idom(n)*$\neq$*n*
- *idom(n)* ist der letzte Dominator auf allen Pfaden von n_s nach n
- Jeder Dominator hat genau einen *idom*

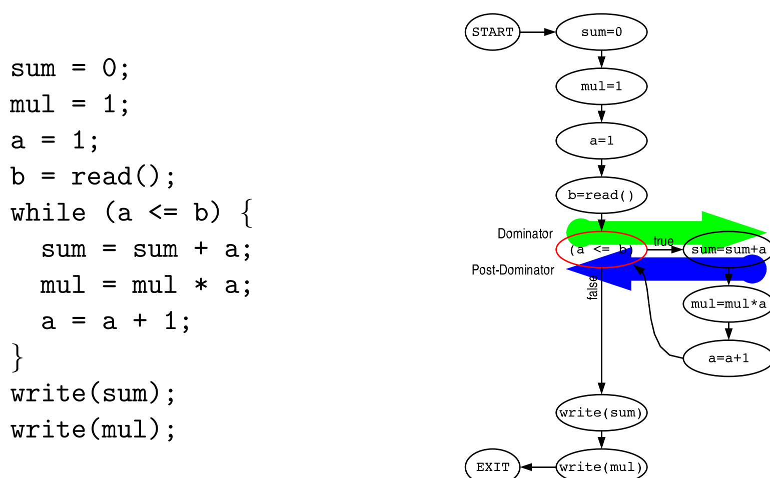


Abbildung 2.2: Dominatoren und Post-Dominatoren

Aus direkten Dominatoren lässt sich z.B. für einen Kontrollflussgraphen ein *Dominatorbaum* erstellen, der die transitiven Dominatorbeziehungen des gesamten Programms widerspiegelt. Hierbei wird der Startknoten n_s zur Wurzel des Baums, und jeder Knoten n aus dem Kontrollflussgraphen erhält als Eltern-Knoten seinen *idom(n)*. Angewandt auf alle Knoten entsteht so eine Baumstruktur. Die Abbildung 2.2 zeigt, dass der Dominator `a<=b` alle innerhalb der Schleife auftretenden Anweisungen dominiert. Er stellt zugleich für diese Anweisungen den Post-Dominator dar.

2.3 Datenfluss

Die Datenflussanalyse propagiert Informationen über die Daten des Programms an andere Programmpunkte. Zu den Einsatzbereichen zählen z.B. Programmoptimierungen wie Eliminierung gemeinsamer Teilausdrücke (engl. common subexpression elimination) und Konstantenpropagation (engl. constant propagation). Datenflussanalyse basiert auf *Erreichbarkeit* (engl. reaching definitions) in einem Flussgraphen G. Eine Variablendefinition (bzw. Speicherbereichsdefinition) $d \in G$ weist (möglicherweise) einen Wert einer Variablen v zu. Man spricht von einer Zuweisung an v in d. Diese Definition erreicht einen Programmpunkt $n \in G$, wenn es einen Pfad $d \rightarrow^* n \in G$ gibt, ohne dass v auf dem Pfad umdefiniert wird. Pfade dieser Art können im CFG existieren, jedoch zur Laufzeit nicht ausführbar sein (engl. infeasible). Der Grund hierfür ist die generelle Unentscheidbarkeit, da Datenflüsse von Eingaben, dem Ergebnis von bedingten Ausdrücken und der Terminierung von Schleifen abhängen. Die Datenflussanalyse kann daher nur Approximationen liefern, die für Pfadbedingungen konservativ sind.

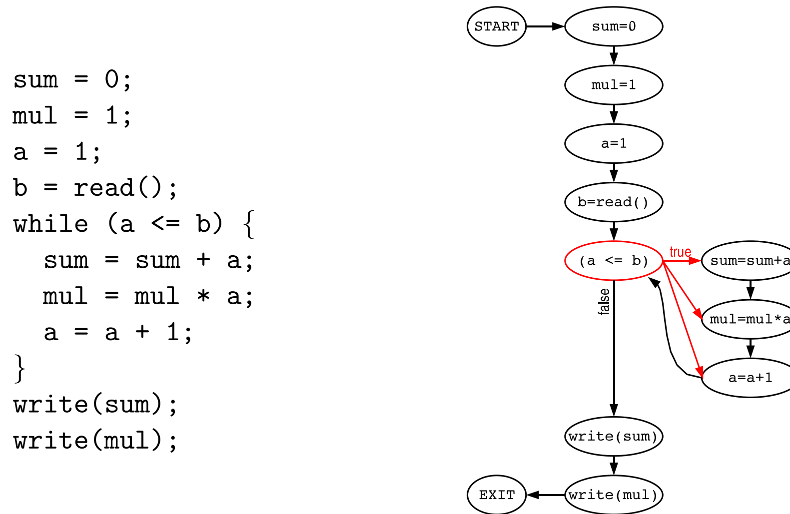


Abbildung 2.3: Kontrollabhängigkeiten

2.4 Kontrollabhängigkeit

Abhängigkeiten zwischen zwei Anweisungen stellen Relationen dar, die die Ausführungsreihenfolge beschränken. Es gibt Kontrollabhängigkeiten, die aus dem Kontrollfluss entstehen und Datenabhängigkeiten, die durch den Datenfluss zwischen Anweisungen bestimmt werden.

Definition 2.1 (Kontrollabhängigkeit)
Sei $G = \{N, E, n_s, n_e, \alpha\}$ ein Kontrollflussgraph. Sei $C \subseteq N$ die Menge der Kontrollprädikatsknoten. Sei $n \in C$ ein Kontrollprädikat, typischerweise die Bedingung eines **if**-, **while**- oder **switch**-Ausdrucks.

Ein Knoten n' ist *kontrollabhängig* vom Kontrollprädikatsknoten n, gdw. die Ausführung von n' abhängig von der Erfüllung von n ist.

Für die Definition über Dominatoren gilt: 1. Es existiert ein Pfad $n \rightarrow^* n'$; 2. n' ist ein Post-Dominator für jeden Knoten auf $n \rightarrow^* n'$, außer n; 3. n' ist kein Post-Dominator für n. Im Beispielprogramm in Abb. 2.3 sind 3 Knoten kontrollabhängig vom Prädikat.

2.5 Datenabhängigkeit

Äquivalent zu Kontrollabhängigkeiten stellen Datenabhängigkeiten die Relation „Definition $\rightarrow$ Verwendung“ dar.

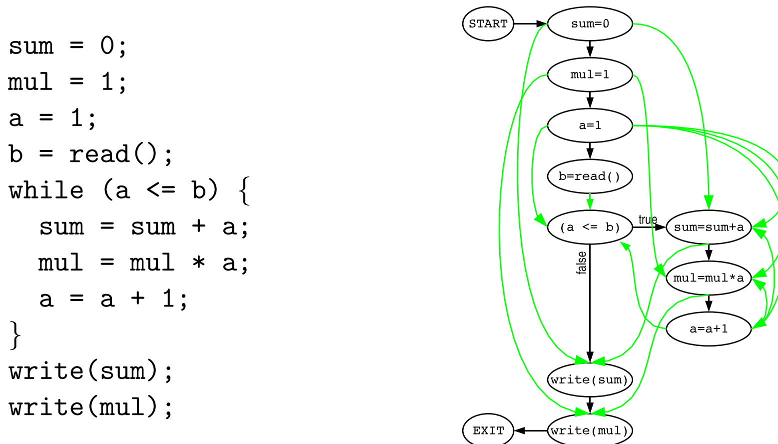


Abbildung 2.4: Datenabhängigkeiten

Definition 2.2 (Datenahängigkeit)
Sei $G = \{N, E, n_s, n_e, \alpha\}$ ein Kontrollflussgraph. Sei $\mathbf{def}() \in N$ eine Variablendefinition und $\mathbf{use}() \in N$ eine Variablenverwendung.

Ein Knoten n' ist *datenabhängig* von Knoten n, gdw.

1. ein Pfad $n \rightarrow^* n'$ existiert,
2. für eine Variable v gilt: $v \in \mathbf{def}(n)$ und $v \in \mathbf{use}(n')$ und
3. $\forall_{k \in n \rightarrow^* n', k \neq n} : v \notin \mathbf{def}(k)$

Die Abbildung 2.4 zeigt entsprechende Abhängigkeiten im Beispielprogramm.

2.6 Programmabhängigkeitsgraph

Die beiden oben genannten Abhängigkeitsrelationen eines Programms können durch einen gerichteten Graphen repräsentiert werden, dem *Programmabhängigkeitsgraphen* (engl. program dependence graph, PDG [OO84, FOW87]). Knoten stellen Anweisungen dar, Kanten Abhängigkeiten.

Definition 2.3 (Kontrollabhängigkeitskante)
Sei $n \in C$ ein Kontrollprädikat. Alle Kontrollabhängigkeiten $n \rightarrow_\alpha n'$ werden durch *Kontrollabhängigkeitskanten* $n \overset{\alpha}{\rightarrow}_n n'$ modelliert. Die Menge der Kontrollabhängigkeitskanten wird mit $\rightarrow_C$ bezeichnet. Sie enthält zudem alle unbedingten Kontrollabhängigkeitskanten.

```
sum = 0;
mul = 1;
a = 1;
b = read();
while (a <= b) {
  sum = sum + a;
  mul = mul * a;
  a = a + 1;
}
write(sum);
write(mul);
```

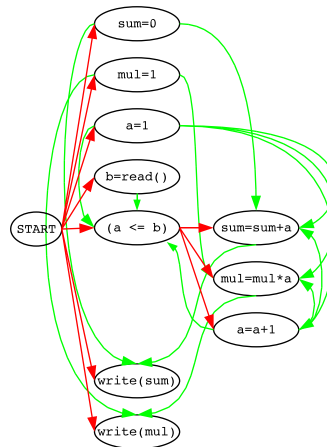


Abbildung 2.5: Der Programmabhängigkeitsgraph

Definition 2.4 (Kontrollabhängigkeitsgraph)
Der gesamte Untergraph $(C, \rightarrow_C)$ stellt den *Kontrollabhängigkeitsgraphen* dar.

Ein Kontrollprädikat $n \in C$ muss zu *true*, *false* oder einem Zahlenwert[1] ausgewertet werden, damit die Programmausführung entlang der Kontrollabhängigkeitskante $n \stackrel{\alpha}{\rightarrow}_n n' \subseteq \rightarrow_C$ stattfinden kann.

Definition 2.5 (Datenabhängigkeitskante)
Alle Datenabhängigkeiten $\mathbf{def}(n) \rightarrow \mathbf{use}(n')$ werden durch *Datenabhängigkeitskanten* modelliert.

Der Programmabhängigkeitsgraph $(N, \rightarrow)$ wird nun folgendermaßen aus dem CFG konstruiert:

- Die Kontrollflusskanten werden entfernt.
- Alle Datenabhängigkeitskanten werden eingefügt.
- Alle Kontrollabhängigkeitskanten werden eingefügt.

Die Abbildung 2.5 zeigt den resultierenden intraprozeduralen PDG. Statt eines Start- und Endknotens existiert nur noch ein Funktionseintrittsknoten. Alle ungeschachtelten Anweisungen sind automatisch von diesem Knoten unbedingt (*true*) kontrollabhängig.

Der in dieser Arbeit verwendete PDG ist feingranular und enthält daher Abhängigkeiten innerhalb einzelner Ausdrücke. Die Feingranularität bietet

[1] In der Programmiersprache ANSI-C dürfen die Zahlenwerte hierbei nur ganzzahlig sein.

für Pfadbedingungen die Möglichkeit, Substitutionen (engl. copy propagation) effizient anzuwenden. Die Repräsentation dieser PDGs ist vollständig, Syntaxbaum- und Quelltext-nah und kommt daher ohne weitere externe Daten aus. Aufgrund der besonderen Anforderungen findet jedoch keine Transformation in die Static-Single-Assignment-Form [CFR+91] statt.

Die grundlegenden Pfadbedingungen basieren vollständig auf dieser Programmrepräsentation, so dass sich der Pfadbedingungsgenerator für beliebige Programmiersprachen, die diesen Standard beibehalten, anwenden lässt. Dies gilt auch für die Erweiterung auf interprozedurale Programmabhängigkeitsgraphen.

2.7 Erweiterung für interprozedurale Programmabhängigkeitsgraphen

Um reale Programme mit Funktionen modellieren zu können, muss der Programmabhängigkeitsgraph durch zusätzliche Knoten und Kanten erweitert werden [HRB90]. Für jede Funktion wird ein eigener PDG berechnet, der einen übergeordneten Funktionseintrittsknoten (entry) erhält. Die Parameter der Funktion werden durch neue (formale) Knoten dargestellt und in Eingangsparameter (formal-in) und Ausgangsparameter (formal-out) unterschieden. Äquivalent zum entry/formal-Konstrukt werden Funktionsaufrufe durch call/actual-Konstrukte realisiert. Jedem actual-in-Knoten wird der korrespondierende formal-in-Knoten durch sog. Parameterkanten zugewiesen. In anderer Richtung werden die formal-out-Knoten mit den actual-out-Knoten verbunden. Zuletzt verbinden Aufrufkanten den call-Knoten mit dem entry-Knoten der aufgerufenen Funktion. Die Parameterübergabe wird somit als „copy-in/copy-out“ durch zusätzliche Zuweisungen modelliert.

Diese Art von Graph nennt man Interprozeduralen Programmabhängigkeitsgraphen (engl. interprocedural program dependence graph, IPDG). Das besondere Merkmal ist das Sharing von aufgerufenen Funktionen, so dass jede Funktion nur genau einmal im IPDG existiert. Ein Inlining findet also nicht statt.

Eine durchgeführte Erweiterung des IPDG ist das Hinzufügen von sog. „zusammenfassende Abhängigkeitskanten“ (engl. summary edges), die in einem call/actual-Konstrukt die transitiven Abhängigkeiten zwischen den formalen Eingangs- und formalen Ausgangsparametern beschreiben. Wenn eine Abhängigkeit vorliegt, wird sie als neue Kante zwischen den entsprechenden aktuellen Parametern in den Graphen eingetragen. Ein Graph mit Summarykanten nennt man System Dependence Graph (SDG).

Im Folgenden wird jedoch der Begriff PDG beibehalten, auch wenn es sich

präzise um einen SDG handelt.

2.8 Beeinflussung und Program-Slicing

Die Grundlage der Berechnung von Pfadbedingungen ist das sog. *Program-Slicing*. Weiser [Wei82, Wei84] führte Program-Slicing als Debugginghilfe ein und bestimmte damit für eine gegebene Anweisung im Programm alle anderen Programmpunkte (Slice), die diese Anweisung beeinflussen können und somit zu einem möglichen Informationsfluss beitragen. Es handelt sich hier um ein typisches Goguen/Meseguer-Modell. Das Programm wird partitioniert in einen Teil, der eine gegebene Anweisung beeinflusst und einen Teil, der außerhalb des Slice liegt und damit keine Information mit der untersuchten Anweisung teilt.

Weiser [Wei84] hat in seiner Definition von Program-Slicing gefordert, dass ein Slice $S(x)$, der aus allen Programmfragmenten y besteht, die die Anweisung x beeinflussen, die identische Auswirkung auf x haben muss, wie der originale Programmtext. Später gab es Anpassungen von Slicing-Algorithmen auf Programmabhängigkeitsgraphen [HRB90, RHSR94b, RR95], die als Ergebnis eine Graph-Knotenmenge liefern.

In dieser Arbeit werden zwei Varianten von Program-Slicing unterschieden:

- *intra*prozedural: Die Abhängigkeitsanalyse beschränkt sich ausschließlich auf die einzelne analysierte Funktion.
- *inter*prozedural: Die Abhängigkeitsanalyse nutzt Übergänge an Funktionsaufrufen, um das Gesamtprogramm zu berücksichtigen.

Beim interprozeduralen Slicing ist es für präzise Ergebnisse notwendig, den Aufrufkontext von Funktionen zu berücksichtigen (kontext-sensitives Slicing), um zu vermeiden, Slices für nicht-realisierbare Pfade zu berechnen. Eine gängige Methode hierfür, die auch in VALSOFT Verwendung findet, ist durch die Erreichbarkeitsanalyse für kontextfreie Sprachen (engl. context-free language reachability, [Rep98, RHSR94a, RR95]) gegeben.

Definition 2.6 (Beeinflussung)
Eine *Beeinflussung* $I(y, x)$ (engl. influence) zwischen den Knoten y und x liegt also genau dann vor, wenn durch Daten- oder Kontrollabhängigkeiten Informationen ausgetauscht werden. Es gilt

$$I(y, x) \text{ gdw. } y \text{ beeinflusst } x.$$

Diese Aussage ist i.d.R. nicht entscheidbar, hat jedoch keine Auswirkungen auf die grundsätzliche Korrektheit von Slices, wenn das Prinzip der *konservativen Approximation* für $I(y, x)$ zugrunde liegt.

Definition 2.7 (Vorwärts-Slice)
Sei Knoten $x \in (N, \to)$ der Startknoten eines Slice (Slice-Kriterium). Dann ist

$$FS(x) = \{y \in (N, \to) \mid x \to^* y\}$$

der *Vorwärts-Slice* von x. Er besteht aus allen Knoten im Programmabhängigkeitsgraphen $(N, \to)$, die von x über realisierbare Pfade erreichbar sind.

Definition 2.8 (Rückwärts-Slice)
Sei Knoten $x \in (N, \to)$ der Startknoten eines Slice (Slice-Kriterium). Dann ist

$$BS(x) = \{y \in (N, \to) \mid y \to^* x\}$$

der *Rückwärts-Slice* von x. Er besteht aus allen Knoten im Programmabhängigkeitsgraphen $(N, \to)$, von denen x über realisierbare Pfade erreichbar ist.

Für Pfadbedingungen ist besonders der Teilgraph des Programmabhängigkeitsgraphen zwischen zwei Knoten x und y interessant, der alle Abhängigkeiten $x \leftrightarrow^* y$ enthält[2].

Definition 2.9 (Chop)
Seien $y, x \in (N, \to)$ zwei Knoten. Dann ist

$$\begin{aligned} CH(y, x) &= \{z \in (N, \to) \mid y \to^* z \to^* x\} \\ &= FS(y) \cap BS(x) \end{aligned}$$

der *Chop*. Er besteht aus allen Knoten in $(N, \to)$, die auf realisierbaren Pfaden zwischen y und x liegen.

Für Chops gelten die folgenden Eigenschaften, die alle äquivalent sind:

1. $y \in BS(x)$
2. $x \in FS(y)$
3. $CH(y, x) \neq \emptyset$

Die Präzision von Slices und Chops hat für nachfolgende Analysen eine große Bedeutung, da sich Ungenauigkeiten nicht präzise vorhersehbar in aufsetzenden Analyseverfahren ausbreiten und damit Ergebnisse stark verfälschen bzw. zu konservativ werden lassen. Ungenauigkeit kann auch zu einer höheren Zahl von Abhängigkeitskanten führen und so aufsetzende Analysen in der

[2] In dieser Arbeit induzieren Knotenmengen, die z.B. durch Program-Slicing berechnet werden, immer die zugehörenden Kanten zwischen den Knoten.

Performance beeinträchtigen. Die Unentscheidbarkeit der Abhängigkeit zwischen zwei Programmpunkten ist ein Fakt, jedoch existieren Verfahren, die Restpräzision so hoch wie möglich zu halten [Kri03].

Program-Slicing ist heute so performant, dass kommerzielle Software in Sprachen wie C, C++ und Java analysiert werden kann. Obwohl die Slicing-Techniken sehr weit fortgeschritten sind, erlauben sie nicht immer die Analyse des vollständigen Sprachumfangs, z.B. beim Einsatz von Zeigerarithmetik. Diese Sprachkonstrukte sind jedoch in sicherheitskritischer Software generell problematisch, da ihre Auswirkungen auch manuell schwer zu erfassen und zu überprüfen sind und dadurch die Wiederverwendbarkeit der Software stark eingeschränkt wird. Dies führt zu höheren Kosten, so dass i.d.R. solche Sprachkonstrukte durch Programmierstandards nicht zugelassen werden.

Kapitel 3

Vom Slicing zu Pfadbedingungen

3.1 Die Grenzen von Program-Slicing

In der Regel zeigen Program-Slices durch die Unentscheidbarkeit der präzisen Abhängigkeitsbestimmung (z.B. Zeigerarithmetik und Schleifenterminierung) mehr Abhängigkeiten, als real vorhanden sind. Dies führt zu der fundamentalen Eigenschaft:

$$I(y, x) \Longrightarrow y \in BS(x)$$

Die umgekehrte Richtung gilt aufgrund der konservativen Approximation im Allgemeinen nicht. In der Praxis wird angestrebt, $BS(x)$ so klein und präzise wie möglich zu berechnen, damit sich auch in dieser Richtung (fast) eine Implikation ergibt. Für realistische Sprachen und Programme können die Slices jedoch sehr konservativ und damit sehr groß werden. Trotz der Praxistauglichkeit und dem Einsatz der effizientesten Algorithmen ist die Grenze des Program-Slicings durch seine *geringe Präzision* gegeben. Durch konservative Approximation sind die Slices oft viel größer als erwartet. In häufigen Fällen sind 75% des Programms im Slice, so dass die Slices viel zu groß und damit unbrauchbar für präzise Aussagen über spezielle Informationsflüsse bzw. Abhängigkeiten werden [BAG00].

Dieses Erkenntnis war die ursprüngliche Motivation, mittels Pfadbedingungen das Program-Slicing entscheidend zu verbessern, indem Knoten nicht-ausgeführter Pfade dem Slice wieder entzogen werden.

Ein weiterer Nachteil von Slices ist die *geringe Aussagekraft* von Knotenmengen. Slices liefern nur eine *binäre Information*: Sie entscheiden, ob eine

Anweisung y eine andere Anweisung x beeinflussen kann, oder ob dies definitiv nicht der Fall ist. Sie ermöglichen jedoch keine Aussage über die *Stärke der Beeinflussung* und den *Umständen der Beeinflussung*.

3.2 Pfadbedingungen als informativeres Program-Slicing

Diese Arbeit wird speziell für die obigen Probleme Lösungen liefern und sie anhand von empirischen Untersuchungen realer Software bewerten. Als Grundlage der Arbeit zur Informationsflusskontrolle dient der Vorschlag, Program-Slicing und Constraint-Solving so miteinander zu kombinieren [Sne96, KS98], dass aussagekräftige Analyseergebnisse bzgl. Informationsflüsse geliefert werden.

Slicing wird daher durch folgende Verfahren ergänzt:

- *Pfadbedingungen* – Für bestimmte Pfade $y \rightarrow^* x$ im Programmabhängigkeitsgraphen werden notwendige Bedingungen $PC(y, x)$ berechnet, die erfüllt sein *müssen*, wenn ein Informationsfluss zwischen y und x möglich sein soll.
- *Constraint-Solving* – Die aussagenlogisch vereinfachten Pfadbedingungen $PC(y, x)$ werden mithilfe von Constraint-Solvern (ggf. nach Eingabevariablen) aufgelöst.

3.2.1 Eigenschaften von Pfadbedingungen

Statt binären Informationen in Form von Kontenmengen liefern Pfadbedingungen die folgenden Aussagen:

- *erfüllt* (*true*) – Zwischen y und x *kann* es einen Informationsfluss geben. Die Wahrscheinlichkeit, dass es in der Realität tatsächlich eine Beeinflussung gibt, ist i.Allg. hoch, so dass prinzipiell von einer hinreichenden Bedingung ausgegangen werden kann.
- *unerfüllt* (*false*) – Zwischen y und x findet definitiv *kein Informationsfluss* und damit keine Beeinflussung statt, selbst wenn im Programmabhängigkeitsgraphen und Slice mögliche Abhängigkeiten gegeben sind.
- *Bedingung* – Die Bedingung kann mittels Constraint-Solving evtl. zu *false* bzw. *true* mit präzisen Variablenbelegungen ausgewertet werden. Die Fallstudien werden zeigen, dass dies in den meisten Fällen nicht

> notwendig ist. Die Interpretation der Bedingungen liefert dem Betrachter detaillierte Informationen über Ereignisse und Abhängigkeiten im Programm zwischen y und x, über die der Informationsfluss realisiert wird und die weit über die binäre Information des Program-Slicing hinausgehen.

Die resultierende Bedingung ist *notwendig* (wenn nicht hinreichend) und liefert ggf. Variablenbelegungen, die mögliche *Zeugen* für eine Beeinflussung von x durch y darstellen. $PC(y, x)$ wird mit der errechneten Belegung erfüllbar und dadurch die Beeinflussung bzw. illegale Manipulation zur Laufzeit sichtbar.

Pfadbedingungen sind also deutlich präziser als Program-Slicing. Sie machen jedoch auch Program-Slices präziser, falls diese Abhängigkeiten aufzeigen, die zur Laufzeit des Programms nicht auftreten können (falsche Positive). Constraint-Solving liefert diese falschen Positive. In [Sne96] werden fundamentale Formeln und Beweise für die Definition und Vereinfachung von Pfadbedingungen beschrieben. Jedoch wurde diese Theorie noch nicht in einem System realisiert und empirisch evaluiert. Es fehlen sowohl eine Implementierung, die Erweiterung für interprozedurale Pfadbedingungen, wirklich effiziente Algorithmen für die Theorie, die Unterstützung des gesamten C-Sprachumfangs und Analysen echter Software, um die Anwendbarkeit und den Nutzen von Pfadbedingungen aufzuzeigen. Die vorgelegte Arbeit wird genau dies leisten.

3.2.2 Das Obstwaagenbeispiel

Als Einstieg dient das angegebene Steuerprogramm einer elektronischen Obstwaage (siehe Abb. 3.1). Es handelt sich um den Auszug einer Software, die von der PTB zur Verfügung gestellt wurde.

In Abschnitt 6.4 wird das Beispiel ausführlich behandelt, so dass hier nur der Unterschied zwischen Program-Slicing und Pfadbedingungen verdeutlicht werden soll. Die Zeile 26 zeigt das ausgegebene Gewicht *u_kg* an, das innerhalb der `while`-Schleife berechnet wird. In Zeile 3 befindet sich der Tastaturport *p_cd*, der sonderbarerweise im Rückwärts-Slice $BS(u_kg)$ liegt. Der Rückwärts-Slice enthält die Programmstellen der Zeilen 2-6 und 9-24, also fast das gesamte Programm.

Die binäre Information bestätigt, dass eine Beeinflussung des Gewichts durch die Tastatur möglich ist. Die Aussagekraft von Program-Slicing endet hiermit. Die Pfadbedingung $PC(p_cd, u_kg)$ hingegen liefert die genauen Ursachen, unter denen die Beeinflussung stattfinden kann. Sie berechnet sich zu:

```
1   ...
2   int p_ab[2] = {0, 1};
3   int p_cd[1] = {0};
4   char e_puf[8];
5   float u_kg;
6   float kal_kg = 1.0;
7   ...
8
9   while (TRUE) {
10    if ((p_ab[CTRL2] & 0x10) == 0) {
11      u = ((p_ab[PB] & 0x0f) << 8) + (unsigned int)p_ab[PA];
12      u_kg = (float) u * kal_kg;
13    }
14    if ((p_cd[CTRL2] & 0x01) != 0) {
15      for (idx=0;idx<7;idx++) {
16        e_puf[idx] = (char)p_cd[PA];
17        if ((p_cd[CTRL2] & 0x10) != 0) {
18          if (e_puf[idx] == '+')
19            kal_kg *= 1.01;
20          else if (e_puf[idx] == '-')
21            kal_kg *= 0.99;
22        }
23      }
24      e_puf[idx] = '\0';
25    }
26    printf("Artikel: %7.7s\n   %6.2f kg    ",e_puf,u_kg);
27  }
28  ...
```

Abbildung 3.1: Obstwaagensteuerung

$$
\begin{aligned}
PC(p_cd, u_kg) = & \\
& \quad p_ab[CTRL2] \ \& \ 0x10 \ = \ 0 \\
& \wedge \ p_cd[CTRL2] \ \& \ 0x01 \ \neq 0 \\
& \wedge \ p_cd[CTRL2] \ \& \ 0x10 \ \neq 0 \\
& \wedge \ idx \ < \ 7 \\
& \wedge \ \big(\mathbf{e_puf}[\mathbf{idx}] \ = \ '+' \ \vee \ \mathbf{e_puf}[\mathbf{idx}] \ = \ '-'\big)
\end{aligned}
$$

Genau dann, wenn sich im Eingabepuffer ein $'+'$ oder $'-'$ befindet, fließen Informationen von der Tastatur zum ausgegebenen Gewicht, sonst nicht. Ein Blick in den Quelltext offenbart mit dieser Information, dass dann sogar das Gewicht um 10% verfälscht werden kann.

Kapitel 4

Einfache Pfadbedingungen

Die fundamentalen Eigenschaften von Pfadbedingungen wurden in [Sne96] gelegt. Sie werden hier teilweise nochmals dargestellt und diskutiert. Weiterhin wird auf Themen wie Static-Single-Assignment-Form (SSA), schwache und starke Pfadbedingungen eingegangen.

4.1 Beeinflussung und Pfadbedingungen

Eine Pfadbedingung $PC(y, x)$ ist eine Bedingung über Programmvariablen v, deren Erfüllung *notwendig* für eine Beeinflussung ist. Es gilt

$$I(y, x) \Rightarrow (\exists v : PC(y, x) = \mathit{true}\ )$$

Die umgekehrte Richtung gilt mit der gleichen Begründung wie beim Program-Slicing nicht. Die Implikation fordert, dass zur Laufzeit bestimmte Programmvariablen Werte annehmen, so dass die Pfadbedingung *true* wird. Sollte die Pfadbedingung zu *false* ausgewertet werden, gibt es tatsächlich keine Möglichkeit, in einem beliebigen Programmlauf für beliebige Werte von v Abhängigkeiten zu produzieren; es liegt dann keine Beeinflussung von x durch y vor.

Im folgenden Beispiel besteht das Programmfragment aus einer Zuweisung an ein Arrayfeld in Zeile 1 und einem Zugriff auf ein Arrayfeld in Zeile 3, der nur dann stattfindet, wenn das Prädikat in Zeile 2 erfüllt ist.

```
1  a[i+3] = y;
2  if (i > 10)
3    x = a[2 * j - 42];
```

Um $I(y, x)$ anzunähern, soll mittels Pfadbedingungen bestimmt werden, ob von y in Zeile 1 ein Informationsfluss nach x in Zeile 3 stattfindet. Von

$a[i+3]$ führt eine Datenabhängigkeitskante zu $a[2*j-42]$. Es gilt daher $\{a[i+3], y\} \in BS(x)$, so dass konservativ approximiert ein Informationsfluss stattfindet. Die interessante Frage ist, ob dieser Informationsfluss auch in einem realen Programmlauf auftreten kann.

Die obige Datenabhängigkeit gilt nur dann, wenn die gelesene Zelle des Arrays $a[2*j-42]$ in Zeile 3 der geschriebenen Zelle in Zeile 1 $a[i+3]$ entspricht. Daher kann der Informationsfluss entlang der Datenabhängkeitskante durch $i+3=2*j-42$ *bedingt* werden:

$$\delta(1,3)^1 \equiv \delta(a[i+3], a[2*j-42]) \equiv (i+3=2*j-42)$$

Datenabhängigkeiten sind i.d.R. immer *unbedingt* (*true*). Sie können jedoch die Pfadbedingungen in Abhängigkeit vom jeweiligen Kontext präzisieren. In Kapitel 6 werden spezielle Anpassungen der Pfadbedingungen an Arrays und abstrakte Datenstrukturen gezeigt.

Neben sog. Datenabhängigkeitsbedingungen δ bestehen Pfadbedingungen aus Kontrollabhängigkeitsbedingungen $\gamma(z)$, die erfüllt sein müssen, damit eine Anweisung z ausgeführt werden kann. Im obigen Beispiel führt eine Kontrollabhängigkeitskante von Zeile 2 zu Zeile 3. Die Anweisung $x = a[2*j-42]$ wird also vom Kontrollprädikat $(i > 10)$ *regiert*. Dadurch ergibt sich die folgende Kontrollabhängigkeitsbedingung:

$$\gamma(3) \equiv \gamma(x = a[2*j-42]) \equiv (i > 10)$$

Für $\gamma(1)$ gilt stattdessen *true*, da der Ausdruck von keinem Prädikat regiert wird. Die Pfadbedingung $PC(y,x)$ besteht nun aus der Konjunktion der Bedingungen über den Kanten zwischen y und x:

$$\begin{aligned} PC(1,3) &\equiv \gamma(1) \wedge \delta(1,3) \wedge \gamma(3) \\ &\equiv \mathit{true} \wedge (i+3=2*j-42) \wedge (i>10) \\ &\equiv (i+3=2*j-42) \wedge (i>10) \end{aligned}$$

Die Variable x wird genau dann von y beeinflusst, wenn Variablenbelegungen die Pfadbedingung zu *true* erfüllen lassen:

$$I(y,x) \Rightarrow \exists i,j : (i+3=2*j-42) \wedge (i>10)$$

Mit $i=1$ und $j=23$ wird die Beeinflussung bestätigt, es gilt $PC(y,x) \equiv$ *true* . Der Quelltext wird nun um einen weiteren Ausdruck ergänzt, wie im folgenden Beispiel zu sehen ist.

[1] In der Einführung werden Variablen bzw. Anweisungen durch ihre Zeilennummern indiziert, später erfolgt die präzise Indizierung anhand der PDG-Knotennummern, um Mehrfachvorkommen derselben Variable innerhalb eines Ausdrucks unterscheiden zu können.

```
1 a[i+3] = y;
2 if (i > 10 && j < 5)
3   x = a[2 * j - 42];
```

Die resultierende Pfadbedingung $PC(y, x)$ ist analog:

$$PC(y, x) \equiv (i + 3 = 2 * j - 42) \wedge (i > 10) \wedge (j < 5)$$

Constraint-Solver zeigen jetzt, dass $I(y, x) \equiv \mathit{false}$ gilt und keine Beeinflussung vorliegt, da sich keine Variablenbelegungen für i und j finden lassen, die $PC(y, x)$ zu *true* auswerten. Dies zeigt, dass Pfadbedingungen Program-Slicing erheblich präziser machen können.

4.2 Static-Single-Assignment-Form (SSA)

Bedingter Kontrollfluss durch Prädikate (**if**, **while**,…) führen dazu, dass Variablen an denselben Programmpunkten unterschiedliche Werte annehmen können, die von verschiedenen Definitionsstellen hervorgerufen werden.

```
1   x = 1;
2   if (b)
3     x = 2;
4   a = x;
```

x_4 kann je nach Auswertung von b entweder 1 oder 2 sein. Diese Mehrfach-Zuweisungen werden durch die Static-Single-Assignment-Form (SSA, [CFR+91]) modelliert. Dabei werden Mehrfach-Zuweisungen durch eine einzige Zuweisung ersetzt, so dass jede Variable eine eindeutige Definitionsstelle besitzt:

```
1   x₁ = 1;
2   if (b₁)
3     x₂ = 2;
4   x₃ = φ(x₁,x₂);
5   a = x₃;
```

Bei Mehrfach-Zuweisungen an Schleifenbedingungen erfolgt die eindeutige Definition innerhalb des Schleifenausdrucks:

```
1 x = a;                      1 x₁ = a₁;
2 while (x < 7) {            2 while (x₂ = φ(x₁,x₃), x₂ < 7) {
3   x = y + x;                3   x₃ = y₁ + x₂;
4   if (x == 8)        ⟶     4   if (x₃ == 8)
5     p(x);                   5     p(x₃);
6 }                           6 }
```

Das x in Zeile 2 wird sowohl in Zeile 1, als auch in Zeile 3 definiert. Durch eine Transformation in SSA-Form wird der Wert der Variablen x in Zeile 2

in dem Sinne eindeutig, dass zu jeder Variablenverwendung nur noch genau eine Datenabhängigkeitskante führt. Für jede Variable existiert also nur noch eine einzige Definitionsstelle (Zuweisung).

ϕ-Funktionen führen Mehrdeutigkeiten zusammen, indem sie Variablendefinitionen mit neuen Indizes einführen, die für nachfolgende Variablenverwendungen gelten. Im Beispiel wird x_2 eingeführt und so die Mehrdeutigkeit aus x_1 und x_3 bereinigt. Semantisch bedeutet $x_2 = \phi(x_1, x_3)$, dass $x_2 = x_1 \vee x_2 = x_3$ gilt.

Der Vorteil der SSA-Form bei Pfadbedingungen wird deutlich, wenn Kontrollabhängigkeitsbedingungen für $p(x_3)$ betrachtet werden. Ohne SSA-Form würde die Bedingung fälschlicherweise $x < 7 \wedge x = 8$ und somit *false* lauten. Mit SSA-Form wird die Bedingung $x_2 < 7 \wedge x_3 = 8$ generiert, die z.B. mit $a = 6$ und $y = 2$ erfüllbar ist.

Der Programmabhängigkeitsgraph von VALSOFT enthält keine echte SSA-Form, da eine der Prioritäten der Graphdarstellung die Quelltextnähe sein sollte. Die generierten Pfadbedingungen nutzen stattdessen die PDG-Knotennummern zur Indizierung aller Variablen, um sie unterscheiden zu können:

```
1  x1 = a2;
2  while (x3 < 7) {
3    x4 = y5 + x6;
4    if (x7 == 8)
5      p(x8);
6  }
```

Hierdurch werden im Hinblick auf Mehrdeutigkeiten unter Verwendung der Datanabhängigkeitskanten dieselben Vorteile der SSA-Form erreicht. $p(x_8)$ ist in VALSOFT nur dann ausführbar, wenn $x_3 < 7 \wedge x_7 = 8$ gilt.

4.3 Grundlegende Formeln für Pfadbedingungen

Pfadbedingungen werden für Chops zwischen zwei Knoten y und x im Programmabhängigkeitsgraphen berechnet. Es gilt hierbei $I(y, x) \Rightarrow CH(y, x) \neq 0$. Die elementaren Bestandteile von Pfadbedingungen sind Kontrollabhängigkeitsbedingungen γ, Ausführungsbedingungen E und Datenabhängigkeitsbedingungen δ.

Definition 4.1 (Kontrollabhängigkeitsbedingung)
Sei $n \in (N, \rightarrow)$ ein Kontrollprädikat. Sei Kante $\xrightarrow{\alpha}_n$ mit $n \xrightarrow{\alpha}_n n'$ eine Kontrollabhängigkeitskante. Dann ist $\gamma(n \xrightarrow{\alpha}_n n')$ eine *Kontrollabhängigkeitsbedingung* der folgenden Form:

- $\gamma(n \xrightarrow{\alpha}_n n') \equiv \mathit{pred}(n) = \mathit{true}$, mit $n \xrightarrow{\mathit{true}}_n n'$
- $\gamma(n \xrightarrow{\alpha}_n n') \equiv \mathit{pred}(n) = \mathit{false}$, mit $n \xrightarrow{\mathit{false}}_n n'$
- $\gamma(n \xrightarrow{\alpha}_n n') \equiv \mathit{pred}(n) = \alpha$, mit $n \xrightarrow{\alpha}_n n'$

α ist z.B. der geforderte Wert eines `switch`-Ausdrucks und $\mathit{pred}(n)$ die mathematische Darstellung des Ausdrucks, den Knoten n repräsentiert.

Definition 4.2 (Datenabhängigkeitsbedingung)
Seien $n, n' \in (N, \rightarrow)$ zwei Knoten. Sei Kante $n \rightarrow_n n'$ eine Datenabhängigkeitskante. Dann ist $\delta(n \rightarrow_n n')$ eine *Datenabhängigkeitsbedingung* der folgenden Form:

- $\delta(n \rightarrow_n n') \equiv \delta_K(n \rightarrow_n n')$, mit Bedingung abhängig vom Kontext K (Funktionsaufrufe, Arrays, abstrakte Datentypen,...)
- $\delta(n \rightarrow_n n') \equiv \mathit{true}$, sonst.

4.3.1 Ausführungsbedingungen

Jede Funktion des analysierten Programms wird durch einen PDG repräsentiert, das gesamte Programm durch die Verknüpfung der PDGs miteinander. Der Gesamt-PDG enthält daher die PDGs jeder Funktion als Subgraphen.

Jeder Subgraph $f_{PDG} \in (N, \rightarrow)$ einer Funktion f hat einen eigenen Startknoten $n_s \in f_{PDG}$, von dem *jeder* Knoten in f_{PDG} (transitiv) kontrollabhängig ist. Die Kontrollabhängigkeiten zwischen dem Startknoten und einem ausgewählten Knoten $x \in f_{PDG}$ werden durch den Kontrollabhängigkeiten-Chop definiert. $(C, \rightarrow_C) \subseteq f_{PDG}$ stellt wieder den Subgraph dar, der nur aus Kontrollabhängigkeitskanten besteht.

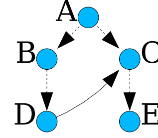
Definition 4.3 (Kontrollabhängigkeiten-Chop)
Seien f_{PDG} und zugehöriger Subgraph $(C, \rightarrow_C)$ gegeben. Der *Kontrollabhängigkeiten-Chop* zwischen dem Startknoten $n_s \in f_{PDG}$ und einem Knoten $x \in f_{PDG}$ wird definiert als

$$\mathit{CP}(x) = \{y \in C \mid n_s \rightarrow^*_C \; y \rightarrow^*_C \; x\}$$

$\mathit{CP}(x)$ enthält ausschließlich *intraprozedurale* Kontrollabhängigkeiten und kann als Subgraph von $(C, \rightarrow_C)$ aufgefasst werden. $\mathit{CP}(x)$ impliziert auch hier wieder seine Kanten.

Es gilt $\mathit{CP}(x) \subseteq \mathit{CH}(n_s, x) \cap (C, \rightarrow_C)$. Die Gleichheit gilt i.d.R nicht, wie das folgende Beispiel zeigt:

Beispiel 4.1
Die gestrichelten Linien stellen Kontrollabhängigkeitskanten dar und die durchgezogene Linie zwischen D und C eine Datenabhängigkeitskante. Es gelten $\{(D,C)\} \subseteq (N, \rightarrow)$ und $\{(A,B),(A,C),(B,D),(C,E)\} \subseteq (C, \rightarrow_C)$.



- $CH(A,E) = \{A,B,C,D,E\}$,
  $CH(A,E) \cap (C, \rightarrow_C) = \{A,B,C,D,E\}$
- $CP(E) = \{A,C,E\}$

Strukturierter Kontrollfluss liefert für $(C, \rightarrow_C)$ einen azyklischen Graphen und $CP(x)$ repräsentiert eine Menge von Pfaden. In Programmen ohne `switch/case`-Ausdrücken stellt $(C, \rightarrow_C)$ sogar einen Baum und $CP(x)$ genau einen Pfad dar. VALSOFT wurde für strukturierten Kontrollfluss entwickelt, so dass unstrukturierter Kontrollfluss, wie er z.B. durch `goto`-Anweisungen entsteht, nicht analysiert werden kann.

Definition 4.4 (Ausführungsbedingung)
Sei $CP(x)$ der Kontrollabhängigkeiten-Chop eines Zielknotens x. Es gilt $x, n_s \in CP(x)$. Dann ist

$$E(x) = \bigvee_{P_i : n_s \rightarrow^* x \in CP(x)} \Big(\bigwedge_{v \rightarrow u \in P_i} \gamma(v \rightarrow u) \Big)$$

die *Ausführungsbedingung* für Knoten x. Die Erfüllung der Bedingung ist notwendig zur Ausführung von x.

Alle Kontrollprädikate $\gamma(v \rightarrow u)$ entlang eines Pfades von n_s bis x müssen erfüllbar sein, damit x über diesen Pfad ausgeführt werden kann. Je nach Modellierung des Programmabhängigkeitsgraphen oder im Fall von unstrukturiertem Kontrollfluss können mehrere dieser Kontrollpfade existieren, so dass ggf. die Bedingungen der einzelnen Pfade disjunktiv miteinander verknüpft werden müssen.

4.3.2 Pfadbedingungen

Ein beliebiger Pfad im Programmabhängigkeitsgraphen wird nur genau dann ausgeführt, wenn entlang der Knoten des Pfades *alle Ausführungsbedingungen für jeden Knoten* erfüllt werden, d.h. alle regierenden Kontrollprädikate je Knoten zu *true* ausgewertet werden. Dies führt zur fundamentalen Definition von Pfadbedingungen:

Definition 4.5 (Pfadbedingung (1. Form))
Sei $CH(y,x)$ der Chop zwischen Knoten y und Knoten x. Seien $P_1, \ldots, P_n \in$

$CH(y, x)$ alle *aufzählbaren* Pfade zwischen y und x. Dann ist

$$PC(y,x) = \bigvee_{P_i: y \rightarrow^* x \in CH(y,x)} \Big(\bigwedge_{z \in P_i} E(z) \Big)$$

die *Pfadbedingung* zwischen y und x. Diese Bedingung ist notwendig und kann aufgrund ihrer Komponenten als stark angesehen werden.

Ein Informationsfluss kann zwischen y und x nur dann stattfinden, wenn es mindestens einen Pfad über Kontroll- und Datenabhängigkeitskanten gibt, auf dem für jeden Knoten die Ausführungsbedingung erfüllbar ist. Da i.d.R. viele Pfade existieren, werden die Bedingungen jedes Pfades disjunktiv miteinander verknüpft. Die Eigenschaften von $PC(y, x)$ wurden bereits in Abschnitt 3.2.1 dargestellt.

4.3.3 Behandlung von Zyklen auf einzelnen Pfaden

Programmabhängigkeitsgraphen enthalten bedingt durch programmiersprachliche Schleifenkonstrukte und gemeinsamer Nutzung von PDG-Repräsentationen einzelner Funktionen viele Zyklen. Es werden solche unterschieden, die Teil der Struktur des PDGs sind und solche, die entlang *einzelner* Pfade auftreten. Mit ersterem wird sich Kapitel 7 intensiv auseinandersetzen. Da durch Zyklen entlang eines Pfades unendlich lange und viele Pfade generiert werden können, wird die folgende Beobachtung in [Sne96] genutzt, um eine endliche Pfadanzahl mit endlicher Pfadlänge zu gewährleisten: Zyklen auf einzelnen Pfaden können ignoriert werden.

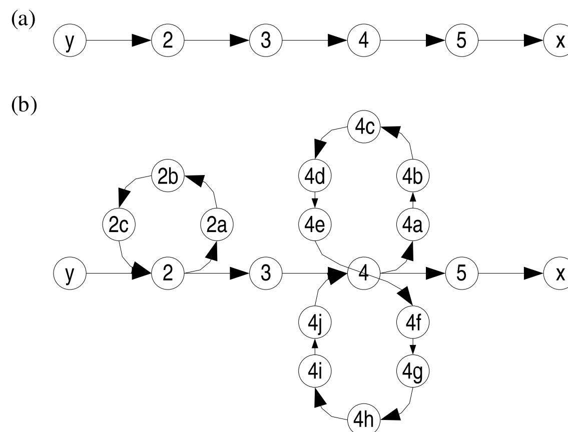


Abbildung 4.1: (a) Einzelner Pfad, (b) Einzelner Pfad mit Zyklen

Beispiel 4.2
Abbildung 4.1 zeigt zwei Pfade von y nach x. Betrachtet man zuerst Pfad

(a), dann ergibt sich nach Definition 4.5 die Pfadbedingung $E(y) \wedge E(2) \wedge E(3) \wedge E(4) \wedge E(5) \wedge E(x)$.

In Pfad (b) wird der Zyklus $2, 2a, 2b, 2c, 2$ *einmalig* durchlaufen. Der Zyklus ab Knoten 4 wird *zweimalig* mit unterschiedlichen Ausführungsbedingungen an den Knoten durchlaufen, da jedem Zyklendurchlauf ggf. ein neuer Kontext für Variablenbelegungen zugeordnet wird. Als Pfadbedingung ergibt sich $E(y) \wedge (E(2) \wedge E(2a) \wedge E(2b) \wedge E(2c) \wedge E(2)) \wedge E(3) \wedge (E(4) \wedge E(4a) \wedge \cdots \wedge E(4e) \wedge E(4) \wedge E(4f) \wedge \cdots \wedge E(4j) \wedge E(4)) \wedge E(5) \wedge E(x)$.

Pfadbedingungen werden immer über *alle* Pfade berechnet, so dass der Pfad mit der schwächsten Bedingung für einen Informationsfluss alle anderen dominiert. Das Beispiel liefert also

$$\begin{gathered} E(y) \wedge E(2) \wedge E(3) \wedge E(4) \wedge E(5) \wedge E(x) \\ \vee \\ E(y) \wedge E(2) \wedge E(2a) \wedge E(2b) \wedge E(2c) \wedge E(2) \wedge E(3) \\ \wedge E(4) \wedge E(4a) \wedge \cdots \wedge E(4e) \wedge E(4) \\ \wedge E(4f) \wedge \cdots \wedge E(4j) \wedge E(4) \wedge E(5) \wedge E(x) \\ \equiv \\ E(y) \wedge E(2) \wedge E(3) \wedge E(4) \wedge E(5) \wedge E(x) \end{gathered}$$

Die Bedingung über dem längeren Pfad liefert also *keinen Informationsgewinn* gegenüber dem kürzeren Pfad. Ausgehend von der Standarddefinition 4.5 der Pfadbedingungsberechnung kann für zyklenhaltige Pfade $y \to^* p_i \to^* p_i \to^* x$ die verbesserte Standarddefinition angegeben werden.

Definition 4.6 (Pfadbedingung (1. Form, verbessert))
Es gelte Definition 4.5. Dann ist

$$PC(y,x) = \bigvee_{\substack{P_i : y \to^* x \in CH(y,x), \\ \forall (n \to n') \in P_i : (n' \to^* n) \notin P_i}} \Big(\bigwedge_{z \in P_i} E(z) \Big)$$

eine *Pfadbedingung* mit explizitem Ausschluss von Zyklen auf einzelnen Pfaden zwischen y und x.

Eine Pfadbedingung über den Zyklus $p_i \to^* p_i$ macht die Pfadbedingung stärker, jedoch stellt p_i im *Programmabhängigkeitsgraphen* eine Verzweigung in mindestens zwei Pfade dar (den ersten Pfad bei p_i am Zyklus vorbei und den zweiten Pfad entlang des Zyklus). Hierdurch greift das Absorptionsgesetz $A \vee (A \wedge B) = A$, wenn A die Pfadbedingung für den ersten Pfad und $A \wedge B$ die Pfadbedingung für den zweiten Pfad darstellt.

Für alle folgenden Definitionen wird implizit die verbesserte 1. Form angenommen, auch wenn nicht ausdrücklich darauf hingewiesen wird.

4.3.4 Pfadbedingungen für SSA-Form

Erst durch die Verwendung der SSA-Form (bzw. SSA-ähnlicher Form) lassen sich korrekte Pfadbedingungen erzeugen, da jede Variablendefinition mit einem eindeutigen Index versehen ist und sich somit gleichlautende Variablen nicht gegenseitig überdecken. Um dies bei Pfadbedingungen zu berücksichtigen werden zusätzliche ϕ-Bedingungen bestimmt, die Definition 4.6 ergänzen.

Definition 4.7 (ϕ-Bedingung, Substitution)
Seien $x_1, x_2, \ldots, x_{n-1}$ Definitionsstellen von x als unterschiedliche SSA-Varianten und x_n eine Verwendungsstelle der Variablen x. Dann beschreibt die ϕ-Funktion $x_n = \phi(x_1, x_2, \ldots, x_{n-1})$ die folgende ϕ-Bedingung:

$$x_n = x_1 \vee x_n = x_2 \vee \cdots \vee x_n = x_{n-1}$$

Die Menge aller ϕ-Bedingungen wird als Φ bezeichnet. Für eine Datenabhängigkeitskante zwischen Definition v und Verwendung u an einem ϕ-Knoten gilt daher $\Phi(v \to u)$. Durch diese Φ-Bedingung kann nun die 2. Form einer Pfadbedingung definiert werden, bei der SSA berücksichtigt wird.

Definition 4.8 (Pfadbedingung (2. Form))
Es gelte Definition 4.6. Dann ist

$$PC(y, x) = \bigvee_{P_i : y \to^* x \in CH(y,x)} \Big(\bigwedge_{z \in P_i} E(z) \ \wedge \bigwedge_{v \to u \in P_i} \Phi(v \to u) \Big)$$

eine *Pfadbedingung* zwischen y und x, die durch SSA-Form getrennte Variablen wieder zusammenführt.

Beispiel 4.3
Als Pfadbedingung für das zweite Beispiel aus Abschnitt 4.2 zwischen a_1 und $p(x_3)$ wird nun $x_2 < 7 \wedge x_3 = 8 \wedge (x_2 = x_1 \vee x_2 = x_3)$ generiert.

In VALSOFT liegen nur Φ-ähnliche Bedingungen vor (siehe Abschnitt 4.2), so dass entlang eines Pfades keine Mehrdeutigkeiten für Variablen auftreten, da keine ϕ-Knoten vorhanden sind. Stattdessen werden in einer initialen Berechnungsphase eindeutige und mehrdeutige Variablendefinitionen anhand ankommender Datenabhängigkeitskanten erkannt. Bei eindeutigen Definitionen (in SSA-Form $x_n = \phi(x_1) \equiv x_n = x_1$) wird für die Auswertung eine *sofortige Substitution* des repräsentativen Verwendungsknotens durch den Definitionsknoten durchgeführt.

In Kapitel 5 und 6 werden Pfadbedingungen für interprozedurale Programmabhängigkeitsgraphen, Arrays und abstrakte Datentypen erweitert.

4.4 Starke und schwache Pfadbedingungen

Die Definition der Pfadbedingungen ist für die Anforderungen dieser Arbeit angemessen. Da die Algorithmen auf Graphen basieren, deren Semantik je nach Problemfeld (z.B. präzise Array-Zugriffsanalyse) oder Genauigkeit (z.B. Variationen in der Points-To-Berechnung) unterschiedlich sein kann, existieren für Pfadbedingungen unterschiedliche dem jeweiligen Kontext entsprechende *notwendige* Bedingungen.

Betrachtet man einen PDG mit $CH(y, x)$, so können für diesen Chop unterschiedliche Pfadbedingungen zwischen y und x berechnet werden, die sich anschließend in Beziehung zueinander setzen lassen.

Definition 4.9 (Starke und schwache Pfadbedingungen)
Sei $PC_1(y, x)$ die erste und $PC_2(y, x)$ die zweite Pfadbedingung in $CH(y, x)$. Es gelte $PC_1 \Rightarrow PC_2$, dann stehen PC_1 und PC_2 in folgender Relation zueinander:

- PC_1 eine *starke* Pfadbedingung (stärker als PC_2)
- PC_2 eine *schwache* Pfadbedingung (schwächer als PC_1)

Im Chop $CH(y, x)$ beschreibt die Menge der Pfadbedingungen $PC_k(y, x)$ innerhalb desselben Systems mit begrenztem $k \in K$ eine partielle Ordnung, wenn Terme durch logische Äquivalenz herausfaktorisiert werden. Da alle Variablen in Pfadbedingungen implizit existenziell quantifiziert sind, existieren keine weiteren Quantoren, so dass die Äquivalenz durch Transformation in Disjunktive Normalform (DNF) entschieden werden kann.

Die Stärke der Pfadbedingungen ist von der Präzision des zugrunde liegenden Graphen abhängig. In VALSOFT wird versucht, die Pfadbedingungen so stark wie möglich zu machen, da jedes weitere Detail die notwendige Bedingung noch präziser macht. Aufgrund des feingranularen PDGs ist die Generierung stärkerer Pfadbedingungen für ANSI-C, die einen echten Mehrwert zur Analyse sicherheitskritischer Software liefern, eher unwahrscheinlich.

In der Praxis haben die durchgeführten Fallstudien ergeben, dass viele Pfadbedingungen nicht nur notwendig, sondern sogar hinreichend sind, so dass bei erfüllbaren Pfadbedingungen tatsächlich ein Informationsfluss im realen Programmlauf stattfindet. Kleine Pfadbedingungen führen bei $y \in BS(x)$ eher zu einer echten Beeinflussung $I(y, x)$ als große Pfadbedingungen.

In der Praxis können Pfadbedingungen nicht immer automatisch zu *true* oder *false* ausgewertet, bzw. die Erfüllbarkeit aufgrund der Restriktionen derzeitiger Constraint-Solver[2] gezeigt werden. Es ist jedoch immer möglich,

[2] Pfadbedingungen können die gleiche Arithmetik enthalten, die in ANSI-C zulässig ist.

Pfadbedingungen zu ordnen und zu vergleichen.

Neben dem Einfluss der Genauigkeit von Programmabhängigkeitsgraphen kann auch der Einfluss unterschiedlich präziser Slicing-Verfahren definiert werden. Bei $CH_1(y,x) \subseteq CH_2(y,x)$ liefert $CH_1(y,x)$ eine stärkere (oder gleichstarke) Pfadbedingung als $CH_2(y,x)$.

Definition 4.10
Sei $CH_1(y,x) \subseteq CH_2(y,x)$ gegeben. Dann gilt für alle Pfade: $P_i : y \rightarrow^* x \in CH_1(y,x) \Rightarrow P_i : y \rightarrow^* x \in CH_2(y,x)$ und somit

$$\bigvee_{P_i:y\rightarrow^* x\in CH_1(y,x)} \bigwedge_{z\in P_i} E(z) \implies \bigvee_{P_i:y\rightarrow^* x\in CH_2(y,x)} \bigwedge_{z\in P_i} E(z)$$

für alle Pfade nach Definition 4.8.

4.5 Pfadbedingungen am Beispiel „mergesort“

In diesem Abschnitt wird anhand des Suchalgorithmus „mergesort“ die Berechnung von Pfadbedingungen im Detail demonstriert. Abbildung 4.2 zeigt den ANSI-C Quelltext einer „mergesort“-Implementierung. Die relevanten Teile des zugehörenden VALSOFT-Programmabhängigkeitsgraphen sind in den Abbildungen zum Startknoten 177 (4.3), zum Aufruf von `mergesort` (4.4), zur Rekursion (4.5), zum Aufruf von `merge` (4.6) und zum Zielknoten 90 (4.7) dargestellt.

Im Startknoten wird die initiale Belegung des Arrays durch die Konstante `999` festgelegt. Den Zielknoten der Pfadbedingung stellt die Zuweisung an `temp` dar.

Die Abbildungen zeigen sehr deutlich, dass feingranulare PDGs schon bei kleinen Programmen sehr groß werden können und zudem eine hohe Anzahl von Abhängigkeitskanten enthalten. Das größte gemessene Verhältnis von Knoten zu Kanten in den Fallstudien beträgt 1:50. Im Beispiel visualisiert daher ein Mini-PDG in Abb. 4.2(rechts) die für die berechnete Pfadbedingung notwendigen Abhängigkeiten in grobgranularer Form (hier von Zeile 45 bis Zeile 21).

Abgerundete längliche Boxen stehen für Anweisungen, Kreise für Bezeichner (Funktionen, Variablen, Konstanten) und Rechtecke für Prädikatsausdrücke. Ausgefüllte schwarze Kreise markieren Start- und Zielknoten, Dreiecke Parameterübergaben bei Funktionsaufrufen. Durchgezogene Pfeile stehen generell für vorwärtsgerichtete Datenabhängigkeiten, gestrichelte Pfeile für rückwärtsgerichtete Datenabhängigkeiten[3] (engl. loop carried) und fettgedruckte Pfeile für Kontrollabhängigkeiten.

[3]Diese Abhängigkeiten werden von Schleifen hervorgerufen.

```
 1 int data[100];
 2 int temp[100];
 3
 4 void move (int* fromlist, int first, int last,
 5            int* tolist, int index) {
 6     while (first <= last)
 7         tolist[index++] = fromlist[first++];
 8 }
 9
10 void merge (int first, int mid, int last) {
11     int index, index1, index2;
12
13     index = 0;
14     index1 = first;
15     index2 = mid + 1;
16
17     while ((index1 <= mid) && (index2 <= last)) {
18         if (data[index1] < data[index2])
19             temp[index++] = data[index1++];
20         else
21             temp[index++] = data[index2++];
22     }
23
24     if (index1 > mid)
25         move (data, index2, last, temp, index);
26     else
27         move (data, index1, mid, temp, index);
28
29     move(temp, 0, last-first, data, first);
30 }
31
32 void mergesort (int left, int right) {
33     int m;
34     m = (left+right) / 2;
35     if (left < right) {
36         mergesort (left, m);
37         mergesort (m + 1, right);
38         merge (left, m, right);
39     }
30 }
41
42 int main () {
43     int i;
44
45     data[0]=999;
46     data[1]=1;
47     data[2]=23;
48     data[3]=55;
49     data[4]=44;
40
51     mergesort (0, 4);
52
53     for (i=0; i < 5; ++i) {
54         printf ("%d ",data[i]);
55     }
56     printf ("\n");
57
58   return 0;
59 }
```

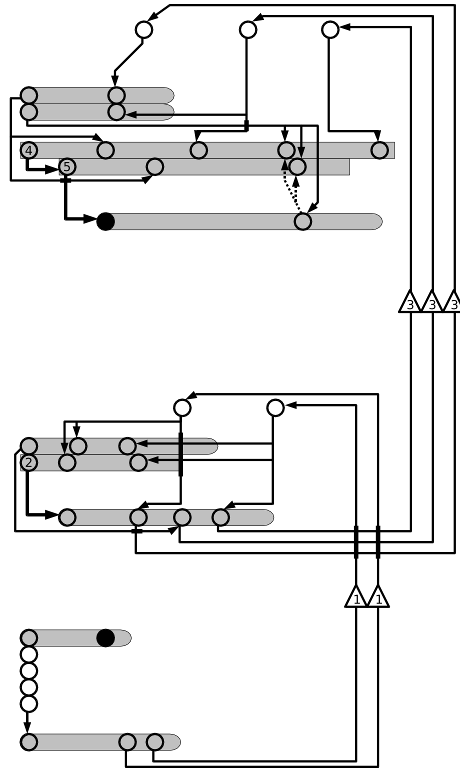


Abbildung 4.2: Der Quelltext zu „mergesort“

Einige Konzepte wie Pfadbedingungen für interprozedurale Programme, präzisere Arraybedingungen und die Behandlung von Rekursion werden erst in Kapitel 5 beschrieben. Daher werden hier Arrays als skalare Variablen betrachtet und der rekursive Aufruf innerhalb von `mergesort` ignoriert. $PC(45, 21)$ repräsentiert damit diejenigen Pfade im Chop $CH(45, 21)$, die nicht über die rekursiven Aufrufe in den Zeilen 36 und 37 führen.

Abbildung 4.2 zeigt auf den Pfaden Zahlenmarkierungen, die relevante Wegpunkte darstellen, die Bedingungen erzeugen. Beginnend bei Zeile 45 werden die Einzelbedingungen in den Zeilen 51, 35, 38, 17 und 18 erzeugt. In Zeile 21 enden die Pfade, da diese Zeile den Zielknoten enthält.

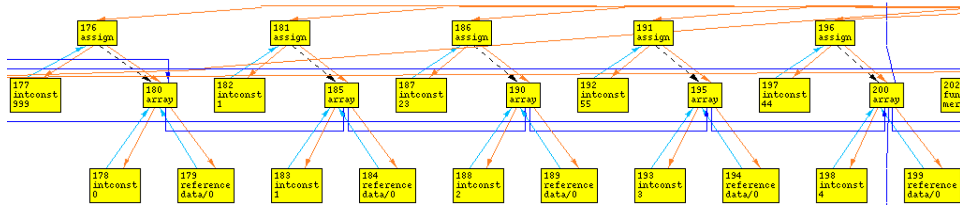


Abbildung 4.3: Der Startknoten 177 in `main`

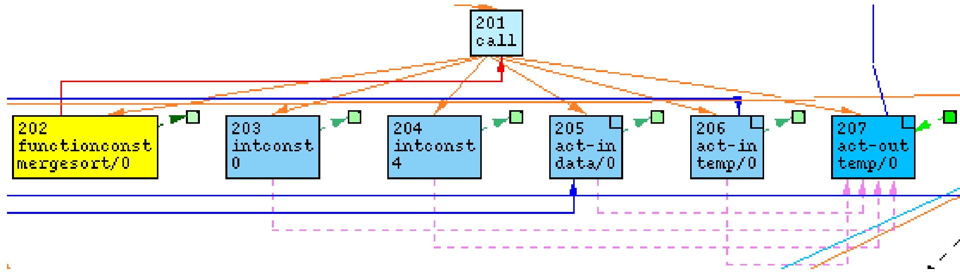


Abbildung 4.4: Der Aufruf von `mergesort` in `main`

Die Definition 4.8 wird direkt angewendet, wobei das Verfahren sowohl für alle Pfade zwischen Start- und Zielknoten gilt, als auch für eine selektive Pfadauswahl, wie in diesem Fall. Die Pfadbedingung stellt dann eine notwendige Bedingung für einen Informationsfluss in einem Teilchop dar. Generell sind Bedingungen für Teilchops stärker als für den gesamten Chop, und ihre Erfüllbarkeit (*true*) impliziert die Erfüllbarkeit des gesamten Chops. Bei *false* gilt diese Aussage nicht.

Die Pfadbedingung setzt sich aus den Ausführungsbedingungen, respektive Kontrollabhängigkeitsbedingungen γ und Φ-Bedingungen entlang der Pfade zusammen. Sie ergeben:

$$\begin{aligned}
51\!:\;& \Phi(51 \to 32) \equiv \{left_{32} = 0_{51}, right_{32} = 4_{51}\}\\
35\!:\;& \Phi(32 \to 35) \equiv \{left_{35} = left_{32}, right_{35} = right_{32}\}\\
& \gamma(35 \to 38) \equiv left_{35} < right_{35}\\
38\!:\;& \Phi(32 \to 38) \equiv \{left_{38} = left_{32}, right_{38} = right_{32}\}\\
& \Phi(34 \to 38) \equiv \{m_{38} = m_{34} = (left_{34} + right_{34})/2\},\\
& \Phi(32 \to 34) \equiv \{left_{34} = left_{32}, right_{34} = right_{32}\})\\
& \Phi(38 \to 10) \equiv \{first_{10} = left_{38}, mid_{10} = m_{38}, last_{10} = right_{38}\}\\
17\!:\;& \Phi(10 \to 17) \equiv \{mid_{17} = mid_{10}, last_{17} = last_{10}\}\\
& \gamma(17 \to 18) \equiv index1_{17} \leq mid_{17} \wedge index2_{17} \leq last_{17}\\
18\!:\;& \gamma(18 \to 21) \equiv data_{45}[index1_{18}] \geq data_{45}[index2_{18}]\\
& \equiv data_{45}[index1_{17}] \geq data_{45}[index2_{17}]
\end{aligned}$$

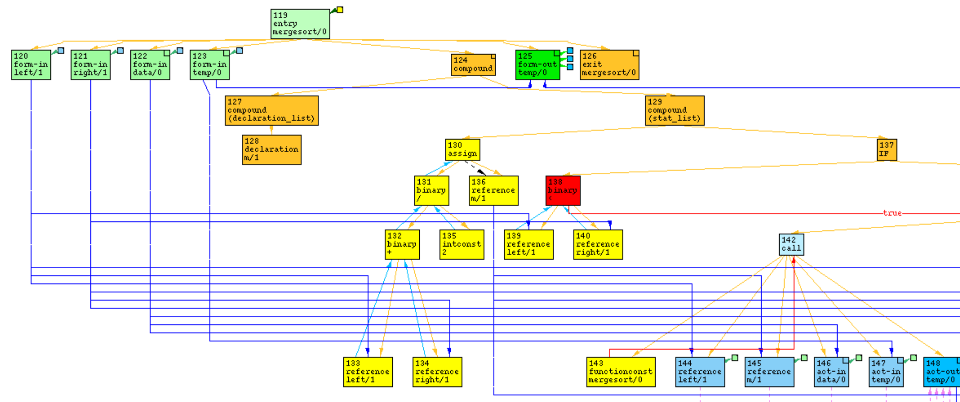


Abbildung 4.5: Die rekursiven Aufrufe in `mergesort`

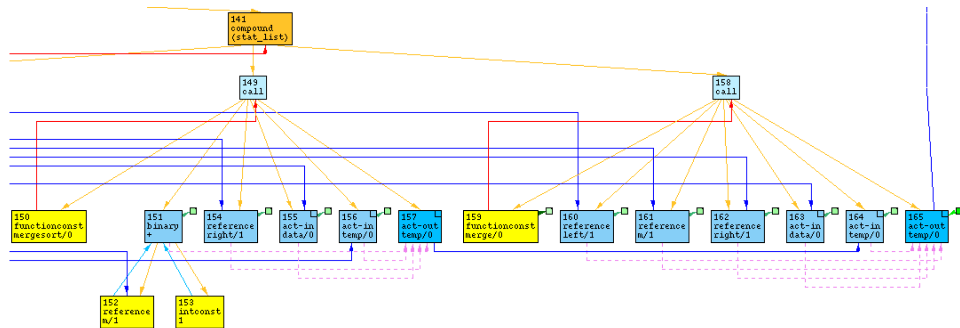


Abbildung 4.6: Der Aufruf von `merge` in `mergesort`

Φ-Bedingungen verknüpfen, durch Indizes unterschiedene, aber gleiche Variablen. Hier werden zudem auch linke Seiten von Zuweisungen automatisch durch rechte Seiten substituiert, um Pfadbedingungen aussagekräftiger zu machen. Besonders gut ist dies an Zeile 34 zu sehen, wo m durch seine rechte Seite ersetzt wird: $\Phi(34 \rightarrow 38) \equiv \{m_{38} = m_{34} = (left_{34} + right_{34})/2\}$. Dieses Verfahren ist auch für interprozedurale Parameterübergaben anwendbar, indem formale Parameter durch aktuelle Parameter ersetzt werden: $\Phi(51 \rightarrow 32) \equiv \{left_{34} = 0_{51}, right_{32} = 4_{51}\}$ und $\Phi(38 \rightarrow 10) \equiv \{first_{10} = left_{38}, mid_{10} = m_{38}, last_{10} = right_{38}\}$.

Die Funktionsweise der Φ-Bedingungen ist bei gegebener Eindeutigkeit $(x_n = \phi(x_1))$ identisch zur *Konstantenpropagation* und wird bei der Berechnung von Pfadbedingungen automatisch angewendet. Die Präzision der Pfadbedingungen wird dadurch erhöht, dass die Propagation der Variablen rückwärts in Richtung der möglichen Eingabevariablen erfolgt.

Mittels der berechneten γ- und Φ-Bedingungen lassen sich nun die Ausführungsbedingungen an den Wegpunkten (Zeilennummern) bestimmen:

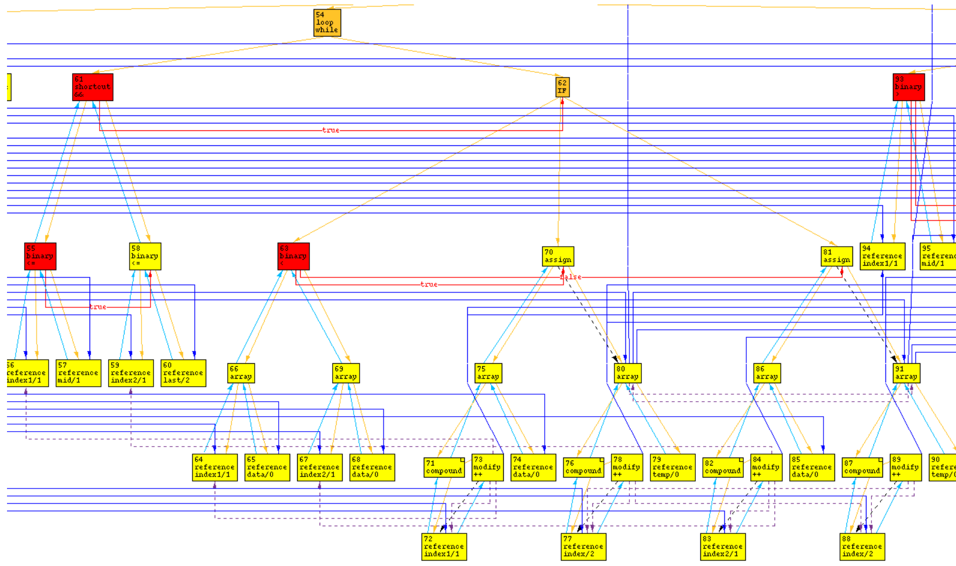


Abbildung 4.7: Der Zielknoten 90 in `merge`

45: $E(45) \equiv true$

51: $E(51) \equiv true$

34: $E(34) \equiv true$

35: $E(35) \equiv true$

38: $E(38) \equiv left_{35} < right_{35} \equiv left_{32} < right_{32} \equiv 0_{51} < 4_{51} \equiv true$

14: $E(14) \equiv true$

15: $E(15) \equiv true$

17: $E(17) \equiv true$

18: $$\begin{aligned} E(18) &\equiv index1_{17} \leq mid_{17} \wedge index2_{17} \leq last_{17} \\ &\equiv index1_{17} \leq mid_{10} \wedge index2_{17} \leq last_{10} \\ &\equiv index1_{17} \leq m_{38} \wedge index2_{17} \leq right_{38} \equiv \ldots \\ &\equiv index1_{17} \leq (left_{34} + right_{34})/2 \wedge index2_{17} \leq right_{32} \\ &\equiv index1_{17} \leq (0_{51} + 4_{51})/2 \wedge index2_{17} \leq 4_{51} \\ &\equiv index1_{17} \leq 2 \wedge index2_{17} \leq 4 \end{aligned}$$

21: $$\begin{aligned} E(21) \equiv\ & index1_{17} \leq 2 \wedge index2_{17} \leq 4 \wedge \\ & data_{45}[index1_{17}] \geq data_{45}[index2_{17}] \end{aligned}$$

Damit berechnet sich die Pfadbedingung zwischen Zeile 45 (Knoten 177) und Zeile 21 (Knoten 90) für alle ausgewählten Pfade disjunktiv verknüpft

zu:

$$\begin{aligned} PC(45,21) \quad \equiv \quad & \mathit{index1}_{17} \leq 2 \wedge \mathit{index2}_{17} \leq 4 \\ & \wedge\ \mathit{data}_{45}[\mathit{index1}_{17}] \geq \mathit{data}_{45}[\mathit{index2}_{17}] \end{aligned}$$

Die Pfadbedingung entspricht damit sogar $E(21)$. Auf die Darstellung der konjunktiven Verknüpfung der Ausführungsbedingungen entlang aller Pfade wird verzichtet, da selbst in diesem Beispiel die Anzahl der möglichen Pfade mit mehr als 50 nur schwer überschaubar ist.

Alle Variablen sind implizit existenziell quantifiziert, und Zugriffe auf Arrayzellen werden als skalare Variablen behandelt. Es lassen sich Variablenbelegungen finden, so dass diese Pfadbedingung *true* und damit ein tatsächlicher Informationsfluss zwischen den Zeilen 45 und 21 sehr wahrscheinlich wird.

Kapitel 5

Komplexe Pfadbedingungen

Bisher wurden Pfadbedingungen nur für *intra*prozedurale Berechnungen gezeigt. Das „mergesort“-Beispiel 4.5 hat demonstriert, wie der *inter*prozedurale Fall durch die Bindung von aktuellen an formale Parameter realisiert wird. Allgemein gültig ist dieses Beispiel jedoch nicht. Daher befasst sich dieses Kapitel mit allen Erweiterungen für eine effiziente interprozedurale Analyse.

5.1 Slicing und „realisierbare Knoten“

Interprozedurale Pfadbedingungen werden innerhalb von interprozeduralen Program-Slices berechnet. Der naive Ansatz beim Program-Slicing ist eine Erreichbarkeitsanalyse, die alle erreichbaren Knoten funktionsübergreifend in den Slice aufnimmt, z.B. durch Tiefensuche. Der Nachteil dieses Ansatzes ist die Ungenauigkeit des Slices, da er die Aufrufkontexte von Funktionen nicht beachtet und so eine Funktion f am Kontext f_A betritt und sie am Kontext f_B verlässt, da es nur eine gemeinsame PDG-Repräsentation von f gibt. Präzise wäre in diesem Fall ausschließlich die Rückkehr in den aufrufenden Kontext f_A.

Pfade, die gegebene Kontexte nicht respektieren, sind nicht realisierbar (engl. *unrealizable path*). Zur Laufzeit des Programms kann es keine Ausführung entlang dieser Pfade geben.

Ein interprozeduraler Slice ist genau dann präzise, wenn alle Knoten des Slice vom Slice-Kriterium erreichbar sind (siehe Abschnitt 2.8). In ähnlicher Weise kommen für interprozedurale Pfadbedingungen nur realisierbare Pfade in Frage. Program-Slices – selbst für präzise Slices unter Berücksichtigung aller Aufrufkontexte – sind für eine naive Pfadaufzählung zur Pfadbedingungsberechnung nicht geeignet, da sie aus Knotenmengen bestehen und

somit nur *realisierbare Knoten* jedoch keine realisierbaren Pfade enthalten. Kontexte müssen für Pfadbedingungen daher explizit beachtet werden.

In [SRK03] wurden Pfadbedingungen unterschieden, je nachdem, ob Start- und Endknoten innerhalb derselben Funktion oder in unterschiedlichen Funktionen liegen. In dieser Arbeit ist diese Unterscheidung nicht notwendig. Es wird immer vom allgemeinen Fall ausgegangen, dass Start- und Endknoten beliebig gewählt sind.

5.2 Interprozedurale Pfadbedingungen

Im Gegensatz zum Program-Slicing werden für Pfadbedingungen auch semantische Informationen innerhalb der Knoten ausgewertet. Bei Funktionsaufrufen ist daher nicht nur die Bewahrung des korrekten Kontextes relevant, sondern zugleich die aktuellen Parameterwerte des Aufrufs, mit denen die Funktion rechnet. Hierzu wird die sog. Parameterbindung von aktuellen an formale Parameter in Programmabhängigkeitsgraphen definiert.

5.2.1 Parameterbindung an Aufrufkontexten

Definition 5.1 (Parameterbindung)
Sei f_{act} ein Aufrufkontext der Funktion f, $PARM(f_{act})$ die Menge der aktuellen Eingangs- und Ausgangsparameter, sowie $PARM(f_{form})$ die Menge der formalen Eingangs- und Ausgangsparameter von f. Dann ist $|PARM(f_{act})| = |PARM(f_{form})|$ und

$$\beta(f_{act}) = \{a_i \mapsto a'_i \mid \forall a_i \in PARM(f_{form}), a'_i \in PARM(f_{act})\}$$

für $1 \leq i \leq |PARM(f_{form})|$ eine *Parameterbindung* der aktuellen Parameter eines Kontextes an die formalen Parameter.

5.2.2 Funktionsaufrufe als Zerlegungspunkte

Die Pfadbedingungsberechnung basiert prinzipiell auf einer Pfadaufzählung innerhalb eines berechneten Chops und ist damit exponentiell in der Berechnungskomplexität. Daher ist es entscheidend, die Länge der aufzuzählenden Pfade soweit wie möglich zu begrenzen, ohne jedoch Pfade auszulassen oder Präzision zu verlieren. Der Programmabhängigkeitsgraph muss daher virtuell an denjenigen Punkten zerlegt werden, die es zulassen, Pfadbedingungen für Folgepfade *wieder zu verwenden* (engl. caching).

Diese Zerlegungspunkte liegen an Funktionsaufrufen. Abbildung 5.1 zeigt eine Funktionsrealisierung f mit den beiden Aufrufkontexten f_A und f_B.

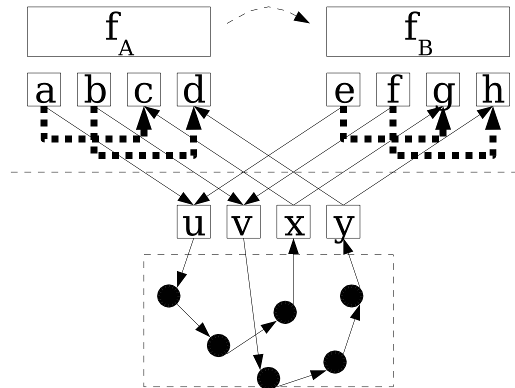


Abbildung 5.1: Symbolisierter Funktionsaufruf mit Summarykanten

Die aktuellen Eingangsparameter a, b, e und f sind mit den formalen Eingangsparametern u und v verbunden, die aktuellen Ausgangsparameter c, d, g und h mit den formalen Ausgangsparametern x und y. Sowohl zwischen u und x als auch zwischen v und y existieren transitive Abhängigkeiten, die durch Summarykanten an den Aufrufstellen (fett gestrichelte Linien) gekennzeichnet sind. An folgenden Kanten bieten sich Zerlegungen an:

- Parametereingangskanten – An ihnen wird in Funktionen abgestiegen. Um den Kontext des Aufrufs zu bewahren, darf keine Parameterausgangskante verfolgt werden, die in einen anderen Kontext führt.
- Summarykanten – Sie fassen die transitiven Abhängigkeiten am Kontext zwischen einem formalen Eingangs- und Ausgangsparameter zusammen.
- Parameterausgangskanten – An ihnen wird in aufrufende Kontexte aufgestiegen.

Im Folgenden werden die Wiederverwendungsmöglichkeiten von Pfadbedingungen für Folgepfade genauer analysiert. Hierfür sind die Definitionen des Tiefen-Chops und der Tiefen-Pfadbedingung notwendig. Beide sind auf eine Analyse ausgelegt, die horizontal den Graphen traversieren kann, vertikal jedoch nur Abstiege in Funktionen durchführt, niemals aber Aufstiege.

Definition 5.2 (Tiefen-Chop)
Sei $CH(y,x) \subseteq (N, \rightarrow)$ ein Chop zwischen Knoten y und Knoten x. Dann ist

$$\underline{CH}(y,x) = CH(y,x) \setminus \{v \in f_{form} \rightarrow u \in f_{act}\}, \forall v,u \in CH(y,x)$$

ein *Tiefen-Chop*, der keine Kanten zwischen formalen Ausgangsparametern und aktuellen Eingangsparametern enthält.

Definition 5.3 (Tiefen-Pfadbedingung)
Sei $\underline{CH}(y,x)$ ein Tiefen-Chop zwischen Knoten y und Knoten x. Seien $P_1,\ldots,P_n \in CH(y,x)$ alle zyklenfreien Pfade zwischen y und x. Dann ist

$$\underline{PC}(x,y) = \bigvee_{P_i : y \to^* x \in \underline{CH}(x,y)} \Big(\bigwedge_{z \in P_i} E(z) \;\wedge \bigwedge_{v \to u \in P_i} \Phi(v \to u)\Big)$$

eine *Tiefen-Pfadbedingung* zwischen y und x, die gemäß Definition 4.8 berechnet wird.

5.2.3 Summarybedingungen

Summarykanten $a_{act} \to b_{act}$ sind redundante Abhängigkeitskanten, da sie einen bestehenden realisierbaren Pfad zwischen $a'_{form} \to b'_{form}$ nochmals „zusammengefasst“ darstellen. Sie können ignoriert werden, jedoch lassen sich dann keine Wiederverwendungspotentiale nutzen. Sie dürfen jedoch nicht wie gewöhnliche Datenabhängigkeitskanten behandelt werden.

Beispiel 5.1
In Abb. 5.1 soll die Pfadbedingung zwischen a und c berechnet werden. Gewöhnlich wird für Bedingungen über Datenabhängigkeitskanten *true* angenommen, so dass sich ergibt: $PC(a,c) \equiv (E(a) \wedge E(u) \wedge \cdots \wedge E(x) \wedge E(c)) \vee (E(a) \wedge E(c)) \equiv E(a) \wedge E(c)$. Durch Absorption würden Bedingungen in aufgerufenen Funktionen neutralisiert; die Pfadbedingung würde daher viel zu konservativ werden.

Um dies zu vermeiden, werden Summarykanten mit Bedingungen belegt.

Definition 5.4 (Summarybedingung)
Sei $a_{act} \to b_{act}$ eine Summarykante des Aufrufkontextes f_{act}. Sei a_{act} ein aktueller Eingangs- und b_{act} ein aktueller Ausgangsparameter, a'_{form} der entsprechende formale Eingangs- und b'_{form} der formale Ausgangsparameter einer Funktion f. Dann ist

$$\delta(a_{act} \to b_{act}) = \underline{PC}(a'_{form}, b'_{form}) \lhd \beta(f_{act})$$

eine *Summarybedingung* über der Kante $a_{form} \to b_{form}$, deren formale Parameter in $\underline{PC}(a'_{form}, b'_{form})$ dem Kontext $\beta(f_{act})$ zugeordnet sind.

Die Tiefen-Pfadbedingung in Summarybedingungen ist *wieder verwendbar*, da für ihre Berechnung keine Historie an Pfaden verwendet wird. Für Abb. 5.1 gilt z.B. $\delta(a \to c) = \underline{PC}(u,x) \lhd \beta(f_A)$ und $\delta(e \to g) = \underline{PC}(u,x) \lhd \beta(f_B)$.

Für die Berechnung von intraprozeduralen Pfadbedingungen, die ihren Start- und Endknoten innerhalb derselben Funktion haben, kann $\delta(a_{act} \to$

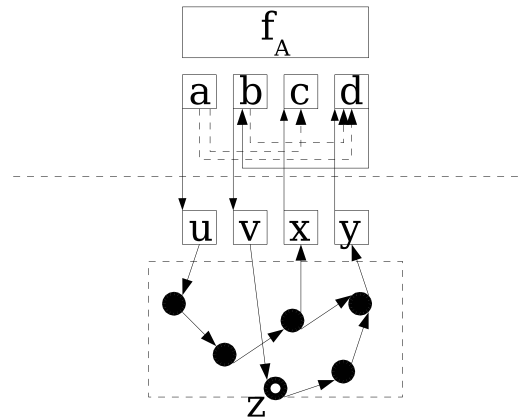


Abbildung 5.2: Zyklische Abhängigkeiten zwischen Parametern

$b_{act}) = true$ gelten, wenn auschließlich Bedingungen innerhalb dieser Funktion gewünscht sind. In diesem Fall ist der Abstieg in aufgerufene Funktionen redundant, so dass generell alle Parametereingangskanten ignoriert werden können.

5.2.4 Parametereingangskanten und Summarybedingungen

Summarybedingungen machen Parametereingangskanten nicht redundant, daher muss die Frage geklärt werden, in welchen Fällen und in welcher Art Parametereingangskanten traversiert werden müssen, wenn Summarykanten am gleichen Knoten vorliegen. Die Abb. 5.1 liefert auch hierfür die Lösung: Parametereingangskanten müssen genau dann traversiert werden,

1. wenn der Zielknoten durch einen Abstieg in die aufgerufene Funktion erreichbar wird und
2. diese Erreichbarkeit nicht durch eine Rückkehr in den Aufrufkontext (Aufstieg) erfolgt[1].

Begründung: 1. ist offensichtlich, da Erreichbarkeit für Abhängigkeiten steht. 2. bedeutet, dass Parameterkanten der Form $v_{form} \rightarrow u_{act}$ generell ignoriert werden können. Abb. 5.2 visualisiert diese Forderung und zeigt einen Funktionsaufruf mit zyklischen Abhängigkeiten, die durch Schleifenkonstrukte realisiert werden. Die Kante $d \rightarrow b$ stellt hierbei eine echte (rückwärtsgerichtete) Datenabhängigkeitskante dar. Die zu berechnende Bedingung $PC(a, z)$ ergibt sich für einen realisierbaren Pfad zu $E(a) \wedge E(u) \wedge \cdots \wedge E(y) \wedge E(d) \wedge E(b) \wedge E(v) \wedge E(z)$. Nach 2. ist z nicht ohne

[1] Aufstiege in andere als den Aufrufkontext liefern generell nicht realisierbare Pfade.

Rückkehr in den Aufrufkontext von a erreichbar. Daher wird die Parametereingangskante $a \to u$ ignoriert. Stattdessen wird berechnet: $E(a) \wedge \delta(a \to d) \wedge E(d) \wedge E(b) \wedge E(v) \wedge E(z)$. $\delta(a \to d)$ liefert $E(u) \wedge \cdots \wedge E(y)$, so dass sie Gesamtbedingung $E(a) \wedge E(u) \wedge \cdots \wedge E(y) \wedge E(d) \wedge E(b) \wedge E(v) \wedge E(z)$ wird und der Originalbedingung entspricht.

Abhängigkeiten gehen nicht verloren, da sie über Summarykanten jederzeit rekonstruiert werden können. Der Vorteil liegt an der möglichen Wiederverwendung von Teilpfadbedingungen (im Beispiel $\underline{PC}(u, y)$).

5.2.5 Summarybedingungen im Falle von Rekursion

Bei selbstrekursiven oder transitiv-rekursiven Aufrufen kann Definition 5.4 zu einem unendlichen Auffalten der Pfadbedingung führen. Als Lösung wird die Rekursionstiefe beschränkt und damit eine legitime Verringerung der Präzision von Summarybedingungen $\delta(a_{act} \to b_{act})$ in Kauf genommen. Im Allgemeinen kann immer $\delta(a_{act} \to b_{act}) = true$ gesetzt werden, um konservativ zu approximieren. Bei Bedarf an erhöhter Genauigkeit kann das Auffalten nach einer endlichen Anzahl von Rekursionsschritten durch $\delta(a \to b) = true$ abgeschlossen werden.

5.2.6 Abstiegsbedingungen

Wenn $a_{act} \to a'_{form}$ eine Parametereingangskante und Teil eines realisierbaren Pfades ist, kann äquivalent zu Summarybedingungen eine Abstiegsbedingung definiert werden, die auf Tiefen-Pfadbedingungen basiert. Die Traversierung von Parametereingangskanten ist dabei an die zwei Forderungen in Abschnitt 5.2.4 gebunden.

Definition 5.5 (Abstiegsbedingung)
Es gelte Definition 4.8. Sei $a_{act} \to a'_{form}$ eine Parametereingangskante von f_{act}. Sei $S = \{a_{act} \to b_{act}\}$ die Menge der Summarykanten an a_{act}. Sei $\underline{CH}(y, a_{act}) \neq \emptyset$. Dann ist

$$\begin{aligned} PC(y,x) \equiv & PC(y, a_{act}) \wedge \\ & \Big(\big(\bigvee_{a_{act} \to b^i_{act} \in S} \delta(a_{act} \to b^i_{act}) \wedge PC(b^i_{act}, x) \big) \\ & \vee \underline{PC}(a'_{form}, x) \lhd \beta(f_{act}) \Big) \end{aligned}$$

eine *Abstiegsbedingung* bei existierenden Summarykanten.

Für den Fall, dass keine Summarykanten im PDG existieren, reduziert sich die Formel zu

$$PC(y,x) \equiv PC(y, a_{act}) \wedge \underline{PC}(a'_{form}, x) \lhd \beta(f_{act})$$

Wie bei Summarykanten kann auch hier $\underline{PC}(a'_{form}, x)$ wieder verwendet werden, da diese Bedingung keine Historie an Pfaden betrachtet. Bei jeder Verwendung wird $\underline{PC}(a'_{form}, x)$ mit dem jeweils aktuellen Kontext β instantiiert.

Im Fall von Rekursion wird das gleiche Vorgehen wie bei Summarykanten durchgeführt (siehe Abschnitt 5.2.5) und $\underline{PC}(a'_{form}, x) \equiv E(x)$ angekommen. $\underline{PC}(a'_{form}, x) \equiv \mathit{true}$ wäre zu konservativ, da x im Falle von realisierbaren Pfaden ausgeführt werden muss.

5.2.7 Aufstiegsbedingungen

Ein Informationsfluss, der eine ggf. nicht explizit entlang von Pfaden aufgerufene Funktion verlässt, ist bei der Pfadbedingungsberechnung ein Sonderfall. Start- und Zielknoten liegen i.d.R. horizontal, also innerhalb derselben Funktion[2] oder top-down, so dass $\underline{CH}(y,x) \neq \emptyset$ gilt.

Ein Informationsfluss in bottom-up Richtung bedeutet, dass aus einer Funktion f heraus *alle* Aufrufstellen dieser Funktion f^K_{act} traversiert werden und es keinen eindeutigen Aufrufkontext der Funktion gibt. Für die Analyse sicherheitskritischer Software ist dieses Vorgehen fragwürdig, da es nicht dem natürlichen Verständnis von Abhängigkeiten und dem Fluss von Informationen entlang von Funktionsaufrufen entspricht.

Da eine Funktion f eines ausgeführten Startknotens $y \in f$ selbst auch ausgeführt werden muss, ist es sinnvoller, den Startknoten in einen explizit gewählten Aufrufkontext (z.B. $y \in f^1_{act}$) unter Berücksichtigung interessierender Abhängigkeiten zu legen.

Definition 5.6 (Aufstiegsbedingung)
Sei $U = \{b'_{form} \rightarrow b_{act}\}$ die Menge der Parameterausgangskanten an b'_{form}
Sei $\underline{CH}(y,x) = \emptyset$ mit $y \neq x$. Dann ist

$$PC(y,x) \equiv \Big(\bigvee_{b'_{form} \rightarrow b^i_{act} \in U} PC(y, b'_{form}) \lhd \beta(f_{act}(b^i_{act}))\Big) \wedge PC(b^i_{act}, x)$$

eine *Aufstiegsbedingung*. $f_{act}(b^i_{act})$ stellt dabei den Kontext derjenigen Funktion dar, die b^i_{act} enthält.

[2] Durch Summarybedingungen werden die Pfadbedingungen interprozedural berechnet.

Wenn $\underline{CH}(y, x) \neq \emptyset$ gilt, werden Aufstiegsfunktionen nie aufgerufen, da nach Abschnitt 5.2.4 alle enthaltenen Abhängigkeiten mit entsprechenden Pfadbedingungen über Summarykanten und Parametereingangskanten realisiert werden.

5.2.8 Behandlung von Funktionsaufrufkanten

Programmabhängigkeitsgraphen enthalten neben Parameterkanten auch Funktionsaufrufkanten, die aktuelle Funktionsaufrufknoten (engl. call) mit formalen Funktionseintrittsknoten (engl. entry) verbinden. Da Funktionseintrittsknoten den „Wurzelknoten" einer Funktion darstellen, sind alle untergeordneten Knoten unbedingt (*true*) kontrollabhängig und somit erreichbar.

Funktionsaufrufkanten werden während der Pfadbedingungsberechnung genau dann traversiert, wenn die Funktionsaufrufknoten (aktuelle Kontexte der Funktionen) über Kontrollabhängigkeitskanten erreicht werden. Aktuelle Parameterknoten werden hingegen ausschließlich über Datenabhängigkeitskanten erreicht.

Pfade, die über Funktionsaufrufkanten in Funktionen absteigen, erzeugen viele redundante Pfade, da Funktionseintrittsknoten strukturelle Vorgänger aller formalen Parameterknoten sind. Dies führt dazu, dass die intraprozedurale Pfadbedingung (nur innerhalb der betretenen Funktion) ausschließlich aus Kontrollabhängigkeitsbedingungen besteht. Datenabhängigkeiten erzeugen durch die Struktur des Graphen nur noch redundante Bedingungen.

Trotz dieser Eigenschaft ist die Traversierung von Funktionsaufrufkanten notwendig. Es können zum einen parameterlose Funktionen im PDG erzeugt werden[3], zum anderen können Zielknoten nicht über Pfade erreichbar sein, die von Eingangsparametern ausgehen.

```
1  int g() {
2    return 10;
3  }
4
5  int f(int a) {
6    int b;
7    b=g();
8    if (a>0)
9      return a*b;
10 }
11 ...
12 y=f(5);
```

[3] Parameterlose Funktionen im PDG greifen nicht auf globale Variablen. Dasselbe gilt auch für von ihnen aufgerufene Funktionen.

Das obige Beispielprogramm verdeutlicht den Nutzen von Funktionsaufrufkanten. Es soll die Pfadbedingung von f_{12} nach 10_2 berechnet werden. In Funktion f ist der Aufruf von g (Zeile 7) nicht vom Parameter a abhängig. Funktion g enthält keine Parameter und liefert immer 10 zurück. Dadurch wird die Pfadbedingung ausschließlich über Kontrollabhängigkeitskanten sowohl in f als auch in g berechnet. Das Ergebnis ist in Originalform

```
1      True  <no formal parameters> (g, call_node_test.c)
2      a:i@10 := 5:i@38; (f, call_node_test.c)
```

Die Pfadbedingung liefert *true*, zusätzlich enthält sie die Information, dass eine Parameterbindung von a in Zeile 5 (Knoten 10, Typ *int*) durch 5 in Zeile 12 durchgeführt wurde.

5.2.9 Behandlung von Funktionsrückgabewerten

Für jeden Kontext f_{act} von Funktionsaufrufen werden durch die Parameterbindung $\beta(f_{act})$ den formalen Parametern die aktuellen Parameter zugeordnet, um für Pfadbedingungen innerhalb von f_{form} kontext-sensitive Bedingungen aufstellen zu können.

Pfadbedingungen lassen sich in einigen Situationen an Aufrufkontexten noch präziser machen, wenn neben der Parameterbindung auch die Rückgabewerte von Funktionsaufrufen berücksichtigt werden.

```
1  int check(int x, int y) {
2    if (x==0)
3      return 0;
4    ...
5    return result;
6  }
7  int f() {
8    ...
9    while (check(a,b) > 0) {
10     ...
```

Der obige Programmauszug generiert innerhalb der Funktion `f` über das `while`-Prädikat die Bedingung $check(a, b)_9 > 0$, ggf. erweitert um Summarybedingungen. Sowohl für die manuelle Inspektion der Pfadbedingung als auch für die Weiterverarbeitung mittels Constraint-Solver ist das Prädikat in dieser Form nicht auswertbar.

Eine Nutzung der `return`-Anweisungen, die im PDG mit einem einzigen formalen Parameterausgangsknoten b'_{exit} gekoppelt sind, löst dieses Problem und kann evtl. sogar konkrete Werte liefern, die den Funktionsaufruf im Schleifenprädikat ersetzen.

Definition 5.7 (Bedingung der Funktionsrückgabewerte)
Sei f_{act} ein aktueller Kontext der Funktion f_{form}. Sei RET_f die Menge der Rücksprungknoten/-werte, und sei $ref \in f_{act}$ der Referenzknoten von f_{act}, aus dem sich das Prädikat bildet. Dann ist mit $ret_i \in RET_f, i = 1 \ldots n$

$$\begin{aligned} ref &= \phi(ret_1, ret_2, \ldots, ret_n) \\ &= \bigvee_i \big(E(ret_i) \rhd \{ref \mapsto ret_i\}\big) \end{aligned}$$

eine *Bedingung der Funktionsrückgabewerte* für einen Funktionsaufruf f_{act} mit neu erzeugter Bindung des Rückgabewertes an den Referenzknoten (analog Abschnitt 5.2.1).

Eine Bindung des Rückgabewertes an den Referenzknoten ist nur dann möglich, wenn die Ausführungsbedingung E des Referenzknotens erfüllbar wird. Angewendet auf obiges Beispiel erhält man statt der nicht weiter auswertbaren Bedingung $check(a,b)_9 > 0$ die präzisere Bedingung:

$$\begin{aligned} &\big(E(0) \rhd \{check(a,b)_9 \mapsto 0\} \vee E(result_5) \rhd \{check(a,b)_9 \mapsto result_5\}\big) \\ &\wedge check(a,b)_9 > 0 \\ = &\big(x = 0 \rhd \{check(a,b)_9 \mapsto 0\} \vee true \ \rhd \{check(a,b)_9 \mapsto result_5\}\big) \\ &\wedge check(a,b)_9 > 0 \\ = &\big(x = 0 \rhd \{check(a,b)_9 \mapsto 0\} \wedge check(a,b)_9 > 0 \ \vee \\ &true \ \rhd \{check(a,b)_9 \mapsto result_5\} \wedge check(a,b)_9 > 0\big) \\ = &\big(x = 0 \wedge 0 > 0 \vee result_5 > 0\big) \\ = &\ result_5 > 0 \end{aligned}$$

Durch Anwendung der obigen Definition wird erkannt, dass Pfade entlang der Anweisung `return 0` das Schleifenprädikat `check(a,b) > 0` nicht erfüllbar werden lassen und somit zur Laufzeit nur ein Informationsfluss über die Anweisung `return result` stattfinden kann.

5.3 Schleifenkontexte und widersprüchliche Pfadbedingungen

Die konjunktive Verknüpfung von Knoten- und Kantenbedingungen entlang von Pfaden erzeugt Pfadbedingungen. Jeder Pfadknoten steht für einen bestimmten Zustand des Programms, der den zeitlichen Ablauf zeigt. Entlang eines Pfades kann es also keine zwei Knoten desselben Zustands geben, da dies ein Zyklus wäre und Zyklen entlang von Pfaden ignoriert werden können.

Jedoch können zwei unterschiedliche Knoten eines Pfades widersprüchliche Bedingungen hervorrufen, wenn sich die beiden Knoten innerhalb von Schleifen (`while, for, ...`) befinden und zu unterschiedlichen Schleifeniterationen und damit zu unterschiedlichen Variablenbelegungen gehören.

Abbildung. 5.3 stellt einen Quelltext mit drei Funktionen dar, die jeweils 2^a berechnen. `pow2` verwendet eine typische Schleife in Zeile 3, `rpow2` ist eine rekursive Realisierung und `nrpow2` eine Realisierung, in der die Schleife in die `main`-Funktion ausgelagert ist. Diese drei Beispiele geben alle Fälle an, in denen durch rückwärtsgerichtete Datenabhängigkeiten (engl. loop-carried) widersprüchliche Bedingungen auftreten können.

Die Beispiele verwenden wieder die grobgranulare Zeilennummer zur Unterscheidung der PDG-Knoten, da diese den direkten Bezug zum Quelltext herstellt.

pow2: Für die Berechnung der Pfadbedingung von a in Zeile 1 zu x in Zeile 8 (symbolisiert als $PC(1,8)$) existiert ein realisierbarer Pfad $1 \rightarrow 4 \rightarrow_{true} 5 \rightarrow 8$. Knoten 5 erreicht 8 über eine Datenabhängigkeitskante von der Definition x zur Verwendung x. Die Knoten des Pfades sind alle ausführbar, und die Konjunktion der Ausführungsbedingungen $E(1) \wedge \dots E(8)$ liefert $\mathit{true} \wedge \mathit{true} \wedge a > 0 \wedge a \leq 0 = \mathit{false}$.

Dieses unerwünschte Ergebnis liegt an der Traversierung der *rückwärtsgerichteten Datenbhängigkeitskante* $x_5 \rightarrow x_8$, die durch die `for`-Schleife hervorgerufen wird. Die Abhängigkeitskante ist rückwärtsgerichtet, da beide Fälle des `if` nicht gleichzeitig ausgeführt werden können. Das Prädikat in Zeile 4 wird daher in verschiedenen Kontexten mit unterschiedlichen Variablenbelegungen für die Ausführungsbedingungen verwendet. Die Lösung ist entweder die Unterscheidung von Schleifeniterationen oder die Substitution des Prädikats innerhalb einer Iteration zu *true*.

rpow2: In dieser rekursiven Funktion wird äquivalent zu `pow2` $a \rightarrow^* x$, also $PC(13,19)$ betrachtet. Der relevante Pfad ist $13 \rightarrow 14 \rightarrow_{true} 16 \rightarrow 13 \rightarrow 14 \rightarrow_{false} 19$, wobei die beiden Funktionsaufrufe in Zeile 13 nicht als Zyklus entlang eines Pfades aufgefasst werden, da jeder Funktionsaufruf in einem anderen Aufrufkontext steht. Die Knoten des Pfades sind auch hier wieder ausführbar, und die Konjunktion der Ausführungsbedingungen liefert *false*.

Im Gegensatz zu `pow` wird die Schleife durch einen rekursiven Aufruf simuliert, der zur Pfadbedingung $PC(13,16) \wedge PC(13,19) \lhd \beta(\mathit{rpow2}_{16})$ führt. Je nach Behandlung von Aufrufkontexten und Zyklen auf einzelnen Pfaden kann die Pfadbedingung zu $PC(13,16) \wedge PC^1(13,16) \lhd \beta(\mathit{rpow2}^1_{16}) \wedge \dots \wedge PC^k(13,16) \lhd \beta(\mathit{rpow2}^k_{16}) \wedge PC(13,19) \lhd \beta(\mathit{rpow2}^{k+1}_{16})$ erweitert wer-

```
 1  int pow2 (int a) {
 2    int x=0;
 3    for (;;) {
 4      if (a>0) {
 5        x+=x+1; --a;
 6      } else {
 7        printf ("2^a= %d\n",x+1);
 8        return (x+1);
 9      }
10    }
11  }
12
13  int rpow2 (int a, int x) {
14    if (a>0) {
15      x+=x+1;
16      rpow2 (--a,x);
17    } else {
18      printf ("2^a= %d\n",x+1);
19      return (x+1);
20    }
21  }
22
23  int nrpow2 (int a, int x) {
24    if (a>0) {
25      x+=x+1;
26      return (x);
27    } else {
28      printf ("2^a= %d\n",x+1);
29      return (x+1);
30    }
31  }
32
33  int main () {
34    int a=24;
35    int x=0;
36
37    pow2 (a);
38    rpow2 (a,0);
39
40    while (a>=0) {
41      x=nrpow2(a--,x);
42    }
43  }
```

Abbildung 5.3: Funktionen, die 2^a berechnen, `pow2` (Schleife), `rpow2` (Rekursion), `nrpow2` (externe Schleife)

den. Eine Erkennung wie in `pow`, dass das Prädikat in Zeile 14 mehrfach für Bedingungen entlang eines Pfades verwendet wird, ist nicht möglich, da die typischen rückwärtsgerichteten Datenabhängigkeitskanten in rekursiven Funktionen der dargestellten Art *nicht* vorkommen. Andererseits ist eine Unterscheidung von Aufrufkontexten bei rekursiven Aufrufen nur dann sinnvoll, wenn sich keine unendlich langen Pfade ($PC^{\infty}(13,16)$) ergeben können.

Eine vorgegebene Rekursionstiefe k stellt hierbei eine Lösungsmöglichkeit dar, mit der sich die Pfadbedingung $a^1 > 0 \wedge a^2 > 0 \wedge \cdots \wedge a^k > 0 \wedge a \leq 0$ generieren lässt.

Im Fall, dass die Unterscheidung von Aufrufkontexten zu komplex wird, kann sowohl $a > 0$ als auch $a \leq 0$ durch *true* substituiert werden.

nrpow2: Die dritte Funktion enthält keine Schleifen und ist nicht rekursiv. Eine externe Schleife in der `main`-Funktion ruft `nrpow2` so oft mit geänderten Parametern auf, bis das Ergebnis vorliegt. Die etwas subtilere Pfadbedingung $PC(34,29)$ wird daher über den Pfad[4] $a_{34} \rightarrow a_{41} \rightarrow a_{23} \rightarrow a_{24} \rightarrow_{true} x_{26} \rightarrow x_{d41} \rightarrow x_{u41} \rightarrow x_{23} \rightarrow x_{29}$ berechnet.

Die Ausführungsbedingungen entlang des Pfades sind $true \wedge a_{40} \geq 0 \wedge true \wedge true \wedge a_{24} > 0 \wedge a_{40} \geq 0 \wedge a_{40} \geq 0 \wedge true \wedge a_{24} \leq 0 = false$.

Dieses unerwünschte Verhalten wird durch die rückwärtsgerichtete Datenabhängigkeitskante von x_{d41} nach x_{u41} hervorgerufen.

Lösungsmöglichkeiten für widersprüchliche Bedingungen

A1 Den Folgepfaden ab rückwärtsgerichtete Datenabhängigkeitskanten können neue Kontexte zugewiesen werden, so dass zwischen Variableninstanzen unterschieden wird. (gilt mit A2)

A2 Funktionsaufrufen können immer neue Kontexte zugewiesen werden, so dass jede Instanz (unabhängig von der Parameterbindung) unterschieden wird. Die Aufrufe müssen dabei in ihrer Anzahl k-limitiert sein.

B Widerspruch verursachende Prädikate können so lange durch *true* substituiert werden, bis der Widerspruch aufgelöst ist. Dies setzt voraus, dass die Bedingungen schon während der Konjunktion geprüft werden und nicht erst nach der Traversierung eines vollständigen Pfades.

[4]Definitionen von Variablen werden bei Uneindeutigkeiten mit einem d als Index markiert, Verwendungen mit u.

Technik zum Erkennen widersprüchlicher Bedingungen
Jeder realisierbare Pfad $p_1 \rightarrow^* p_{i-1} \rightarrow p_i \rightarrow^* p_n$ ist ausführbar. Daher darf die unausgewertete Pfadbedingung über dem Pfad nicht *false* werden, sondern entweder *true* oder ein Term. Sei $E(p_{i-1}) \neq \mathit{false}$ und $PC(p_i \rightarrow p_n) \neq \mathit{false}$. Dann gilt:

$$E(p_{i-1}) \wedge PC(p_i \rightarrow p_n) \equiv \begin{cases} \mathit{false} & \text{widersprüchliche Bedingung,} \\ \neq \mathit{false} & \text{korrekte Konjunktion} \end{cases}$$

Bei einer Realisierung der Pfadbedingungen, die auf Pfadaufzählung per Tiefensuchalgorithmus basiert, reicht es nun aus, $E(p_{i-1}) = \mathit{true}$ zu setzen, um die widersprüchliche Bedingung mit minimalem Informationsverlust zu verhindern. Im Hinblick auf den Tradeoff zwischen Präzision und Größe der zu untersuchenden Programme wird in dieser Arbeit die obige Technik verwendet, da aussagenlogische Operationen $(\wedge, \vee, \neg)$ in ihrer Zeit- und Speicherkomplexität von der Anzahl an Prädikaten abhängt, die mit A1 und A2 exponentiell steigen kann.

Kapitel 6

Pfadbedingungen für spezielle Datenstrukturen

Die bisherigen Pfadbedingungen basierten auf Ausführungsbedingungen, Φ-Bedingungen und Bedingungen für Aufrufkontexte im interprozeduralen Fall. Die Präzision wurde hauptsächlich durch Knotenbedingungen bestimmt, die schon eine umfassende Aussage über Informationsflüsse liefern. Für eine Reihe von speziellen Datenstrukturen bzw. Datentypen können jedoch zusätzliche Bedingungen generiert werden, die den Informationsfluss über Kanten präziser beschreiben.

6.1 Pfadbedingungen mit Datenabhängigkeitsinformationen

In Abschnitt 4.1 zeigte ein Beispiel ($i + 3 = 2 * j - 42$), wie Datenabhängigkeitsbedingungen über Arrays Pfadbedingungen präziser machen, wenn Zugriffe auf Array-Indizes berücksichtigt werden, so dass die einzelnen Arrayzellen unterschieden werden können. Aus Datenabhängigkeiten der Form `array`[`expression`$_1$] $\rightarrow$ `array`[`expression`$_2$] wird dabei die Datenbedingung *expression$_1$* = *expression$_2$* generiert.

Die allgemeine Definition 4.8 von Pfadbedingungen kann daher um zusätzliche δ-Bedingungen erweitert werden, die Pfadbedingungen entlang von Kanten stärker machen.

Definition 6.1 (Pfadbedingung (3. Form))
Es gelte Definition 4.8. Dann ist

$$PC(y,x) = \bigvee_{P_i : y \rightarrow^* x \in CH(y,x)} \Big(\bigwedge_{z \in P_i} E(z) \wedge \bigwedge_{v \rightarrow u \in P_i} \Phi(v \rightarrow u) \wedge \bigwedge_{v \rightarrow u \in P_i} \delta(v \rightarrow u) \Big)$$

eine *Pfadbedingung* zwischen y und x, die zusätzlich zu Definition 4.8 Datenabhängigkeitsbedingungen δ für spezielle Datenstrukturen und -typen enthält.

6.2 Pfadbedingungen für Arrays

Der Ausdruck $A(i) = expr$ bezeichnet den Index *expr* eines Arrayfeld-Zugriffs $array[expr]$ im Knoten i. Für Arrayindizes gilt dann im allgemeinen Fall, der nur Datenabhängigkeiten zwischen Definitionen und Verwendungen ($def \rightarrow use$) zulässt, folgende δ-Bedingung.

$$\delta(v \rightarrow u) = \begin{cases} A(v) = A(u) & \text{wenn } v \text{ Definition und } u \text{ Verwendung,} \\ \delta(v \rightarrow u) & \text{sonst.} \end{cases}$$

Diese Definition ist für Nicht-Array-Kanten unbestimmt. Generell kann immer die konservative Definition $\delta(v \rightarrow u) = true$ verwendet werden.

Die Gleichung $A(v) = A(u)$ kann beliebige in ANSI-C mögliche Integer-Arithmetik enthalten, so dass ihre Auflösung theoretisch nicht berechenbar ist. In der Praxis sind die Ausdrücke oft weniger komplex bzw. trivial, so dass Constraint-Solver wie z.B. ECLiPSe (Constraint-Logic-Programming) oder Reduce/Redlog (Quantorenelimination) Lösungen berechnen können (siehe hierzu Abschnitt 9.1). Im Fall von Presburger Arithmetik [Wei97a, PW98] sind diese Gleichungen immer lösbar.

Wenn es für eine Arrayverwendung (Arrayfeld-Zugriff) nur genau eine Definitionstelle gibt, gelten die obigen Aussagen. In der Regel existieren entweder mehrere Definitionsstellen oder es ist statisch unentscheidbar, ob eine Definitionsstelle eine Arrayzelle beschreibt.

Beispiel 6.1

```
1 a[i] = x;
2 a[j] = y;
3 z = a[k];
```

Der Programmauszug beschreibt zwei Definitionen von Arrayzellen in Zeile 1 und 2, sowie eine Arrayverwendung in Zeile 3. Die zugehörende Abbildung 6.1 zeigt in (a) den Programmabhängigkeitsgraphen zum dargestellten Programmauszug. Die Modellierung orientiert sich am Standard nach [ADS91], bei der Datenabhängigkeiten in Arrays über *non-killing*-Definitionen realisiert werden. Alle möglichen Definitionen behalten an der Verwendungsstelle ihre Gültigkeit.

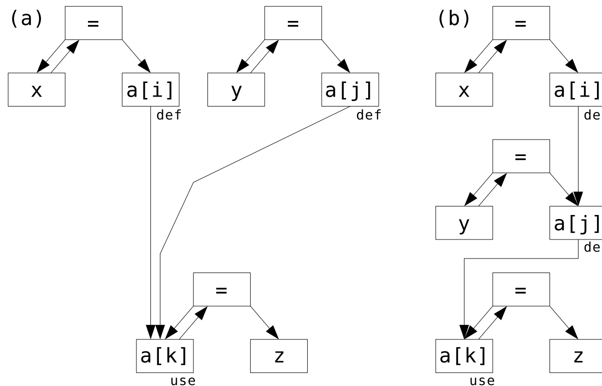


Abbildung 6.1: Unterschiedliche Arrayrealisierungen in Programmabhängigkeitsgraphen, (a) Standard, (b) speziell für Pfadbedingungen

Als Pfadbedingung über die Datenabhängigkeitskante von x_1 nach z_3 erhält man $PC(x_1, z_3) \equiv (i = k)$, und für y_2 nach z_3 erhält man $PC(y_2, z_3) \equiv (j = k)$. Die Pfadbedingung ist korrekt, jedoch zu konservativ, da mögliche Überdeckungen wie $i = j$ nicht betrachtet werden. Bei $i = j$ gilt $PC(x_1, z_3) \equiv \mathit{false}$, da Arrayzelle i in Zeile 2 überschrieben wird.

Um dieses Problem zu lösen, modelliert der PDG Arrayzugriffe nach einer *killing*-Definition. Hierbei sind sowohl $\mathit{def} \rightarrow \mathit{use}$ als auch $\mathit{def}_1 \rightarrow \mathit{def}_2$ durch Datenabhängigkeitskanten miteinander verbunden. Die Definition einer Arrayzelle entspricht zugleich einer Verwendung des Arrays.

Die neue Modellierung liefert Datenabhängigkeiten entlang der Zeilen $1 \rightarrow 2 \rightarrow 3$ (siehe Abbildung 6.1 (b)). Die Definition 6.2 ist nur noch für $2 \rightarrow 3$ gültig, nicht jedoch für $1 \rightarrow 2$. Daher ist eine Erweiterung dieser Definition notwendig, um $PC(x_1, z_3)$ korrekt berechnen zu können.

δ-Bedingungen zwischen $\mathit{def}_1 \rightarrow \mathit{def}_2$ stellen *Negationen* von Definition 6.2 dar, um ein Überdecken von def_1 durch def_2 zu verhindern. Eine mögliche Verwendungsstelle kann von def_1 nur dann abhängig sein, wenn die Arrayindizes von def_1 und def_2 nicht gleich sind. δ-Bedingungen zwischen Definitionen werden nicht beim Erreichen einer entsprechenden $\mathit{def}_1 \rightarrow \mathit{def}_2$ Kante berechnet, sondern beim Erreichen einer $\mathit{def} \rightarrow \mathit{use}$ Kante.

Im Folgenden bezeichnet $v \overset{dd}{\rightarrow} u$ eine Datenabhängigkeitskante zwischen den Arraydefinitionen v und u und $v \overset{du}{\rightarrow} u$ eine Datenabhängigkeitskante zwischen Arraydefinition v und -verwendung u.

Die Kantenbedingung $\delta(i \overset{du}{\rightarrow} j)$ wird durch zusätzliche Bedingungen $i_1 \overset{dd}{\rightarrow} i_2$, $i_2 \overset{dd}{\rightarrow} i_3, \ldots$ angereichert, die auf den direkten Vorgängerkanten des *aktuellen Pfades* liegen.

Definition 6.2 (Array-Bedingung)
Sei P_i ein Pfad in $CH(y,x)$ und $v_1 \stackrel{dd}{\rightarrow} v_2 \stackrel{dd}{\rightarrow} \cdots \stackrel{dd}{\rightarrow} v_k \stackrel{du}{\rightarrow} u$ ein maximaler Teilpfad auf P_i. Dann ist

$$\delta_{P_i}(v_k \stackrel{du}{\rightarrow} u) \equiv \bigvee_{i=1\dots k} \Big((A(v_i) = A(u)) \wedge \bigwedge_{j=i+1\dots k} (A(v_i) \neq A(v_j))\Big)$$

eine *Array-Bedingung* für $v_k \stackrel{du}{\rightarrow} u$ und den gegebenen Pfad $v_1 \rightarrow^* u$.

Da diese δ-Bedingungen nur über $v \stackrel{du}{\rightarrow} u$ Kanten berechnet werden, reicht es nicht aus, nur v_1 als Definitionsstelle zu betrachten, da entlang des Pfades jedes $v_1 \dots v_k$ die alleinige Definitionsstelle für u darstellen kann.

Der maximale Teilpfad ist in Definition 6.1 immer eindeutig bestimmt, so dass $\delta(v \rightarrow u) = \delta_{P_i}(v \stackrel{du}{\rightarrow} u)$ geschrieben werden kann. Das obige Beispiel ergibt $PC(1,3) \equiv \delta(2,3) \equiv (i = k) \wedge (i \neq j)$ und stellt die stärkste Bedingung für einen Informationsfluss dar.

Unabhängig von speziellen Pfaden kann es nützlich sein, für eine $v \stackrel{du}{\rightarrow} u$ Kante alle möglichen Definitionsstellen innerhalb eines Chops $CH(y,x)$ zu berücksichtigen. Damit lässt sich eine allgemeine Aussage unabhängig von Pfaden treffen, unter welchen Umständen ein Informationsfluss entlang einer $v \stackrel{du}{\rightarrow} u$ Kante stattfinden kann.

Definition 6.3 (Array-Bedingung (global))
Seien $v_1^l \stackrel{dd}{\rightarrow} v_2^l \stackrel{dd}{\rightarrow} \cdots \stackrel{dd}{\rightarrow} v_{k-1}^l \stackrel{dd}{\rightarrow} v_k \stackrel{du}{\rightarrow} u$ alle l maximalen Teilpfade für $v_k \stackrel{du}{\rightarrow} u$ in $CH(y,x)$. Dann ist

$$\delta_G(v_k \stackrel{du}{\rightarrow} u) \equiv \bigvee_{l} \Big(\bigvee_{i=1\dots k} \big((A(v_i) = A(u)) \wedge \bigwedge_{j=i+1\dots k} (A(v_i) \neq A(v_j))\big)\Big)$$

eine *globale Array-Bedingung*, die Teilbedingungen für alle Abhängigkeiten innerhalb von $CH(y,x)$ berücksichtigt.

Beispiel 6.2

```
1 a[i] = x;
2 if (a<0)
3   a[i2] = x2;
4 else
5   a[i3] = y2;
6 a[j] = y;
7 z = a[k];
```

Eine Erweiterung des Beispielquelltextes zu obiger Form liefert bei globaler Array-Bedingung $\delta_G(6,7) \equiv \big(i = k \wedge i \neq i2 \wedge i \neq j\big) \vee \big(i = k \wedge i \neq i3 \wedge i \neq j\big) \vee \big(i2 = k \wedge i2 \neq j\big) \vee \big(i3 = k \wedge i3 \neq j\big) \vee \big(j = k\big)$.

6.2.1 Array-Bedingungen am Beispiel

```
1  void main () {
2    int i;
3    int a[100];
4    int k=40;
5    int l=53;
6    int x;
7
8    a[0]=100;
9    for (i = 1;
10     i < a[0]; ++i) {
11       a[i]=255;
12   }
13   a[5]=5;
14   if (1)
15       a[10] = 10;
16   else
17       a[20] = 20;
18   a[30] = 30;
19   a[k] = 40;
20   a[50] = 50;
21
22   if (a[l] == x) {
23     ;
24   }
25 }
```

Abbildung 6.2: Komplexes Beispiel für präzise Array-Bedingungen

Als Abschluss wird das Beispiel in Abbildung 6.2 betrachtet. Vielfache Arrayzugriffe und -definitionen machen die manuelle Programmanalyse kompliziert, da Variablen zusätzlich in Felder eingeteilt werden. Im Beispiel soll die Pfadbedingung zwischen `100` in Zeile 8 und `a[l]` in Zeile 22 berechnet werden. Es gilt die Frage zu klären, unter welchen Umständen ein Informationsfluss stattfindet, da Program-Slicing die binäre Information *true* für Abhängigkeiten zwischen Start- und Endknoten liefert.

Die unausgewertete und manuell erstellte Pfadbedingung ergibt für das Beispiel: $\big((0 = 0 \wedge i < a[0] \wedge i = l \wedge i \neq 5 \wedge i \neq 10 \wedge i \neq 30 \wedge i \neq k \wedge i \neq 50) \vee (0 = 0 \wedge i < a[0] \wedge i = l \wedge i \neq 5 \wedge i \neq 20 \wedge i \neq 30 \wedge i \neq k \wedge i \neq 50) \vee (0 = l \wedge 0 \neq 5 \wedge 0 \neq 10 \wedge 0 \neq 30 \wedge 0 \neq k \wedge 0 \neq 50) \vee (0 = l \wedge 0 \neq 5 \wedge 0 \neq 20 \wedge 0 \neq 30 \wedge 0 \neq k \wedge 0 \neq 50)\big) \wedge \big(a[0] = 100 \wedge l = 53 \wedge k = 40\big)$.

Der Pfadbedingungsgenerator liefert unter Verwendung von Φ, γ und den obigen δ-Bedingungen sowie Substitutionen innerhalb von Zuweisungen automatisch:

$$PC(100_8, \texttt{a[l]}_{22}) \equiv i = 53$$

Die obigen logischen Bedingungen über Integervariablen werden vom Pfadbedingungsgenerator bereits vereinfacht, so dass kein externer Anschluss eines Constraint-Solvers notwendig ist.

Die vermutlich unerwartete Bedingung $i = 53$ wird deutlich, wenn die Abhängigkeiten genauer untersucht werden. Zwischen $8 \rightarrow 13 \rightarrow 15 \rightarrow 18 \rightarrow 19 \rightarrow 20 \rightarrow 22$ und $8 \rightarrow 13 \rightarrow 17 \rightarrow 18 \rightarrow 19 \rightarrow 20 \rightarrow 22$ findet kein Informationsfluss statt, da $l \neq 0$ gilt. `a[0]` in Zeile 10 ist jedoch datenabhängig vom Startknoten in Zeile 8 (es gilt die Array-Bedingung $0 = 0$). `a[i]` in

Zeile 11 ist kontrollabhängig von Zeile 10. Der Zielknoten `a[l]` in Zeile 22 ist nun datenabhängig von `a[i]`. Die Array-Bedingung liefert hierfür die exakte Aussage $i = l \wedge i \neq 5 \wedge (i \neq 10 \vee i \neq 20) \wedge i \neq 30 \wedge i \neq k \wedge i \neq 50)$ und vereinfacht zu $i = 53$.

6.3 Pfadbedingungen für Abstrakte Datentypen

Ein Abstrakter Datentyp (ADT,[LEW96]) stellt in der Praxis einen Container mit spezifischen Daten und Funktionen für einen bestimmten Zweck dar, wie z.B. Mengen, Listen, Maps oder Stacks. Im mathematischen Sinne werden Abstrakte Datentypen zustands-basiert und axiomatisch spezifiziert. Sie definieren dabei mit ihren Axiomen eine Theorie.

In Programmiersprachen wie C++ (STL) oder Java sind Bibliotheken mit Abstrakten Datentypen Teil des Sprachumfangs. In ANSI-C gibt es hierfür keinen Standard. Jedoch können anhand der theoretischen Definition von Abstrakten Datentypen Pfadbedingungen in bestimmten Fällen stärker gemacht werden.

Der Abstrakte Datentyp *STACK* ist z.B. folgendermaßen definiert:

$$\begin{aligned}
\mathit{NewStack} &: \rightarrow \mathit{STACK} \\
\mathit{IsEmpty} &: \mathit{STACK} \rightarrow \mathit{BOOL} \\
\mathit{Push} &: \mathit{STACK} \times \mathit{SYMBOL} \rightarrow \mathit{STACK} \\
\mathit{Pop} &: \mathit{STACK} \rightarrow \mathit{STACK} \\
\mathit{Top} &: \mathit{STACK} \rightarrow \mathit{SYMBOL} \\
\mathit{IsEmpty}(\mathit{NewStack}) &= \mathit{true} \\
\mathit{IsEmpty}(\mathit{Push}(s,n)) &= \mathit{false} \\
\mathit{Pop}(\mathit{Push}(s,n)) &= s \\
\mathit{Top}(\mathit{Push}(s,n)) &= n \\
\mathit{Pop}(\mathit{NewStack}) &= \text{✗} \\
\mathit{Top}(\mathit{NewStack}) &= \text{✗}
\end{aligned}$$

Zur Erweiterung von Pfadbedingungen kann angenommen werden, dass bestimmte Abstrakte Datentypen im Quelltext als Bibliothek vorliegen und sich präzise an ihre axiomatischen Definitionen halten. Die Auswertung dieser Definition erfolgt analog [BHK89], indem die Gleichungen von links nach rechts orientiert betrachtet werden, um Rewrite-Regeln anwenden zu können. Dieses Term-Rewriting erfolgt immer dann, wenn der Pfadbedingungsgenerator Terme entsprechend der axiomatischen Definition mittels Pattern-Matching im Programmabhängigkeitsgraphen erkennt.

Beispiel 6.3

```
1  n = readkey();
2  if (...)
3    s = Push(s,n);
4  else
5    s = NewStack();
6  ...
7  x = top(s);
8  ...
9  while (x == 'q')
10   p();
```

Betrachtet man die Datenabhängigkeitskanten zwischen $3 \rightarrow 7$ und $5 \rightarrow 7$ so erhält man ohne Term-Rewriting in beiden Fällen als Datenabhängigkeitsbedingung *true*. Dies ist korrekt, jedoch nicht so präzise wie möglich, da der Inhalt der Stack-Elemente unbeachtet bleibt. Sobald Term-Rewriting für ADT Stack eingesetzt wird, ergeben die Datenabhängigkeitsbedingungen stattdessen $\delta(3 \rightarrow 7) \equiv (x = n) \equiv (x = \mathit{readkey}())$ und $\delta(5 \rightarrow 7) \equiv$ ✗.

Entlang der Pfade werden ADT-spezifische Informationen weiterpropagiert und damit nicht nur ein genaueres Lösen der Pfadbedingungen ermöglicht, sondern auch mögliche fehlerträchtige Programmstellen aufgedeckt. Die Pfadbedingung $PC(1, 10)$ wird ohne ADT-Unterstützung zu $x =$ 'q' ausgewertet. Mit ADT-Unterstützung liefert sie $x = \mathit{readkey}() \wedge x =$ 'q' und damit $\mathit{readkey}() =$ 'q'. Das Ergebnis zeigt, dass Pfadbedingungen unter Berücksichtigung von Abstrakten Datentypen deutlich aussagekräftiger werden können.

δ-Bedingungen für allgemeine Abstrakte Datentypen lassen sich wie folgt definieren:

Definition 6.4 (ADT-Bedingung)
Sei l die linke Seite und r die rechte Seite einer axiomatischen Definition eines ADT. Sei $(u_i := l)$ ein Element aus der Menge der Zuweisungen (*Assignments*). Sei $(u_i = v_j) \in \Phi(i \rightarrow j)$ eine Φ-Bedingung zwischen u_i und v_j. Dann ist

$$\frac{u_i := l \in \mathit{Assignments},\ l \mapsto r, (u_i = v_j) \in \Phi(i \rightarrow j)}{\delta(i \rightarrow j) \equiv (v_j = r)}$$

eine *ADT-Bedingung* für $\delta(i \rightarrow j)$. Das Ziel der Datenabhängigkeitskante v_j darf durch die rechte Seite der axiomatischen Definition r substituiert werden. Dies gilt auch für *implizite Zuweisungen* innerhalb von Prädikaten, in denen $u_i = v_j$ gilt.

Zur Erkennung der axiomatischen Definitionen werden alle möglichen Quellen für Datenabhängigkeiten einer Variablen mittels Pattern-Matching im

PDG ermittelt und durch bedingte Kanten entsprechend der Axiome der gegebenen Abstrakten Datentypen angereichert. Diese Ermittlung kann nur in dem Fall vorgenommen werden, wenn keine zyklischen Substitutionen eingeführt werden, die leicht durch zyklische Datenabhängigkeiten innerhalb des PDGs erkannt werden können.

6.3.1 ADT-Bedingungen am Beispiel

Abschließend wird anhand eines realen Beispiels der direkte Vergleich zwischen einer normalen Pfadbedingungsberechnung und der Verwendung spezieller ADT-Bedingungen demonstriert.

Abbildung 6.3 zeigt ein typisches Taschenrechnerprogramm, das hier aus Platzgründen nur die Operatoren + und − erkennt. Die Funktion `eval()` liefert das Ergebnis für einen Ausdruck, der in der Variablen `expr` abgelegt ist. `get_token` wandelt ein eingelesenes Zeichen in ein entsprechendes Symbol um. Im Hauptprogramm `main()` wird `eval()` parameterlos aufgerufen und das Ergebnis ausgegeben. Der Taschenrechner selbst verwendet für seine Berechnungen einen typischen Stack, dem Operanden und Operatoren entnommen werden und das Ergebnis wieder auf den Stack gelegt wird.

Für die Untersuchung soll die interprozedurale Pfadbedingung zwischen $expr_{90}$ und dem Ergebnis aus $eval()_{92}$ ermittelt werden. Ohne Behandlung von axiomatischen Definitionen liefert der Pfadbedingungsgenerator in disjunktiver Normalform $PC(expr_{90}, eval()_{92}) =$

$$\begin{array}{ll} & token_{58} \neq eos \;\wedge\; token_{59} = operand \\ \vee & token_{58} \neq eos \;\wedge\; token_{62} = plus \;\wedge\; token_{59} \neq operand \\ \vee & token_{58} \neq eos \;\wedge\; token_{62} = minus \;\wedge\; token_{62} \neq plus \\ & \wedge\; token_{59} \neq operand \end{array}$$

Diese Pfadbedingung ist, wie erwartet, erfüllbar und zeigt präzise die 3 Fälle, entlang derer Abhängigkeiten zwischen dem Start- und Zielpunkt möglich sind.

Wenn die axiomatischen Definitionen für $STACK$ als ADT-Bedingungen eingesetzt werden, liefert die Pfadbedingungsberechnung:

$$\begin{array}{ll} & token_{58} \neq eos \;\wedge\; token_{59} = operand \;\wedge\; \mathbf{symbol}_{55} - \mathbf{'0'} = \mathbf{return}_{85} \\ \vee & token_{58} \neq eos \;\wedge\; token_{62} = plus \;\wedge\; token_{59} \neq operand \\ & \wedge\; \mathbf{top}_{66} + \mathbf{top}_{64} = \mathbf{return}_{85} \\ \vee & token_{58} \neq eos \;\wedge token_{62} = minus \;\wedge\; token_{62} \neq plus \\ & \wedge\; token_{59} \neq operand \wedge\; \mathbf{top}_{73} = \mathbf{0} \;\wedge\; \mathbf{return}_{85} = \mathbf{1} \\ \vee & token_{58} \neq eos \;\wedge token_{62} = minus \;\wedge\; token_{62} \neq plus \\ & \wedge\; token_{59} \neq operand \wedge\; \mathbf{top}_{73} \neq \mathbf{0} \;\wedge\; \mathbf{return}_{85} = \mathbf{top}_{73} - \mathbf{top}_{71} \end{array}$$

```
1  typedef enum {plus, minus, eos,
2          operand} precedence;
3
4  struct adt_stack {
5    int s[100];
6    int i;
7  };
8
9  struct adt_stack stack;
10 char* expr;
11
12 struct adt_stack push (int e,
13        struct adt_stack stack) {
14   ++(stack.i);
15   stack.s[stack.i] = e;
16   return stack;
17 }
18
19 struct adt_stack pop (struct adt_stack
20        stack) {
21   --(stack.i);
22   return stack;
23 }
24
25 int top (struct adt_stack stack) {
26   return stack.s[stack.i];
27 }
28
29 int isempty (struct adt_stack stack) {
30   return (stack.i==0);
31 }
32
33 struct adt_stack newstack () {
34        struct adt_stack s;
35   s.i = 0;
36   return s;
37 }
38
39 precedence get_token (char* symbol,
40            int* n) {
41   *symbol = expr[(*n)++];
42   switch (*symbol) {
43     case '+': return plus;
44     case '-': return minus;
45     case '\0': return eos;
46     default: return operand;
47   }
48 }
49 int eval (void) {
50   precedence token;
51   char symbol;
52   int op1, op2;
53   int n=0;
54
55   token = get_token (&symbol, &n);
56   assert (token==plus || token==minus ||
57           token==eos  || token==operand);
58   while (token != eos) {
59     if (token == operand) {
60       stack = push(symbol-'0', stack);
61     } else {
62       switch(token) {
63         case plus:
64           op2 = top(stack);
65           stack = pop(stack);
66           op1 = top(stack);
67           stack = pop(stack);
68           stack = push(op1+op2,stack);
69           break;
70         case minus:
71           op2 = top(stack);
72           stack = pop(stack);
73           op1 = top(stack);
74           stack = pop(stack);
75           if (op1==0) {
76             stack = push(1,stack);
77           } else {
78             stack = push(op1-op2,stack);
79           }
80           break;
81       }
82     }
83     token = get_token (&symbol, &n);
84   }
85   return (top(stack));
86 }
87
88 int main () {
89     stack = newstack();
90     expr = "75-123-++1-";
91     printf ("'%s' results to %d\n",
92             expr, eval());
93
94     return 0;
95 }
```

Abbildung 6.3: Stärkere Pfadbedingungen mittels ADT-Bedingungen

Die Pfadbedingung wird mit ADT-Bedingungen deutlich stärker und liefert tiefergehende Informationen über die möglichen Abhängigkeiten im Programm. Der Pfadbedingungsgenerator hat aus den vorgegebenen axiomatischen Definitionen $Top(Push(s,n)) = n$ im Programmabhängigkeitsgraphen herausfiltern können. Die neuen ADT-Bedingungen sind hervorgehoben dargestellt, und alle vorherigen Bedingungen erhalten geblieben.

Im *operand*-Fall in Zeile 60 ist zu sehen, dass eine als Zeichen codierte Zahl auf den Stack gelegt wird. Der *plus*-Fall zeigt nicht nur, dass tatsächlich addiert wird, sondern auch, um welche SSA-Variablen es sich tatsächlich handelt. Interessant ist der *minus*-Fall, der ein obskures Verhalten bei speziellen Eingaben zeigt. Dieser Fall ist nun in zwei Fälle zerlegt. Die Subtraktion findet nur dann statt, wenn der Minuend größer Null ist. Im anderen Fall wird keine Subtraktion durchgeführt, sondern nur der Wert 1 zurückgeliefert, so dass tatsächlich ein falscher Wert berechnet wird.

Man sieht sehr deutlich, dass ADT-Bedingungen in der Regel zu wesentlich stärkeren und aussagekräftigeren Pfadbedingungen führen können. Aber auch Pfadbedingungen ohne die Behandlung von axiomatischen Definitionen sind natürlich hilfreich, da immer der Vergleich zum reinen Program-Slicing mit seinem binären Ergebnis (hier *true*) betrachtet werden muss.

6.4 Pfadbedingungen für Assertions

Pfadbedingungen waren bisher immer nur so präzise, wie Abhängigkeiten aus dem Quelltext extrahiert werden konnten. Abstrakte Datentypen erweitern zwar die Aussagekraft auf Basis einer axiomatischen Definition für Standarddatentypen, jedoch deckt das Verfahren z.B. keine absichtliche Beschränkung von Wertebereichen einzelner Variablen ab. Daher ist eine Erweiterung von Pfadbedingungen wünschenswert, mit der z.B. Informationen über die eingesetzte Hardwareplattform integriert werden können.

Aus diesem Grund wurde ein Verfahren entwickelt, wie die Konzepte anderer Verifikationsverfahren auf Pfadbedingungen übertragen werden können. LCLint [EGHT94, Eva94] ist z.B. ein Werkzeug, bei dem ANSI-C-Programme mit verschiedenen Arten von formaler Spezifikation angereichert werden können, um Inkonsistenzen aufzudecken. LCLint dient hierbei in erster Linie zum Programmverstehen und aussagekräftigeren Dokumentieren. Im Gegensatz zu reinen Verifikationssystemen, die ein hohes Maß an Benutzerwissen voraussetzen, kann bei LCLint der Trade-Off zwischen Aufwand der Spezifikation und Umfang der Überprüfung vom Benutzer festgelegt werden.

Ein anderes System ist Larch/C++ [Lea96, CL94], das als Spezifikationssprache für C++-Interfaces designed wurde und auch das Verhalten von Funktionen beschreiben kann. Die formale Spezifikation ist so ausgelegt, dass sie von allen Implementierungsdetails abstrahiert. Für jede Funktion können Vor- und Nachbedingungen angegeben werden, die, wie z.B. bei LCLint, automatisch bewiesen (oder widerlegt) werden können. Eine typische Larch/C++-Spezifikation für eine Funktion zum Berechnen von Zinsen sieht beispielsweise folgendermaßen aus [CL94]:

```
1  float interest(float x, float rate = 0.05)
2  {
3    requires 0.0 <= rate /\ rate <= 1.0;
4    ensures result = x * rate;
5  }
```

Die Spezifikation fordert als Vorbedingung, dass die Zinsrate *rate* – falls angegeben – immer ≥ 0.0 und ≤ 1.0 ist. Als Nachbedingung sichert sie zu, dass das Ergebnis der Multiplikation der Zinsrate mit dem übergebenen Betrag x ist.

Analog hierzu soll der Pfadbedingungsgenerator die Möglichkeit bieten, ANSI-C Quelltexte mit prädikatenlogischen Bedingungen einschl. Arithmetik und Quantoren zu annotieren, so dass Pfadbedingungen um Benutzerwissen angereichert und damit auch Constraint-Solvern bessere Lösungsmöglichkeiten gegeben werden. Man kann von Zusicherungen an Programmvariablen sprechen, die jeweils an ausgezeichneten Quellcodestellen gültig sind.

Um dies zu realisieren werden sog. *XAssertion-Bereiche* eingeführt, die Hintergrundwissen für diejenigen Teilbedingungen der Pfadbedingung enthalten, die im definierten Bereich erzeugt wurden. XAssertions können sowohl für vollständige Funktionen gelten, als auch für mehrere disjunkte Bereiche einer Funktion. XAssertions dürfen sich nicht überlappen, jedoch ineinander geschachtelt sein. Jeder Geltungsbereich ist disjunkt von den anderen, so dass widersprüchliche Aussagen, die allein durch Schachtelung von XAssertions hervorgerufen werden, ausgeschlossen sind. Eine Spezialisierung durch Schachtelung von XAssertions erfolgt durch Kopieren von äußeren Bedingungen in innere.

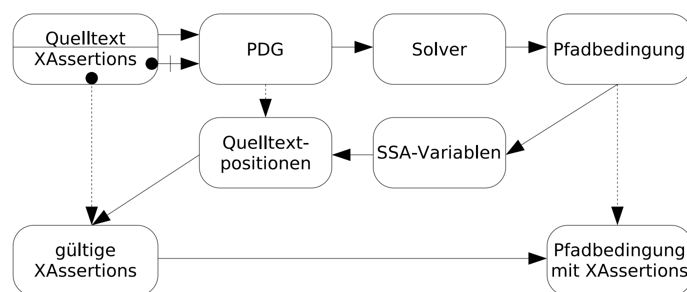


Abbildung 6.4: Formale Spezifikation in Pfadbedingungen

Abbildung 6.4 veranschaulicht die Generierung von XAssertions in Pfadbedingungen. Durchgehende Linien geben den allgemeinen Ablauf an, gestrichelte Linien die Verwendung von Daten.

XAssertions werden als Kommentar in den Quelltext eingefügt und haben

folgenden Aufbau und Eigenschaften:

- Variablen und Werte werden in Nicht-SSA-Form (z.B. x) angegeben. Die Zuordnung zu SSA-Variablen ($x_1, \ldots, x_n$) erfolgt für jeden Block automatisch.
- Prädikatenlogische Vergleichsoperatoren sind $=, \neq, <, \leq, >, \geq$ (textuell: `==, not, <, >=, >, >=`)
- Boolsche Operatoren sind $\neg, \wedge\vee$ (textuell: `not`, `and`, `or` sowohl in Infix- als auch in Prefix-Schreibweise)
- Quantoren sind $\forall, \exists$ (textuell: `forall x,y,...:(...)`, `exists x,y,...:(...)`)

Für alle Variablen, Konstanten und Operatoren der Originalpfadbedingung wird bestimmt, ob sie zu XAssertions gehören. Dies ist genau dann der Fall, wenn ein zugehöriger PDG-Knoten von XAssertions regiert wird. Damit wird die Mengenauswahl der *aktiven* XAssertions getroffen.

Definition 6.5 (XAssertion-Bedingung)
Sei M ein Monom der Pfadbedingung PC (disjunktive Normalform). Sei *Var(M)* $= \{u_1, u_2, \ldots\}$ die Menge der Variablen in SSA-Form von *M*. Sei X eine XAssertion-Bedingung mit $Var(X) = \{v_1, v_2, \ldots\}$ ohne SSA-Form. Sei $Var(XAssertion\text{-}Region(X)) = \{s_1, s_2, \ldots\}$ die Menge der Variablen innerhalb der XAssertion-Region von X in SSA-Form. Dann erweitert

$$E_{v_i} = \begin{cases} \{v_i \mapsto u_j\} & v_i \in Var(X), u_j \in Var(M), \\ & v_i = u_j \text{ bis auf SSA-Index von } u_j, \\ & u_j \in XAssertion\text{-}Region(X) \\ \{v_i \mapsto s_k\} & v_i \in Var(X), s_k \notin Var(M), \\ & \forall_j : v_i \not\mapsto u_j, \\ & v_i = s_k \text{ bis auf SSA-Index von } s_k, \\ & s_k \in XAssertion\text{-}Region(X) \\ \{v_i \mapsto v_i\} & sonst. \end{cases}$$

mit

$$M \equiv M^{OLD} \;\wedge\; X \lhd\; E_{v_1}, E_{v_2}, \ldots$$

das originale Monom M um zugehörende *XAssertion-Bedingungen.*

Während der XAssertion-Berechnung wird für jeden Knoten im PDG seine XAssertion-Region bestimmt, so dass ausgehend von einer Pfadbedingung jeder Term präzise einer/keiner Region zugeordnet werden kann.

Die Dreiteilung von $\{v_i \mapsto u_j\}$ ermöglicht eine Präzisierung der Pfadbedingungen, ohne zu viel unnötige Informationen zu liefern. Im ersten Fall

werden XAssertion-Variablen nur auf diejenigen SSA-Variablen abgebildet, die auch in den Pfadbedingungen vorkommen. So werden nur Bedingungen generiert, die Gemeinsamkeiten mit den originalen Pfadbedingungen aufweisen. Erst wenn dies nicht möglich sein sollte, werden im zweiten Fall die XAssertion-Variablen auf SSA-Variablen im Quelltext abgebildet, die sich innerhalb der aktuellen XAssertion-Region befinden. Falls es keine Gemeinsamkeiten zwischen den XAssertion-Variablen und Quelltext-Variablen gibt, wird keine SSA-Ersetzung der SSA-Variablen durchgeführt.

6.4.1 XAssertions am Beispiel

Abbildung 6.7 zeigt einen Auszug aus einer Steuerungssoftware für elektronische Waagen (siehe auch Abb. 3.1 auf Seite 24). Innerhalb einer `while`-Schleife wird über den Tastaturport `p_cd` und Analog-Digital-Wandler `p_ab` ein Messwert in Zeile 19 eingelesen und anschließend kalibriert. Zeile 38 gibt den anzuzeigenden Messwert und Zeile 27 die übergebene Warengruppe aus. Das Programm entstammt [Sne96] und gilt als intraprozeduraler VALSOFT-Klassiker.

Die Pfadbedingung $PC(\texttt{p_cd}, \texttt{u_kg})$ von Zeile 11 nach Zeile 38 (in Abb. 6.7) ist in Abb. 6.5 zu sehen. Die Pfadbedingung legt eine Beeinflussung des ausgegebenen Gewichts durch den Tastaturport offen. Diese Manipulation wird genau dann verursacht, wenn im Tastaturpuffer '+' bzw. '−' stehen. Bei '+' wird der Kalibrierungsfaktor um 1% erhöht, bei '−' entsprechend reduziert.

1 $(\texttt{p_cd}_{12}[\texttt{0}_{54}]\ \texttt{\&}_{53}\ \texttt{1}_{57}) \neq_{52} \texttt{0}_{58}$
2 $(\texttt{p_cd}_{12}[\texttt{0}_{79}]\ \texttt{\&}_{78}\ \texttt{16}_{82}) \neq_{77} \texttt{0}_{83}$
3 $(\texttt{p_ab}_{8}[\texttt{0}_{27}]\ \texttt{\&}_{26}\ \texttt{16}_{30}) =_{25} \texttt{0}_{31}$
4 $\texttt{e_puf}_{87}[\texttt{idx}_{86}] =_{85} \texttt{'+'}_{89}$
5 $\texttt{idx}_{65} <_{64} \texttt{7}_{66}$
6 $\vee$
7 $(\texttt{p_cd}_{12}[\texttt{0}_{54}]\ \texttt{\&}_{53}\ \texttt{1}_{57}) \neq_{52} \texttt{0}_{58}$
8 $(\texttt{p_cd}_{12}[\texttt{0}_{79}]\ \texttt{\&}_{78}\ \texttt{16}_{82}) \neq_{77} \texttt{0}_{83}$
9 $\texttt{e_puf}_{96}[\texttt{idx}_{95}] =_{94} \texttt{'-'}_{98}$
10 $(\texttt{p_ab}_{8}[\texttt{0}_{27}]\ \texttt{\&}_{26}\ \texttt{16}_{30}) =_{25} \texttt{0}_{31}$
11 $\texttt{e_puf}_{87}[\texttt{idx}_{86}] \neq_{85} \texttt{'+'}_{89}$
12 $\texttt{idx}_{65} <_{64} \texttt{7}_{66}$

Abbildung 6.5: Der Klassiker

Die Pfadbedingung des Solvers enthält statt Zeilennummern als Index die realen PDG-Knotennummern, da zeilenweise Indizes wegen möglicher Seiteneffekte innerhalb einzelner Zeilen nicht präzise genug sind. Für Variablen sind dies SSA-Indizes. Die Einführung von Indizes für alle Operanden und

Operatoren ist sinnvoll, wenn präzise Rückbezüge in den Quelltext unter Einbeziehung von SSA-Informationen notwendig sind. Durch automatische Rückwärtssubstitutionen von Variablen weist in einer Pfadbedingung i.Allg. nur ein Operator – nicht eine Variable – auf die genaue Zeile hin, aus deren Prädikat ein Pfadbedingungsterm erzeugt wurde.

Der Klassiker 6.5 ist an sich selbsterklärend und enthält Bedingungen über alle Prädikate des Quelltextes. Er ist in zwei Disjunktionen aufgeteilt, die nur durch '+' bzw. '−' unterschieden werden.

Für das Beispiel wurde der Quelltext 6.7 nun mit zwei geschachtelten XAssertions angereichert. Die außenliegende XAssertion gilt für die gesamte Funktion `main`. Die innenliegende XAssertion gilt für die `for`-Schleife von Zeile 26 bis 34. XAssertions lassen prädikatenlogische Aussagen für beliebige Variablen zu, wobei die Variablen generell in Nicht-SSA-Form angegeben werden und daher für alle Instanzen der Variablen gültig sind. Der Pfadbedingungsgenerator generiert für jede Variableninstanz eine entsprechende Bedingung.

Die erste Forderung der außenliegenden XAssertion spezifiziert ein funktionsübergreifendes Variablenverhalten, das z.B. aufgrund hardwaretechnischer Restriktionen gilt. Beispielsweise kann der Pufferspeicher auf 10 Zeichen begrenzt sein und die Tastatur nur die Symbole 'A'-'Z' enthalten. In der geschachtelten XAssertion wird zudem das Hintergrundwissen eines Entwicklers eingebracht, dass Warenabkürzungen nur aus 7 Buchstaben bestehen. Die neue Pfadbedingung mit Spezifikation nach XAssertions ist in Abbildung 6.6 dargestellt.

Die Originalbedingung 6.5 ist eingebettet und mit • gekennzeichnet. Die neue Pfadbedingung ist deutlich größer als die Originalbedingung. Die Ursache liegt an den feingranularen SSA-Indizes, die zu einer notwendigen Vervielfältigung von Variablen und damit Bedingungen führen.

Erläuterungen zu Abbildung 6.6

Sei X als `e_puf[idx]>=65 and e_puf[idx]<=90` aus Zeile 25 definiert. M steht für folgendes Monom:

$$(\texttt{p_cd}_{12}[\texttt{0}_{54}]\ \texttt{\&}_{53}\ \texttt{1}_{57}) \neq_{52} \texttt{0}_{58} \quad \wedge \quad (\texttt{p_cd}_{12}[\texttt{0}_{79}]\ \texttt{\&}_{78}\ \texttt{16}_{82}) \neq_{77} \texttt{0}_{83}$$
$$\wedge \quad (\texttt{p_ab}_{8}[\texttt{0}_{27}]\ \texttt{\&}_{26}\ \texttt{16}_{30}) =_{25} \texttt{0}_{31} \quad \wedge \quad \texttt{e_puf}_{87}[\texttt{idx}_{86}] =_{85} \texttt{'+'}_{89}$$
$$\wedge \quad \texttt{idx}_{65} <_{64} \texttt{7}_{66}$$

Dann gilt:
$Var(X) = \{\texttt{e_puf}, \texttt{idx}\}$,
$Var(M) = \{\texttt{p_cd}_{12}, \texttt{p_ab}_{8}, \texttt{e_puf}_{87}, \texttt{idx}_{86}, \texttt{idx}_{65}\}$ und
$E_{\texttt{e_puf}} = \{\texttt{e_puf} \mapsto \texttt{e_puf}_{87}\}$, da innerhalb der XAssertion-Region $\texttt{e_puf}_{87}$ in der Pfadbedingung vorkommt.

Weiterhin ist $E_{\texttt{idx}} = \{\texttt{idx} \mapsto \texttt{idx}_{86}, \texttt{idx} \mapsto \texttt{idx}_{65}\}$, so dass sich die XAssertion-Bedingung $X \lhd E_{\texttt{e_puf}}, E_{\texttt{idx}}$ berechnet zu:

$$\texttt{e_puf}_{87}[\texttt{idx}_{86}] \geq 65 \;\wedge\; \texttt{e_puf}_{87}[\texttt{idx}_{86}] \leq 90$$
$$\texttt{e_puf}_{87}[\texttt{idx}_{65}] \geq 65 \;\wedge\; \texttt{e_puf}_{87}[\texttt{idx}_{65}] \leq 90$$

Daraus folgt mit $M = M^{OLD}$:

$$(\texttt{p_cd}_{12}[0_{54}]\ \&_{53}\ 1_{57}) \neq_{52} 0_{58} \;\wedge\; (\texttt{p_cd}_{12}[0_{79}]\ \&_{78}\ 16_{82}) \neq_{77} 0_{83}$$
$$\wedge\; (\texttt{p_ab}_{8}[0_{27}]\ \&_{26}\ 16_{30}) =_{25} 0_{31} \;\wedge\; \texttt{e_puf}_{87}[\texttt{idx}_{86}] =_{85} \texttt{'+'}_{89}$$
$$\wedge\; \texttt{idx}_{65} <_{64} 7_{66}$$
$$\wedge\; \texttt{e_puf}_{87}[\texttt{idx}_{86}] \geq 65 \;\wedge\; \texttt{e_puf}_{87}[\texttt{idx}_{86}] \leq 90$$
$$\wedge\; \texttt{e_puf}_{87}[\texttt{idx}_{65}] \geq 65 \;\wedge\; \texttt{e_puf}_{87}[\texttt{idx}_{65}] \leq 90$$

Führt man dies konsequent weiter, gelangt man zur Pfadbedingung in Abbildung 6.6. Die Manipulation im Quelltext unter Berücksichtigung der SSA-Indizes zeigt, dass ein Informationsfluss nur genau dann stattfinden kann, wenn $\texttt{e_puf}_{87}[\texttt{idx}_{86}]$ den Wert '+' annimmt ('+' ist in ANSI-C vom Typ „Char" mit Wert 43). Mit den Zusicherungen der XAssertions über das tatsächliche Verhalten der Software (und Hardware) findet nun trotz bestehender Manipulation kein illegaler Informationsfluss statt. Zu sehen ist dies am Widerspruch der Bedingungen in den Zeilen 4 und 17 (Abb. 6.6) und ebenso im zweiten Monom in den Zeilen 22 uns 29. Das bedeutet, dass beim Einsatz von einfacher Hardware, die mit XAssertions spezifiziert wurde, die Pfadbedingung *false* wird und es keinen illegalen Informationsfluss geben kann.

```
1  • (p_cd12[0_54] &_53 1_57) ≠_52 0_58
2  • (p_cd12[0_79] &_78 16_82) ≠_77 0_83
3  • (p_ab8[0_27] &_26 16_30) =_25 0_31
4  • e_puf87[idx86] =_85 '+'_89
5  • idx65 <_64 7_66
6    idx17 < 10
7    idx107 < 10
8    e_puf15[idx17]≥65 ∧ e_puf15[idx17]≤90
9    e_puf15[idx107]≥65 ∧ e_puf15[idx107]≤90
10   e_puf108[idx17]≥65 ∧ e_puf108[idx17]≤90
11   e_puf108[idx107]≥65 ∧ e_puf108[idx107]≤90
12   e_puf113[idx17]≥65 ∧ e_puf113[idx17]≤90
13   e_puf113[idx107]≥65 ∧ e_puf113[idx107]≤90
14   p_ab8[0]<5000
15   idx86 ≥ 0 ∧ idx86 < 7
16   idx65 ≥ 0 ∧ idx65 < 7
17   e_puf87[idx86]≥65 ∧ e_puf87[idx86]≤90
18   e_puf87[idx65]≥65 ∧ e_puf87[idx65]≤90
19 ∨
20 • (p_cd12[0_54] &_53 1_57) ≠_52 0_58
21 • (p_cd12[0_79] &_78 16_82) ≠_77 0_83
22 • e_puf96[idx95] =_94 '-'_98
23 • (p_ab8[0_27] &_26 16_30) =_25 0_31
24 • e_puf87[idx86] ≠_85 '+'_89
25 • idx65 <_64 7_66
26   → 6...14
27   idx95 ≥ 0 ∧ idx95 < 7
28   → 15...16
29   e_puf96[idx95]≥65 ∧ e_puf96[idx95]≤90
30   e_puf96[idx86]≥65 ∧ e_puf96[idx86]≤90
31   e_puf96[idx65]≥65 ∧ e_puf96[idx65]≤90
32   e_puf87[idx95]≥65 ∧ e_puf87[idx95]≤90
33   → 17...18
```

Abbildung 6.6: Der Klassiker mit XAssertions

```
1  #define TRUE 1
2  #define CTRL2 0
3  #define PB 0
4  #define PA 1
5  void printf();
6
7  /* [XASSERT] idx<10; e_puf[idx]>=65 and e_puf[idx]<=90; p_ab[0]<5000 */
8  void main()
9  {
10   int p_ab[2] = {0, 1};
11   int p_cd[1] = {0};
12   char e_puf[8];
13   int u;
14   int idx;
15   float u_kg;
16   float kal_kg = 1.0;
17
18   while(TRUE) {
19     if ((p_ab[CTRL2] & 0x10)==0) {
20       u = ((p_ab[PB] & 0x0f) << 8) + (unsigned int)p_ab[PA];
21       u_kg = (float) u * kal_kg;
22     }
23     if ((p_cd[CTRL2] & 0x01) != 0) {
24       /* [XASSERT] idx>=0 and idx<7;                   */
25       /*           e_puf[idx]>=65 and e_puf[idx]<=90; */
26       for (idx=0;idx<7;idx++) {
27         e_puf[idx] = (char)p_cd[PA];
28         if ((p_cd[CTRL2] & 0x10) != 0) {
29           if (e_puf[idx] == '+')
30             kal_kg *= 1.01;
31           else if (e_puf[idx] == '-')
32             kal_kg *= 0.99;
33         }
34       }
35       /* [END XASSERT] */
36       e_puf[idx] = '\0';
37     }
38     printf("Artikel: %7.7s\n   %6.2f kg    ",e_puf,u_kg);
39   }
40 }
41 /* [END XASSERT] */
```

Abbildung 6.7: Steuerungssoftware mit XAssertions angereichert

Kapitel 7

Pfadbedingungen für reale Programme

> *Analysis of factors determining path condition:* This chapter considers the factors that actually determine the condition of paths and, in doing so, provides reasons why paths in some areas are in better condition than in others [TCA].

7.1 Anforderungen realer Programme

Die Berechnung von Pfadbedingungen skaliert für reale Programme nicht, wenn ihre Realisierung strikt nach den formalen Grundlagen der vorherigen Kapitel (besonders Definition 4.8) umgesetzt wird, da ihre Definition vollständige Pfade zwischen Start- und Zielknoten voraussetzt. Selbst unter der bewiesenen Aussage, dass Zyklen entlang eines Pfades ignoriert werden können (Abschnitt 4.3.3), existieren in realen Programmabhängigkeitsgraphen exponentiell viele Pfade, so dass eine vollständige Pfadaufzählung zwischen zwei Knoten i.d.R. unmöglich ist. In diesem Kapitel werden Verfahren vorgeschlagen und empirisch untersucht, mit denen sich trotz der exponentiellen Komplexität Pfadbedingungen für reale Programme berechnen lassen.

Für die empirischen Untersuchungen werden sowohl die Programme aus der Testsuite von Barbara G. Ryder, als auch Steuerungssoftware mit echtem sicherheitskritischen Anspruch herangezogen. Die Programmauswahl basiert darauf, dass es sich bei den meisten Programmen um gängige Benchmark-Programme für ANSI-C-Analysen handelt und somit ein sehr breites Spektrum an Funktionalität und Programmierstilen abgedeckt wird. Hierdurch werden nicht nur reale Programme, sondern auch vielfältige PDG-Strukturen untersucht.

Programm	Knoten	Kanten	LOC	Funkt.	Aufrufe
mergesort	244	640	59	5	10
calculator	267	716	115	8	15
triple_des	5146	20353	1103	22	50
ctags	12958	55290	2933	100	300
assembler	16049	263053	3178	72	523
gnugo	7786	23762	3305	37	293
agrep	22820	79938	3968	87	424
wackeltisch	10590	30210	4563	67	497
flex	48098	738129	7640	119	621
patch	30774	246754	7998	163	856
bison	33158	156926	8313	157	906
larn	171904	1000811	10410	276	2416
moria	364986	1742234	25754	465	4861
palme	15709	46538	6457	131	726

Tabelle 7.1: Die Testsuite

Obwohl Pfadbedingungen in erster Linie zur Untersuchung sicherheitskritischer Software gedacht sind, die innerhalb geschlossener technische Systeme laufen und daher relativ klein sind, spiegeln die anderen Benchmarkprogramme das Verhalten von Pfadbedingungen in größeren Systemen wider, selbst mit intensiver Verwendung von Zeigern.

Die Tabelle 7.1 zeigt die untersuchten Programme und gibt ihre Größe in Zeilen (LOC) sowie definierte Funktionen und enthaltene Funktionsaufrufe an. Die Größe der entsprechenden Programmabhängigkeitsgraphen ist durch die Knotenzahl und Kantenzahl gegeben.

Die Struktur der obigen Programmabhängigkeitsgraphen ist *intraprozedural zyklisch*, da Schleifenkonstrukte wie `while`, `do` und `for` zyklische Datenabhängigkeiten einführen. Außerdem ist die Struktur *interprozedural zyklisch*, da jede Funktion genau einen PDG realisiert, der mit all seinen Aufrufstellen über Parameter- und Aufrufkanten verbunden ist. Dieses gemeinsame Nutzen (Sharing) von Funktions-PDG ist in der reinen PDG-Struktur kontext-insensitiv und führt daher strukturelle Zyklen ein, die erst durch kontext-sensitive Verfahren vermieden werden (siehe Kapitel 5).

Dieses Kapitel beschäftigt sich daher hauptsächlich mit einer effizienteren Berechnung von intraprozeduralen Zyklen. Die Gesamtkomplexität kann durch die notwendige Pfadaufzählung definitionsbedingt nicht besser als (stückweise) exponentiell sein. Jedoch wird mit einer Kombination von verschiedenen Verfahren dennoch das Ziel erreicht, für reale Programme Pfadbedingungen zu berechnen.

Im ersten Abschnitt wird analysiert, inwieweit die Verwendung von Binären

Programm	Knoten	Kanten	Funkt.	Aufrufe	SCC	maxKn	maxKa
mergesort1	69	156	2	4	10	26	53
mergesort2	99	227	3	7	14	26	53
calculator1	178	487	5	13	14	61	123
calculator2	66	169	2	5	8	13	25
triple_des1	897	3265	7	14	19	20	49
triple_des2	136	331	7	11	1	2	2
ctags1	4299	16762	53	118	254	515	2155
ctags2	2958	11994	45	99	176	515	2155
assembler1	6169	181470	34	103	42	27	56
assembler2	2704	54900	22	61	80	27	56
gnugo1	4649	14103	23	74	451	301	823
gnugo2	5500	16788	28	87	587	301	823
agrep1	11031	32897	39	85	5	98	203
agrep2	10954	32539	39	84	19	336	794
wackeltisch1	3466	9865	30	130	348	172	564
wackeltisch2	3413	9835	26	124	338	174	580
flex1	6032	45326	45	127	46	180	394
flex2	10508	249915	49	168	73	41	62
patch1	14000	128527	106	405	1203	1302	30834
patch2	13943	128271	106	405	1201	1310	30951
bison1	5742	51658	51	213	416	440	24154
bison2	5378	50043	47	206	375	440	24154
larn1	36382	222695	172	883	8	11	23
larn2	36451	222864	172	884	9	11	23
moria1	15376	59571	138	588	1476	1195	4077
moria2	16644	63194	143	621	794	371	639
palme1	508	1287	16	31	76	15	29
palme2	747	1915	21	42	98	15	29
palme3	818	2101	20	48	108	15	29
palme4	946	2417	25	64	130	15	29
palme5	1405	3590	31	94	180	43	76
palme6	973	2530	25	74	144	15	29

Tabelle 7.2: Untersuchte Chops der Testsuite

Entscheidungsgraphen einen Einfluss auf die Performance von Pfadbedingungen hat. Anschließend werden unterschiedliche Zyklenzerlegungsverfahren untersucht, mit denen eine Intervallanalyse zur Effizienzsteigerung realisiert wird. Der letzte Abschnitt untersucht die Auswirkungen einer heuristischen Pfadauswertungsbegrenzung mit unterschiedlichem Parameter k.

Aus den Ergebnissen kann anschließend auch ermittelt werden, in welchem Maße die Faktoren, wie Programmgröße, Größe und Struktur von PDGs sowie Größe der Chops an Performance-Gewinnen und -Verlusten beim Berechnen von Pfadbedingungen beteiligt sind.

Tabelle 7.2 zeigt 32 Chops, für die Pfadbedingungen berechnet wurden. Die Auswahl der Chops für die sicherheitsrelevanten Anwendungen (wackeltisch und palme) erfolgte nach intensivem Quelltextstudium. Kapitel 8 widmet sich diesen beiden Anwendungen im Detail. Alle anderen Chops wurden

nach folgenden Kriterien bestimmt:

- Start- und Zielknoten mussten inhaltlich sinnvoll sein. Hierfür wurde der Quelltext nach interessanten Punkten durchsucht, genauso, wie es ein Prüfingenieur machen würde.

- Die Pfadbedingung sollte nicht *true* oder *false* ergeben, da in diesem Fall nur die Aussagekraft von Program-Slicing erreicht und dadurch der Pfadbedingungsgenerator nur unzureichend ausgelastet wird.

- Chops sollten im Vergleich zur PDG-Größe nicht zu klein sein, um aussagekräftige Zahlen erheben zu können.

Die Tabelle zeigt die allgemeine Struktur der analysierten Chops anhand der Knotenzahl, Kantenzahl, Anzahl der enthaltenen Funktionen und Funktionsaufrufen. Der kleinste sinnvolle Chop hat 2,5% der Knotenzahl des zugehörigen PDG (triple des 2), der größte Chop enthält 70% des PDG (gnugo 2). Der Durchschnitt über alle Chops liegt bei 25,5% der PDG-Größe.

Die Kantenzahl wird in empirischen Studien zu Program-Slicing selten herangezogen, da die Verfahren i.Allg. linear sind. Die Berechnung von Pfadbedingungen hat exponentielle Komplexität, die in erster Linie aus der exponentiellen Anzahl an Pfaden innerhalb von Zyklen herrührt. Um die Struktur von Chops für Pfadbedingungen miteinander vergleichen zu können, ist es daher notwendig, die Anzahl von Kanten bei der Beurteilung von Laufzeiten zu berücksichtigen.

Neben der Größe des Graphen werden für die späteren Zerlegungsverfahren auch die Anzahl der starken Zusammenhangskomponenten (SCC, engl. strongly connected components) mit ihrer eigenen maximalen Knoten- und Kantenzahl dargestellt. Jede starke Zusammenhangskomponente repräsentiert einen intraprozeduralen Zyklus. Die Anzahl der starken Zusammenhangskomponenten schwankt sehr und ist ausschließlich von den Schleifenausdrücken im Programm abhängig.

Die Fallstudie „wackeltisch“ hat z.B. eine sehr große Anzahl an starken Zusammenhangskomponenten, ganz im Gegensatz zu Fallstudie „flex“, bei der im untersuchten Bereich kaum Zyklen liegen. „patch“ als nächstgrößeres Programm enthält von allen Fallstudien die meisten Zyklen mit der größten Anzahl an Knoten. Man sieht anhand der Knoten-Kanten-Ratio, wie stark die Struktur der Graphen und Zyklen innerhalb der Fallstudien variiert. Sie liegt häufig bei 1:2, steigt jedoch z.B. bei „bison“ auf bis zu 1:50 an.

```
 1  void looptest (int a[10]) {
 2    int max=0;
 3    int go=0;
 4    int i=0;
 5
 6    while (i<10 && go<5) {
 7      if (a[i] > max) {
 8        max = a[i];
 9        (max);
10        ++go;
11      }
12      ++i;
13    }
14  }
15
16  int main() {
17    int a[10];
18    looptest(a);
19    return 0;
20  }
```

Abbildung 7.1: Ein durchgehendes Beispiel („looptest“)

7.2 Binäre Entscheidungsgraphen

Pfadbedingungen stellen boolesche Funktionen dar. Die Verwaltung dieser Funktionen erfordert Datenstrukturen, die eine effiziente Repräsentation und Manipulation der Formeln ermöglichen. Aus der grundlegenden Formel (Definition 6.1)

$$PC(y,x) = \bigvee_{P_i : y \to^* x \in CH(y,x)} \Big(\bigwedge_{z \in P_i} E(z) \wedge \bigwedge_{v \to u \in P_i} \Phi(v \to u) \wedge \bigwedge_{v \to u \in P_i} \delta(v \to u) \Big)$$

geht hervor, dass Ausführungsbedingungen E für Knoten und Φ-, δ- sowie weitere Bedingungen für Kanten berechnet werden. Die Anzahl und Schachtelung der Prädikate im Programm entscheidet über die Größe der Ausführungsbedingungen. Da Prädikate i.d.R. größere Programmteile regieren, werden die Knoten von Teilpfaden größtenteils von denselben Ausführungsbedingungen regiert.

7.2.1 Redundante Terme

Das Beispiel in Abb. 7.1[1] (39 Knoten in `looptest`) enthält die beiden Prädikate $i < 10 \wedge go < 5$ in Zeile 6 und $a[i] > max$ in Zeile 7. Daraus ergibt sich,

[1] Der Programmabhängigkeitsgraph ist in Abbildung 7.7 auf Seite 90 abgebildet.

dass die Zeilen 8-10 (10 Knoten) vom Prädikat $a[i] > max$ regiert werden und die Zeilen 7-12 (20 Knoten) vom Prädikat $i < 10 \wedge go < 5$.

26% aller PDG-Knoten von `looptest` werden also von beiden Prädikaten (Zeile 6 und 7) regiert und 51% der Knoten nur vom Prädikat in Zeile 6. Insgesamt werden 59% der Knoten ausschließlich von einem der folgenden 3 nichttrivialen Ausführungsbedingungen regiert (ohne Indizes):

1. $i < 10$
2. $i < 10 \wedge go < 5$
3. $i < 10 \wedge go < 5 \wedge a[i] > max$

Die starke Redundanz in den Bedingungen ist offensichtlich und wirkt sich auch auf die Pfadbedingungen aus, bei der für jeden Knoten entlang eines Pfades E konjunktiv verknüpft wird. Die Redundanz in den Bedingungen wird durch leicht unterschiedliche Ausführungsbedingungen noch verstärkt und tritt dann auf, wenn der Pfad entlang von geschachtelten Prädikaten verläuft. Die Anwendung von Idempotenz- und Absorptionsgesetzen sind daher elementar wichtig.

Binäre Entscheidungsgraphen (engl. binary decision diagrams (BDD), [Bry86, DB98]) sind Datenstrukturen, die bei geschickter Anwendung (siehe Abschnitt 7.2.3) die obigen Forderungen erfüllen. BDDs haben sich bereits beim Model Checking bewährt, für das sie effizient Zustandsräume von Automaten darstellen können [HS96].

7.2.2 Pfadbedingungen und boolesche Algebra

In Pfadbedingungen entsprechen die einzelnen Prädikate booleschen Werten aus $\mathbb{B} = \{0, 1\}$, die durch Literale $x_1, \ldots, x_n$ repräsentiert werden können. Ein *Monom* ist eine konjunktive Verknüpfung von Literalen. Eine disjunktive Normalform ist eine disjunktive Verknüpfung von Monomen. Boolesche Funktionen $f : \mathbb{B}^n \rightarrow \mathbb{B}$ über einer Literalmenge $X_n = \{x_1, \ldots, x_n\}$ können nun in Binären Entscheidungsgraphen dargestellt werden, bei denen es sich um zusammenhängende azyklische Graphen handelt. Ein Blatt enthält entweder den Wert 0 (*false*) oder 1 (true) und keine weiteren abgehenden Kanten. Alle anderen Knoten repräsentieren ein $x_i \in X_n$ und besitzen zwei abgehende Kanten. Die *low*-Kante steht für die Negation von x_i ($\overline{x_i}$), die *high*-Kante steht für die Erfüllung von x_i. Die Auswertung beginnt bei der Wurzel des BDD und endet bei einem Blatt.

Da alle inneren Knoten der BDDs zwei abgehende Kanten haben, entsteht eine Baumstruktur (Binärer Entscheidungsbaum). Die relevante Eigenschaft von BDD ist es, isomorphe Untergraphen zu bestimmen und merfach zu

nutzten (Sharing). Diesen Vorgang, bei dem die Knotenzahl drastisch verringert wird, nennt man Reduktion. Aus einem reduzierten Entscheidungsbaum wird ein Entscheidungsgraph.

Binäre Entscheidungsgraphen sind ideal für die Speicherung von Pfadbedingungen, da sie die booleschen Operationen *und/oder* in polynomialer Zeit ermöglichen, *not* sowie die Tests auf *true* und *false* in konstanter Zeit.

In dieser Arbeit werden OBDD[2] (engl. ordered binary decision diagrams) eingesetzt, die eine Ordnung $x_1 < x_2 < \cdots < x_n$ auf den Literalen beschreiben. Jedes Literal tritt so auf jedem Pfad von der Wurzel bis zum Blatt höchstens einmal auf.

7.2.3 Transformation von Prädikaten zu BDD-Knoten

Die Sprache ANSI-C unterstützt zusammengesetzte Prädikate (*und/oder*), die in VALSOFT-PDGs feingranular repräsentiert werden. Um BDDs effizient nutzen zu können, werden diese zusammengesetzten γ-Bedingungen in ihre atomaren Bestandteile zerlegt, so dass keine Konjunktionen und Disjunktionen mehr enthalten sind. Jede atomare Kontrollabhängigkeitsbedingung wird durch eine eindeutige Id (eindeutige Nummer) identifiziert, die unabhängig vom aktuellen Kontext der Berechnung ist. Somit besteht ein Pool von Ids, der alle atomaren Prädikate aller γ-Bedingungen des Programms repräsentiert.

Beispiel 7.1 (Gemeinsame Nutzung von Ids)
Eine Ausführungsbedingung $a \wedge b \vee c$, deren Literale z.B. durch die eindeutigen Ids $a \mapsto 1, b \mapsto 2, c \mapsto 3$ repräsentiert sind, wird demnach in einem BDD als $1 \wedge 2 \vee 3$ dargestellt. In einem zweiten Prädikat $(a \wedge b \vee c) \wedge e$ ist unter Verwendung des Id-Pools nur z.B. $e \mapsto 4$ neu, so dass die zweite Ausführungsbedingung im BDD als $(1 \wedge 2 \vee 3) \wedge 4$ gespeichert werden kann. Da Pfadbedingungen aus der konjunktiven bzw. disjunktiven Verknüpfung von Ausführungsbedingungen bestehen, zeigt sich der Gewinn bei der Verknüpfung der beiden Ausführungsbedingungen. Im konjunktiven Fall wird nur $(1 \wedge 2 \vee 3) \wedge 4$ gespeichert, im disjunktiven Fall $1 \wedge 2 \vee 3$. Das Problem der leicht unterschiedlichen Prädikate entlang von Pfaden ist damit behoben.

Durch die starke gemeinsame Nutzung von Teilausdrücken gilt die Speicherung von Binären Entscheidungsgraphen als effizient. Jedoch hat die Ordnung der Literale einen entscheidenen Einfluss auf die Größe der BDD. Beim Pfadbedingungsgenerator wird die Ordnung allein durch die Graphtraversierung und dem verwendeten BDD-Paket bestimmt. Eine explizite Vorgabe

[2] Der Begriff BDD wird für geordnete Literalmengen verwendet.

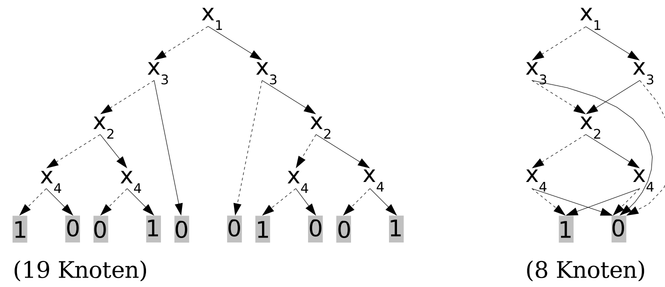


Abbildung 7.2: BDD mit Literalordnung $x_1 < x_3 < x_2 < x_4$

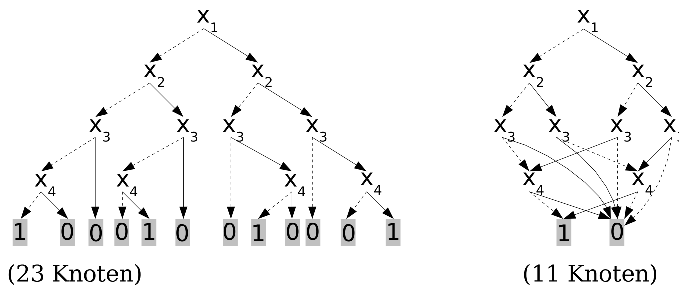


Abbildung 7.3: BDD mit Literalordnung $x_1 < x_2 < x_3 < x_4$

ist nicht sinnvoll, da sich nicht abschätzen lässt, in welcher Reihenfolge welche Prädikate im PDG traversiert werden und wie die Auswirkung auf die Größe des BDD ist.

Beispiel 7.2 (Einfluss der Literalordnung)
Dieses Beispiel zeigt den Einfluss der Literalordnung auf die BDD-Größe und den Speicherplatzgewinn durch die gemeinsame Nutzung von Teilausdrücken. Die Bedingung $x_1 \Leftrightarrow x_3 \wedge x_2 \Leftrightarrow x_4 \equiv (\neg x_1 \vee x_3) \wedge (\neg x_3 \vee x_1) \wedge (\neg x_2 \vee x_4) \wedge (\neg x_4 \vee x_2)$ ist als BDD in den Abbildungen 7.2 und 7.3 dargestellt. Die jeweils linke Seite präsentiert den Entscheidungsbaum und die rechte Seite den reduzierten Entscheidungsgraphen. Die Wahrheitstafel zur Auswertung nach *true* ist:

x_1	x_2	x_3	x_4	*true*
false	*false*	*false*	*false*	✔
false	*false*	*true*	*true*	✔
true	*true*	*false*	*false*	✔
true	*true*	*true*	*true*	✔

Der Einfluss der Literalordnung ist erheblich. Allein in dieser kleinen Bedingung kann durch eine günstige Wahl der Ordnung die Knotenzahl von 11 auf 8 Knoten reduziert werden. Wenn das verwendete BDD-Paket ein automatisches Umsortieren der Ids zur Verfügung stellt, führt dies i.d.R. zu kleineren BDDs, so dass sich die elementaren Pfadbedingungsoperationen (*und*/*oder*) schneller durchführen lassen.

Der Pfadbedingungsgenerator setzt das BDD-Paket BuDDy [LN03] ein. Auf dem Markt befinden sich weitere freie BDD-Pakete wie z.B. [Som04, EC94, Ran98].

7.2.4 BDD vs. DNF

Neben der Disjunktiven Normalform als Ergebnis der Pfadbedingungsberechnung wird zusätzlich der Binäre Entscheidungsgraph mit einer Reportdatei ausgegeben, die die verwendeten BDD-Knotennummern auf die echten Literale der Pfadbedingung abbildet. Die BDDs liegen im „dot“-Format [LR04] vor, so dass diese Schnittstelle zur automatischen Weiterverarbeitung geeignet ist. Die textuelle Repräsentation von BDDs ist für Pfadbedingungen i.d.R. sehr klein.

Die Pfadbedingung des „looptest“-Beispiels aus Abb. 7.1 zwischen a in Zeile 18 und go in Zeile 10 berechnet sich zu:

```
(go:i@20 <:i@19 5:i@21)
(a:p@47[i:i@26]:i >:i@25 max:i@29)
(i:i@17 <:i@16 10:i@18)
a:p@5 := a:p@47; (looptest, looptest.c)
```

Durch den Einsatz von BDDs wurden die Redundanzen in den leicht unterschiedlichen Ausführungsbedingungen auf den Pfaden entfernt. Übrig bleibt eine einzige konjunktiv verknüpfte Formel, die alle 3 atomaren Prädikate enthält. Die letzte Definition weist dem formalen Parameter aus `looptest` den aktuellen Parameter aus `main` zu. Da Variable a eindeutig ist, wurde sie in der 2. Bedingung automatisch durch den aktuellen Parameter ersetzt. Die Reportdatei zum BDD liefert:

```
BDD REPORT

1:        (go:i@20 <:i@19 5:i@21)
2:        (i:i@17 <:i@16 10:i@18)
3:        (a:p@47[i:i@26]:i >:i@25 max:i@29)
4:        a:p@5 := a:p@47; (looptest, looptest.c)
```

Die Abb. 7.4 zeigt den Binären Entscheidungsgraphen zu „looptest“. Der geordnete BDD enthält keine Redundanzen mehr und verwendet genau die vier Ids der Reportdatei.

Zum Zweck der direkten Interpretation von Pfadbedingungen sind BDDs nicht geeignet, da sie als Auswertungsbaum die Erfüllung bzw. Nicht-Erfüllung für jedes Literal angeben. Pfade im BDD können somit Literale enthalten, die nicht den Literalen entlang von Pfaden im PDG entsprechen,

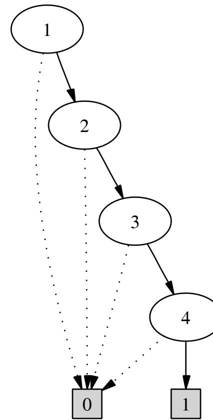


Abbildung 7.4: Der BDD zum Programm „looptest“

da die gesamte Pfadbedingung für alle PDG-Pfade innerhalb eines einzigen Entscheidungsgraphen gespeichert wird.

Um dieses Problem zu lösen, wird die Disjunktive Normalform aus dem BDD generiert. Die Interpretation der Pfadbedingung wird dadurch für den Anwender möglich. Jeder konjunktiv verknüpfte Block (Monom) steht i.d.R. für eine Menge von Pfaden, die von derselben Bedingung regiert wird. Wenn kein Monom erfüllbar ist, findet kein Informationsfluss zwischen dem Start- und Endknoten der Pfadbedingung statt. Die Ausgabe in DNF dient zugleich als Eingabe für angeschlossene Constraint-Solver, um Wertebelegungen für Pfadbedingungen zu ermitteln, die sie erfüllbar machen.

Der Nachteil der DNF ist ihre möglicherweise exponentielle Größe, die trotz der Minimierung nach Quine/McCluskey und der Berechnung von Minimalpolynomen auftreten kann.

7.2.5 Empirische Untersuchung

Der Einsatz von Binären Entscheidungsgraphen reicht alleine nicht aus, um Pfadbedingungen effizient berechnen zu können, da bei einer Pfadaufzählung mit zyklischen Programmabhängigkeitsgraphen exponentiell viele Pfade zu untersuchen sind. Daher wird der Einsatz von BDDs im Zusammenhang mit einer Behandlung von Zyklen in Abschnitt 7.4.3 auf Seite 89 untersucht.

Als vorweggenommenes Fazit hat sich gezeigt, dass für Pfadbedingungen der Einsatz von Binären Entscheidungsgraphen dazu führt, dass während der Pfadaufzählung – unabhängig von Zerlegungs- und Vereinfachungsstrategien – das exponentielle „Aufblähen“ von Pfadbedingungen verhindert wird. Die Untersuchungen zeigen, dass der Einsatz von Binären Entscheidungsgraphen

ein *Schlüssel* zum effizienten Berechnen von Pfadbedingungen ist.

7.3 Dominatoren

In der ursprünglichen Arbeit von [Sne96] wurde vorgeschlagen, gemeinsame Teilpfade herauszufaktorisieren. Dieser Faktorisierungsvorschlag lässt jedoch die Frage der Realisierung offen. Die in [Sne96] dargelegten Vereinfachungsmöglichkeiten betreffen nur strukturierten Kontrollfluss und werden daher in dieser Arbeit nicht weiter betrachtet, da sie der Struktur von Programmabhängigkeitsgraphen nicht entsprechen.

Zwei Faktorisierungsansätze für Programmabhängigkeitsgraphen werden hier tiefergehend untersucht. Der erste ermittelt als Faktorgrenzen längste gemeinsame Teilpfade, und der zweite Ansatz ermittelt Dominatoren:

- Teilpfade können mittels LCS-Algorithmen [CLR01] (engl. longest common subsequence) berechnet werden. Da die Anwendung von LCS auf zwei Sequenzen beruht, nutzt dies bei l Pfaden i.d.R. nichts. Die Komplexität von LCS ist $O(mn)$ mit Sequenzlänge $|s_1| = m$ und $|s_2| = n$. Um eine maximale gemeinsame Sequenz auf allen Pfaden zwischen y und x zu bestimmen, würde die Komplexität daher auf $O(\binom{l}{2}mn)$ wachsen. Bei einer teilweise exponentiellen Pfadanzahl ist dies nicht praktikabel.

- Dominatoren liefern natürliche Knoten im PDG, an denen Pfade, ausgehend von einem Startknoten, wieder zusammenlaufen. Der Dominator stellt damit einen Startknoten für einen gemeinsamen Teilpfad zum Zielknoten dar. Es gibt intraprozedurale und interprozedurale Dominatoren (siehe Abb. 7.5).

Analog zum Abschnitt 2.2 werden Dominatoren nun für Programmabhängigkeitsgraphen in einer intuitiven Form definiert und anschließend gezeigt, wie Pfadbedingungen Dominatoren nutzen können.

Definition 7.1 (Dominator)
Ein Knoten p stellt einen *Dominator* für x dar, wenn für jeden Pfad $y \to^* x$ im PDG gilt $y \to^* p \to^* x$ [LT79]. Im Falle eines Kontrollflussgraphen steht y für den *START*-Knoten.

Definition 7.2 (Postdominator)
Ein Knoten p stellt einen *Postdominator* für y dar, wenn für jeden Pfad $y \to^* x$ im PDG gilt $y \to^* p \to^* x$ [LT79]. Im Falle eines Kontrollflussgraphen steht x für den *END*-Knoten.

In einem Chop $CH(y, x)$ stellen die Dominatoren von x und die Postdominatoren von y per Definition dieselbe Knotenmenge dar, da y und x Start- und Zielpunkt der Pfade sind.

Definition 7.3 (Pfadbedingung für einen Dominator)
Sei z ein Dominator für $x \in CH(y, x)$. Dann ist

$$PC(y, x) \equiv PC(y, z) \wedge PC(z, x)$$

eine *Pfadbedingung für einen Dominator z.*

Im Falle von mehreren Dominatoren $z_1, \ldots, z_n$ verhält sich die Pfadbedingungsberechnung äquivalent zu Def. 7.3, da die Dominatoren $z_1, \ldots, z_n$ für $y \rightarrow^* x$ definitionsgemäß in Reihe geordnet sind und die Dominatoren $z_1, \ldots, z_{n-1}$ ebenso Dominatoren für z_n sind.

Definition 7.4 (Pfadbedingung für mehrere Dominatoren)
Seien $z_1, \ldots, z_n$ Dominatoren für $x \in CH(y, x)$. Dann ist

$$PC(y, x) \equiv PC(y, z_1) \wedge \bigwedge_{i=1}^{n-1} PC(z_i, z_{i+1}) \wedge PC(z_n, x)$$

eine *Pfadbedingung für mehrere Dominatoren $z_1, \ldots, z_n$.*

Knoten können nicht nur von *einzelnen* Knoten dominiert werden, sondern auch von Knotenmengen. In diesem Fall spricht man von einer *Dominatorfront $z^1, \ldots, z^n$*, wobei die Knoten der Menge $\{z^1, \ldots, z^n\}$ nicht in einer Ordnung zueinander stehen müssen.

Definition 7.5 (Pfadbedingung für eine Dominatorfront)
Sei $z^1, \ldots, z^n$ eine Dominatorfront für $x \in CH(y, x)$, so dass jeder Pfad $y \rightarrow^* x$ mindestens ein z^i enthält. Dann ist

$$PC(y, x) \equiv \bigvee_{i=1}^{n} PC(y, z^i) \wedge PC(z^i, x)$$

eine *Pfadbedingung für eine Dominatorfront $z^1, \ldots, z^n$.*

7.3.1 Bewertung

Die Berechnung von längsten gemeinsamen Teilpfaden scheidet aufgrund der hohen Komplexität aus, da die Vorbedingung von LCS die Aufzählung und Speicherung aller Pfade ist. Eine exponentielle Anzahl an Pfaden und deren paarweises Vergleichen macht das Verfahren nicht praktikabel.

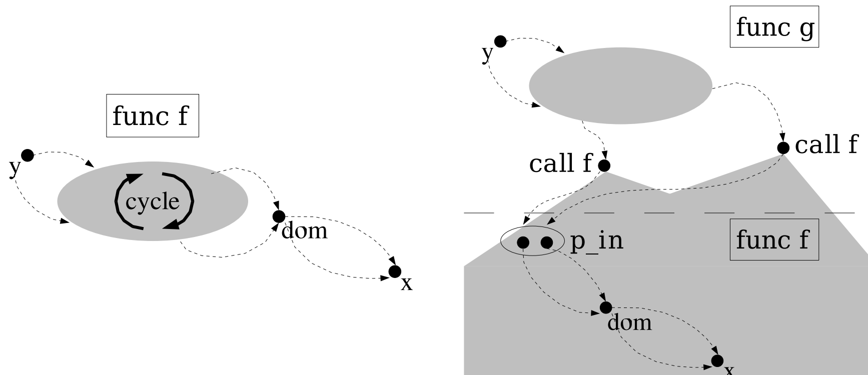


Abbildung 7.5: Mögliche Positionen für intraprozedurale Dominatoren (links) und interprozedurale Dominatoren (rechts)

Performance-Gewinne können dadurch erreicht werden, dass die vollständige Pfadaufzählung vermieden wird. Dominatoren sind praktikabel, liefern jedoch zu wenig Faktorisierungspunkte in Programmabhängigkeitsgraphen. Die Abbildung 7.5 zeigt mögliche Positionen für intra- und interprozedurale Dominatoren.

Intraprozedurale Dominatoren für einen ausgewählten Endknoten x haben den Nachteil, dass sie nicht innerhalb von Zyklen auftreten. Die VALSOFT-Programmabhängigkeitsgraphen enthalten viele Zyklen (siehe Tabelle 7.2 auf Seite 75). Daher wird die Anzahl möglicher Dominatoren stark reduziert. Interprozedurale Dominatoren, die Programmpunkte über Funktionsaufrufe hinweg dominieren, leiden an der gemeinsamen Nutzung von Funktions-PDGs. Die Abbildung zeigt einen Dominator, der erst innerhalb der x enthaltenen Funktion existiert, da durch mehrfache Funktionsaufrufe der Funktion f, die Pfade erst spät wieder zusammenlaufen.

Dominatoren sind nach obigen Definitionen auch für Teilchops definiert, die stellvertretend für eine eingeschränkte Menge von Pfaden $y_i \rightarrow^* x_i$ sind ($CH(y_i, x) \subseteq CH(y, x)$). Zur Untersuchung des Nutzens von Dominatoren wurde ein rekursives Verfahren realisiert, das stückweise Dominatoren für Teilchops $CH(y_i, x)$ entlang der regulären Pfadaufzählung berechnet und nutzt. Der Triple-DES-Algorithmus[3] enthielt für die gewählten Endknoten der Chops im interprozeduralen Fall genau einen Hauptdominator auf den Pfaden $y \rightarrow^* x$, so dass der exponentielle Aufwand zum Berechnen der Pfadbedingung nur minimal reduziert wurde. Die Berechnungszeit der Pfadbedingung lag bei 15 Sekunden (im Vergleich zu <1s mit einer Zyklenbehandlung, die im nächsten Kapitel vorgestellt wird).

Die Dominatorberechnung hat sich als nicht ergiebig erwiesen, da Domi-

[3] Zum Zeitpunkt der Berechnung lagen keine größeren PDGs vor.

natoren für $CH(y, x)$ nur ausserhalb von Zyklen liegen können. Die Pfadbedingungsberechnung im azyklischen Teil eines Graphen lässt sich jedoch mittels dynamischer Programmierung immer linearisieren, da einmal berechnete Teilpfadbedingungen wiederverwendet werden können. Die exponentielle Berechnungskomplexität bleibt damit nur im zyklischen Teil des Graphen vorhanden und erfordert daher eine entsprechende Erkennung und Behandlung.

7.4 Zyklen

Dominatoren haben sich als unwirksam erwiesen. Jedoch kann der Ansatz bestehen bleiben, wenn statt Dominatoren nun komplette Zyklen Faktorisierungspunkte darstellen, an denen gemeinsame Teilpfade beginnen. Dies erfordert eine Erweiterung der Definitionen des letzten Abschnitts um die Behandlung von Ein- und Austrittsknoten der Zyklen.

Wie oben beschrieben entsteht durch die Faltung von Zyklen ein zyklenfreier Graph, der sich topologisch sortieren lässt. Dies wiederum lässt die Berechnung von Pfadbedingungen durch dynamische Programmierung in $O(n)$ im azyklischen Bereich zu. Die exponentielle Pfadaufzählung findet also nur noch in zyklischen Bereichen statt.

Es gibt zwei Arten von Zyklen:

- *Reduzible* Zyklen enthalten genau einen Zykleneintritts- und beliebig viele Austrittsknoten.
- *Irreduzible* Zyklen enthalten beliebig viele Zykleneintritts- und Austrittsknoten.

Analog der Definition 7.3 können nun Pfadbedingungen für Zyklen definiert werden.

Definition 7.6 (Pfadbedingung für Zyklen)
Sei $C \subset CH(y, x)$ ein Zyklus mit Eintrittsknoten $in_1, \ldots, in_m \in CH(y, x)$ und Austrittsknoten $out_1, \ldots, out_n \in CH(y, x)$[4]. Auf jedem Pfad $y \to^* x$ im PDG gilt $y \to^* C \to^* x$. Dann ist

$$PC(y, x) = \bigvee_{i=1}^{m} \Big(PC(y, in_i) \wedge \bigvee_{j=1}^{n} \big(PC(in_i, out_j) \wedge PC(out_j, x) \big) \Big)$$

eine *Pfadbedingung für Zyklen.*

[4]Eintritts- und Austrittsknoten müssen nicht disjunkt sein.

Zyklen implizieren hierbei, ebenso wie Chops, alle enthaltenen Kanten, so dass der $\in$-Operator sowohl für Knoten als auch für Kanten verwendet wird. Die obige Definition bzw. ihre Realisierung kann erheblich verbessert werden, indem Pfade, deren Pfadbedingungen aufgrund von Absorbtionsregeln redundant sind, nicht unnötig betreten werden. Dies erreicht man, indem Pfade von y zu den Zykleneintrittsknoten den Zyklus nicht selbst durchlaufen.

Definition 7.7 (Pfadbedingung für Zyklen (effizienter))
Es gelte Def. 7.6. Mit

$$(y \to^* in_i) \cap C \setminus \{in_1, \ldots, in_m\} = \emptyset \text{ und } (out_j \to^* x) \cap C \setminus \{out_1, \ldots, out_n\} = \emptyset$$

ist $PC(y, x)$ *effizienter*, da redundante Pfadbedingungen nicht berechnet werden. Es existieren keine Pfade mehr der Form $in_{1,\ldots,i-1,i+1,m} \in y \to^* in_i$ und $out_{1,\ldots,j-1,j+1,n} \in out_j \to^* x$.

Begründung: $PC(y, in_i)$ enthält nicht nur Pfade der Form $y \to^* in_i$ mit $(y \to^* in_i) \cap C \setminus \{in_1, \ldots, in_m\} = \emptyset$. Legale Pfade sind auch $y \to^* in_k \to^* in_i$, wobei der Pfad $in_k \to^* in_i$ innerhalb von C liegt. Nach Def. 7.6 wird $PC(y \to^* in_k \to^* in_i) \wedge PC(in_i \to^* out_j)$ berechnet. Diese Pfadbedingung ist jedoch bereits Bestandteil von $PC(y \to^* in_k) \wedge PC(in_k \to^* out_j)$.

7.4.1 Lineare Pfadbedingungen im azyklischen PDG

Durch die Zyklenberechnung wird der PDG in zyklische und azyklische Bereiche separiert. Für die zyklischen Bereiche gilt Definition 7.7, wobei $PC(in_i, out_j)$ exponentiell bleibt. Die beiden Restpfadbedingungen $PC(y, in_i)$ und $PC(out_j, x)$ können weitere Zyklen überdecken, für die wiederum Def. 7.7 gilt. Für den azyklischen Bereich des PDG lassen sich jedoch $PC(y, in_i)$ und $PC(out_j, x)$ in linearer Zeit durch dynamische Programmierung berechnen. Da der azyklische Bereich topologisch sortierbar ist, muss jede Kante maximal einmal durchlaufen werden.

Hierfür lässt sich die Definition für Dominatorfronten 7.5 adaptieren, indem sich eine Front aus den direkten Vorgängerknoten eines Knotens zusammensetzt. Entsprechendes gilt auch für Nachfolgerknoten.

Definition 7.8 (Lineare Pfadbedingung für Vorgänger)
Sei $CH^{\otimes}(y, x)$ der azyklische Bereich von $CH(y, x)$. Sei *pred(x)* ein Prädikat, das die Menge der direkten Vorgängerknoten von $x \in CH^{\otimes}(y, x)$ liefert, dann gilt

$$PC(y, x) \equiv \bigvee_{z \in pred(x) \in CH^{\otimes}(y,x)} PC(y, z) \wedge PC(z, x)$$

wobei durch eine topologische Sortierung jedes $PC(z,x)$ genau einmal berechnet werden muss und zugleich die Pfadbedingung für genau eine Kante darstellt.

Definition 7.9 (Lineare Pfadbedingung für Nachfolger)
Analog zu Def. 7.8 gilt wegen der Symmetrie für *Nachfolger* (succ) von $y \in CH^{\otimes}(y,x)$

$$PC(y,x) \equiv \bigvee_{z \in succ(y) \in CH^{\otimes}(y,x)} PC(y,z) \wedge PC(z,x)$$

wobei durch eine topologische Sortierung jedes $PC(y,z)$ genau einmal berechnet werden muss und zugleich die Pfadbedingung für genau eine Kante darstellt.

7.4.2 Intraprozedurale oder interprozedurale Zyklen?

Der Unterschied zwischen intraprozeduralen und interprozeduralen Zyklen ist die Berücksichtigung von Parametereingangs- und -ausgangskanten, sowie Funktionsaufrufkanten. Intraprozedurale Zyklen enthalten nur Daten- und Kontrollabhängigkeitskanten, sowie Summarykanten.

Der Abschnitt 5.2 hat sich schon mit realisierbaren Pfaden beschäftigt, die auch Zyklen betreffen. Algorithmen zum Berechnen von Zyklen basieren auf Graphen und beachten im universellen Fall die Bedeutung der Kantensemantik nicht. Bei Programmabhängigkeitsgraphen spielt diese jedoch eine bedeutende Rolle, da jeder Funktions-PDG im Graphen genau einmal vorkommt und von all seinen Aufrufstellen gemeinsam genutzt wird. Diese gemeinsame Nutzung erfolgt durch Parameterkanten und Funktionsaufrufkanten, wobei nur Parameterkanten je Aufrufkontext in beide Richtungen zeigen, wie in Abbildung 7.6 dargestellt wird.

Die Definition von Zyklen innerhalb eines Graphen sagt aus, dass verschiedene Zyklen disjunkt sein müssen, sonst würden sie zu einem einzigen Zyklus verschmolzen. Abbildung 7.6 zeigt hierzu zwei Funktionsaufrufe (`call h`) und den Funktions-PDG (`function h`). Jeder Funktionsaufruf findet in einer anderen Funktion statt. O.B.d.A. kann angenommen werden, dass jede der dargestellten 3 Funktionen intraprozedurale Zyklen besitzt. Ein interprozeduraler Zyklus könnte nun in `f` von `call h` über die aktuellen Parameter `act` zu `form` führen und über `body` von `h` zurück zum Aufrufkontext von `call h` in `f`. Dieser Zyklus würde einem realisierbaren Pfad entsprechen. Nach der Zyklendefinition müsste der Zyklus von `call h` in `f` nun vollständig mit der Funktion `h` verschmolzen werden, da keine disjunkten Bereiche vorliegen. Mit derselben Argumentation folgt jedoch auch ein Verschmelzen mit `call h` in `g`, so dass als Resultat alle drei Funktionen verschmolzen wären. In der

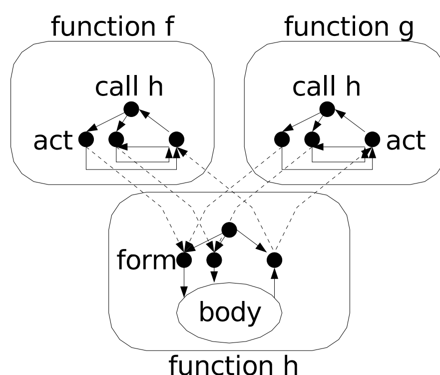


Abbildung 7.6: Sharing von Funktionen

Praxis bleibt bei Programmabhängigkeitsgraphen *ein* großer Zyklus übrig, und die Möglichkeit der linearen Pfadbedingungsberechnung geht verloren.

Mit der gleichen Begründung ist auch die im nächsten Abschnitt vorgestellte hierarchische Zyklenzerlegung interprozedural nicht sinnvoll, da man hierdurch eine Möglichkeit schaffen würden, dass beide Aufrufkontexte `call h` in `f` und `g` mit einem Teil aus `h` einen eigenen geschachtelten Zyklus bilden könnten. Dies könnte Pfade zwischen beliebigen Kontexten und Regionen des PDG schaffen, die dem realen Informationsfluss nicht mehr entsprechen würden.

Als Konsequenz werden Zyklen immer intraprozedural berechnet. Durch den Einsatz von Tiefen-Chops werden lokale Zyklen nicht mehr mit Zyklen in aufrufenden Funktionen verschmolzen, so dass nur noch legale Zyklen und realisierbare Pfade entstehen können.

Die Abbildung 7.7 zeigt den irreduziblen Zyklus in der Funktion `looptest` für den Chop zwischen *start* und *end*. Der Zyklus enthält 25 Knoten und damit 78% der Knoten von `looptest`. Die Knoten 27 und 33 stellen Zykleneintrittsknoten dar, die durch Pfade vom formalen Parametereingangsknoten 5 erreichbar sind. Verlassen wird der Zyklus an den beiden Knoten 37 und 38, die direkt zum Zielknoten der Pfadbedingung führen. Aufgrund von Idempotenz- und Absorptionsgesetzen ist der Zielknoten der Pfadbedingung nicht Bestandteil von Zyklen.

7.4.3 Empirische Untersuchung

Die Tabelle 7.3 zeigt Berechnungszeiten und Speicherplatzverbrauch für die in Tabelle 7.2 auf Seite 75 dargestellten Chops. Die erste Ergebnisspalte enthält die Daten für eine Berechnung ohne Verwendung von Binären

Abbildung 7.7: Programmabhängigkeitsgraph und Zyklus zu „looptest“

Entscheidungsgraphen, für die zweite Ergebnisspalte wurden BDDs eingesetzt. Ein „-“-Symbol sagt aus, dass sich die Ergebnisse nicht innerhalb von zwei Stunden berechnen ließen. Für die praktische Anwendbarkeit ist dieser Schwellwert angemessen, da die Berechnung von Pfadbedingungen ein halb-interaktiver Prozess ist, wenn z.B. für die Zulassungsprüfung Herstellerangaben überprüft werden sollen.

Es wurde darauf verzichtet, Performanceangaben zu Pfadbedingungen *ohne* BDDs und *ohne* die Berechnung von Zyklen zu machen, da sich Programme,

	−BDD +SCC		+BDD +SCC	
Programm	Zeit (s)	Speicher (MB)	Zeit (s)	Speicher (MB)
mergesort1	0	0,9	0	0,9
mergesort2	0	1	0	1,1
calculator1	227	208,7	0	1,3
calculator2	0	0,8	0	0,8
triple_des1	0	4,6	0	4,7
triple_des2	0	3,8	1	3,9
ctags1	-	-	-	-
ctags2	-	-	-	-
assembler1	-	-	6	22,3
assembler2	-	-	4	15,9
gnugo1	-	-	-	-
gnugo2	-	-	-	-
agrep1	7	20,8	0	21,2
agrep2	-	-	-	-
wackeltisch1	-	-	1673	12,1
wackeltisch2	-	-	164	11,0
flex1	-	-	-	-
flex2	-	-	19	41,6
patch1	-	-	-	-
patch2	-	-	-	-
bison1	26	26,2	4	20,7
bison2	-	-	-	-
larn1	208	79,3	4	80,5
larn2	207	79,4	4	80,6
moria1	-	-	-	-
moria2	86	99,9	30	104,9
palme1	-	-	1	7,1
palme2	-	-	1	7,9
palme3	-	-	5	17,2
palme4	-	-	262	159,8
palme5	-	-	23	32,3
palme6	-	-	203	100,8

Tabelle 7.3: Performancevergleich mit/ohne BDD und unter Berücksichtigung von Zyklen (Starke Zusammenhangskomponenten)

die größer als „triple_des“ waren, nicht berechnen ließen. Die Berechnung von Zyklen ist daher eine Vorbedingung für eine effiziente Berechnung von Pfadbedingungen.

Die Tabelle zeigt, dass sich von den 32 Pfadbedingungen nur 34% ohne den Einsatz von BDDs berechnen ließen. Im Gegensatz dazu war es mit der Zyklenbehandlung möglich, 69% der Pfadbedingungen zu berechnen. Auffällig an den Zahlen ist, dass es keine offensichtliche Korrelation zwischen der Größe der Chops und den Zeiten mit und ohne BDDs gibt. Dies zeigt, dass bei den berechenbaren Pfadbedingungen nicht nur die Chopgröße alleine ausschlaggebend ist, sondern auch die Anzahl an enthaltenen Prädikaten. Die Reihenfolge, in der die Programmabhängigkeitsgraphen bei der Pfadbedin-

gungsberechnung traversiert werden, ist für die Reihenfolge der Prädikate und damit auch für die Größe der BDDs relevant (siehe Abschnitt 7.2.3) und damit auch für den Aufwand bei der anschließenden Minimierung und Ausgabe in Disjunktiver Normalform. Beim Pfadbedingungsgenerator wird diese Reihenfolge jedoch nicht vorgegeben, sondern ist von der internen Datenstruktur des Programmabhängigkeitsgraphen bzw. der Reihenfolge der Kanten (eines Knotens) im Graphen abhängig.

Bei dem kleinen Testfall *calculator*1 sind die 227 Sekunden Berechnungszeit interessant, die sich durch den hohen Aufwand ergeben, wenn Formeln durch eine Baumstruktur verwaltet werden und durch die Transformation in DNF zur Minimierung nach Quine/McCluskey exponentiell aufblähen.

Mit dem Einsatz von BDDs sind die Ausführungszeiten bis auf „wackeltisch1“ sehr gering, selbst bei großen Programmen. In den meisten Fällen liegt die Zeit bei wenigen Sekunden. Diese Untersuchung belegt, dass der Einsatz von Binären Entscheidungsgraphen eine Grundvoraussetzung zur Berechnung von Pfadbedingungen ist. Allerdings reicht die Behandlung von Zyklen nicht aus, um alle Pfadbedingungen berechnen zu können, da das Verfahren für 31% der Testfälle nicht anwendbar ist.

Die Ursache liegt in der exponentiellen Berechnungskomplexität innerhalb der Zyklen. Große Zyklen führen dazu, dass die Pfadaufzählung innerhalb des vorgegebenen Zeitrahmens nicht mehr möglich ist. Im nächsten Abschnitt wird daher eine Erweiterung der Zyklenberechnung untersucht, die noch bessere Ergebnisse liefern wird. Die Daten zur Größe und Art der Pfadbedingungen aller Chops werden zudem ausführlich dargestellt.

7.5 Intervallanalyse

Der letzte Abschnitt zeigte, dass die Faltung von Zyklen nicht ausreicht, um Pfadbedingungen effizient berechnen zu können. Der azyklische Bereich des Programmabhängigkeitsgraphen lässt sich in linearer Zeit berechnen. Realisiert wird dies durch konsequente Wiederverwendung einmal berechneter Pfadbedingungen (Caching) nach Abschnitt 7.4.1. Im zyklischen Bereich ist der Einsatz von Caching nicht gestattet, da per Definition eines Zyklus die topologische Sortierung der Kanten nicht möglich ist.

Das Ziel dieses Abschnitts ist es, ein Verfahren vorzustellen, das Caching innerhalb von Zyklen ermöglicht, indem die Zyklen an fest definierten Punkten zerlegt werden.

Die Nutzung einer Zyklenzerlegung bei der Pfadbedingungsberechnung bezeichnet man als *Intervallanalyse*. Dabei werden die Starken Zusammenhangskomponenten aus Abschnitt 7.4 in eine Hierarchie von ineinander ge-

schachtelten reduziblen und irreduziblen Zyklen (engl. nested loop forest) zerlegt und mittels Definition 7.7 die Pfadbedingungen nach der Divide-and-Conquer-Methode berechnet.

Starke Zusammenhangskomponenten haben selbst fest definierte Punkte, und Pfadbedingungen zwischen diesen Punkten können (und werden) ebenso wiederverwendet. Diese Punkte sind die Ein- und Austrittsknoten der Zyklen. Jedes $PC(in_i, out_j)$ aus Definition 7.7 lässt sich mit exponentieller Komplexität berechnen und wiederverwenden.

Beispiel 7.3 (Caching in Starken Zusammenhangskomponenten)
Sei C ein Zyklus mit den Zykleneintrittsknoten i_1 und i_2 und einem Zyklenaustrittsknoten o. Im Zyklus existiert ein Pfad $i_1 \rightarrow^* i_2$, sowie die Pfade $i_1 \rightarrow^* o$ und $i_2 \rightarrow^* o$. Während der Pfadaufzählung im azyklischen Bereich wird C an i_2 betreten. Die resultierende Pfadbedingung *über den gesamten Zyklus* ist $PC(i_2, o)$. Wenn zu einem späteren Zeitpunkt vom azyklischen Bereich $i1$ erreicht wird, erfolgt die Berechnung von $PC(i_1, o)$. Möglicherweise wird hierbei der Pfad $i_1 \rightarrow^* i_2$ durchlaufen, an dessen Endpunkt $PC(i_2, o)$ eingesetzt werden kann, da diese Bedingung ohne Historie, also über den gesamten Zyklus, gültig ist.

Dieses Beispiel zeigt, wie Caching in Starken Zusammenhangskomponenten funktioniert. Jedoch hat der letzte Abschnitt demonstriert, dass diese Methode nicht ausreichend für eine effiziente Berechnung ist, da die Zyklen noch zu groß sind. Das Ziel ist die Anwendung des Verfahrens auf eine Hierarchie von geschachtelten Zyklen, um deutlich mehr Cachingpotenzial nutzen zu können.

Tarjan [Tar74] führte Intervallanalyse für reduzible Zyklen, insbesondere Kontrollflussgraphen, ein. Aufgrund der Notwendigkeit einer hierarchischen Zerlegung für irreduzible Zyklen wurden bereits unterschiedliche Verfahren entwickelt, die auf beliebige Zyklen angewendet werden können [SGL96, Ste93, Hav97, Ram99, Ram02].

In dieser Arbeit werden vier Zerlegungsverfahren auf Programmabhängigkeitsgraphen angewendet. Mittels des obigen Caching-Verfahrens wird empirisch untersucht, ob sich ein größerer Caching-Effekt ergibt, als mit unzerlegten Zyklen. Die folgenden Studien zeigen, dass sich die vier Verfahren unterschiedlich auf die Performance des Pfadbedingungsgenerators auswirken, jedoch der Verwendung von unzerlegten Zyklen deutlich überlegen sind. Die Berechnungskomplexität für Pfadbedingungen bleibt i.Allg. exponentiell, jedoch lassen sich trotzdem in der Praxis Pfadbedingungen effizient berechnen.

7.5.1 Eigenschaften von ineinander geschachtelten Zyklen

Die vier Zerlegungsverfahren sind von Sreedhar, Gao, Lee [SGL96], Steensgaard [Ste93], Havlak [Hav97] und Ramalingam [Ram99, Ram02]. Alle Verfahren haben gemeinsam, dass sie die Starken Zusammenhangskomponenten eines Graphen in unterschiedliche ineinander geschachtelte Zyklen zerlegen. Ein geschachtelter Zyklus ist per Definition kleiner als sein übergeordneter Zyklus und vollständig im übergeordneten Zyklus enthalten. Weiterhin können mehrere Zyklenhierarchien nebeneinander existieren, wobei jede Hierarchie disjunkt zu anderen ist. In geschachtelten Zyklen findet daher nie eine Überlappung von zwei Zyklen statt. Die einzelnen Zyklen sind entlang einer Hierarchie beliebig reduzibel oder irreduzibel.

Da [SGL96, Ste93, Hav97, Ram99, Ram02] bei ihren Algorithmen von (virtuell) gefalteten Zyklen ausgehen, sind die Klassifizierungen nach Reduzibilität und Irreduzibilität nur *lokal* gültig. Um Pfadbedingungen durch die Zyklenzerlegung effizienter zu machen, müssen die *realen* Ein- und Austrittsknoten der Zyklen bekannt sein. Ein gefalteter Zyklus kann daher vom Algorithmus lokal als reduzibel klassifiziert werden, wenn sein einziger Eintrittsknoten durch einen *gefalteten* Zyklus dargestellt wird. Real kann der gefaltete Zyklus irreduzibel sein und beim Auffalten seinen übergeordneten Zyklus ebenfalls irreduzibel werden lassen. Solch eine Implikation (reduzibel(irreduzibel)→irreduzibel) liegt jedoch nur dann vor, wenn die Eintrittsknoten des gefalteten Zyklus Randknoten des übergeordneten Zyklus darstellen.

Die Aussage im Abschnitt 4.3.3, dass Zyklen auf einzelnen Pfaden ignoriert werden können, gilt unabhängig von der Zyklenzerlegung weiter, da auch weiterhin die reguläre Pfadaufzählung innerhalb der geschachtelten Zyklen durchzuführen ist. Die begriffliche Trennung von Zyklen auf Pfaden im PDG und Zyklen im Graphen ist jedoch bedeutsam. Die einzige Gemeinsamkeit dieser zwei Begriffe liegt in reduziblen Zyklen mit einem einzigen Austrittsknoten, der dem Eintrittsknoten entspricht. In diesem Fall kann bis auf den Eintrittsknoten der gesamte Zyklus als Zyklus entlang eines Pfades angesehen und damit ignoriert werden.

7.5.2 Faktorisierungspunkte geschachtelter Zyklen

Der Algorithmus von Steensgaard unterscheidet sich von den anderen drei Verfahren, da er *top-down* arbeitet, alle anderen *bottom-up*. Steensgaards Verfahren beginnt daher mit der Starken Zusammenhangskomponente und berechnet durch die Entfernung von Kanten so lange kleinere Zyklen, bis das Verfahren terminiert. Im Gegensatz dazu berechnen Sreedhar/Gao/Lee,

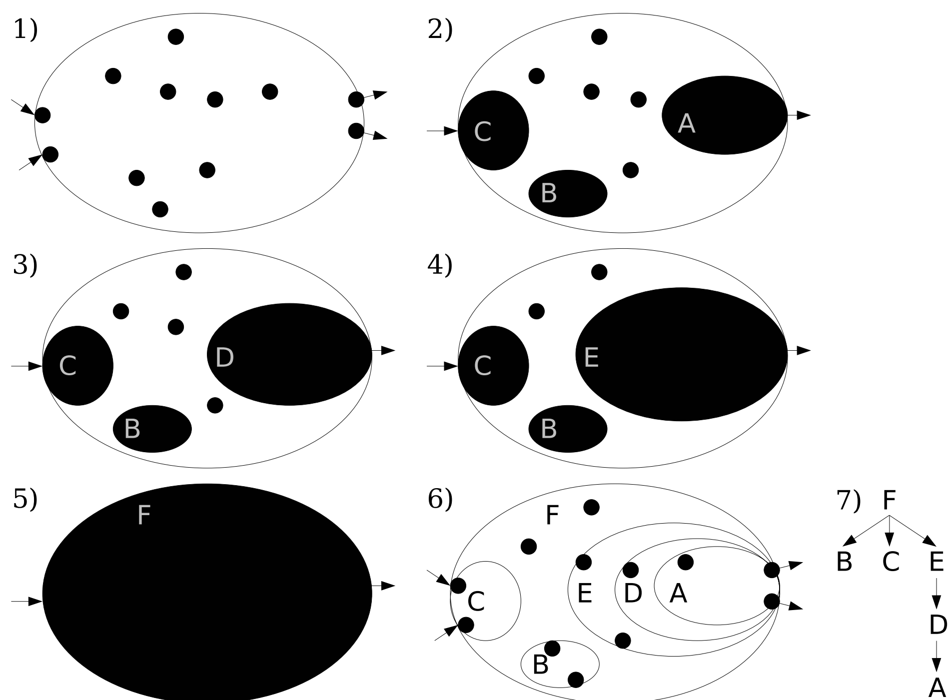


Abbildung 7.8: Hierarchie geschachtelter Zyklen

Havlak und Ramalingam nach ihrer Art möglichst kleine Zyklen, die anschließend zu Knoten gefaltet werden und dadurch eine weitere Zyklensuche erlauben, die so lange fortgeführt wird, bis die Starke Zusammenhangskomponente erreicht ist.

Die Abbildung 7.8 zeigt die Zerlegung einer Starken Zusammenhangskomponente durch ein Bottom-up-Verfahren in eine Hierarchie von geschachtelten Zyklen. Der Graph in 1) ist real irreduzibel, da er zwei Eintrittsknoten besitzt. In 2) werden die drei Unterzyklen A,B und C ermittelt und zu einzelnen (virtuellen) Knoten gefaltet[5]. Der Restgraph enthält nun weniger Knoten. Dies hat zur Folge, dass C ein Eintrittsknoten geworden ist und A ein Austrittsknoten, da nur über sie der gesamte Zyklus betreten bzw. verlassen werden kann. Für den Algorithmus wird daher der Restzyklus reduzibel. Die reale Irreduzibilität muss in einem späteren Schritt bei der Pfadbedingungsberechnung berücksichtigt werden, wenn die Auffaltung der Ein- und Austrittsknoten entlang der Zyklenhierarchie stattfindet. In 3) und 4) ergeben sich weitere Hierarchieebenen um den Ausgangszyklus A. Im letzten Schritt in 5) werden alle gefundenen realen oder virtuellen Knoten zum Gesamtzyklus F verschmolzen.

Die Zerlegungverfahren liefern für Pfadbedingungen die reinen Daten der

[5] Die Reihenfolge der Zyklenerkennung ist o.B.d.A. A-B-C-D-E-F.

Algorithmus 7.1 Zyklenzerlegung nach Steensgaard

Input: $G_C \in (N, \rightarrow)$ intraprozeduraler Zyklus im PDG
$entry \in G_C$ beliebiger Zykleneintrittsknoten
$hidden_edges \subseteq G_C$ Zyklen-schließende Kanten, initial {}
Output: C *Hierarchie von geschachtelten Zyklen*

proc STEENSGAARD($entry, hidden_edges$):
Initialisierung
Berechne $SCC(entry, hidden_edges)$ *ohne Kanten aus* $hidden_edges$
foreach $S \in SCC$ *mit* $|S| > 1 \wedge S \notin C$ **do**
  foreach $n \in S$ **do**
    if n *ist ein Eintrittsknoten von* S **then**
      $S_{entry}.insert(n)$
      foreach $m \rightarrow n$ *mit* $m \rightarrow n \in S$ **do**
        $hidden_edges.insert(m \rightarrow n)$
    if n *ist ein Austrittsknoten von* S **then**
      $S_{exit}.insert(n)$
  if $|S_{entry}| > 1$ **then**
    markiere S *als irreduzibel*
  else
    markiere S *als reduzibel*
  $C.insert(S, S_{entry}, S_{exit})$
  foreach $n \in S_{entry}$ **do**
    $C = C \cup$ STEENSGAARD($n, hidden_edges$)
return C *Die Hierarchie der geschachtelten Zyklen*

Zyklen (reale innere Knoten, Ein- und Austrittsknoten) in 6). Die Pfadbedingungsberechnung kann die geschachtelten Ein- und Austrittsknoten als *Faktorisierungspunkte* zum Wiederverwenden von bereits berechneten Pfadbedingungen nutzen. Die resultierende Hierarchie aus der Zyklenfaltung wird in 7) dargestellt.

7.5.3 Zerlegung nach Steensgaard

Steensgaard hat in [Ste93] ein Verfahren zur Sequenzialisierung von Programmen aus grafischen Programmrepräsentationen vorgeschlagen, das zur Code-Generierung in Compilern notwendig ist. Bisherige Verfahren von [SAF90, SF93] basierten auf schleifenfreien Kontrollabhängigkeitsgraphen (engl. Control Dependency Graphs, CDG [BM92]). Steensgaards Variante arbeitet mit beliebigen CDGs, die irreduzible Zyklen enthalten können. Als Nebenprodukt dieser Arbeit entstand ein Algorithmus zum Zerlegen von irreduzible Zyklen in eine Hierarchie von geschachtelten Zyklen, die ausschließlich auf der Berechnung von Starken Zusammenhangskomponenten basiert.

Steensgaards Algorithmus wird nur in Prosaform angegeben und beschreibt nicht eindeutig, wie er algorithmisch mit gefundenen Eintrittsknoten umgeht. In jedem Rekursionsschritt werden dem Graphen diejenigen Kanten entfernt, die zu allen aktuellen Eintrittsknoten führen. Es bleibt unklar, ob die SCC-Berechnung für *jeden* Eintrittsknoten rekursiv durchgeführt wird, oder nur für einen beliebigen. Der Restgraph ohne die entfernten Kanten bleibt zwar gleich, jedoch können sich je nach gewähltem Eintrittsknoten unterschiedlich viele Starke Zusammenhangskomponenten ergeben. In dieser Arbeit wurde daher der Algorithmus so realisiert, dass für alle Eintrittsknoten der rekursive Aufruf stattfindet, um möglichst viele geschachtelte Zyklen zu finden. Die Bedingung $S \notin C$ verhindert dabei eine doppelte Aufnahme von bereits berechneten Starken Zusammenhangskomponenten im Untergraphen. Der Algorithmus ist in 7.1 dargestellt.

7.5.4 Zerlegung nach Sreedhar, Gao, Lee

Sreedhar, Gao und Lee haben Tarjans [Tar74] ursprünglichen Algorithmus für reduzible Graphen auf irreduzible Graphen erweitert [SGL96]. Hierfür berechnen sie zuerst den Dominatorbaum, so dass dieser Algorithmus für andere Einsatzgebiete, wie z.B. der Dominatorbaumberechnung, die sich selbst mittels Zyklenhierarchien effizient berechnen lässt, eher ungeeignet ist [Ram02]. Für die Zerlegung von Programmabhängigkeitsgraphen stellt dies jedoch kein Problem dar.

Der Algorithmus verwendet im Gegensatz zu den anderen Verfahren eine spezielle Datenstruktur, den sog. DJ-Graphen, der drei verschiedene Kantentypen beinhaltet (Dominatorkanten, Cross-Join-Kanten und Back-Join-Kanten). Um den Algorithmus für Pfadbedingungen anwenden zu können, wird daher der PDG so in einen DJ-Graphen eingebettet, dass jeder notwendigen PDG-Kante genau eine DJ-Graph-Kante zugeordnet wird. Durch zusätzliche Dominatorinformationen kann der DJ-Graph jedoch mehr Kanten enthalten als der PDG. Diese Kanten müssen bei der Bestimmung der realen Ein- und Austrittsknoten von PDG-Zyklen berücksichtigt werden.

Der Dominatorbaum nach Lengauer/Tarjan [LT79] liefert im DJ-Graphen eine Hierarchie, die bottom-up untersucht wird, wobei Dominatorknoten potenzielle Zykleneintrittsknoten darstellen. Eine zusätzliche Tiefensuche bestimmt Rückwärtskanten, um geschachtelte irreduzible Zyklen zu identifizieren.

Algorithmus 7.2 beschreibt das realisierte Verfahren. Die Funktionen $findEntry()$ und $findExit()$ dienen als Hilfsfunktionen, um die Ein- und Austrittsknoten der jeweiligen geschachtelten Zyklen zu bestimmen. Ein- bzw. Austrittsknoten liegen genau dann vor, wenn die Knoten der ankom-

Algorithmus 7.2 Zyklenzerlegung nach Sreedhar, Gao und Lee

Req: Einbettung des PDG in einen DJ-Graphen
Jede Kante hat dann einen der folgenden Typen: DOM, BJ, CJ
Input: $G_C \in (N, \rightarrow)_{DJ}$ intraprozeduraler Zyklus im DJ-Graph
$d_1, \ldots, d_n$ Ebenen der Dominatorbaumhierarchie
entry $\in G_C$ beliebiger Zykleneintrittsknoten
Output: C *Hierarchie von geschachtelten Zyklen*

proc SREEDHAR_GAO_LEE(*entry*):
Initialisierung
Führe Tiefensuche ab entry zur Rückwärtskanten-Berechnung durch
foreach $d \in d_n, \ldots, d_1$ *bottom-up in der Dominatorbaumhierarchie* **do**
  $irreducible_cycle = false$
  foreach $n \in G_C$ *mit* $level(n) = d$ **do**
    Initialisiere leeren Zyklus S
    foreach $m \rightarrow n$ *mit* $m \rightarrow n \in G_C$ **do**
      if $type(m \rightarrow n) = CJ \wedge m \rightarrow n$ *Rückwärtskante* **do**
        $irreducible_cycle = true$ *(irreduzibler Zyklus)*
      if $type(m \rightarrow n) = BJ$ **do**
        $\forall k \in n \rightarrow^* k \rightarrow^* m : S.insert(k)$ *(reduzibler Zyklus)*
        Füge alle erreichbaren Knoten unterhalb von n ein.
        $S.insert(m, n)$
    if $|S| > 1|$ **do**
      markiere S als reduzibel
      foreach $p \in S$ **do**
        if *p ist ein Eintrittsknoten von S* **then**
          $S_{entry}.insert(p)$
        if *p ist ein Austrittsknoten von S* **then**
          $S_{exit}.insert(p)$
      $C.insert(S, S_{entry}, S_{exit})$
      Kollabiere Zyklus S im DJ-Graphen
  if $irreducible_cycle = true$ **do**
    covered = {}
    foreach $n \in G_C$ *mit* $level(n) = d \wedge n \notin covered$ **do**
      Berechne SCC S_n *ab n für alle Knoten mit* $level() \geq d$
      if $|S_n| > 1$ **do**
        $S_{n.entry}.insert(p \in S_n | level(p) = d)$
        $S_{n.exit}.insert(findExit(S_n))$
        markiere S_n *als irreduzibel*
        $C.insert(S_n, S_{n.entry}, S_{n.exit})$
        $covered = covered \cup S_n$
        Kollabiere S_n
return *C Die Hierarchie der geschachtelten Zyklen*

menden bzw. abgehenden intraprozeduralen Kanten außerhalb des aktuellen Zyklus liegen, i.d.R. im übergeordneten Zyklus oder azyklischen Bereich.

7.5.5 Zerlegung nach Havlak

Der dritte Zyklenzerlegungsalgorithmus ist von Havlak [Hav97] und stellt eine Erweiterung von Tarjans Algorithmus zur Überprüfung auf Reduzibilität dar [Tar74]. Im Unterschied zum Algorithmus von Sreedhar, Gao und Lee hängt hier das Ergebnis der hierarchischen Zerlegung bei irreduziblen Zyklen vom Tiefensuchbaum und damit von der Reihenfolge der bearbeiteten Knoten ab. Zwei Programmläufe mit unterschiedlichen Startknoten können daher bei irreduziblen Zyklen zu unterschiedlichen Zerlegungen kommen.

Der Algorithmus wurde als „fast linear“ beschrieben, jedoch hat Ramalingam in [Ram99] gezeigt, dass unter bestimmten Umständen die Komplexität quadratisch sein kann. Ramalingam hat den Algorithmus dahingehend modifiziert, dass er tatsächlich fast linear wird. Algorithmus 7.3 zeigt die Ramalingamsche Version von Havlaks Verfahren.

7.5.6 Zerlegung nach Ramalingam

Bei der vierten Zyklenzerlegung handelt es sich wiederum um eine Modifikation der Havlakschen Zerlegung [Hav97, Ram02]. Diese Zerlegungsart ist von Ramalingam entwickelt worden, um für die Probleme der Dominatorbaum- und Dominatorfrontberechnung einen Algorithmus zu haben, der sowohl beinahe linear in der Zyklenzerlegung, als auch linear in der Anwendungskomplexität ist.

Der Algorithmus berechnet eine kleinere Version der Havlak-Hierarchie, indem er die in Havlak geschachtelten Zyklen genau dann wieder zusammenfaltet, wenn sie *gemeinsame* Eintrittsknoten teilen. Der im Pfadbedingungsgenerator realisierte Algorithmus ist also identisch zu 7.3, bis auf die Hilfsfunktion $findEntry$. Jeder bereits registrierte Eintrittsknoten wird in $findEntry$ mit seinem zugehörigen Zyklus vermerkt. Wenn nun ein neu registrierter Eintrittsknoten schon ein Eintrittsknoten eines vorhandenen und damit geschachtelten Zyklus war, werden die beiden Zyklen miteinander verschmolzen.

7.5.7 Vier Zyklenzerlegungen am Beispiel „looptest“

In Abbildung 7.7 auf Seite 90 ist die Starke Zusammenhangskomponente als einziger Zyklus der Funktion `looptest` eingezeichnet. Die Anwendung der vier Zerlegungsverfahren auf diesen Zyklus ist in den Abbildungen 7.9, 7.10, 7.11 und 7.12 zu sehen. Zahlen auf schwarzem Grund stellen die Zyklennummern dar, wobei die Berechnungsreihenfolge eingehalten wird. Wenn PDG-Knoten mit Ein- und Austrittsmarken gekennzeichnet sind, so gelten diese

Algorithmus 7.3 Zyklenzerlegung nach Havlak (Ramalingam Version)

proc COLLAPSE($body, header$):
foreach $z \in body$ **do**
  $loop_parent(z) = header$
  $S.insert(z)$
  $union(z, header)$ *header ist Repräsentant der Menge*
$S.insert(header)$
$S_{entry}.insert(findEntry(S))$
$S_{exit}.insert(findExit(S))$
Markiere S in Abhängigkeit von $|S_{entry}| > 1$ *als irreduzibel bzw. reduzibel*
$C.insert(S, S_{entry}, S_{exit})$

▶

proc FIND_LOOP($header$):
$body = \{\}$
$worklist = \{find(y) | (y \rightarrow header)$ *ist eine Rückwärtskante*$\} - \{\text{header}\}$
while $\neg worklist.empty()$ **do**
  Entferne beliebigen Knoten y aus worklist
  $body.insert(y)$
  foreach $z \rightarrow y \in G_C$ *mit* $z \rightarrow y$ *ist keine Rückwärtskante* **do**
    if $find(z) \notin (body \cup \{header\} \cup worklist)$ **do**
      worklist.insert(find(z))
if $\neg body.empty()$ **do**
  return COLLAPSE($body, header$)

▶

Input: $G_C \in (N, \rightarrow)$ intraprozeduraler Zyklus im PDG
  $entry \in G_C$ beliebiger Zykleneintrittsknoten
Output: C *Hierarchie von geschachtelten Zyklen*

proc HAVLAK_RAMALINGAM($entry$):
Initialisierung
Führe Tiefensuche ab entry zur Rückwärtskanten-Berechnung durch
Berechnung der kleinsten gemeinsamen Vorgänger (LCA,[CLRS01])
foreach $n \in G_C$ **do**
  $loop_parent(n) = NULL$
foreach $m \rightarrow n \in G_C$ *mit* $m \rightarrow n$ *keine Rückwärtskante* **do**
  (nur Vorwärts- und Kreuzkanten)
  Entferne $m \rightarrow n$ *aus* G_C
  $crossFwdEdges[LCA(m, n)].insert(m \rightarrow n)$
foreach $n \in G_C$ *in umgedrehter Tiefensuch-Reihenfolge* **do**
  foreach $x \rightarrow y \in crossFwdEdges[n]$ **do**
    Füge Kante $find(x) \rightarrow find(y)$ *in* G_C *ein*
  $C = C \cup$ FIND_LOOP(n)
return C *Die Hierarchie der geschachtelten Zyklen*

für den am nahe liegendsten Zyklus (bis auf Steensgaard, wo aufgrund der Berechnungsweise die Zyklen explizit vermerkt sind). Wenn Zyklennummern mit Ein- und Austrittsmarken gekennzeichnet sind, dann bedeutet dies, dass

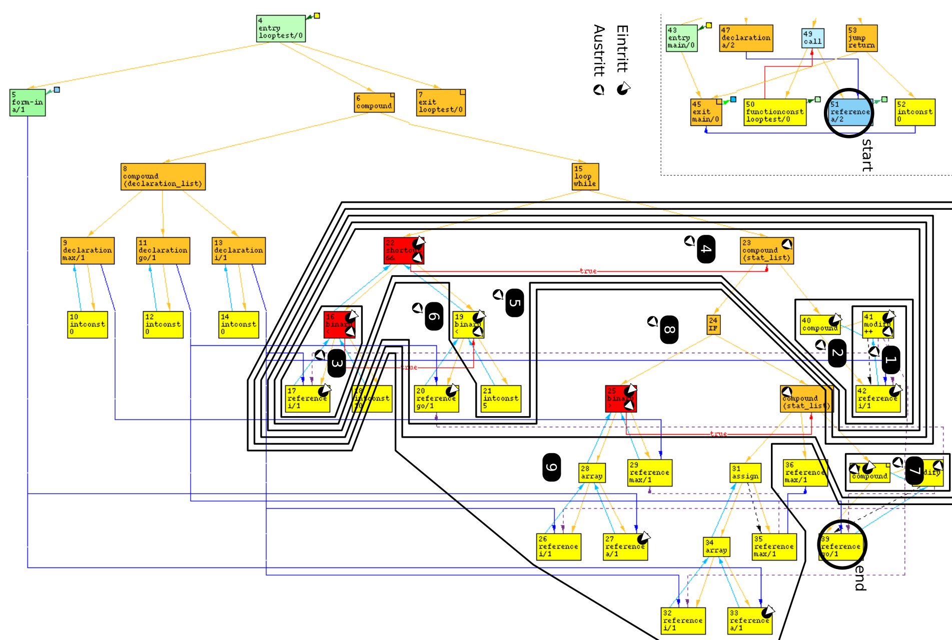


Abbildung 7.9: Zyklenhierarchie nach Sreedhar, Gao und Lee

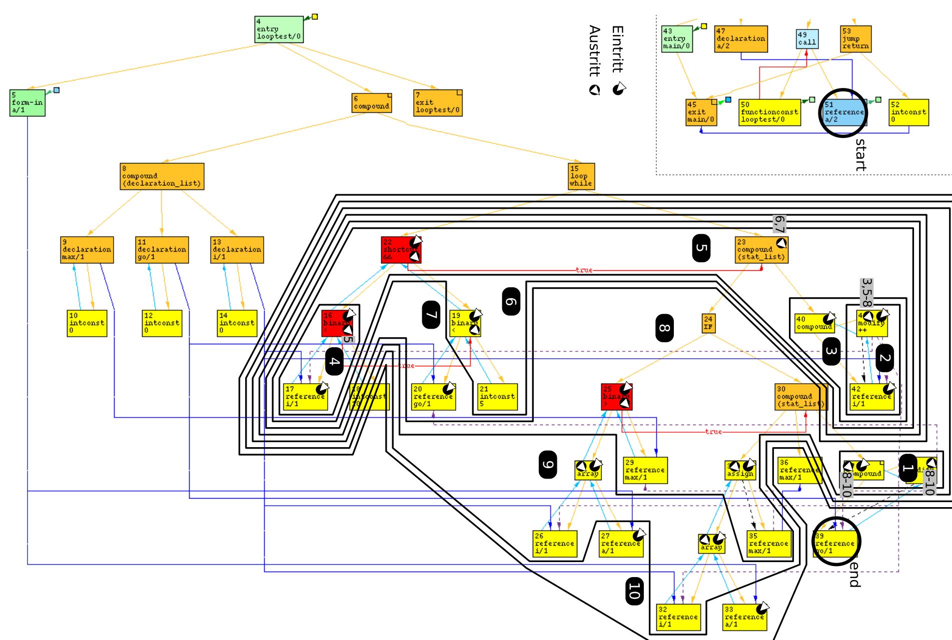


Abbildung 7.10: Zyklenhierarchie nach Steensgaard

Abbildung 7.11: Zyklenhierarchie nach Havlak

Abbildung 7.12: Zyklenhierarchie nach Ramalingam

der Zyklus in gefalteter Form selbst einen Ein- bzw. Austrittsknoten darstellt. Für Ramalingam wurde hierbei die Zyklennummerierung von Havlak verwendet, wobei fehlende Nummern die Verschmelzung von Zyklen bedeuten.

Es ist deutlich zu sehen, dass trotz der wenigen Knoten und Kanten des Originalzyklus aus Abbildung 7.7 die Zerlegungsverfahren zu den folgenden sehr unterschiedlichen Ergebnissen kommen:

Zerlegung	reduzibel	irreduzibel	$\sum$
Steensgaard	5 {1,3,5,6,7}	5 {2,4,8,9,10}	10
Sreedhar, Gao und Lee	6 {2,4,5,6,7,8}	3 {1,3,9}	9
Havlak	5 {2,4,5,6,7}	7 {1,3,8,9,10,11,12}	12
Ramalingam	5 {2,4,5,6,7}	3 {1,3,12}	8

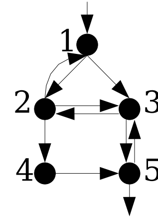
Die Klassifizierung der Zyklen nach Reduzibilität bzw. Irreduzibilität orientiert sich an realen Zyklen, nicht gefalteten. Auffällig an den Verfahren ist, dass sie für bestimmte Teile des Graphen (oberer linker und rechter Bereich) identische Zerlegungen finden und dabei bis auf Steensgaard, der als einziger top-down arbeitet, die gleiche Reihenfolge beibehalten. Offensichtlich gibt es wenig Spielraum für Unterschiede bei kleinen Zyklen. Jedoch differieren die Zerlegungsverfahren bei größeren Zyklen stark auseinander, vor allem im Bereich der zwei Haupteintrittsknoten 27 und 33 (unterer Bereich).

7.5.8 Intervallanalyse, Spieletheorie und Komplexität im Detail

Prinzipiell müssen reduzible und irreduzible Zyklen nicht unterschieden werden. Die Pfadbedingungsberechnung kann mit einer generellen Klassifizierung als „irreduzibel“ erfolgen. Jedoch haben reduzible Zyklen bestimmte Eigenschaften, die die Pfadbedingungsberechnung beschleunigen.

In reduziblen Zyklen können Rückwärtskanten generell ignoriert werden, da sie laut Definition nur zu Zykleneintrittsknoten zurückführen bzw. automatisch in einem Pfad-Zyklus enden, der bis zum Austrittsknoten der Rückwärtskante führt. Der resultierende Restzyklus ohne Rückwärtskanten kann jedoch weiterhin Zyklen enthalten und lässt daher eine generelle topologische Sortierung von reduziblen Zyklen nicht zu.

Die nebenstehende Abbildung zeigt einen reduziblen Zyklus mit 5 Knoten. 1 ist der Eintrittsknoten und 5 der Austrittsknoten. Wenn die Rückwärtskante zwischen 2 und 1 entfernt wird, bleibt ein Zyklus bestehen, so dass keine topologische Sortierung der Kanten und damit eine Pfadbedingungsberechnung mit linearer Komplexität möglich wird.



Reduzible Zyklen haben den Vorteil, dass sie nur einen Eintrittsknoten haben. Nach Definition 7.6 kann daher für die Restpfadbedingung $PC(out_j, x)$ auch innerhalb von geschachtelten Zyklen der gesamte reduzible Zyklus ignoriert werden, da er unweigerlich in einen Zyklus entlang eines Pfades führen würde. Dies gilt auch für irreduzible Zyklen mit einem einzigen Austrittsknoten.

Die Ein- und Austrittsknoten geschachtelter Zyklen stellen Faktorisierungspunkte für wiederverwendbare Pfadbedingungen dar. Es ist jedoch nicht sinnvoll, $|in| \times |out|$ Pfadbedingungen je Zyklus vorzuberechnen, da während der Pfadbedingungsberechnung nicht alle Eintrittsknoten eines Zyklus betreten werden müssen. Die Pfadtraversierung für Teilpfade wird generell – auch im azyklischen Bereich – genau dann vermieden, wenn

- Absorptionsgesetze *an Knoten* gelten. Dies ist genau dann der Fall, wenn von einem Knoten y mehrere Pfade $y \to y^1 \to^* x$, $y \to y^2 \to^* x$, $\ldots$, $y \to y^n \to^* x$ abgehen, und ein Pfad über y^i die Bedingung *true* liefert. Durch die Disjunktion der Pfadbedingungen werden alle Pfadbedingungen der Pfade $1 \ldots i-1$ ebenfalls zu *true* und die Pfade $i+1 \ldots n$ müssen nicht traversiert werden.

- Absorptionsgesetze *an Pfaden* gelten. Die Pfadaufzählung speichert für alle Pfade $y \to^* x$ ihre Bedingungen. Liegen die Bedingungen für $y \to y^1 \to^* x, \ldots, y \to y^i \to^* x$ vor, werden diese disjunktiv verknüpft, um die schwächste Bedingung PC_{weak} zu ermitteln. An jedem Knoten der Teilpfade $y \to y^{i+1} \to^* x$, $\ldots$, $y \to y^n \to^* x$ wird überprüft, ob die Gleichung $PC_{weak} = PC_{weak} \vee PC_{current}$ gilt. $PC_{current}$ stellt dabei die Bedingung des Pfades bis zum aktuellen Knoten dar. Wenn die Gleichung erfüllt ist, kann die aktuelle Bedingung des Pfades nur noch stärker als PC_{weak} werden, so dass der aktuelle Pfad nicht weiter traversiert werden muss. Dieses Verfahren der Suchbaumbegrenzung entstammt der Spieletheorie (z.B. beim Schach), in der ungünstigere Pfade des Ergebnisbaums abgeschnitten werden.

Pfadbedingungen werden weiterhin durch einen typischen Tiefensuchalgorithmus zur Pfadaufzählung generiert, so dass gemeinsame Prefixe automatisch herausfaktorisiert werden.

Die Komplexität der Pfadbedingungsberechnung setzt sich hauptsächlich aus dem Aufwand zur Berechnung von Ausführungsbedingungen, zur Durchführung von BDD-Operationen, der Pfadaufzählung und der anschließenden Berechnung einer minimierten Disjunktiven Normalform zusammen. Wie schon beschrieben, werden Ausführungsbedingungen für traversierte Knoten einmalig[6] in $O(|N|)$ für einen PDG (N, E) berechnet und bei Bedarf wiederverwendet. Da Kontrollabhängigkeitskanten im PDG einen azyklischen Graphen erzeugen, ist die lineare Komplexität möglich. Äquivalent zu Ausführungsbedingungen werden auch die Bedingungen für $\delta(y \rightarrow x)$, $\beta(y)$ und $\Phi(y \rightarrow x)$ einmalig berechnet und wiederverwendet, wenn sie pfadunabhängig sind.

Binäre Entscheidungsgraphen haben für die Standardoperationen $\wedge$ und $\vee$ eine polynomiale Komplexität, die bei Pfadbedingungen von $|N|$ abhängt. Im Durchschnitt liegt sie bei $O(|N|^k)$ für ein nicht näher bestimmtes aber festes k.

Die Pfadaufzählung im azyklischen Bereich ist nach Definition 7.8 immer linear $O(|N|)$.

Geschachtelte Zyklen L enthalten Eintrittsknoten $in_1, \ldots, in_m$ und Austrittsknoten $out_1, \ldots, out_n$, wobei Ein- und Austrittsknoten nicht disjunkt sein müssen. Alle zyklenfreien Pfade zwischen in_i und $out_1, \ldots, out_n$ werden durch das Standardverfahren nach Definition 7.6 *innerhalb des aktuellen Zyklus* berechnet. Jedes Ergebnis $PC(in_i, out_1), \ldots, PC(in_i, out_n)$ steht nun nachfolgenden Berechnungen zur Verfügung. Die Berechnungskomplexität ist von der Anzahl der möglichen Pfade p und von der Größe des Zyklus $|L|$ abhängig und damit innerhalb von Zyklen mit $O(p * |L|)$ exponentiell. Der Einfluss von Caching-Effekten und die Behandlung von reduziblen Zyklen wirkt sich positiv auf die praktische Anwendbarkeit aus, reduziert jedoch die theoretische Komplexität nicht.

Die berechneten Pfadbedingungen $PC^l(in_i, out_j)$ einer Hierarchieebene werden nur in derselben Hierarchieebene l wiederverwendet, da sie für einen bestimmten Zyklus gültig sind. Jede Hierarchieebene hat daher unabhängige wiederverwendbare Pfadbedingungen, die sogar identische Ein- und Austrittsknoten haben können. So kann es ein $PC^{l+1}(in_i, out_j)$ zwischen identischen Knoten geben, das sich jedoch dadurch zu $PC^l(in_i, out_j)$ unterscheidet, dass der geschachtelte Zyklus $l + 1$ kleiner als l ist und daher die Wiederverwendung von $PC^{l+1}(in_i, out_j)$ in l (statt $PC^l(in_i, out_j)$) illegal wäre.

Als Voraussetzung für die korrekte Anwendung von geschachtelten Zyklen müssen, wie in den Abbildungen 7.9, 7.10, 7.11 und 7.12 dargestellt, die Ein- und Austrittsknoten von den inneren Zyklen in die äußeren Zyklen „hochpropagiert“ werden, da die Zyklen nur symbolisch gefaltet sind. Kommen

[6] Die Berechnung erfolgt on-demand.

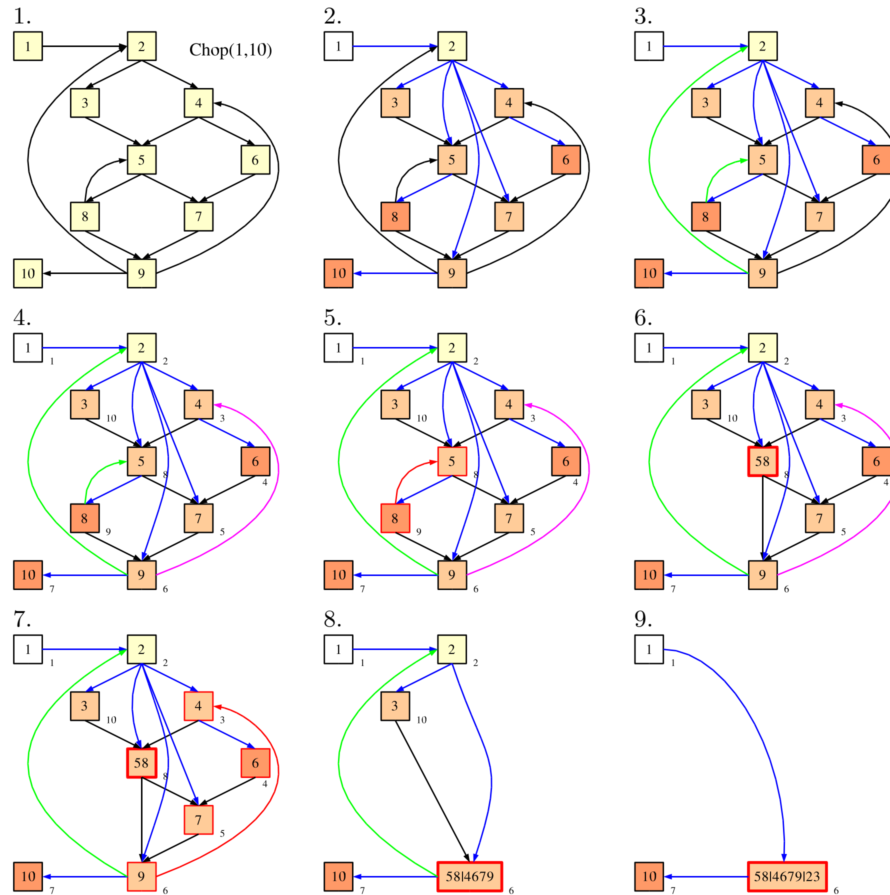


Abbildung 7.13: Pfadbedingung mit Zyklenzerlegung und Caching-Effekt

nun gefaltete Zyklen in übergeordneten Zyklen als symbolische Ein- oder Austrittsknoten vor, erhöht sich die Zahl der realen Ein- und Austrittsknoten des übergeordneten Zyklus maximal um die Anzahl der Ein- und Austrittsknoten des gefalteten Zyklus, wenn es sich um reale „Randknoten“ des übergeordneten Zyklus handelt.

7.5.9 Zyklenzerlegung und Caching-Effekt am Beispiel

Anhand des Graphen in Abbildung 7.13 (1) soll die hierarchische Zyklenzerlegung (beispielhaft nach Sreedhar, Gao und Lee) mit der anschließenden Einbindung der geschachtelten Zyklen in die Pfadbedingungsberechnung durchgeführt werden.

Der dargestellte PDG besteht aus 10 Knoten mit einem einfachen reduziblen Zyklus $\{2, \ldots, 9\}$. Die Definition 7.7 liefert $PC(1, 10) \equiv PC(1, 2) \wedge PC(2, 9) \wedge$

$PC(9,10)$. $PC(1,2)$ und $PC(9,10)$ lassen sich in linearer Zeit berechnen.

Der einfache Beispielgraph in (1) lässt sich topologisch sortieren, wenn seine Rückwärtskanten entfernt werden. Bei Zyklen in PDGs ist diese Eigenschaft jedoch nicht generell gegeben, da deren Struktur beliebig komplex sein kann. Deshalb wird von der besonderen Struktur von (1) kein Gebrauch gemacht und eine allgemeine exponentielle Berechnungskomplexität angenommen.

Jeder Knoten i steht für eine Ausführungsbedingung $E(i)$. δ-, β- und Φ-Bedingungen werden hierbei nicht betrachtet, da sie zur Demonstration des Nutzens einer Intervallanalyse nicht von Belang sind. Die resultierenden 5 Pfade innerhalb des Zyklus sind $2 \rightarrow 3 \rightarrow 5 \rightarrow 8 \rightarrow 9$, $2 \rightarrow 3 \rightarrow 5 \rightarrow 7 \rightarrow 9$, $2 \rightarrow 4 \rightarrow 5 \rightarrow 8 \rightarrow 9$, $2 \rightarrow 4 \rightarrow 5 \rightarrow 7 \rightarrow 9$ und $2 \rightarrow 4 \rightarrow 6 \rightarrow 7 \rightarrow 9$. Selbst innerhalb dieses kleinen Zyklus ist die Redundanz in den Bedingungen und der Pfadtraversierung offensichlich, da die Folgepfade ab Knoten 5 mehrfach durchlaufen werden.

In Phase 1 wird der Dominatorbaum berechnet (Abb. 7.13 (2)) und die Dominatorkanten dem Graphen hinzugefügt. Der Graph enthält nun 3 zusätzliche Kanten $2 \rightarrow 5$, $2 \rightarrow 7$ und $2 \rightarrow 9$, die keine realen Abhängigkeiten darstellen. Durch die Dominatorberechnung wird der Zyklus in sog. Dominatorregionen eingeteilt, die sich aus dem Dominatorbaum ergeben. Innerhalb des Zyklus liegt Knoten 2 auf Ebene 1, die Knoten 3, 4, 5, 7, 9 auf Ebene 2 und Knoten 6, 8 auf Ebene 3. DJ-Graphen bestehen aus den drei Kantenarten DOM- (für Dominatorkanten), BJ- (back-join, für eine besondere Form von Rückwärtskanten) und CJ-Kanten (cross-join, für alle übrigen Kanten). Abb. 7.13 (3) enthält neben den 3 Dominatorkanten die beiden BJ-Kanten $8 \rightarrow 5$ und $9 \rightarrow 2$, da zwischen den Knotenpaaren in umgekehrter Richtung eine *direkte* Dominatorbeziehung besteht. Alle anderen Kanten stellen CJ-Kanten dar, so dass hiermit ein DJ-Graph vorliegt.

Die regulären Rückwärtskanten werden durch einen Tiefensuchalgorithmus bestimmt (Abb. 7.13 (4)) und später zur Identifizierung von irreduziblen Zyklen benötigt[7]. Die drei ermittelten Kanten sind $8 \rightarrow 5$, $9 \rightarrow 2$ und $9 \rightarrow 4$.

Bottom-up der Dominatorhierarchie werden alle Knoten auf ankommende BJ-Kanten untersucht, so dass bei Knoten 5 der erste reduzible Zyklus aus den Knoten 5 und 8 erkannt wird (Abb. 7.13 (5)). Für die Generierung von Pfadbedingungen reicht es aus, die Zyklenknoten, sowie Ein- und Austrittsknoten (*in(5), out(5,8)*) zu speichern. Der Zyklenrumpf wird bei reduziblen Zyklen durch das Markieren aller Knoten – ausgehend vom Zykleneintrittsknoten – unterhalb der aktuellen Dominatorhierarchie erreicht. Der gefaltete Zyklus ist in (Abb. 7.13 (6)) dargestellt.

[7] Im Gegensatz zu Havlaks Algorithmus sind die berechneten Zyklen unabhängig von der Tiefensuchreihenfolge.

Da in der aktuellen Dominatorhierarchie keine ankommenden BJ-Kanten mehr vorliegen, wird der einzige irreduzible Zyklus durch das Zusammenfallen von CJ-Kante und Rückwärtskante $9 \rightarrow 4$ erkannt (Abb. 7.13 (7)). Der irreduzible Zyklus besteht aus den Knoten $\overline{5,8}$,4,6,7,9 und ist gefaltet in Abb. 7.13 (8) zu sehen. Die Ein- und Austrittsknoten sind *in(4,*$\overline{\mathbf{5,8}}$*)* und *out(9)*.

In der obersten Dominatorhierarchie wird durch die ankommende BJ-Kante an Knoten 2 der letzte reduzible Zyklus aus den Knoten $2,3,\overline{\overline{5,8}},4,6,7,9$ erkannt. Für diesen Zyklus wird *in(2), out(*$\overline{\overline{\mathbf{5,8}},\mathbf{4,6,7,9}}$*)* gespeichert. Der resultierende azyklische Graph in Abb. 7.13 (9) zeigt, dass eine weitere Zerlegung nicht möglich ist.

Verwendung der geschachtelten Zyklen zur effizienten Berechnung von Pfadbedingungen

Die zerlegten Zyklen liefern noch keinen Performancegewinn. Der Performancegewinn ergibt sich erst während der Pfadbedingungsberechnung, wenn eine Wiederverwendung bereits berechneter Teilpfadbedingungen stattfindet. Die folgende Tabelle zeigt in 18 Schritten symbolisch den Ablauf der Pfadbedingungsberechnung des Beispiels:

Die Propagation der Eintritts- und Austrittsknoten aller Zyklen bottom-up liefert für Zyklus {5,8}: *in(5), out(5,8)*, für Zyklus {$\overline{5,8}$,4,6,7,9}: *in(4,5), out(9)* und für Zyklus $\{2,3,\overline{\overline{5,8}},4,6,7,9\}$: *in(2), out(9)*.

1. $PC(1,10) \equiv E(1) \wedge PC(2,10)$
2. Bei $PC(2,10)$ wird Zyklus Z_3 $\{2,3,\overline{\overline{5,8},4,6,7,9}\}$ erreicht
3. $PC(2,10) \equiv PC_{Z_3}(2,9) \wedge PC_{|9}(9,10)$, Austrittsknoten 9
4. $PC(9,10) \equiv E(9) \wedge E(10)$
5. $PC_{Z_3}(2,9) \equiv E(2) \wedge (E(3) \wedge PC_{Z_3}(5,9) \vee PC_{Z_3}(4,9))$
6. Bei $PC_{Z_3}(5,9)$ wird Zyklus $\{Z_2\ \overline{5,8}$,4,6,7,9} erreicht
7. $PC_{Z_3}(5,9) \equiv PC_{Z_2}(5,9) \wedge PC_{Z_3|9}(9,9)$, Austrittsknoten 9
8. $PC_{Z_3|9}(9,9) \equiv E(9)$
9. Bei $PC_{Z_2}(5,9)$ wird Zyklus Z_1 {5,8} erreicht
10. $PC_{Z_2}(5,9) \equiv PC_{Z_1}(5,8) \wedge PC_{Z_2|8}(8,9) \vee PC_{Z_1}(5,5) \wedge PC_{Z_2|5}(5,9)$, Austrittsknoten 8 und 5
11. $PC_{Z_1}(5,5) \equiv E(5)$

12. $PC_{Z_2|5}(5,9) \equiv E(5) \wedge E(7) \wedge E(9)$
    Realisierter Pfad $1 \rightarrow 2 \rightarrow 3 \rightarrow 5 \rightarrow 7 \rightarrow 9 \rightarrow 10$
13. $PC_{Z_1}(5,8) \equiv E(5) \wedge E(8)$
14. $PC_{Z_2|8}(8,9) \equiv E(8) \wedge E(9)$
    Realisierter Pfad $1 \rightarrow 2 \rightarrow 3 \rightarrow 5 \rightarrow 8 \rightarrow 9 \rightarrow 10$
15. Bei $PC_{Z_3}(4,9)$ (siehe 5.) wird Zyklus $\{Z_2\ \overline{5,8}$,4,6,7,9$\}$ erreicht
16. $PC_{Z_3}(4,9) \equiv PC_{Z_2}(4,9) \wedge PC_{Z_3|9}(9,9)$
17. $PC_{Z_2}(4,9) \equiv PC_{Z_2}(5,9) \vee E(4) \wedge E(6) \wedge E(7) \wedge E(9)$
    Realisierter Pfad $1 \rightarrow 2 \rightarrow 4 \rightarrow 6 \rightarrow 7 \rightarrow 9 \rightarrow 10$
18. Cache: $PC_{Z_2}(5,9)$ liegt bereits vor.
    Realisierter Pfad $1 \rightarrow 2 \rightarrow 4 \rightarrow 5 \rightarrow 7 \rightarrow 9 \rightarrow 10$
    Realisierter Pfad $1 \rightarrow 2 \rightarrow 4 \rightarrow 5 \rightarrow 8 \rightarrow 9 \rightarrow 10$

$\Rightarrow$ Alle 5 Pfade sind berechnet, die Redundanz wurde bis auf die Kante $7 \rightarrow 9$ vermieden, die sowohl in 12. als auch in 17. traversiert wurde.

Da Ausführungsbedingungen nur einmal berechnet werden, stellt die mehrfache Angabe von Ausführungsbedingungen in obiger Tabelle keinen Komplexitätsanstieg dar. Die hierarchische Zyklenzerlegung zur Pfadbedingungsberechnung zeigt, dass unabhängig von der sporadischen Linearisierbarkeit reduzibler Zyklen, das Verfahren (fast) linear bei dieser Form von Graphen sein kann, jedoch kann es vor allem für nichtlinearisierbare Zyklen durch Caching-Effekte deutlich effizienter als ohne hierarchische Zerlegung von Zyklen sein.

7.5.10 Empirische Untersuchung

In diesem Abschnitt werden die vier Zerlegungsverfahren anhand der Chops aus Tabelle 7.2 auf Seite 75 empirisch untersucht. Jeder Chop hat eine durch das Programm bedingte eigenständige Struktur. Die Eigenschaften der Chops in Bezug auf Variablenreihenfolge für BDDs und Kantenstruktur für Zerlegbarkeit sind damit unterschiedlich und liefern eine Basis für die Übertragbarkeit der Ergebnisse auf andere Graphen.

Zyklenzerlegungen

Die Tabellen 7.4 (Sreedhar, Gao, Lee), 7.5 (Steensgaard), 7.6 (Havlak) und 7.7 (Ramalingam) zeigen die Anzahl der reduziblen Zyklen (redC) mit der jeweils größten Knoten- und Kantenzahl eines Zyklus (maxKn, maxKa) sowie die Daten der irreduziblen Zyklen (irrC). Die rechte Spalte mit rredC

Programm	redC	maxKn	maxKa	irrC	maxKn	maxKa	rredC	rirrC
mergesort1	15	4	7	4	26	53	15	4
mergesort2	19	4	7	5	26	53	19	5
calculator1	26	4	6	11	61	123	26	11
calculator2	8	4	6	5	13	25	8	5
triple_des1	35	8	14	9	20	49	35	9
triple_des2	1	2	2	0	0	0	1	0
ctags1	577	11	21	139	515	2155	577	139
ctags2	452	55	284	86	515	2155	452	86
assembler1	90	27	56	29	24	49	90	29
assembler2	116	9	16	48	27	56	116	48
gnugo1	700	130	258	298	301	823	700	298
gnugo2	862	130	258	358	301	823	862	358
agrep1	15	98	203	3	94	195	15	3
agrep2	148	336	794	17	316	729	148	17
wackeltisch1	509	15	22	265	172	564	509	265
wackeltisch2	497	174	580	274	173	572	497	274
flex1	85	9	17	42	180	394	85	42
flex2	65	6	10	57	41	62	65	57
patch1	1598	23	42	787	1302	30834	1598	787
patch2	1599	1310	30951	780	1309	30864	1599	780
bison1	537	11	21	282	440	24154	537	282
bison2	521	11	21	250	440	24154	521	250
larn1	15	5	8	2	11	23	15	2
larn2	16	5	8	2	11	23	16	2
moria1	2570	1195	4077	693	1182	4056	2570	693
moria2	1226	371	639	416	234	458	1226	416
palme1	73	6	10	49	15	29	73	49
palme2	98	6	10	65	15	29	98	65
palme3	114	6	10	68	15	29	114	68
palme4	135	6	10	77	15	29	135	77
palme5	187	6	10	103	43	76	187	103
palme6	143	6	10	85	15	29	143	85

Tabelle 7.4: Zerlegung nach Sreedhar, Gao und Lee

und rirrC zeigt die *realen* reduziblen und irreduziblen Zyklen, die entstehen, wenn die Eintrittsknoten gefalteter Zyklen in übergeordnete Zyklen propagiert werden.

Wie bei den Starken Zusammenhangskomponenten ist die Anzahl der geschachtelten Zyklen prinzipiell unabhängig von der Chop- bzw. Programmgröße. Ausschlaggebend für die Anzahl ist die Struktur der Chops, wobei Zyklen nur durch Schleifenkonstrukte, wie z.B. `while`, verursacht werden. Geringe redC- und irrC-Werte werden durch eine geringe Anzahl an Starken Zusammenhangskomponenten verursacht. Programmen wie z.B. Palme (siehe Abschnitt Fallstudien 8) mit insgesamt 25 `for`-Schleifen und 10 `while`-Schleifen enthalten im ersten Chop nur 76 SCC, die sich in insgesamt 50 reduzible und 60 irreduzible Zyklen nach Sreedhar, Gao, Lee zerlegen lassen. Im Vergleich dazu enthält der erste Chop des „wackeltisch“ 348 SCC,

Programm	redC	maxKn	maxKa	irrC	maxKn	maxKa	rredC	rirrC
mergesort1	11	4	5	9	26	53	11	9
mergesort2	14	4	5	11	26	53	14	11
calculator1	8	4	6	23	61	123	8	23
calculator2	6	4	6	7	13	25	6	7
triple_des1	25	8	14	25	20	49	25	25
triple_des2	1	2	2	0	0	0	1	0
ctags1	372	11	21	265	515	2155	372	265
ctags2	314	55	284	196	515	2155	314	196
assembler1	34	4	6	43	27	56	34	43
assembler2	46	4	6	75	27	56	46	75
gnugo1	394	130	258	574	301	823	394	574
gnugo2	862	130	258	358	301	823	862	358
agrep1	2	2	2	21	98	203	2	21
agrep2	26	3	4	62	336	794	26	62
wackeltisch1	222	6	10	467	172	564	222	467
wackeltisch2	210	174	580	474	173	572	210	474
flex1	26	4	6	97	180	394	26	97
flex2	26	4	6	82	41	62	26	82
patch1	998	20	40	1320	1302	30834	998	1320
patch2	1012	1310	30951	1275	1309	30864	1012	1275
bison1	384	11	21	460	440	24154	384	460
bison2	377	11	21	424	440	24154	377	424
larn1	9	4	6	7	11	23	9	7
larn2	10	4	6	7	11	23	10	7
moria1	1927	9	14	1008	1195	4077	1927	1008
moria2	925	8	14	571	371	639	925	571
palme1	50	5	8	60	15	29	50	60
palme2	66	5	8	80	15	29	66	80
palme3	82	5	8	83	15	29	82	83
palme4	98	5	8	94	15	29	98	94
palme5	136	5	8	132	43	76	136	132
palme6	100	5	8	105	15	29	100	105

Tabelle 7.5: Zerlegung nach Steensgaard

die in 222 reduzible und 467 irreduzible Zyklen zerlegt werden. Die höhere Anzahl an Zyklen liegt an den 67 `for`-Schleifen und 24 `while`-Schleifen im gesamten Programm.

Die Formel $SCC \leq (rredC + rirrC)$ stellt zwischen geschachtelten Zyklen und SCCs eine Verbindung her. Weiterhin gilt entweder $maxN(SCC) = maxN(redC)$ oder $maxN(SCC) = maxN(irrC)$. Das Verhältnis zwischen SCC und geschachtelten Zyklen schwankt zwischen 1:1,5 und 1:4,5. Der Havlak-Algorithmus ist mit Abstand der stärkste Zerlegungsalgorithmus. Danach folgt Sreedhar/Gao/Lee und an dritter Stelle Steensgaard. Ramalingam liefert die wenigsten geschachtelten Zyklen. In der Sektion über Pfadbedingungen im nächsten Abschnitt wird die Auswirkung der Zerlegungszahl auf die Performance weiter untersucht.

Die rechten beiden Spalten rredC und rirrC zeigen interessante Eigenschaf-

Programm	redC	maxKn	maxKa	irrC	maxKn	maxKa	rredC	rirrC
mergesort1	11	4	5	19	26	53	11	19
mergesort2	15	4	5	24	26	53	15	24
calculator1	12	4	6	42	61	123	12	42
calculator2	5	4	6	11	13	25	5	11
triple_des1	28	8	14	47	20	49	28	47
triple_des2	1	2	2	0	0	0	1	0
ctags1	387	11	21	766	515	2155	387	766
ctags2	306	55	284	640	515	2155	306	640
assembler1	30	4	6	125	27	56	30	125
assembler2	45	4	6	163	27	56	45	163
gnugo1	340	130	258	1384	301	823	340	1384
gnugo2	862	130	258	358	301	823	862	358
agrep1	2	2	2	56	98	203	2	56
agrep2	15	3	4	197	336	794	15	197
wackeltisch1	197	15	22	979	172	564	197	979
wackeltisch2	183	174	580	973	173	572	183	973
flex1	21	4	6	172	180	394	21	172
flex2	35	4	6	125	41	62	35	125
patch1	934	21	42	3965	1302	30834	934	3965
patch2	954	1310	30951	3928	1309	30864	954	3928
bison1	290	11	21	1583	440	24154	290	1583
bison2	283	11	21	1509	440	24154	283	1509
larn1	2	3	4	24	11	23	2	24
larn2	3	3	4	24	11	23	3	24
moria1	1886	57	90	2549	1195	4077	1886	2549
moria2	857	57	90	1478	371	639	857	1478
palme1	48	6	10	116	15	29	48	116
palme2	60	6	10	153	15	29	60	153
palme3	74	6	10	157	15	29	74	157
palme4	90	6	10	184	15	29	90	184
palme5	127	6	10	239	43	76	127	239
palme6	90	6	10	192	15	29	90	192

Tabelle 7.6: Zerlegung nach Havlak

ten der Zyklenzerlegungsverfahren. Ein als reduzibel klassifizierter Algorithmus kann real irreduzibel werden, wenn sein einziger Zykleneintrittsknoten aus einem gefalteten irreduziblen Knoten besteht. Dieser Effekt tritt jedoch ausschließlich bei Ramalingam auf, wo Zyklen mit gemeinsamen Eintrittsknoten verschmolzen werden. In den anderen Verfahren bleibt die Klassifizierung während der Zerlegung auch später noch gültig.

Im Anhang B ab Seite 177 werden für alle in diesem Kapitel vorgestellten Zyklenzerlegungsverfahren detaillierte Grafiken präsentiert, die die besonderen Charakteristika der Zerlegungen hervorheben.

Programm	redC	maxKn	maxKa	irrC	maxKn	maxKa	rredC	rirrC
mergesort1	14	26	53	2	2	2	11	5
mergesort2	17	26	53	3	9	17	14	6
calculator1	15	61	123	4	3	4	7	12
calculator2	7	13	25	3	3	4	5	5
triple_des1	32	20	49	2	2	2	25	9
triple_des2	1	2	2	0	0	0	1	0
ctags1	419	515	2155	48	56	286	310	157
ctags2	343	515	2155	25	36	77	264	104
assembler1	41	27	56	8	10	18	19	30
assembler2	69	27	56	21	10	18	34	56
gnugo1	428	301	823	203	27	72	307	324
gnugo2	862	130	258	358	301	823	862	358
agrep1	4	4	6	2	98	203	1	5
agrep2	22	336	794	5	4	7	11	16
wackeltisch1	314	172	564	86	17	32	130	270
wackeltisch2	304	174	580	88	17	32	116	276
flex1	33	39	75	29	180	394	9	53
flex2	62	23	46	20	41	62	21	61
patch1	1283	1302	30834	343	450	1387	806	820
patch2	1295	1310	30951	345	450	1387	822	818
bison1	397	440	24154	138	257	2137	249	286
bison2	381	440	24154	115	257	2137	242	254
larn1	8	11	23	1	2	2	2	7
larn2	9	11	23	1	2	2	3	7
moria1	2084	1195	4077	262	472	1069	1653	693
moria2	1027	234	458	181	371	639	778	430
palme1	83	15	29	13	7	12	47	49
palme2	105	15	29	19	7	12	59	65
palme3	118	15	29	23	7	12	73	68
palme4	142	15	29	24	7	12	89	77
palme5	198	43	76	31	7	12	124	105
palme6	145	15	29	29	7	12	89	85

Tabelle 7.7: Zerlegung nach Ramalingam

Performance

Die Tabelle 7.8 zeigt für die vier Zerlegungsverfahren die Laufzeiten zur Berechnung der jeweils 32 Pfadbedingungen. Die linke Spalte (SCC) stellt einen direkten Vergleich mit reinen Starken Zusammenhangskomponenten dar. Die Spalte MAX zeigt die prozentuale Geschwindigkeit der besten Zerlegung im Vergleich mit SCCs. Die Spalte MIN zeigt die Geschwindigkeit zur ungünstigsten Zerlegung.

Es ist nicht zu erwarten, dass die gemessenen Zeiten bei einer hierarchischen Zerlegungen zwangsläufig besser sind – im Vergleich zu reinen SCCs. Das Zusammensetzen geschachtelter Zyklen erzeugt einen Overhead und führt durch die Faktorisierung des Graphen zu einer völlig geänderten Reihenfolge der Berechnung von Teilpfadbedingungen im Vergleich zur Pfadaufzählung

Programm	SCC (s)	Sreed. (s)	Steen. (s)	Havl. (s)	Ramal. (s)	MAX (%)	MIN (%)
mergesort1	0	0	0	1	0	100	50
mergesort2	0	0	0	0	0	100	100
calculator1	0	0	0	1	0	100	50
calculator2	0	0	0	0	0	100	100
triple_des1	1	1	1	1	1	100	100
triple_des2	1	0	0	0	0	200	200
ctags1	-	-	-	15	-	**NEU**	**NEU**
ctags2	-	-	-	9	-	**NEU**	**NEU**
assembler1	6	6	4	5	5	150	100
assembler2	4	3	3	4	4	133	100
gnugo1	-	-	-	68	-	**NEU**	**NEU**
gnugo2	-	-	-	94	-	**NEU**	**NEU**
agrep1	0	1	0	1	1	100	50
agrep2	-	26	24	3	-	**NEU**	**NEU**
wackeltisch1	1673	27	317	18	215	9294	528
wackeltisch2	164	344	418	17	151	965	39
flex1	-	2546	5	7	-	**NEU**	**NEU**
flex2	19	19	11	15	15	173	100
patch1	-	-	-	-	-	-	-
patch2	-	-	-	-	-	-	-
bison1	4	4	5	33	31	100	12
bison2	-	-	-	-	-	-	-
larn1	4	3	2	4	4	200	100
larn2	4	3	2	4	4	200	100
moria1	-	-	-	-	-	-	-
moria2	30	30	57	483	482	100	6
palme1	1	0	1	1	1	200	100
palme2	1	0	1	2	2	200	50
palme3	5	5	4	6	6	125	83
palme4	262	261	261	290	290	100	90
palme5	23	22	21	26	26	110	88
palme6	203	203	203	224	225	100	90

Tabelle 7.8: Performancevergleich der Zyklenzerlegungsverfahren

im SCC. Daher müssen zum einen Verfahren der Suchbaumbegrenzung nicht so greifen wie bisher, zum anderen wird auch die Reihenfolge der Pfade geändert, so dass sich dies auf die Reihenfolge der Variablen in den BDDs auswirken kann.

Die Ergebnisse der Tabelle sind jedoch überaus positiv. Der MAX-Durchschnitt zur Berechnung von Pfadbedingungen zeigt eine Geschwindigkeitssteigerung von 1161% im Verhältnis zu reinen SCCs (bei Laufzeiten $> 1s$). Dieser besonders hohe Wert wird durch die extrem kurzen Laufzeiten der Wackeltisch-Analyse verursacht. Ohne Betrachtung des Wackeltisches liegt die Geschwindigkeitssteigerung bei 56% zur Originalgeschwindigkeit, sie wäre damit jedoch zu zurückhaltend angegeben, da der Wackeltisch nicht das einzige herausragende Ergebnis ist. Mittels der hierarchischen Zyklen-

zerlegung ist es im direkten Vergleich erstmals möglich, sechs weitere Fallstudien zu berechnen und damit eine Fallstudienabdeckung von 87,5% zu erzielen. Mit SCCs war nur eine Abdeckung von 69% möglich. Die sechs neuen Fallstudien zu ctags, gnugo, agrep und flex sind in der angegebenen Geschwindigkeitssteigerung noch gar nicht vertreten und steigern den Wert der Zyklenzerlegung umso mehr, da es sich um relativ große Programme handelt.

Die MIN-Werte sind durchaus akzeptabel, bis auf die zwei „Ausreißer“ bison1 und moria2. In Zeiten von Multiprozessormaschinen und der Verfügbarkeit von vier Zyklenzerlegungsverfahren stellt dies jedoch kein Manko dar, da die empirische Untersuchung belegt, dass mindestens ein Verfahren genauso schnell – oder schneller – als die Analyse unzerlegter Zyklen ist.

Der Speicherplatzverbrauch ist in den Tabellen nicht mehr angegeben, da er in etwa dem Verbrauch aus Tabelle 7.3 für Starke Zusammenhangskomponenten entspricht.

Die Zahlen zeigen, dass sich aus der Größe der Programme bzw. Chops keine Rückschlüsse auf die Laufzeiten zur Pfadbedingungsberechnung ziehen lassen. Selbst für mehrere tausend Zeilen große Chops sind Zeiten von wenigen Sekunden gängig. Aber auch Minuten stellen für die Sicherheitsanalyse im Vergleich zum manuellen Code-Review einen gravierenden Fortschritt dar.

Der folgende Quelltext demonstriert, warum Laufzeiten von Sekunden überhaupt möglich sind.

```
1 int f(int *a)
2   while (...)
3     if (...)
4       *a=...
5   return *a
```

Abhängigkeitsgraphen sind nicht so intuitiv wie Kontrollflussgraphen. Zwischen dem Zeiger `a` in Zeile 1 und `a` in den Zeilen 4 und 5 liegen Abhängigkeiten. Die Pfadbedingung $PC(a_1, a_5)$ liefert *true*, da die Kante $a_1 \to a_5$ unter keiner Bedingung steht. Wird dieser Pfad während der Tiefensuche vor dem Pfad $a_1 \to a_4 \to^* a_5$ gewählt, erfolgt aufgrund von Absorptionsregeln keine Traversierung der `while`-Schleife und des `if`-Ausdrucks. So werden große Bereiche des Graphen übergangen, da sie die Pfadbedingung nicht präziser machen können.

Ähnliche Laufzeiten über alle Zerlegungsverfahren, wie z.B. bei Palme, deuten darauf hin, dass der Aufwand zur Verwaltung der BDDs dominiert, und die Zyklen eine geringere Bedeutung für die Laufzeit haben.

Zusammenfassend zeigen die Zahlen, dass Pfadbedingungen für zerlegbare Chops mit kleinen Starken Zusammenhangskomponenten schnell be-

Programm	Disj	Konj	Neg	Literal	BddKn	BddVar	rBddVar
mergesort1	2	15	3	18	250	8	8
mergesort2	4	20	1	25	366	13	10
calculator1	0	1	0	2	253	9	2
calculator2	0	2	0	3	209	6	3
triple_des1	3	24	4	28	402	13	8
triple_des2	0	1	1	2	207	4	2
ctags1	-	-	-	-	-	-	-
ctags2	-	-	-	-	-	-	-
assembler1	17	177	96	195	373	21	14
assembler2	5	59	16	65	1094	46	14
gnugo1	-	-	-	-	-	-	-
gnugo2	-	-	-	-	-	-	-
agrep1	0	4	4	5	209	5	5
agrep2	1	12	5	14	832	25	8
wackeltisch1	14	323	126	338	4105	132	38
wackeltisch2	11	148	63	160	4678	128	36
flex1	21	191	85	213	1047	27	15
flex2	0	1	2	2	746	59	2
patch1	-	-	-	-	-	-	-
patch2	-	-	-	-	-	-	-
bison1	2	12	3	15	331	10	6
bison2	-	-	-	-	-	-	-
larn1	1	9	5	11	426	54	6
larn2	1	1	1	3	455	55	2
moria1	-	-	-	-	-	-	-
moria2	0	2	0	3	5334	311	3
palme1	5	21	4	27	581	32	17
palme2	5	93	58	99	874	42	29
palme3	36	367	219	404	2898	65	43
palme4	50	649	305	700	3979	79	69
palme5	730	11722	6535	12453	4364	65	40
palme6	505	6432	2738	6938	3475	74	47

Tabelle 7.9: Pfadbedingung mit Sreedhar/Gao/Lee-Zerlegung

stimmt werden können, die durchschnittliche Zeit liegt im Minutenbereich. Die Chopgröße spielt für die Berechnungszeit eine geringere Rolle, als die tatsächliche Struktur und Kantenzahl der Chops. Große Zyklen stellen generell ein Problem dar, besonders dann, wenn zugleich die Knoten-Kanten-Ratio wie z.B. bei *ctags*, *patch* und *bison* ungünstig ist. Der direkte Vergleich der vier Zyklenzerlegungsverfahren offenbart jedoch, dass eine höhere Zahl von geschachtelten Zyklen in den meisten Fällen eine Steigerung der Performance zur Folge hat. Für Pfadbedingungen nehmen die Zerlegungen nach Havlak und Sreedhar, Gao, Lee die Favoritenrolle ein.

Programm	Disj	Konj	Neg	Literal	BddKn	BddVar	rBddVar
mergesort1	2	15	3	18	250	8	8
mergesort2	4	20	1	25	361	13	10
calculator1	0	1	0	2	239	9	2
calculator2	0	2	0	3	209	6	3
triple_des1	3	24	4	28	398	13	8
triple_des2	0	1	1	2	207	4	2
ctags1	-	-	-	-	-	-	-
ctags2	-	-	-	-	-	-	-
assembler1	17	177	96	195	373	21	14
assembler2	5	59	16	65	1100	46	14
gnugo1	-	-	-	-	-	-	-
gnugo2	-	-	-	-	-	-	-
agrep1	0	4	4	5	209	5	5
agrep2	1	12	5	14	472	25	8
wackeltisch1	12	303	120	316	4175	130	36
wackeltisch2	11	148	63	160	4465	126	36
flex1	23	214	97	238	1670	27	15
flex2	0	1	2	2	746	59	2
patch1	-	-	-	-	-	-	-
patch2	-	-	-	-	-	-	-
bison1	2	12	3	15	331	10	6
bison2	-	-	-	-	-	-	-
larn1	1	9	5	11	426	54	6
larn2	1	1	1	3	455	55	2
moria1	-	-	-	-	-	-	-
moria2	0	2	0	3	5334	311	3
palme1	5	21	4	27	581	32	17
palme2	5	93	58	99	874	42	29
palme3	36	367	219	404	2898	65	43
palme4	50	649	305	700	3979	79	69
palme5	730	11722	6535	12453	4364	65	40
palme6	505	6432	2738	6938	3475	74	47

Tabelle 7.10: Pfadbedingung mit Steensgaard-Zerlegung

Pfadbedingungen und BDDs

Die vier Tabellen 7.9, 7.10, 7.11 und 7.12 zeigen für die berechneten Pfadbedingungen Details zu den resultierenden prädikatenlogischen Formeln und BDDs.

Alle Pfadbedingungen liegen in disjunktiver Normalform vor. Die Tabellen enthalten Spalten für die Anzahl der Disjunktionen (Disj), Konjunktionen (Konj), Negationen (Neg) und Bedingungen (Literal). In einer Baumstruktur stellen die Bedingungen die Blätter dar und setzen sich aus β-, δ- und γ-Bedingungen[8] zusammen. Die Fallstudien zeigen, dass erwartungsgemäß Pfadbedingungen nicht von der Programmgröße abhängen.

[8] Ausführungsbedingungen werden durch γ-Bedingungen repräsentiert.

Programm	Disj	Konj	Neg	Literal	BddKn	BddVar	rBddVar
mergesort1	2	15	3	18	250	8	8
mergesort2	4	20	1	25	365	13	10
calculator1	0	1	0	2	230	9	2
calculator2	0	2	0	3	209	6	3
triple_des1	3	24	4	28	417	13	8
triple_des2	0	1	1	2	207	4	2
ctags1	1	1	2	3	703	81	2
ctags2	1	15	3	17	385	34	9
assembler1	17	177	96	195	373	21	14
assembler2	5	59	16	65	1094	46	14
gnugo1	0	4	0	5	32725	245	5
gnugo2	0	9	2	10	36681	311	10
agrep1	0	4	4	5	209	5	5
agrep2	1	12	5	14	791	25	8
wackeltisch1	14	323	126	338	4419	130	38
wackeltisch2	11	148	63	160	4593	126	36
flex1	23	214	97	238	1559	27	15
flex2	0	1	2	2	746	59	2
patch1	-	-	-	-	-	-	-
patch2	-	-	-	-	-	-	-
bison1	2	12	3	15	331	10	6
bison2	-	-	-	-	-	-	-
larn1	1	9	5	11	426	54	6
larn2	1	1	1	3	455	55	2
moria1	-	-	-	-	-	-	-
moria2	0	2	0	3	5334	311	3
palme1	5	21	4	27	581	32	17
palme2	5	93	58	99	874	42	29
palme3	36	367	219	404	2898	65	43
palme4	50	649	305	700	3979	79	69
palme5	730	11722	6535	12453	4364	65	40
palme6	505	6432	2738	6938	3475	74	47

Tabelle 7.11: Pfadbedingung mit Havlak-Zerlegung

Ein hoher LOC-Wert liefert i.d.R. einen größeren Graphen, jedoch steigt damit nicht automatisch die problematische Knoten-Kanten-Ratio, die für die Performance relevant ist. Die Anzahl an Prädikaten ist wiederum nur von der Art und Funktion des Programms abhängig, nicht prinzipiell von der Größe des Graphen. Graphen bzw. Chops mit wenigen Schleifen, Verzweigungen und Funktionsaufrufen enthalten somit wenig „Ausgangsmaterial“ für Pfadbedingungen, da sie für Ausführungsbedingungen in höherem Maße *true* liefern würden.

Es ist eine Korrelation zwischen der Anzahl an Disjunktionen und Konjunktionen erkennbar. Die Zahl der Konjunktionen liegt bei den Pfadbedingungen um den Faktor 10...20 höher als die Anzahl der Disjunktionen. Auf den Graphen übertragen bedeutet dies, dass Pfadbündel, die unter gemeinsamen Bedingungen stehen, i.d.R. von 10-20 Prädikaten regiert werden. Im

Programm	Disj	Konj	Neg	Literal	BddKn	BddVar	rBddVar
mergesort1	2	15	3	18	250	8	8
mergesort2	4	20	1	25	361	13	10
calculator1	0	1	0	2	257	9	2
calculator2	0	2	0	3	209	6	3
triple_des1	3	24	4	28	398	13	8
triple_des2	0	1	1	2	207	4	2
ctags1	-	-	-	-	-	-	-
ctags2	-	-	-	-	-	-	-
assembler1	17	177	96	195	373	21	14
assembler2	5	59	16	65	1096	46	14
gnugo1	-	-	-	-	-	-	-
gnugo2	-	-	-	-	-	-	-
agrep1	0	4	4	5	209	5	5
agrep2	-	-	-	-	-	-	-
wackeltisch1	14	323	126	338	4581	132	38
wackeltisch2	11	148	63	160	4481	128	36
flex1	-	-	-	-	-	-	-
flex2	0	1	2	2	746	59	2
patch1	-	-	-	-	-	-	-
patch2	-	-	-	-	-	-	-
bison1	2	12	3	15	331	10	6
bison2	-	-	-	-	-	-	-
larn1	1	9	5	11	426	54	6
larn2	1	1	1	3	455	55	2
moria1	-	-	-	-	-	-	-
moria2	0	2	0	3	5334	311	3
palme1	5	21	4	27	581	32	17
palme2	5	93	58	99	874	42	29
palme3	36	367	219	404	2891	64	43
palme4	50	649	305	700	3972	78	69
palme5	730	11722	6535	12453	4364	65	40
palme6	505	6432	2738	6938	3475	74	47

Tabelle 7.12: Pfadbedingung mit Ramalingam-Zerlegung

Abschnitt 8 werden einige Pfadbedingungen vorgestellt und ausführlich besprochen.

Abschnitt 4.4 führte starke und schwache Pfadbedingungen ein, da es zwischen zwei Knoten bis zur maximalen konservativen Approximation *true* mehr als eine Bedingung geben kann. Unterschiedliche Pfadbedingungen sind sehr selten, können jedoch auftreten und werden durch eine konservative Approximation an rückwärtsgerichteten Datenabhängigkeitskanten verursacht (siehe Abschnitt 5.3). Je nach Tiefensuchreihenfolge und Zerlegungsverfahren – auch bei SCCs – können so Approximationen eingeführt und in einem gewissen Rahmen weiterpropagiert werden.

Dieser Effekt ist bei den Pfadbedingungen jedoch nur an zwei Stellen zu sehen. Bei Sreedhar, Gao und Lee besteht flex1 aus 21 Disjunktionen, 191 Konjunktionen, bei allen anderen Verfahren aus 23 Disjunktionen und 214

Konjunktionen. Das zweite Mal tritt der Effekt bei Steensgaard auf, wo wackeltisch1 aus 12 Disjunktionen und 303 Konjunktionen besteht, bei allen anderen Verfahren aus 14 Disjunktionen und 323 Konjunktionen.

Die Tabellen enthalten neben den Daten zur Pfadbedingungsgröße auch Daten zu den Binären Entscheidungsgraphen. Die Spalte BddKn enthält die Anzahl der tatsächlich generierten BDD-Knoten. Diese Zahl ist immer von der Reihenfolge der BDD-Variablen abhängig und wird für Pfadbedingungen von der Tiefensuchreihenfolge zur Pfadaufzählung bestimmt. Die Spalte BddVar zeigt die Anzahl der generierten symbolischen BDD-Variablen, die sich aus β-, δ- und γ-Bedingungen zusammensetzen. rBddVar zeigt die Zahl der resultierenden BDD-Variablen, aus denen die endgültige Pfadbedingung aufgebaut wird. Es gilt daher immer BddVar $\geq$ rBddVar.

Die Pfadbedingungsberechnung mittels einer hierarchischen Zyklenzerlegung ist für 87,5% der Testfälle effizient einsetzbar. Nur 12,5% lassen sich noch nicht effizient berechnen, da ihre Graphstruktur zu komplex ist. Im nächsten Abschnitt wird daher ein Verfahren vorgestellt, das konservativ bleibt, jedoch eine Abschwächung von Pfadbedingungen zulässt.

7.6 Pfadauswertungsbegrenzung

Die Struktur von feingranularen Programmabhängigkeitsgraphen ist so ausgelegt, dass Funktionsaufrufe, die sich in Schleifen befinden, sehr viele bzw. große Zyklen produzieren. Die Ursache liegt an der Propagation von globalen Variablen in Funktionen hinein. Jede Variable wird als Eingangsparameter und ggf. auch Ausgangsparameter realisiert. Summarykanten verbinden Eingangs- mit Ausgangsparametern, und Datenabhängigkeitskanten führen durch Schleifen den Informationsfluss von den Ausgangsparametern zu den Eingangsparametern zurück und produzieren dadurch eine Vielzahl von Zyklen zwischen den Parametern.

Das folgende Codestück entstammt dem Programm „patch“ und der Datei „patch.c“:

```
1      for (
2          open_patch_file (patchname);
3          there_is_another_patch();
4          reinitialize_almost_everything()
5      ) {
```

Der unscheinbare Funktionsaufruf `there_is_another_patch()` enthält im PDG durch globale Parameter 312 Parameterknoten, 15321 Summarykanten und 667 Datenabhängigkeitskanten von Ausgangs- zu Eingangsparametern. Pfade, die eine derartige Funktion über Parametereingangskanten erreichen

und z.B. an Parameterausgangskanten weiterführen oder in die Funktion absteigen, werden mit einem immens großen Zyklus konfrontiert. Dieser gesamte Zyklus mit all seinen Parameterknoten steht unter einer einzigen Ausführungsbedingung und generiert über Summarykanten evtl. weitere Bedingungen derselben Funktion. Da es höchst unwahrscheinlich ist, zu jeder Summarykante eine andere Bedingung zu generieren, werden die meisten Bedingungen entlang der Pfade des Zyklus gleich sein.

Für einen Zyklus S kann daher ausgehend von seinen Zykleneintrittsknoten in_i die Pfadaufzählung nur $k \times |S|, k \leq 1.0$ Schritte durchgeführt werden. Bei $k = 1.0$ beträgt die maximale Pfadlänge innerhalb des Zyklus $|S|$ Schritte und kann nicht größer werden, da sonst der Pfad selbst zyklisch wäre.

Die Pfadbegrenzungsauswertung approximiert immer konservativ, da Pfade nicht abgeschnitten werden, sondern die Erzeugung von Bedingungen nur eine bestimmte Anzahl von Schritten erfolgt und für den Restpfad die Bedingung *true* angenommen wird. Alle unberührten Knoten und Kanten erzeugen daher automatisch die β-, δ- und γ-Bedingungen *true*. Das Ziel dieses Verfahrens ist eine Beschleunigung der Pfadbedingungsberechnung mit relativ geringem Informationsverlust in den resultierenden Pfadbedingungen.

7.6.1 Empirische Untersuchung

Die Tabellen 7.13 (Sreedhar/Gao/Lee, ab Seite 123), 7.14 (Steensgaard, ab Seite 125), 7.15 (Havlak, ab Seite 127) und 7.16 (Ramalingam, ab Seite 130) zeigen die Ergebnisse für die Pfadbedingungen. Für jedes Zyklenzerlegungsverfahren wurde die Pfadbedingungsberechnung mit verschiedenen Werten für k (in Prozent) durchgeführt. Standard sind $k = 5$, $k = 2,5$, $k = 1$ und $k = 0,5$. In besonderen Fällen wurde das Intervall auf $20 \ldots 0,05$ erweitert.

Bei der Wahl von k ist zu beachten, dass geschachtelte Zyklen betrachtet werden und keine globalen SCCs. Für jeden erreichten geschachtelten Zyklus wird die Pfadlänge in Abhängigkeit des aktuellen Zyklus bestimmt, wie weit dieser traversiert werden darf. k kann daher deutlich kleiner als 100% (1,0) gewählt werden, um eine vollständige Pfadaufzählung zu erreichen, da k für $|S \backslash \{S_i | S_i \subset S\}|$ gilt.

Überraschenderweise haben viele Pfadbedingungen dieselbe Präzison wie bei vollständiger Auswertung der Pfade, jedoch benötigen sie z.T. nur einen Bruchteil der bisherigen Berechnungszeit. Die folgende Tabelle zeigt für jedes Zerlegungsverfahren die maximale Testfallüberdeckung und gibt für die 4 Standard-k-Werte die erreichte Präzision im Vergleich zur vollständigen Pfadauswertung an:

Zerlegung	Überdeckung	Präzision $k = 5,0$	$k = 2,5$	$k = 1,0$	$k = 0,5$
Sreedhar,...	100%	95,8%	95,8%	95,8%	95,8%
Steensgaard	100%	95,8%	95,8%	91,7%	91,7%
Havlak	87,5%	96,4%	92,9%	92,9%	92,9%
Ramalingam	100%	95,5%	95,5%	95,5%	95,5%

Mit der Pfadauswertungsbegrenzung können bei drei Zerlegungsverfahren alle Pfadbedingungen realisiert werden, also auch die fehlenden 12,5%. Nur mit dem Verfahren von Havlak ist eine weitere Auswertung nicht mehr möglich. Die Ursache liegt darin, dass Havlak mehr geschachtelte Zyklen als die anderen drei Verfahren berechnet. Die Pfadauswertungsbegrenzung hat eine natürliche untere Grenze, die bei genau einer Kante festgelegt ist. Null Kanten würden dazu führen, dass kein Pfad verfolgt wird. Wenn die Anzahl der geschachtelten Zyklen so hoch ist, dass nach jeder Kante ein neuer geschachtelter Zyklus erreicht wird, erfolgt eine reguläre Pfadaufzählung, ohne dass jemals eine Pfadlänge reduziert werden kann.

Die Präzision ist trotz der unvollständigen Auswertung mit über 90% sehr gut. Nur die drei Testfälle „wackeltisch1“, „ctags2“ und „flex1“ verlieren mit abnehmendem k an Präzision.

Ein Prüfingenieur erhält mit der Pfadauswertungsbegrenzung ein flexibles Werkzeug, das ihn in die Lage versetzt, sehr schnell konservativ abgeschätzte Ergebnisse zu produzieren, ohne die vollständige Berechnung abwarten zu müssen. Analysen sollten daher mit einem kleinen k begonnen werden, das bei Bedarf vergrößert wird. Durch die hohe Präzision ist ggf. schon bei kleinem k eine Beurteilung der Pfadbedingung möglich, z.B. genau dann, wenn verdächtige Prädikate schon in der konservativen Bedingung auftreten. Die Fallstudie „wackeltisch2“ wird in Abschnitt 8 ausführlich behandelt. Ohne Pfadauswertungsbegrenzung benötigt z.B. der Sreedhar/Gao/Lee-Algorithmus 344 Sekunden zur Berechnung, der Steensgaard-Algorithmus sogar 418 Sekunden. Mit Pfadauswertungsbegrenzung und $k = 0,5$ liegt das Ergebnis mit identischer Präzision bereits nach 1 Sekunde vor.

Die Parametrisierung der zu berücksichtigenden Pfadlänge ermöglicht somit eine effektive und konservative Abschätzung der Pfadbedingung, ohne eine exponentielle Laufzeit innerhalb der Zyklen in Kauf nehmen zu müssen.

Pfadauswertungsbegrenzung mit Sreedhar/Gao/Lee-Zerlegung

Tabelle 7.13: Pfadauswertungsbegrenzung mit Sreedhar/Gao/Lee-Zerlegung

Programm	Disj	Konj	Neg	Literal	Kn	Var	rVar	(s)	k(%)
mergesort1	2	15	3	18	250	8	8	0	5
	2	15	3	18	250	8	8	0	2,5
	2	15	3	18	250	8	8	0	1
	2	15	3	18	250	8	8	0	0,5
mergesort2	4	20	1	25	366	13	10	0	5
	4	20	1	25	366	13	10	0	2,5
	4	20	1	25	366	13	10	0	1
	4	20	1	25	366	13	10	1	0,5
calculator1	0	1	0	2	225	9	2	0	5
	0	1	0	2	217	9	2	1	2,5
	0	1	0	2	217	9	2	0	1
	0	1	0	2	217	9	2	0	0,5
calculator2	0	2	0	3	209	6	3	0	5
	0	2	0	3	209	6	3	1	2,5
	0	2	0	3	209	6	3	0	1
	0	2	0	3	209	6	3	1	0,5
triple_des1	3	24	4	28	393	13	8	0	5
	3	24	4	28	393	13	8	0	2,5
	3	24	4	28	393	13	8	0	1
	3	24	4	28	393	13	8	0	0,5
triple_des2	0	1	1	2	207	4	2	0	5
	0	1	1	2	207	4	2	1	2,5
	0	1	1	2	207	4	2	0	1
	0	1	1	2	207	4	2	0	0,5
ctags1	1	1	2	3	534	80	2	8	5
	1	1	2	3	933	80	2	7	2,5
	1	1	2	3	933	80	2	7	1
	1	1	2	3	933	80	2	7	0,5
ctags2	1	15	3	17	315	34	9	4	5
	1	11	3	13	367	34	7	4	2,5
	1	11	3	13	367	34	7	4	1
	1	11	3	13	367	34	7	4	0,5
assembler1	17	177	96	195	373	21	14	6	5
	17	177	96	195	373	21	14	5	2,5
	17	177	96	195	373	21	14	6	1
	17	177	96	195	373	21	14	5	0,5
assembler2	5	59	16	65	978	39	14	3	5
	5	59	16	65	978	39	14	4	2,5
	5	59	16	65	978	39	14	4	1
	5	59	16	65	978	39	14	3	0,5
gnugo1	0	4	0	5	20245	245	5	23	5
	0	4	0	5	21808	243	5	5	2,5
	0	4	0	5	11732	243	5	5	1
	0	4	0	5	10947	243	5	4	0,5
gnugo2	0	9	2	10	13810	311	10	28	5
	0	9	2	10	40714	311	10	7	2,5
	0	9	2	10	32009	311	10	6	1
	0	9	2	10	31616	311	10	6	0,5
agrep1	0	4	4	5	209	5	5	1	5
	0	4	4	5	209	5	5	1	2,5
	0	4	4	5	209	5	5	1	1
	0	4	4	5	209	5	5	0	0,5
agrep2	1	12	5	14	815	25	8	27	5
	1	12	5	14	1551	25	8	2	2,5
	1	12	5	14	654	25	8	1	1
	1	12	5	14	437	25	8	1	0,5

Tabelle 7.13: Pfadauswertungsbegrenzung mit Sreedhar/Gao/Lee-Zerlegung

Programm	Disj	Konj	Neg	Literal	Kn	Var	rVar	(s)	k(%)
wackeltisch1	14	323	126	338	4990	129	38	24	10
	14	323	126	338	4897	129	38	21	8,5
	11	175	77	187	5028	129	36	12	5
	0	4	3	5	3721	82	5	2	2,5
	0	4	3	5	384	21	5	1	1
	0	4	3	5	384	21	5	1	0,5
wackeltisch2	11	148	63	160	4156	125	36	221	5
	11	148	63	160	2157	64	36	2	2,5
	11	148	63	160	1303	60	36	2	1
	11	148	63	160	1303	60	36	1	0,5
flex1	21	191	85	213	1309	27	15	5	5
	21	191	85	213	866	27	15	2	2,5
	21	191	85	213	713	27	15	2	1
	21	191	85	213	713	27	15	2	0,5
flex2	0	1	2	2	746	59	2	18	5
	0	1	2	2	746	59	2	18	2,5
	0	1	2	2	746	59	2	18	1
	0	1	2	2	746	59	2	18	0,5
patch1	-	-	-	-	-	-	-	-	5
	-	-	-	-	-	-	-	-	2,5
	-	-	-	-	-	-	-	-	1
	-	-	-	-	-	-	-	-	0,5
	103	1734	967	1838	1293	66	22	15	0,1
	103	1734	967	1838	1293	66	22	14	0,05
patch2	-	-	-	-	-	-	-	-	5
	-	-	-	-	-	-	-	-	2,5
	15	254	64	270	1077	157	28	68	1
	15	254	64	270	1041	157	28	61	0,5
bison1	2	12	3	15	331	10	6	3	5
	2	12	3	15	331	10	6	3	2,5
	2	12	3	15	331	10	6	4	1
	2	12	3	15	331	10	6	3	0,5
bison2	-	-	-	-	-	-	-	-	5
	-	-	-	-	-	-	-	-	2,5
	0	9	3	10	197526	341	10	510	1
	0	9	3	10	191271	341	10	101	0,5
larn1	1	9	5	11	426	54	6	3	5
	1	9	5	11	426	54	6	4	2,5
	1	9	5	11	426	54	6	3	1
	1	9	5	11	426	54	6	3	0,5
larn2	1	1	1	3	455	55	2	4	5
	1	1	1	3	455	55	2	4	2,5
	1	1	1	3	455	55	2	3	1
	1	1	1	3	455	55	2	4	0,5
moria1	-	-	-	-	-	-	-	-	5
	-	-	-	-	-	-	-	-	2,5
	-	-	-	-	-	-	-	-	1
	-	-	-	-	-	-	-	-	0,5
	0	2	1	3	672	104	3	24	0,1
moria2	0	2	0	3	5334	311	3	30	5
	0	2	0	3	5334	311	3	30	2,5
	0	2	0	3	5334	311	3	31	1
	0	2	0	3	5334	311	3	30	0,5
palme1	5	21	4	27	581	32	17	1	5
	5	21	4	27	581	32	17	0	2,5
	5	21	4	27	581	32	17	0	1
	5	21	4	27	581	32	17	1	0,5
palme2	5	93	58	99	874	42	29	1	5
	5	93	58	99	874	42	29	0	2,5

Tabelle 7.13: Pfadauswertungsbegrenzung mit Sreedhar/Gao/Lee-Zerlegung

Programm	Disj	Konj	Neg	Literal	Kn	Var	rVar	(s)	k(%)
palme2	5	93	58	99	874	42	29	1	1
	5	93	58	99	874	42	29	1	0,5
palme3	36	367	219	404	2891	64	43	4	5
	36	367	219	404	2891	64	43	4	2,5
	36	367	219	404	2891	64	43	5	1
	36	367	219	404	2891	64	43	4	0,5
palme4	50	649	305	700	3947	75	69	262	5
	50	649	305	700	3947	75	69	264	2,5
	50	649	305	700	3947	75	69	300	1
	50	649	305	700	3947	75	69	262	0,5
palme5	730	11722	6535	12453	4373	65	40	22	5
	730	11722	6535	12453	4373	65	40	22	2,5
	730	11722	6535	12453	4373	65	40	22	1
	730	11722	6535	12453	4373	65	40	22	0,5
palme6	505	6432	2738	6938	3475	72	47	203	5
	505	6432	2738	6938	3475	72	47	203	2,5
	505	6432	2738	6938	3475	72	47	205	1
	505	6432	2738	6938	3475	72	47	203	0,5

Pfadauswertungsbegrenzung mit Steensgaard-Zerlegung

Tabelle 7.14: Pfadauswertungsbegrenzung mit Steensgaard-Zerlegung

Programm	Disj	Konj	Neg	Literal	Kn	Var	rVar	(s)	k(%)
mergesort1	2	15	3	18	250	8	8	0	5
	2	15	3	18	250	8	8	0	2,5
	2	15	3	18	250	8	8	0	1
	2	15	3	18	250	8	8	0	0,5
mergesort2	4	20	1	25	361	13	10	0	5
	4	20	1	25	361	13	10	0	2,5
	4	20	1	25	361	13	10	0	1
	4	20	1	25	361	13	10	0	0,5
calculator1	0	1	0	2	222	9	2	0	5
	0	1	0	2	217	9	2	0	2,5
	0	1	0	2	217	9	2	0	1
	0	1	0	2	217	9	2	0	0,5
calculator2	0	2	0	3	209	6	3	0	5
	0	2	0	3	209	6	3	0	2,5
	0	2	0	3	209	6	3	0	1
	0	2	0	3	209	6	3	0	0,5
triple_des1	3	24	4	28	389	13	8	0	5
	3	24	4	28	389	13	8	0	2,5
	3	24	4	28	389	13	8	0	1
	3	24	4	28	389	13	8	0	0,5
triple_des2	0	1	1	2	207	4	2	0	5
	0	1	1	2	207	4	2	0	2,5
	0	1	1	2	207	4	2	0	1
	0	1	1	2	207	4	2	0	0,5
ctags1	1	1	2	3	535	80	2	8	5
	1	1	2	3	933	80	2	7	2,5
	1	1	2	3	933	80	2	7	1
	1	1	2	3	933	80	2	6	0,5

Tabelle 7.14: Pfadauswertungsbegrenzung mit Steensgaard-Zerlegung

Programm	Disj	Konj	Neg	Literal	Kn	Var	rVar	(s)	k(%)
ctags2	1	15	3	17	303	34	9	4	5
	1	11	3	13	367	34	7	4	2,5
	1	11	3	13	367	34	7	4	1
	1	11	3	13	367	34	7	4	0,5
assembler1	17	177	96	195	373	21	14	5	5
	17	177	96	195	373	21	14	4	2,5
	17	177	96	195	373	21	14	4	1
	17	177	96	195	373	21	14	5	0,5
assembler2	5	59	16	65	978	39	14	2	5
	5	59	16	65	978	39	14	3	2,5
	5	59	16	65	978	39	14	3	1
	5	59	16	65	978	39	14	3	0,5
gnugo1	0	4	0	5	17920	245	5	57	5
	0	4	0	5	10343	244	5	8	2,5
	0	4	0	5	10857	243	5	4	1
	0	4	0	5	20593	243	5	4	0,5
gnugo2	0	9	2	10	13810	311	10	28	5
	0	9	2	10	40714	311	10	7	2,5
	0	9	2	10	32009	311	10	6	1
	0	9	2	10	31616	311	10	5	0,5
agrep1	0	4	4	5	209	5	5	1	5
	0	4	4	5	209	5	5	0	2,5
	0	4	4	5	209	5	5	1	1
	0	4	4	5	209	5	5	0	0,5
agrep2	1	12	5	14	446	25	8	24	5
	1	12	5	14	446	25	8	24	2,5
	1	12	5	14	437	25	8	1	1
	1	12	5	14	437	25	8	1	0,5
wackeltisch1	12	303	120	316	3831	126	36	287	10
	12	303	120	316	3853	126	36	279	8,5
	12	291	114	304	4464	125	36	178	5
	21	449	175	471	7825	80	42	84	2,5
	0	4	3	5	384	21	5	1	1
	0	4	3	5	384	21	5	1	0,5
wackeltisch2	11	148	63	160	4437	121	36	207	5
	11	148	63	160	4510	121	36	20	2,5
	11	148	63	160	1303	60	36	2	1
	11	148	63	160	1303	60	36	1	0,5
flex1	23	214	97	238	1625	27	15	5	5
	23	214	97	238	1704	27	15	2	2,5
	21	191	85	213	713	27	15	2	1
	21	191	85	213	713	27	15	1	0,5
flex2	0	1	2	2	746	59	2	10	5
	0	1	2	2	746	59	2	10	2,5
	0	1	2	2	746	59	2	11	1
	0	1	2	2	746	59	2	10	0,5
patch1	-	-	-	-	-	-	-	-	5
	-	-	-	-	-	-	-	-	2,5
	-	-	-	-	-	-	-	-	1
	-	-	-	-	-	-	-	-	0,5
	103	1734	967	1838	1293	66	22	17	0,1
	103	1734	967	1838	1293	66	22	16	0,05
patch2	-	-	-	-	-	-	-	-	5
	-	-	-	-	-	-	-	-	2,5
	-	-	-	-	-	-	-	-	1
	15	254	64	270	1041	157	28	54	0,5
bison1	2	12	3	15	331	10	6	5	5
	2	12	3	15	331	10	6	5	2,5
	2	12	3	15	331	10	6	5	1

Tabelle 7.14: Pfadauswertungsbegrenzung mit Steensgaard-Zerlegung

Programm	Disj	Konj	Neg	Literal	Kn	Var	rVar	(s)	k(%)
bison1	2	12	3	15	331	10	6	5	0,5
bison2	-	-	-	-	-	-	-	-	5
	-	-	-	-	-	-	-	-	2,5
	0	9	3	10	197542	341	10	511	1
	0	9	3	10	191281	341	10	100	0,5
larn1	1	9	5	11	426	54	6	2	5
	1	9	5	11	426	54	6	2	2,5
	1	9	5	11	426	54	6	2	1
	1	9	5	11	426	54	6	2	0,5
larn2	1	1	1	3	455	55	2	2	5
	1	1	1	3	455	55	2	2	2,5
	1	1	1	3	455	55	2	2	1
	1	1	1	3	455	55	2	2	0,5
moria1	-	-	-	-	-	-	-	-	5
	-	-	-	-	-	-	-	-	2,5
	-	-	-	-	-	-	-	-	1
	-	-	-	-	-	-	-	-	0,5
	0	2	1	3	672	104	3	78	0,1
moria2	0	2	0	3	5334	311	3	58	5
	0	2	0	3	5334	311	3	58	2,5
	0	2	0	3	5334	311	3	57	1
	0	2	0	3	5334	311	3	57	0,5
palme1	5	21	4	27	581	32	17	0	5
	5	21	4	27	581	32	17	0	2,5
	5	21	4	27	581	32	17	1	1
	5	21	4	27	581	32	17	1	0,5
palme2	5	93	58	99	874	42	29	1	5
	5	93	58	99	874	42	29	0	2,5
	5	93	58	99	874	42	29	1	1
	5	93	58	99	874	42	29	1	0,5
palme3	36	367	219	404	2891	64	43	4	5
	36	367	219	404	2891	64	43	5	2,5
	36	367	219	404	2891	64	43	4	1
	36	367	219	404	2891	64	43	4	0,5
palme4	50	649	305	700	3947	75	69	260	5
	50	649	305	700	3947	75	69	261	2,5
	50	649	305	700	3947	75	69	260	1
	50	649	305	700	3947	75	69	261	0,5
palme5	730	11722	6535	12453	4373	65	40	22	5
	730	11722	6535	12453	4373	65	40	21	2,5
	730	11722	6535	12453	4373	65	40	21	1
	730	11722	6535	12453	4373	65	40	22	0,5
palme6	505	6432	2738	6938	3475	72	47	214	5
	505	6432	2738	6938	3475	72	47	202	2,5
	505	6432	2738	6938	3475	72	47	202	1
	505	6432	2738	6938	3475	72	47	207	0,5

Pfadauswertungsbegrenzung mit Havlak-Zerlegung

Tabelle 7.15: Pfadauswertungsbegrenzung mit Havlak-Zerlegung

Programm	Disj	Konj	Neg	Literal	Kn	Var	rVar	(s)	k(%)
mergesort1	2	15	3	18	250	8	8	0	5
	2	15	3	18	250	8	8	0	2,5

Tabelle 7.15: Pfadauswertungsbegrenzung mit Havlak-Zerlegung

Programm	Disj	Konj	Neg	Literal	Kn	Var	rVar	(s)	k(%)
mergesort1	2	15	3	18	250	8	8	0	1
	2	15	3	18	250	8	8	0	0,5
mergesort2	4	20	1	25	365	13	10	0	5
	4	20	1	25	365	13	10	0	2,5
	4	20	1	25	365	13	10	0	1
	4	20	1	25	365	13	10	0	0,5
calculator1	0	1	0	2	223	9	2	1	5
	0	1	0	2	217	9	2	0	2,5
	0	1	0	2	217	9	2	0	1
	0	1	0	2	217	9	2	1	0,5
calculator2	0	2	0	3	209	6	3	0	5
	0	2	0	3	209	6	3	0	2,5
	0	2	0	3	209	6	3	0	1
	0	2	0	3	209	6	3	0	0,5
triple_des1	3	24	4	28	400	13	8	0	5
	3	24	4	28	400	13	8	0	2,5
	3	24	4	28	400	13	8	0	1
	3	24	4	28	400	13	8	0	0,5
triple_des2	0	1	1	2	207	4	2	1	5
	0	1	1	2	207	4	2	0	2,5
	0	1	1	2	207	4	2	0	1
	0	1	1	2	207	4	2	0	0,5
ctags1	1	1	2	3	693	81	2	12	5
	1	1	2	3	709	81	2	14	2,5
	1	1	2	3	709	81	2	12	1
	1	1	2	3	709	81	2	13	0,5
ctags2	1	15	3	17	301	34	9	9	5
	1	11	3	13	367	34	7	7	2,5
	1	11	3	13	367	34	7	7	1
	1	11	3	13	367	34	7	7	0,5
assembler1	17	177	96	195	373	21	14	4	5
	17	177	96	195	373	21	14	5	2,5
	17	177	96	195	373	21	14	5	1
	17	177	96	195	373	21	14	4	0,5
assembler2	5	59	16	65	1094	46	14	4	5
	5	59	16	65	1094	46	14	4	2,5
	5	59	16	65	1094	46	14	4	1
	5	59	16	65	1094	46	14	4	0,5
gnugo1	0	4	0	5	45430	244	5	65	5
	0	4	0	5	20070	244	5	56	2,5
	0	4	0	5	7046	244	5	23	1
	0	4	0	5	7839	244	5	23	0,5
gnugo2	0	9	2	10	13810	311	10	28	5
	0	9	2	10	40714	311	10	7	2,5
	0	9	2	10	32009	311	10	5	1
	0	9	2	10	31616	311	10	6	0,5
agrep1	0	4	4	5	209	5	5	1	5
	0	4	4	5	209	5	5	0	2,5
	0	4	4	5	209	5	5	1	1
	0	4	4	5	209	5	5	1	0,5
agrep2	1	12	5	14	732	25	8	3	5
	1	12	5	14	703	25	8	3	2,5
	1	12	5	14	1351	25	8	2	1
	1	12	5	14	595	25	8	2	0,5
wackeltisch1	14	323	126	338	4675	130	38	18	20
	9	160	69	170	4660	130	36	18	15
	9	160	69	170	4534	130	36	18	10
	9	145	69	155	4526	130	33	17	8,5
	9	145	69	155	4526	130	33	18	5

Tabelle 7.15: Pfadauswertungsbegrenzung mit Havlak-Zerlegung

Programm	Disj	Konj	Neg	Literal	Kn	Var	rVar	(s)	k(%)
wackeltisch1	11	166	80	178	4554	130	33	17	2,5
	0	4	3	5	3096	96	5	11	1
	0	4	3	5	3096	96	5	10	0,5
wackeltisch2	11	148	63	160	4911	126	36	17	5
	11	148	63	160	4215	126	36	17	2,5
	11	148	63	160	1291	65	36	6	1
	11	148	63	160	1291	65	36	5	0,5
flex1	23	214	97	238	1455	27	15	6	5
	21	191	85	213	1261	27	15	6	2,5
	21	191	85	213	1306	27	15	5	1
	21	191	85	213	1306	27	15	6	0,5
flex2	0	1	2	2	746	59	2	16	5
	0	1	2	2	746	59	2	15	2,5
	0	1	2	2	746	59	2	15	1
	0	1	2	2	746	59	2	16	0,5
patch1	-	-	-	-	-	-	-	-	5
	-	-	-	-	-	-	-	-	2,5
	-	-	-	-	-	-	-	-	1
	-	-	-	-	-	-	-	-	0,5
	-	-	-	-	-	-	-	-	0,1
	-	-	-	-	-	-	-	-	0,05
patch2	-	-	-	-	-	-	-	-	5
	-	-	-	-	-	-	-	-	2,5
	-	-	-	-	-	-	-	-	1
	-	-	-	-	-	-	-	-	0,5
bison1	2	12	3	15	331	10	6	33	5
	2	12	3	15	331	10	6	33	2,5
	2	12	3	15	331	10	6	32	1
	2	12	3	15	331	10	6	33	0,5
bison2	-	-	-	-	-	-	-	-	5
	-	-	-	-	-	-	-	-	2,5
	-	-	-	-	-	-	-	-	1
	-	-	-	-	-	-	-	-	0,5
larn1	1	9	5	11	426	54	6	4	5
	1	9	5	11	426	54	6	3	2,5
	1	9	5	11	426	54	6	4	1
	1	9	5	11	426	54	6	4	0,5
larn2	1	1	1	3	455	55	2	3	5
	1	1	1	3	455	55	2	4	2,5
	1	1	1	3	455	55	2	4	1
	1	1	1	3	455	55	2	3	0,5
moria1	-	-	-	-	-	-	-	-	5
	-	-	-	-	-	-	-	-	2,5
	-	-	-	-	-	-	-	-	1
	-	-	-	-	-	-	-	-	0,5
	-	-	-	-	-	-	-	-	0,1
moria2	0	2	0	3	5334	311	3	482	5
	0	2	0	3	5334	311	3	481	2,5
	0	2	0	3	5334	311	3	482	1
	0	2	0	3	5334	311	3	484	0,5
palme1	5	21	4	27	581	32	17	1	5
	5	21	4	27	581	32	17	2	2,5
	5	21	4	27	581	32	17	2	1
	5	21	4	27	581	32	17	1	0,5
palme2	5	93	58	99	874	42	29	2	5
	5	93	58	99	874	42	29	2	2,5
	5	93	58	99	874	42	29	2	1
	5	93	58	99	874	42	29	2	0,5
palme3	36	367	219	404	2898	65	43	6	5

Tabelle 7.15: Pfadauswertungsbegrenzung mit Havlak-Zerlegung

Programm	Disj	Konj	Neg	Literal	Kn	Var	rVar	(s)	k(%)
palme3	36	367	219	404	2898	65	43	6	2,5
	36	367	219	404	2898	65	43	6	1
	36	367	219	404	2898	65	43	6	0,5
palme4	50	649	305	700	3979	79	69	292	5
	50	649	305	700	3979	79	69	290	2,5
	50	649	305	700	3979	79	69	288	1
	50	649	305	700	3979	79	69	294	0,5
palme5	730	11722	6535	12453	4373	65	40	26	5
	730	11722	6535	12453	4373	65	40	26	2,5
	730	11722	6535	12453	4373	65	40	26	1
	730	11722	6535	12453	4373	65	40	25	0,5
palme6	505	6432	2738	6938	3475	74	47	225	5
	505	6432	2738	6938	3475	74	47	225	2,5
	505	6432	2738	6938	3475	74	47	225	1
	505	6432	2738	6938	3475	74	47	226	0,5

Pfadauswertungsbegrenzung mit Ramalingam-Zerlegung

Tabelle 7.16: Pfadauswertungsbegrenzung mit Ramalingam-Zerlegung

Programm	Disj	Konj	Neg	Literal	Kn	Var	rVar	(s)	k(%)
mergesort1	2	15	3	18	250	8	8	0	5
	2	15	3	18	250	8	8	0	2,5
	2	15	3	18	250	8	8	0	1
	2	15	3	18	250	8	8	0	0,5
mergesort2	4	20	1	25	361	13	10	0	5
	4	20	1	25	361	13	10	0	2,5
	4	20	1	25	361	13	10	1	1
	4	20	1	25	361	13	10	0	0,5
calculator1	0	1	0	2	217	9	2	0	5
	0	1	0	2	217	9	2	0	2,5
	0	1	0	2	217	9	2	0	1
	0	1	0	2	217	9	2	0	0,5
calculator2	0	2	0	3	209	6	3	0	5
	0	2	0	3	209	6	3	0	2,5
	0	2	0	3	209	6	3	0	1
	0	2	0	3	209	6	3	0	0,5
triple_des1	3	24	4	28	389	13	8	1	5
	3	24	4	28	389	13	8	0	2,5
	3	24	4	28	389	13	8	0	1
	3	24	4	28	389	13	8	0	0,5
triple_des2	0	1	1	2	207	4	2	0	5
	0	1	1	2	207	4	2	0	2,5
	0	1	1	2	207	4	2	0	1
	0	1	1	2	207	4	2	0	0,5
ctags1	1	1	2	3	534	80	2	13	5
	1	1	2	3	933	80	2	12	2,5
	1	1	2	3	933	80	2	12	1
	1	1	2	3	933	80	2	12	0,5
ctags2	1	15	3	17	303	34	9	7	5
	1	11	3	13	367	34	7	7	2,5
	1	11	3	13	367	34	7	7	1
	1	11	3	13	367	34	7	7	0,5

Tabelle 7.16: Pfadauswertungsbegrenzung mit Ramalingam-Zerlegung

Programm	Disj	Konj	Neg	Literal	Kn	Var	rVar	(s)	k(%)
assembler1	17	177	96	195	373	21	14	5	5
	17	177	96	195	373	21	14	5	2,5
	17	177	96	195	373	21	14	5	1
	17	177	96	195	373	21	14	5	0,5
assembler2	5	59	16	65	978	39	14	4	5
	5	59	16	65	978	39	14	4	2,5
	5	59	16	65	978	39	14	4	1
	5	59	16	65	978	39	14	3	0,5
gnugo1	0	4	0	5	16907	245	5	11	5
	0	4	0	5	21207	243	5	10	2,5
	0	4	0	5	11112	243	5	9	1
	0	4	0	5	10907	243	5	9	0,5
gnugo2	0	9	2	10	13810	311	10	28	5
	0	9	2	10	40714	311	10	6	2,5
	0	9	2	10	32009	311	10	6	1
	0	9	2	10	31616	311	10	5	0,5
agrep1	0	4	4	5	209	5	5	1	5
	0	4	4	5	209	5	5	0	2,5
	0	4	4	5	209	5	5	1	1
	0	4	4	5	209	5	5	1	0,5
agrep2	1	12	5	14	521	25	8	1	5
	1	12	5	14	451	25	8	1	2,5
	1	12	5	14	437	25	8	1	1
	1	12	5	14	437	25	8	1	0,5
wackeltisch1	14	323	126	338	4417	129	38	202	10
	14	315	122	330	4560	129	38	190	8,5
	11	177	78	189	3395	84	36	10	5
	0	4	3	5	3877	82	5	6	2,5
	0	4	3	5	384	21	5	5	1
	0	4	3	5	384	21	5	5	0,5
wackeltisch2	11	148	63	160	1364	66	36	7	5
	11	148	63	160	2141	64	36	6	2,5
	11	148	63	160	1303	60	36	5	1
	11	148	63	160	1303	60	36	6	0,5
flex1	21	191	85	213	1246	27	15	5	5
	21	191	85	213	895	27	15	5	2,5
	21	191	85	213	713	27	15	5	1
	21	191	85	213	713	27	15	4	0,5
flex2	0	1	2	2	746	59	2	15	5
	0	1	2	2	746	59	2	15	2,5
	0	1	2	2	746	59	2	15	1
	0	1	2	2	746	59	2	15	0,5
patch1	-	-	-	-	-	-	-	-	5
	-	-	-	-	-	-	-	-	2,5
	-	-	-	-	-	-	-	-	1
	-	-	-	-	-	-	-	-	0,5
	103	1734	967	1838	1293	66	22	89	0,1
	103	1734	967	1838	1293	66	22	85	0,05
patch2	-	-	-	-	-	-	-	-	5
	-	-	-	-	-	-	-	-	2,5
	15	254	64	270	1071	157	28	148	1
	15	254	64	270	1041	157	28	140	0,5
bison1	2	12	3	15	331	10	6	31	5
	2	12	3	15	331	10	6	30	2,5
	2	12	3	15	331	10	6	31	1
	2	12	3	15	331	10	6	32	0,5
bison2	-	-	-	-	-	-	-	-	5
	-	-	-	-	-	-	-	-	2,5
	0	9	3	10	191702	341	10	539	1

Tabelle 7.16: Pfadauswertungsbegrenzung mit Ramalingam-Zerlegung

Programm	Disj	Konj	Neg	Literal	Kn	Var	rVar	(s)	k(%)
bison2	0	9	3	10	190886	341	10	126	0,5
larn1	1	9	5	11	426	54	6	3	5
	1	9	5	11	426	54	6	4	2,5
	1	9	5	11	426	54	6	4	1
	1	9	5	11	426	54	6	3	0,5
larn2	1	1	1	3	455	55	2	3	5
	1	1	1	3	455	55	2	4	2,5
	1	1	1	3	455	55	2	4	1
	1	1	1	3	455	55	2	3	0,5
moria1	-	-	-	-	-	-	-	-	5
	-	-	-	-	-	-	-	-	2,5
	-	-	-	-	-	-	-	-	1
	0	2	1	3	674	104	3	954	0,5
	0	2	1	3	672	104	3	1010	0,1
moria2	0	2	0	3	5334	311	3	481	5
	0	2	0	3	5334	311	3	482	2,5
	0	2	0	3	5334	311	3	483	1
	0	2	0	3	5334	311	3	482	0,5
palme1	5	21	4	27	581	32	17	2	5
	5	21	4	27	581	32	17	2	2,5
	5	21	4	27	581	32	17	1	1
	5	21	4	27	581	32	17	2	0,5
palme2	5	93	58	99	874	42	29	2	5
	5	93	58	99	874	42	29	2	2,5
	5	93	58	99	874	42	29	2	1
	5	93	58	99	874	42	29	2	0,5
palme3	36	367	219	404	2891	64	43	6	5
	36	367	219	404	2891	64	43	6	2,5
	36	367	219	404	2891	64	43	6	1
	36	367	219	404	2891	64	43	6	0,5
palme4	50	649	305	700	3947	75	69	290	5
	50	649	305	700	3947	75	69	290	2,5
	50	649	305	700	3947	75	69	290	1
	50	649	305	700	3947	75	69	288	0,5
palme5	730	11722	6535	12453	4373	65	40	26	5
	730	11722	6535	12453	4373	65	40	26	2,5
	730	11722	6535	12453	4373	65	40	26	1
	730	11722	6535	12453	4373	65	40	25	0,5
palme6	505	6432	2738	6938	3475	72	47	227	5
	505	6432	2738	6938	3475	72	47	227	2,5
	505	6432	2738	6938	3475	72	47	226	1
	505	6432	2738	6938	3475	72	47	225	0,5

7.7 Fazit

Die empirischen Untersuchungen haben deutlich gezeigt, dass sich interprozedurale Pfadbedingungen mit obigen Techniken trotz der prinzipiell exponentiellen Komplexität für reale Programme in kurzer Zeit berechnen lassen. Die durchschnittlichen Berechnungszeiten liegen bei Sekunden bis wenigen Minuten und machen Pfadbedingungen sogar für den interaktiven Einsatz interessant.

Der Schlüssel zum erfolgreichen Einsatz liegt in der Kombination von Binären Entscheidungsgraphen, einer interprozeduralen Analyse mit wiederverwendbaren Pfadbedingungen, einer Intervallanalyse von Zyklen, der Anwendung von Verfahren aus der Spieletheorie und der Nutzung einer Pfadauswertungsbegrenzung für unzerlegbare Zyklen.

Ohne die Kombination dieser Techniken ließen sich bisher nur Pfadbedingungen für Chops von weniger als 200 Knoten berechnen. Die Untersuchungen haben gezeigt, dass jetzt sogar Chops mit 30000 Knoten kein Problem mehr darstellen müssen. Für die Berechnungszeit ist zum einen die Struktur des Chops verantwortlich und zum anderen die Anzahl der enthaltenen Prädikate. Die Struktur ist wiederum von der Anzahl der Kanten und der Art ihrer Verknüpfung abhängig. Globale Variablen erzeugen i.d.R. viele Funktionsparameter und damit zugleich große Zyklen an Aufrufkontexten. Große Schleifenrümpfe erzeugen erwartungsgemäß auch große Zyklen. Die Verwendung von Zeigern erhöht die Anzahl an Kanten, da sowohl Repräsentanten für referenzierte als auch dereferenzierte Objekte erzeugt werden.

Aufgrund dieser Vielfalt an Faktoren, die die Struktur von Chops beeinflussen, wird kein einzelnes Zerlegungsverfahren als Favorit vorgeschlagen. Es bietet sich an, Chops zuerst nach Havlak zu zerlegen, da aufgrund der höchsten Hierarchieebenen die Pfadbedingungsberechnung bei einem Teil der Testfälle mit Abstand am besten abschnitt. Auf der anderen Seite war diese hohe Anzahl an geschachtelten Zyklen zugleich ein Nachteil bei anderen Testfällen. Daher sind mindestens zwei Zyklenzerlegungsverfahren notwendig, um Pfadbedingungen effizient zu berechnen. Von den anderen drei Verfahren ist die Zerlegung nach Sreedhar, Gao, Lee am besten, da sie mit konstant 95,8% bei der Pfadauswertungsbegrenzung die höchste Präzision liefert. Zerlegungen nach Steensgaard und Ramalingam bieten sich in Einzelfällen an und sollten – neben den oben genannten – dann eingesetzt werden, wenn konkurrierende Pfadbedingungsberechnungen auf Mehrprozessormaschinen durchgeführt werden.

Die Pfadbegrenzungsauswertung liefert eine überraschend hohe Präzision und sollte daher immer zum Einsatz kommen. Da sich die Pfadbedingung mit steigendem k-Wert einem Fixpunkt nähert, kann mit sehr kleinem k begonnen werden, das anschließend stückweise erhöht wird, wenn die Notwendigkeit einer höheren Präzision besteht.

Kapitel 8

Fallstudien

In diesem Kapitel wird die bisherige Theorie auf real existierende Programme angewandt, um zu zeigen, wie Pfadbedingungen in der Praxis einsetzbar sind und welche Erkenntnisse sie liefern. Das Kapitel bedient sich dabei der Sichtweise eines potentiellen Prüfingenieurs, um Programme zu untersuchen. Beim ersten Programm handelt es sich um ein Echtzeitsystem, beim zweiten um eine elektronische Geldbörse. Beide Programme sind keine echten sicherheitskritischen Programme, wie z.B. Steuerungssoftware für Autopiloten in Flugzeugen oder klinische Tomographen, bei denen die Gesundheit von Personen gefährdet werden kann. Jedoch zeigen sie Charakteristiken echter sicherheitskritischer Systeme, da sie zum einen Echtzeitsysteme sind und zum anderen durch Manipulation wirtschaftliche Schäden verursachen können.

8.1 Wackeltisch

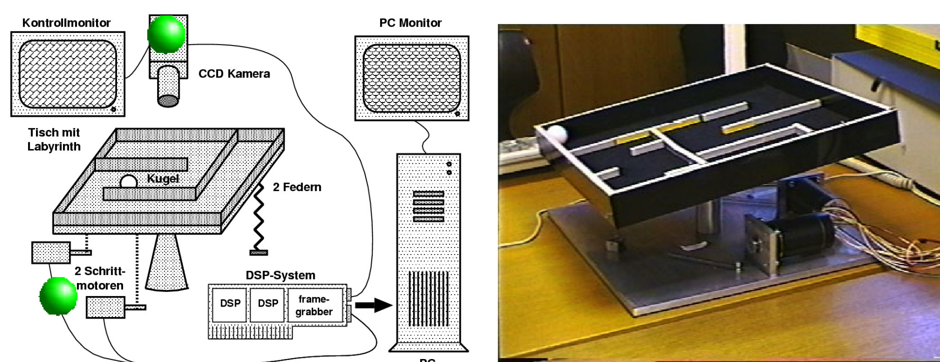


Abbildung 8.1: Der „Wackeltisch“

Die Abbildung 8.1 zeigt den sog. Wackeltisch[SKRS98]. Es handelt sich um ein größeres studentisches Gruppenprojekt für ein komplexes Echtzeitsystem. Das Ziel des Systems ist die kontrollierte Führung eines Balls durch ein Labyrinth zum Zielpunkt, ohne die Banden zu berühren. Hierfür wird ein durch zwei Schrittmotoren horizontal und vertikal schwenkbarer Tisch verwendet, der das Labyrinth darstellt. Mit einer Stereokamera über dem Tisch wird kontinuierlich die aktuelle Position des Balls bestimmt und anschließend ein optimaler Pfad zum Ziel berechnet. Aus den Pfadangaben werden mithilfe eines neuronalen Netzwerks entsprechende Winkel für die beiden Schrittmotoren bestimmt und diese durch Signale aktiviert.

Beim Wackeltisch handelt es sich um ein ANSI-C Programm mit 4500 Zeilen Quelltext. Der PDG besteht aus über 10000 Knoten und 30000 Kanten. Fehlende Bibliotheksfunktionen wurden nach bewährter Art durch Stubs approximiert, die transitive Abhängigkeiten zwischen Eingangs- und Ausgangsparametern in Funktionsaufrufen darstellen und somit reale Abhängigkeiten in Funktionsrümpfen simulieren.

8.1.1 Informationsfluss von der Kamera zur Motorsteuerung

Die erste Fallstudie soll zeigen, ob der erwartete Informationsfluss zwischen der Kamera (Abb.8.1, obere Markierung) und den Schrittmotoren (Abb.8.1, untere Markierung) mit VALSOFT berechnet werden kann. Wenn ein Informationsfluss stattfindet, bedeutet dies, dass der Schrittmotor von der Kamera beeinflusst wird.

Die Abbildung 8.4 auf Seite 142 stellt einen Auszug aus der zentralen Berechnungsschleife dar. Die Zeilenangaben entsprechen daher nicht den tatsächlichen Zeilennummern, dienen jedoch im Folgenden zur Erklärung der Funktionalität.

Der zu realisierende Pfad wird berechnet (Zeile 3) und anschließend die Hauptschleife (Zeile 13) betreten. Sie wird solange durchlaufen, bis die Kugel ihren festgelegten Endpunkt erreicht hat. Bei jedem Durchlauf wird die aktuelle Ballposition von der Kamera gelesen und in diskrete Tischkoordinaten übersetzt (Zeile 14). Nach dem Test, ob die Winkel des Tisches im legalen Bereich sind (Zeile 21), wird der nächste diskrete Zwischenzielpunkt für den Ball bestimmt (Zeile 27). Ein neuronales Netz bestimmt anschließend die neuen Winkel für die beiden Schrittmotoren, um den anvisierten Zwischenzielpunkt zu erreichen (Zeile 34). Die neuen Daten werden den Schrittmotoren zur Ausführung übergeben (Zeile 42). Wenn der Endpunkt erreicht ist, wird der Tisch waagerecht gestellt, damit die Ballbewegung terminiert (Zeile 54).

Der zu untersuchende Informationsfluss führt vom Ergebnispfad der Kamera

$PC(kamera, motor) \equiv$

$$
\begin{aligned}
& \textit{ziel_nicht_erreicht}_{9263} = \textit{TRUE} \quad \text{regler.c} \\
\wedge\ & \text{dspkommEmpfangeDoublePunkt}_{9267}(\text{calloc}_{9093}(1,8),\ldots) \neq 0 \quad \text{regler.c} \\
\wedge\ & |\textit{platte}_{9289}.x_{9290})| \leq 250 \quad \text{regler.c} \\
\wedge\ & |\textit{platte}_{9296}.y_{9297})| \leq 250 \quad \text{regler.c} \\
\wedge\ & \text{pfadNaechsterZielpunkt}_{9325}(\text{dspkommEmpfangePfad}_{9132}(\ldots), \text{calloc}_{9093}(1,8), \\
& \quad \text{calloc}_{9100}(1,8),\ldots) = \textit{TRUE} \quad \text{regler.c} \\
\wedge\ & i_{208} < |-\textit{platte}_{9570}.x_{9571}| + |-\textit{platte}_{9575}.y_{9576}| \quad \text{calc.c, regler.c} \\
\wedge\ (& \quad \text{sqrt}((\text{dspkommEmpfangePfad}_{9132}(\ldots).\textit{root}_{7874}.\textit{pos}_{7876}.x_{7878} - \text{calloc}_{9093}(1,8).x_{7880})* \\
& \quad (\text{dspkommEmpfangePfad}_{9132}(\ldots).\textit{root}_{7884}.\textit{pos}_{7886}.x_{7888} - \text{calloc}_{9093}(1,8).x_{7890}) + \\
& \quad (\text{dspkommEmpfangePfad}_{9132}(\ldots).\textit{root}_{7895}.\textit{pos}_{7897}.y_{7899} - \text{calloc}_{9093}(1,8).y_{7901}) * \\
& \quad (\text{dspkommEmpfangePfad}_{9132}(\ldots).\textit{root}_{7905}.\textit{pos}_{7907}.y_{7909} - \text{calloc}_{9093}(1,8).y_{7911})) \\
& \quad < \textit{MAX_ZIELPUNKTABSTAND}_{9330} \quad \text{pfad.c, regler.c} \\
& \wedge |\text{calloc}_{9107}(1,8).x_{7923}| < \textit{MAX_ZIELPUNKTGESCHWIND}_{9331} \quad \text{pfad.c, regler.c} \\
& \wedge |\text{calloc}_{9107}(1,8).y_{7932}| < \textit{MAX_ZIELPUNKTGESCHWIND}_{9331} \quad \text{pfad.c, regler.c} \\
& \wedge \text{dspkommEmpfangePfad}_{9132}(\ldots).\textit{root}_{7941}.\textit{naechster}_{7943} = 0 \quad \text{pfad.c, regler.c} \\
& \vee \\
& \quad i_{4518} < \textit{anzahl_neuronen}_{6160} \quad \text{neuronal.c} \\
& \wedge\ \textit{anzahl_neuronen_in_schicht}_{9424}[0] \neq 2 \quad \text{neuronal.c} \\
& \wedge\ \textit{anzahl_neuronen_in_schicht}_{9424}[0] \neq 3 \quad \text{neuronal.c} \\
& \wedge\ \textit{anzahl_neuronen_in_schicht}_{9424}[0] = 8 \quad \text{neuronal.c} \\
)
\end{aligned}
$$

Abbildung 8.2: Zwei Monome der Pfadbedingung von der Kamera zur Schrittmotorsteuerung

(*kamera*) zur Ansteuerung der Schrittmotoren (*motor*). Die Abbildung 8.2 zeigt zwei von 14 Monomen der resultierenden Pfadbedingung und ist erwartungsgemäß nicht *false*, da zum Betrieb Informationen von der Kamera zu den Schrittmotoren übertragen werden müssen. Die Pfadbedingung stellt zwischen Start- und Zielpunkt eine notwendige Bedingung für einen Informationsfluss dar und sagt aus, dass es tatsächlich nur dann eine Abhängigkeit geben kann, wenn die Bedingung erfüllt wird.

Die SSA-Indizes an Variablen (*kursiv*) und Funktionsaufrufen (normal) stehen für Knotennummern des PDG, die sich bei Bedarf in Sourcecode-Positionen überführen lassen. Dateiangaben je Term liefern Informationen darüber, woher die Komponenten des Terms kommen. Eindeutige Φ-Bedingungen werden automatisch substituiert (Konstantenpropagation), mehrdeutige Φ-Bedingungen werden erst bei Bedarf gezeigt. Weitere Details zur Pfadbedingung bzgl. der Berechnungszeit und Größe werden in den Tabellen 7.4, 7.8 und 7.9 (Seite 110, 114 und 116) gegeben.

Für die manuelle Auswertung von Pfadbedingungen wird immer die Dis-

junktive Normalform berechnet, da Monome besonders leicht zu interpretieren sind: Jedes Monom beschreibt eine eigenständige und abgeschlossene Bedingung, die ein Satz von Pfaden gemeinsam hat. Zur Bestimmung von Wertebereichen für die Variablen können Constraint-Solver die gegebene Pfadbedingung noch weiter auswerten und versuchen, Zeugen zu berechnen. In diesem Fall ist dies nicht notwendig, da die Pfadbedingung relativ klein, übersichtlich und aufgrund einer exzellenten Namensgebung lesbar und verständlich ist.

Die Pfadbedingung sagt aus, dass für einen möglichen Informationsfluss das Ziel des Balls nicht erreicht sein darf, der Ball eine legale Position haben muss, die horizontale und vertikale Ausrichtung des Tisches eine Grenze von 250 Einheiten nicht überschreiten darf und der nächste Zwischenzielpunkt berechnet sein muss. Diese Anforderungen hätte man möglicherweise durch ein Studium des Quelltextes 8.4 herausfinden können, jedoch ist die manuelle Inspektion entlang von Abhängigkeitspfaden (nicht Kontrollflusspfaden!) sehr aufwändig und fehlerträchtig. Die nachfolgenden Bedingungen sind ebenso notwendig und entstammen aus Prädikaten, die in anderen Funktionen und Dateien definiert werden, jedoch auf den untersuchten Abhängigkeitspfaden liegen. Sie sagen aus, dass der euklidische Abstand zwischen der aktuellen Ballposition und dem Zwischenzielpunkt einen fest vorgegebenen Maximalabstand nicht überschreiten darf und dass die maximale Geschwindigkeit, mit der die Kugel bewegt werden darf, nicht überschritten wird und weiterhin der nächste Zielpunkt des Pfades 0 ist. Für das zweite Monom gilt, dass einige Bedingungen des neuronalen Netzes erfüllt sein müssen, wie z.B. die begrenzte Anzahl der Neuronen und die Anzahl von Neuronen je Schicht. Die SSA-Rückbezüge in den Quelltext offenbaren, dass die Anzahl der „feuernden“ Neuronen verschiedene Fälle bzgl. horizontaler und vertikaler Ausrichtungen, Zielabstände und Ballgeschwindigkeiten unterscheiden.

Die Untersuchung der Testsuite in Kapitel 7 hat gezeigt, dass Pfadbedingungen i.Allg. weniger komplex als vielleicht angenommen sind und sich relativ einfach „lesen“ lassen. Einen Beitrag hierzu leisten natürlich Code-Konventionen, nach denen heutzutage programmiert wird. Lesbarer Code sollte daher i.d.R. auch lesbare Pfadbedingungen erzeugen.

Der Prüfingenieur könnte nun die berechneten Informationen auf Plausibilität überprüfen oder gegen die Herstellerspezifikation[1] testen. Hierbei erfordert das Verstehen von Pfadbedingungen immer ein grundlegendes Programmverständnis der zu untersuchenden Software, das jedoch im direkten Vergleich mit dem reinen Code-Review geringer ausfällt, da Pfadbedingungen die relevanten Informationen entlang der interessanten Pfade bündeln.

[1] Für zertifizierungspflichtige Software müssen Hersteller sicherheitskritische Pfade (sog. Eichpfade) angeben.

8.1.2 Pfadbedingungen reduzieren Program-Slices

Pfadbedingungen liefern nicht nur eine Methode, um präzise notwendige Bedingungen oder Zeugen für Informationsflüsse zu liefern, sie können auch Program-Slices reduzieren und damit präziser machen. Die binäre Information von Program-Slices enthält keine Information über Beziehungen und Werte zwischen den enthaltenen Knoten.

Eine genaue Betrachtung des ersten Monoms in Abbildung 8.2 mit dem Wurzelaufruf „sqrt“ offenbart eine Inkonsistenz in der Bedingung. Einerseits wird gefordert, dass der nächste Zwischenzielpunkt berechnet sein soll (pfadNaechsterZielpunkt() $=$ *true*), andererseits existiert eine Bedingung, dass der nächste Zielpunkt des Pfades 0 ist (dspkommEmpfangePfad()$.root.naechster = 0$). Dies deutet auf einen Widerspruch hin, da letztere Forderung i.Allg. impliziert, dass das Ziel bereits erreicht ist.

Ein Blick in den Quelltext der Datei „pfad.c“ offenbart:

```
1  if ((eukl_abstand < MAX_ZIELPUNKTABSTAND) &&
2      (abs(geschwindigkeit->x) < MAX_ZIELPUNKTGESCHWINDIGKEIT) &&
3      (abs(geschwindigkeit->y) < MAX_ZIELPUNKTGESCHWINDIGKEIT)) {
4      /* Neuen Zielpunkt wählen */
5      if (weg->root->naechster == 0l) {
6        /* Wir sind am Ende! */
7        return 0;
8      }
9  ...
```

Die Rückgabe des Wertes 0 führt sofort zur Nichterfüllung von pfadNaechsterZielpunkt() $=$ *true* , so dass dieses Monom immer zu *false* ausgewertet wird. Dies ist ein Beleg dafür, dass der obige Quelltext im Program-Slice liegt, zur Laufzeit jedoch *kein* Informationsfluss zwischen der Kamera und der Motorsteuerung entlang dieses Quelltextes existieren kann. Die Auswertung der Pfadbedingung hat somit den Program-Slice verkleinert und präziser gemacht. Mit den Techniken aus Abschnitt 5.2.9 kann die Pfadbedingung automatisch reduziert werden.

8.1.3 Entdeckung einer Sicherheitsverletzung

In dieser Fallstudie geht es um die Überprüfung, ob „externe Agenten“ den Schrittmotor beeinflussen können. Die Datei „regler.c“ des Wackeltisches enthält überraschenderweise eine Variablendeklaration innerhalb der `main`-Funktion (`char* taste;`). Die Tastatur wurde während der Entwicklung für

Debugging-Zwecke verwendet und in der Release-Version des Wackeltisches alle Referenzen an die Variable `taste` entfernt.

Zur Überprüfung der Entfernung aller Referenzen wurde `taste` als neuer Startpunkt einer Pfadbedingung gewählt. Der Zielpunkt blieb die Ansteuerung der beiden Schrittmotoren aus der vorherigen Fallstudie. Die Pfadbedingung $PC(\texttt{taste}, motor)$ liefert erwartungsgemäß *false*, so dass mit Sicherheit kein Informationsfluss zwischen der Tastatureingabe und der Schrittmotorsteuerung vorliegt.

Für einen Praxistest wurde eine Programmänderung vorgenommen und `taste` wieder reaktiviert. Die Herausforderung lag darin, herauszufinden, ob Pfadbedingungen im Gegensatz zu Program-Slicing aussagekräftig genug sind, um evtl. Abhängigkeiten informativ zu beschreiben.

Nach der Programmänderung wurde die Pfadbedingung $PC(\texttt{taste}, motor)$ ein weiteres Mal berechnet und weder zu *false* noch zu *true* ausgewertet. Die resultierende Pfadbedingung bestand aus 11 Disjunktionen und 148 Konjunktionen mit 160 Termen in Disjunktiver Normalform. Drei typische Monome der Pfadbedingung sind in Abbildung 8.3 zu sehen. Es liegt also tatsächlich eine Beeinflussung der Schrittmotorsteuerung durch eine Tastatureingabe vor.

Die Pfadbedingung ist in einigen Bereichen ähnlich zu Abb 8.2, was durch denselben Zielknoten *motor* verursacht wird. Neben den Dateinamen stehen nun auch die entsprechenden Zeilennummern, an denen Komponenten für nebenstehende Terme zuletzt eindeutig definiert wurden. Unterhalb der Pfadbedingung ist ein Teil der berechneten Φ-Bedingungen angegeben, die komplexe SSA-Substitutionen – vor allem interprozedurale – beschreiben.

Aus der Pfadbedingung selbst geht kein Hinweis auf die Ursache der Beeinflussung hervor. Jedoch beschreiben die Φ-Bedingungen alle möglichen SSA-Substitutionen und enthalten die Definitionsstelle des Startknotens `taste` ($taste_{9184}$). Da Φ-Bedingungen alle Ketten von Datenabhängigkeiten liefern, erhält man die Verknüpfung zwischen $taste_{9224}$ und $ping_{9517}$, die weitergeführt auf $ping_{3498}$ in der Datei „neuronal.c“ verweist.

An dieser Stelle kann die Verbindung zur Pfadbedingung hergestellt, werden, da dort in jedem der 12 Monome der Term $*ping_{3498}\&128 > 0$ gefordert wird. Somit ist gezeigt, dass dieser Term ausschlaggebend für die gesuchte Ursache der Beeinflussung ist, da er direkt über Datenabhängigkeiten mit der Tastatureingabe verbunden ist. Der Schrittmotor wird also durch Tastatureingaben über die Variable *ping* manipuliert.

Über SSA-Indizes und Zeileninformationen kann direkt im Quelltext die Auswirkung dieser Manipulation überprüft werden:

$PC(\texttt{taste}, motor) \equiv$

$i_{207} < |m_x_{582}| + |m_y_{583}|$ calc.c : 80, 157, 137
$\wedge$ $\text{schichtVon}_{3591}(neuron_id_{3497}, \ldots) \neq 0$ neuronal.c : 696, 719
$\wedge$ $\underline{*ping_{3498}\&128 > 0}$ neuronal.c : 696, 731
$\wedge$ $ziel_nicht_erreicht_{9361} = TRUE$ regler.c : 202
$\wedge$ $\text{dspkommEmpfangeDoublePunkt}_{9365}(\text{calloc}_{9188}(1, 8), \ldots) \neq 0$ regler.c : 160, 205, 208
$\wedge$ $|platte_{9387}.x_{9388})| \leq 250$ regler.c : 215
$\wedge$ $|platte_{9394}.y_{9395})| \leq 250$ regler.c : 215
$\wedge$ $\text{pfadNaechsterZielpunkt}_{9423}(\text{dspkommEmpfangePfad}_{9230}(\ldots), \text{calloc}_{9188}(1, 8),$
$\text{calloc}_{9195}(1, 8), \ldots) = TRUE$ regler.c : 160 . . . , 179, 228
$\wedge$ ($anzahl_neuronen_in_schicht_{5298}[0] = 2$ neuronal.c : 1338
$\vee$ $anzahl_neuronen_in_schicht_{5298}[0] = 3$ neuronal.c : 1338
$\vee$ $anzahl_neuronen_in_schicht_{5298}[0] = 8$ neuronal.c : 1338
)

$\Phi \equiv m_x_{582} = -platte_{9671}.x_{9672},$
$m_y_{583} = -platte_{9676}.y_{9677},$ calc.c : 137, regler.c : 272, 273
$neuron_id_{3497} = \text{erstesNeuronIn}_{5373}(anzahl_schichten_{9523} - 1),$
neuronal.c : 696, 1448
$ping_{3498} = ping_{5492} = ping_{5291},$
$ping_{5291} = ping_{9517} = taste_{9224} = \underline{taste_{9184}}$ *Definitionsstelle*
regler.c : 158, 171, 240, neuronal.c : 696, 1316, 1363

Abbildung 8.3: Drei Monome der Pfadbedingung von der Tastatureingabe zur Schrittmotorsteuerung

```
1 if ((*ping) & 0x80 > 0) {
2   wert *= 1.2;
3 }
```

Dieses Codestück in „neuronal.c“ erhöht den Skalierungsfaktor des Neuronalen Netzes um 20%, wenn das 8. Bit von `*ping` gesetzt ist. Man sieht, dass `wert` selbst nicht in der Pfadbedingung enthalten sein muss (es ist nicht Bestandteil eines Prädikats), um trotzdem mit Pfadbedingungen sowohl die Ursachen als auch die Auswirkungen von Sicherheitsverletzungen relativ leicht bestimmen zu können.

Neben der Änderung von „neuronal.c“ wurden zusätzlich in der Datei „variable.h“ die Deklaration `extern int* ping`, in „dspkomm.c“ die Deklaration `int* ping` und in „regler.c“ die Anweisung `ping = (int*) taste` hinzugefügt.

Obwohl die Manipulation keine Sicherheitsverletzung real eingesetzter Software ist, zeigt diese Fallstudie jedoch, dass die Aufdeckung einer über meh-

rere Dateien verteilte Manipulation ohne Einsatz von Pfadbedingungen ein sehr mühseliges und fehlerträchtiges Unterfangen wäre, selbst für Experten.

```
1  int main() {
2    ...
3    weg_ans_ziel = dspkommEmpfangePfad(); (kamera)
4    if (weg_ans_ziel == 01) {
5      ...
6    }
7
8    statInit(weg_ans_ziel);
9    alter_mittelpunkt = weg_ans_ziel->root->wegpunkt;
10   weg_startpunkt = weg_ans_ziel->root->wegpunkt;
11   reglerInit();
12
13   while (ziel_nicht_erreicht) {
14     ret = dspkommEmpfangeDoublePunkt(mittelpunkt);
15
16     if (ret == 0) {
17       ziel_nicht_erreicht = 0;
18       continue;
19     }
20
21     if (abs(platte.x) > 250 || abs(platte.y) > 250) {
22       ...
23       ziel_nicht_erreicht = 0;
24       continue;
25     }
26
27     ziel_nicht_erreicht = pfadNaechsterZielpunkt(weg_ans_ziel, mittelpunkt,
28                                                  abstand, geschwindigkeit);
29     aktueller_zielpunkt = weg_ans_ziel->root->wegpunkt;
30     geschwindigkeit->x = mittelpunkt->x - alter_mittelpunkt.x;
31     geschwindigkeit->y = mittelpunkt->y - alter_mittelpunkt.y;
32
33     ...
34     reglerBerechneMotorschritte(abstand->x, geschwindigkeit->x, platte.x,
35                                 abstand->y, geschwindigkeit->y, platte.y,
36                                 &schritte_x, &schritte_y);
37     ...
38     statUpdate(weg_ans_ziel->root, *mittelpunkt, alter_mittelpunkt, zyklen);
39     vektor = calcSteuerungsVektor(schritte_x, schritte_y);
40
41     if (ziel_nicht_erreicht) {
42       dspkommSendeMotorschritte(vektor, 0, vektorlaenge);
43       platte.x = platte.x + schritte_x;
44       platte.y = platte.y + schritte_y;
45     }
46     free(vektor);
47     alter_mittelpunkt = *mittelpunkt;
48   }
49
50   schritte_x = -platte.x;
51   schritte_y = -platte.y;
52   pckommPrintString("Stelle Platte waagerecht.\n");
53   vektor = calcSteuerungsVektor(schritte_x, schritte_y);
54   dspkommSendeMotorschritte(vektor, vektorlaenge, 0); (motor)
55   ...
56 }
```

Abbildung 8.4: Quelltextauszug Wackeltisch (Hauptfunktion)

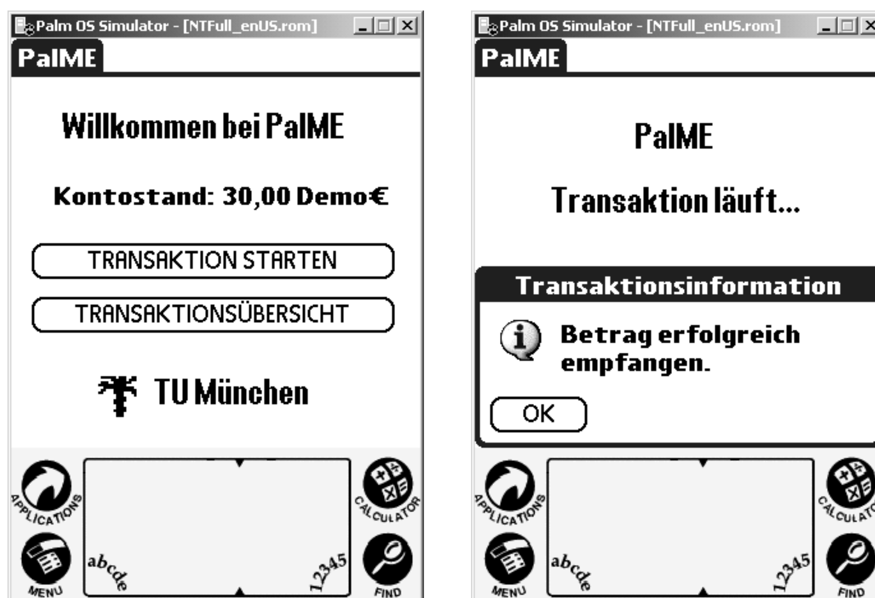


Abbildung 8.5: Die elektronische Geldbörse „PalME“

8.2 Elektronische Geldbörse

Bei dieser Fallstudie geht es um die Überprüfung von Pfadbedingungen für die elektronische Geldbörse PalME (Secure Palm-based Money Exchange, [LSS01]). PalME ist ein Softwareprojekt, das eine elektronische Geldbörsenanwendung für einen PalmOS basierten PDA ermöglicht. Der PDA kann virtuelles Geld wie eine Geldkarte speichern, jedoch auch ohne Kartenautomat Beträge zwischen zwei virtuellen Geldbörsen via Infrarot übertragen (engl. off-line purse-to-purse transaction). Es handelt sich also um ein wirtschaftlich interessantes System, bei dem Überprüfungen sinnvoll erscheinen. Aus diesem Grund wurden im gesamten Entwicklungsprozess der Anwendung die Anforderungen der Common Criteria [BSI03a] berücksichtigt, da für Entwicklung und mögliche Zertifizierung die Sicherheit eine zentrale Rolle spielt.

In Abbildung 8.5 sind zwei Screenshots des Systems zu sehen. PalME besteht aus den folgenden zentralen Komponenten:

- PalME – Hauptprogramm, in dem die Benutzeroberfläche gestartet wird und ankommende Events innerhalb einer zentralen Schleife bearbeitet werden
- GUI – Benutzeroberfläche, die Eingaben und Ausgaben zulässt
- Control – Protokollhandler für den sicherheitsunkritischen Teil

  - PH_Control – Anwendungslogik für den unsicheren Teil. Bei sensitiven Daten agiert PH_Control als Vermittler zwischen der Komponente SmartCard und dem GUI.
  - Transfer – Datenübertragung zur Infrarotschnittstelle des PDA
- SmartCard – Protokollhandler für den sicherheitskritischen Teil. Wenn Daten an Komponenten außerhalb der SmartCard weitergereicht werden sollen, erfolgt eine Verschlüsselung mithilfe der enthaltenen Crypto-Komponente.
  - Account – Konto mit allen notwendigen Status- und Transaktionsfunktionen
  - Crypto – Algorithmus zum Ver- und Entschlüsseln
  - PH_SmartCard – Öffentliches Interface, Anwendungslogik für den sicheren Teil und Kapselung aller sicherheitskritischen Daten und Funktionen
  - Storage – Sicherer Datenbankspeicher auf der SmartCard

Bei PalME handelt es sich um eine objektorientierte Anwendung in C++, die fast ausschließlich Singleton-Objekte verwendet. Aus diesem Grund war es möglich, mit moderatem Aufwand PalME nach ANSI-C zu konvertieren. Zum Einsatz kamen sowohl die PRC-Tools [PRC03, Pal04b] als Cross-Compiler für PalmOS-Software als auch MetroWorks CodeWarrior [Met] mit dem PalmOS Simulator [Pal04a]. Die Screenshots entstammen dem nach ANSI-C konvertierten PalME, ausgeführt im PalmOS Simulator (siehe Abb. 8.5).

PalME ist den sicherheitskritischen Programmen zuzuordnen. Interessanterweise enthält das Programm für seine Größe von fast 6500 Zeilen Quelltext nur relativ wenige Prädikate (153 `if`, 25 `for`, 10 `while`, 35 `switch` und 128 `case`). Die geringe Anzahl von Schleifen wirkt sich direkt auf die Anzahl der Zyklen im PDG aus. Für die berechneten sechs Chops ergeben sich nur 76, 98, 108, 130, 180 und 144 Zyklen. Diese Erkenntnis muss sich in der Theorie auf eine relativ konstante Berechnungszeit, unabhängig von der Wahl der Zyklenzerlegung, auswirken und wird durch die Vergleichstabelle in Abb. 7.8 bestätigt. Erstaunlicherweise bedeuten aber eine geringe Anzahl von Prädikaten keineswegs eine ebenso geringe Anzahl an Disjunktionen, Konjunktionen und Termen, wie die Tabellen in Abb. 7.9, 7.10, 7.11 und 7.12 (Seite 116, 117, 118 und 119) zeigen. Man kann daher von einer Bündelung von Prädikaten sprechen.

Zu Demonstrationszwecken und als Beleg des „Program-Understanding“ werden in diesem Abschnitt vier interprozedurale Pfadbedingungen berechnet, wobei Start- und Endknoten jeweils in verschiedenen Dateien liegen.

Als sinnvoller und allen Pfadbedingungen gemeinsamer Startknoten wird die Funktion `ControlHandleData` der Komponente „PH_Control“ gewählt, da sie die gesamte Anwendungslogik kapselt und hiermit den zentralen Punkt für kritische und unkritische Informationsflüsse darstellt.

Die vier Pfadbedingungen sind:

1. ControlHandleData → HandleData (PH_SmartCard)
   Weiterleitung einer Anfrage an PH_SmartCard
2. ControlHandleData → AccoundDeposit (Account)
   Gutschrift eines empfangenen Betrags
3. ControlHandleData → SendEncrypted (PH_SmartCard)
   Senden einer Anfrage (z.B. ChallengeResponseRequest)
4. ControlHandleData → Send (Transfer)
   Senden einer Anfrage über die Infrarotschnittstelle

Für die erste Pfadbedingung ist in Abbildung 8.10 auf Seite 151 ein Auszug aus der minimierten Originalpfadbedingung mit allen Zusatzinformationen wie Typinformationen, Rückbezüge in den Quelltext, etc. angegeben, so wie sie der Pfadbedingungsgenerator konstruiert.

Die Pfadbedingung besteht aus den notwendigen Bedingungen am Anfang eines jeden Monoms, bei denen elementare Substitutionen bereits durchgeführt sind. Durch die Minimierung nach Quine/McCluskey und der Berechnung von Minimalpolynomen liegt eine minimale DNF vor. Die Interpretationsweise wird anhand des folgenden Beispiels beschrieben:

```
1  [r][deref]ctrl@6741.AmIGG:cu@3632
2  ([f]DecryptStr:p@12806(crypto:s@12807, data:p@12686, len:iu@12687,
3    ...
4  *sha1.Corrupted@12834, *sha1.Message_Block_Index@12835) ==:i@12855 0:i@12857)
5
6  sc:p@12685 := [r][deref]ctrl@6741.SmartCardPtr:p@3927;
7  data:p@12686 := [r][deref]pCBParms:p@5575.rxBuff:p@6717;
```

Zeile 1 fordert, dass das Ergebnis des `struct`-Zugriffs ([r]ecord) `ctrl.AmIGG` *true* ist. Da `ctrl` einen Zeiger darstellt, wird er hierfür dereferenziert ([deref]er). Der Gesamttyp des Ausdrucks ist „unsigned char“ (cu). Die Rückverkettung an die Quelltextposition ist durch PDG-Knotenummer 3632 gegeben, die bei PalME für Datei „PH_Control.c“, Zeile 260, Spalte 7-21 steht. Der entsprechende Quelltext für Zeile 1 lautet:

`if (control->AmIGG) //Sicht des Geldgebers`.

Die Knotennummer 6741 zeigt noch die letzte eindeutige Definitionsstelle von `ctrl` an.

Die Zeilen 2-4 sind vom selben Aufbau, mit dem Unterschied, dass der Term das Ergebnis eines Funktionsaufrufs auswertet ([f]unction). Die aufgerufene Funktion lautet `DecryptSrt` und liefert einen Zeiger zurück ([p]ointer). Der Aufruf der Funktion erfolgt an Knotennummer 12806. Innerhalb der folgenden Klammern sind die bereits substituierten aktuellen Parameter der Funktion nach gleichem Schema angegeben. Hierbei besteht die Parameterliste sowohl aus realen Parametern als auch aus Parametern, die durch die Propagation von globalen Variablen hinzugefügt wurden. Das Ergebnis des Rückgabewertes wird an Knotennummer 12855 auf Gleichheit mit dem Wert 0 von Knotennummer 12857 überprüft. Da 0 vom Typ Integer (i) ist und der Rückgabewert ein Zeiger, der hier dieselbe Speicherrepräsentation hat, ist der Gesamttyp des Ausdrucks ebenfalls Integer. Ein Blick in den Quelltext liefert:

```
h = DecryptStr(&crypto, (unsigned char*)data, len);
  if (h == NULL) {
```

Die präzise Klassifizierung des Operators (`==`) ist notwendig, um bei unterschiedlichen Typen von Operanden den Ergebnistyp zu kennen und vor dem Aufruf von Constraint-Solvern ggf. eigene Typcasts vorzunehmen. Weiterhin ist die exakte Positionierung mittels der Knotennummer nützlich, da sich die letzten eindeutigen Definitionsstellen der Operanden an völlig verschiedenen Positionen im Programm befinden können. Eine Lokalisierung des Prädikats wäre ansonsten in einigen Fällen schwierig.

Die Zeilen 6 und 7 stehen für Parameterzuweisungen von aktuellen an formale Parameter entlang der Pfade, für die das gegebene Monom steht. In Zeile 6 wird z.B. der Variablen `sc` als formaler Parameter an Position 12685 (Datei „PH_SmartCard.c“, Zeile 782, Spalte 37-39) der aktuelle Parameter `ctrl->SmartCardPtr` an Position 3927 (Datei „PH_Control.c“, Zeile 333, Spalte 32-53) zugewiesen.

8.2.1 Pfadbedingung PalME 1

Diese einfache Pfadbedingung in Abbildung 8.6 entstammt hauptsächlich zwei geschachtelten *switch*-Anweisungen, die auf den Pfaden liegen, wobei jede unter einem *if*-Prädikat bzw. seiner negierten Form steht. Bis auf den letzten Fall zeigen alle anderen Monome, dass die nachfolgenden Pfade bis zum Zielknoten fast unbedingt sind, also *true* als Bedingung haben. Es existiert nur genau ein gemeinsames Prädikat, das auf allen Pfaden vorkommt. Interessant ist im letzten Monom die Zahl 20, die nirgendwo im Quelltext vorkommt, da sie einer `sizeof`-Anweisung entstammt und vom Compiler automatisch berechnet wird.

$\mathrm{DecryptStr}_{12806}(\mathrm{crypto}_{12807},\ \ldots) \ = \ 0$

$\wedge$

$ctrl_{6741}.AmIGG_{3632} \ \neq \ 0$
$ctrl_{6741}.State_{3636} \ = \ 1$

$\vee$

$ctrl_{6741}.AmIGG_{3632} \ = \ 0$
$ctrl_{6741}.State_{4506} \ = \ 2$

$\vee$

$ctrl_{6741}.AmIGG_{3632} \ \neq \ 0$
$ctrl_{6741}.State_{3636} \ = \ 2$

$\vee$

$ctrl_{6741}.AmIGG_{3632} \ = \ 0$
$ctrl_{6741}.State_{4506} \ = \ 3$

$\vee$

$ctrl_{6741}.AmIGG_{3632} \ \neq \ 0$
$ctrl_{6741}.State_{3636} \ = \ 3$

$\vee$

$ctrl_{6741}.AmIGG_{3632} \ = \ 0$
$20 \ = \ pCBParms_{5575}.rxLen_{6720}$
$ctrl_{6741}.State_{4506} \ = \ 1$
$\mathrm{StrCompare}_{4861}(\mathrm{pCBParms}_{5575}.rxBuff_{6717},\ \ldots) \ \neq \ 0$
$ctrl_{6741}.confirm_{4588} \ \neq \ 0$

Abbildung 8.6: Pfadbedingung PalME 1

Aufgrund des Umfangs der Pfadbedingungen werden für die PalME-Beispiele nur interessante Auszüge und generell keine Parametersubstitutionen (siehe Abb. 8.6) angegeben.

8.2.2 Pfadbedingung PalME 2

Die zweite Pfadbedingung in Abbildung 8.7 hat viele Gemeinsamkeiten mit der ersten, da zwischen Start- und Zielknoten die Funktion `HandleData` (PH_SmartCard) aus der ersten Pfadbedingung durchlaufen wird. Die Disjunktionen sind mit der ersten Pfadbedingung identisch, die gemeinsamen

$\mathrm{DecryptStr}_{12806}(\mathrm{crypto}_{12807},\ \ldots) \ \neq \ 0$
$\mathrm{MemHandleSize}_{12953}(\mathrm{DecryptStr}_{12806},\ \ldots) \ = \ 32$
$MemHandleLock_{12947}(DecryptStr_{12806},\ \ldots).Betrag_{12105} \ = \ sc_{12685}.Amount_{12108}$
$\mathrm{CheckChallengeResponse}_{12373}(\mathrm{sc}_{12685},\ \ldots,\ \ldots) \ = \ 0$
$\mathrm{MemHandleSize}_{12953}(\mathrm{DecryptStr}_{12806},\ \ldots) \ = \ 24$
$\mathrm{AccountCheckDepositLimit2}_{7569}(\mathrm{sc}_{12685}.storage_{12566},\ \ldots,\ \ldots) \ = \ 1$
$\mathrm{CheckChallengeResponse}_{12529}(\mathrm{sc}_{12685},\ \ldots,\ \ldots) \ = \ 0$
$sc_{12685}.state_{12013} \ = \ 2$
$\mathrm{MemHandleSize}_{12953}(\mathrm{DecryptStr}_{12806},\ \ldots) \ = \ 24$
$\mathrm{CheckChallengeResponse}_{12065}(\mathrm{sc}_{12685},\ \ldots,\ \ldots) \ = \ 0$
$sc_{12685}.AmIGG_{12961} \ = \ 0$

$\wedge$

Disjunktionen sind identisch zu Pfadbedingung PalME 1

Abbildung 8.7: Pfadbedingung PalME 2

$$
\begin{array}{l}
\quad ctrl_{6741}.AmIGG_{3632} = 0 \\
\quad ctrl_{6741}.State_{4506} = 2 \\
\quad \mathrm{DecryptStr}_{12806}(\mathrm{crypto}_{12807},\ \ldots) = 0 \\
\vee \\
\quad ctrl_{6741}.AmIGG_{3632} \neq 0 \\
\quad \mathrm{StrCompare}_{3643}(\mathrm{pCBParms}_{5575}.rxBuff_{6717},\ \ldots) \neq 0 \\
\quad ctrl_{6741}.TimeoutOccurred_{3723} = 0 \\
\quad \mathrm{CheckWithdrawLimit}_{10583}(\mathrm{ctrl}_{6741}.SmartCardPtr_{3759}.storage_{10585},\ \ldots) \neq 0 \\
\quad ctrl_{6741}.confirm_{3746} \neq 0 \\
\quad ctrl_{6741}.confirm_{3694} \neq 0 \\
\quad ctrl_{6741}.State_{3636} = 0 \\
\vee \\
\quad ctrl_{6741}.AmIGG_{3632} \neq 0 \\
\quad sc_{12685}.AmIGG_{12961} = 0 \\
\quad \mathrm{MemHandleSize}_{12953}(\mathrm{DecryptStr}_{12806},\ \ldots) = 32 \\
\quad MemHandleLock_{12947}(DecryptStr_{12806},\ \ldots).Betrag_{12105} = sc_{12685}.Amount_{12108} \\
\quad \mathrm{MemHandleSize}_{12953}(\mathrm{DecryptStr}_{12806},\ \ldots) = 24 \\
\quad sc_{12685}.state_{12013} = 2 \\
\quad \mathrm{CheckChallengeResponse}_{12529}(\mathrm{sc}_{12685},\ \ldots,\ \ldots) = 0 \\
\quad ctrl_{6741}.State_{3636} = 2 \\
\quad \mathrm{MemHandleSize}_{12953}(\mathrm{DecryptStr}_{12806},\ \ldots) = 24 \\
\quad \mathrm{CheckChallengeResponse}_{12373}(\mathrm{sc}_{12685},\ \ldots,\ \ldots) = 0 \\
\quad sc_{12685}.state_{12013} = 1 \\
\quad \mathrm{CheckChallengeResponse}_{12065}(\mathrm{sc}_{12685},\ \ldots,\ \ldots) = 0 \\
\quad sc_{12685}.state_{12013} = 0 \\
\quad \mathrm{DecryptStr}_{12806}(\mathrm{crypto}_{12807},\ \ldots) \neq 0
\end{array}
$$

Abbildung 8.8: Pfadbedingung PalME 3 (Auszug)

Prädikate sind jedoch deutlich umfangreicher. Die Pfadbedingung enthält das gemeinsame Prädikat der ersten Bedingung in negierter Form, ferner Vergleiche zwischen zwei Strukturen (Betrag und Amount) sowie eine Überprüfung, ob die durchzuführende Gutschrift möglich ist und das festgelegte Limit des Kontos nicht überschritten wird.

8.2.3 Pfadbedingung PalME 3

Diese Pfadbedingung in Abbildung 8.8 enthält keine gemeinsamen Prädikate, die auf allen Pfaden liegen. In der Gesamtbedingung (nicht dargestellt) ist die `if/switch`-Kaskade der ersten Pfadbedingung enthalten, jedoch liegen zwischen Start- und Endknoten deutlich mehr Pfadverzweigungen, was aus der hohen Anzahl an Disjunktionen hervorgeht. Anhand der dargestellten drei Monome ist die Vielfältigkeit der unterschiedlichen Pfade zu sehen, die durch verschiedendste Typen von Requests verursacht wird.

8.2.4 Pfadbedingung PalME 4

Die vierte Bedingung in Abbildung 8.9 enthält wieder ein gemeinsames Prädikat, das prüft, ob die Infrarotschnittstelle schon initialisiert

$$
\begin{array}{l}
\quad IrInitializedYet_{6267} \neq 0 \\
\wedge \\
\quad ctrl_{6741}.AmIGG_{3632} = 0 \\
\quad 20 = pCBParms_{5575}.rxLen_{6720} \\
\quad \text{CheckDepositLimit}_{4614}(\text{ctrl}_{6741}.SmartCardPtr_{4616}.storage_{4618}, \ldots) \neq 0 \\
\quad ctrl_{6741}.Error_{4755} = 0 \\
\quad ctrl_{6741}.confirm_{4712} \neq 0 \\
\quad ctrl_{6741}.confirm_{4588} \neq 0 \\
\quad ctrl_{6741}.State_{4506} = 0 \\
\vee \\
\quad ctrl_{6741}.AmIGG_{3632} = 0 \\
\quad \text{StrCompare}_{4528}(\text{ControlGetVariant}_{4530}, \text{"Bank"}_{4534}) = 0 \\
\quad 20 = pCBParms_{5575}.rxLen_{6720} \\
\quad ctrl_{6741}.confirm_{4588} = 0 \\
\quad \text{ControlGetBalance}_{4556}(\text{ctrl}_{6741}, \ldots) + pCBParms_{5575}.rxBuff_{6717}.Betrag_{4569} > 0 \\
\quad \text{ControlGetBalance}_{4540}(\text{ctrl}_{6741}, \ldots) \neq 0 \\
\quad ctrl_{6741}.State_{4506} = 0 \\
\vee \\
\quad ctrl_{6741}.AmIGG_{3632} \neq 0 \\
\quad \text{StrCompare}_{3643}(\text{pCBParms}_{5575}.rxBuff_{6717}, \ldots) \neq 0 \\
\quad ctrl_{6741}.TimeoutOccurred_{3723} = 0 \\
\quad ctrl_{6741}.confirm_{3746} \neq 0 \\
\quad \text{CheckWithdrawLimit}_{10583}(\text{ctrl}_{6741}.SmartCardPtr_{3759}.storage_{10585}, \ldots) \neq 0 \\
\quad ctrl_{6741}.confirm_{3694} \neq 0 \\
\quad ctrl_{6741}.State_{3636} = 0
\end{array}
$$

Abbildung 8.9: Pfadbedingung PalME 4 (Auszug)

wurde. Da über diese Schnittstelle sowohl virtuelle Geldbeträge empfangen als auch überwiesen werden können, existieren Pfade für beide Transaktionen (siehe CheckDepositLimit und CheckWithdrawLimit). Da das Überweisen sinnvollerweise eine Bestätigung erfordert, ob der Betrag angekommen ist, wird diese Transaktion über einen Timeout gesteuert ([r][deref]ctrl.TimeoutOccurred). Das mittlere Monom ist bei den Pfadbedingungen für PalME zum ersten Mal nicht rein prädikatenlogisch, sondern enthält Arithmetik. Wie schon beim Wackeltisch zu sehen war, kommt Arithmetik nicht sehr häufig vor, jedoch dürfen alle in ANSI-C realisierbaren arithmetischen Ausdrücke auftreten. Der obige Term prüft, ob das elektronische Geld vermehrt wird (Karte aufladen). In diesem Fall wird ein Bestätigungsdialog eingeblendet, damit der Benutzer die Transaktion „Bargeld → elektronisches Geld“ steuern kann.

8.3 Fazit

Die Fallstudien haben gezeigt, dass die Berechnung von Pfadbedingungen kein vollautomatisches Verfahren ist, um Sicherheitsverletzungen aufzudecken. Program-Slicing ist automatisch, liefert jedoch nur binäre Informationen über Abhängigkeiten im Programm. Detaillierte Informationen über In-

formationsflüsse im Programm, die zum Aufdecken von Sicherheitsverletzungen führen können oder die Semantik von Programmen bzgl. Abhängigkeiten verdeutlichen sollen, lassen sich jedoch hervorragend mit Pfadbedingungen berechnen. Die Fallstudien zeigen, dass trotz der allgemeinen exponentiellen Komplexität Pfadbedingungen tatsächlich realisierbar sind.

Der Einsatz von Pfadbedingungen setzt dabei je nach Einsatzziel ein grundlegendes Verständnis des zu analysierenden Programms voraus, da der Pfadbedingungsgenerator das Ergebnis aus dem analysierten Programm konstruiert. Die Interpretation des Ergebnisses – wenn nötig mit Auflösung der Pfadbedingungen nach Eingabevariablen – liefert dann Variablenbelegungen als Zeugen, mit denen der Informationsfluss sichtbar gemacht werden kann. Dies erfolgt bei Eingabevariablen durch entsprechende Wahl der Parameter, bei lokalen Variablen durch Laufzeitbelegung im Debugger.

Im Anhang A ab Seite 171 werden für alle in diesem Kapitel vorgestellten Pfadbedingungen die Binären Entscheidungsgraphen gezeigt.

```
1  [r][deref]ctrl@6741.AmIGG:cu@3632
2  ([f]DecryptStr:p@12806(crypto:s@12807, data:p@12686, len:iu@12687,
3  *crypto.initialized@12811, *crypto.S@12812, *crypto.P@12813,
4  *sha1.H@12814, *sha1.Corrupted@12815, *sha1.Computed@12816,
5  *sha1.H@12817, *sha1.Computed@12818, *sha1.Corrupted@12819,
6  *sha1.Message_Block@12820, *sha1.Message_Block_Index@12821,
7  *sha1.Length_Low@12822, *sha1.Length_High@12823,
8  *sha1.Message_Block@12824, *sha1.H@12825,
9  *sha1.Message_Block_Index@12826, *sha1.Message_Block@12827,
10 *sha1.Length_High@12828, *sha1.Length_Low@12829,
11 *sha1.Length_Low@12830, *sha1.Length_High@12831,
12 *sha1.Message_Block_Index@12832, *sha1.Computed@12833,
13 *sha1.Corrupted@12834, *sha1.Message_Block_Index@12835) ==:i@12855 0:i@12857)
14 [r][deref]ctrl@6741.State:iu@3636 == 1
15
16 sc:p@12685 := [r][deref]ctrl@6741.SmartCardPtr:p@3927; data:p@12686 := [r][deref]pCBParms:p@5575.rxBuff:p@6717;
17 len:iu@12687 := [r][deref]pCBParms:p@5575.rxLen:iu@6720; *id.iddata@12688 := *id.iddata@3931;
18 *id1.iddata@12689 := *id1.iddata@3932; *id2.iddata@12690 := *id2.iddata@3933;
19 refNum@12691 := refNum@3934; IrInitializedYet@12692 := IrInitializedYet@3935;
20 buff@12693 := buff@3936; buflen@12694 := buflen@3937; *crypto.initialized@12695 := *crypto.initialized@3938;
21 *crypto.initialized@12696 := *crypto.initialized@3939; *crypto.S@12697 := *crypto.S@3940;
22 *crypto.P@12698 := *crypto.P@3941; *crypto.P@12699 := *crypto.P@3942;
23 *packet.Challenge@12700 := *packet.Challenge@3943; *packet.Response@12701 := *packet.Response@3944;
24 *sc.LastChallenge@12702 := *sc.LastChallenge@3945; *sc.PartnerID@12703 := *sc.PartnerID@3946;
25 *sc.Amount@12704 := *sc.Amount@3947; *sc.AmIGG@12705 := *sc.AmIGG@3948;
26 *sc.TransactionNo@12706 := *sc.TransactionNo@3949; *sc.storage@12707 := *sc.storage@3950;
27 *bufP@12708 := *bufP@3951; *sc.state@12709 := *sc.state@3952;
28 *pMErr.FehlerText@12710 := *pMErr.FehlerText@3953; *pChResp.Challenge@12711 := *pChResp.Challenge@3954;
29 *sc.storage@12712 := *sc.storage@3955; *sc.Amount@12713 := *sc.Amount@3956;
30 *sc.state@12714 := *sc.state@3957; *pPSO.Header.Response@12715 := *pPSO.Header.Response@3958;
31 *pPSO.Betrag@12716 := *pPSO.Betrag@3959; *sc.Amount@12717 := *sc.Amount@3960;
32 *pPSO.Header.Challenge@12718 := *pPSO.Header.Challenge@3961;
33 *pPSO.TransaktionsNR@12719 := *pPSO.TransaktionsNR@3962; *pChResp.Challenge@12720 := *pChResp.Challenge@3963;
34 *sc.storage@12721 := *sc.storage@3964; *sc.AmIGG:cu@12722 := *sc.AmIGG@3965;
35 *sha1.H@12723 := *sha1.H@3966; *sha1.Corrupted@12724 := *sha1.Corrupted@3967;
36 *sha1.Computed@12725 := *sha1.Computed@3968; *sha1.H@12726 := *sha1.H@3969;
37 *sha1.Computed@12727 := *sha1.Computed@3970; *sha1.Corrupted@12728 := *sha1.Corrupted@3971;
38 *sha1.Message_Block@12729 := *sha1.Message_Block@3972;
39 *sha1.Message_Block_Index@12730 := *sha1.Message_Block_Index@3973;
40 *sha1.Length_Low@12731 := *sha1.Length_Low@3974; *sha1.Length_High@12732 := *sha1.Length_High@3975;
41 *sha1.Message_Block@12733 := *sha1.Message_Block@3976; *sha1.H@12734 := *sha1.H@3977;
42 *sha1.Message_Block_Index@12735 := *sha1.Message_Block_Index@3978;
43 *sha1.Message_Block@12736 := *sha1.Message_Block@3979; *sha1.Length_High@12737 := *sha1.Length_High@3980;
44 *sha1.Length_Low@12738 := *sha1.Length_Low@3981; *storage.PalMEDB@12739 := *storage.PalMEDB@3982;
45 *storage.PalMEDB@12740 := *storage.PalMEDB@3983; *storage.PalMEDB@12741 := *storage.PalMEDB@3984;
46 *storage.PalMEDB@12742 := *storage.PalMEDB@3985; *storage.PalMEDB@12743 := *storage.PalMEDB@3986;
47 *storage.PalMEDB@12744 := *storage.PalMEDB@3987; irPack.len@12745 := irPack.len@3988;
48 irPack.buff@12746 := irPack.buff@3989; *packet.ID@12747 := *packet.ID@3990;
49 *sc.LastChallenge@12748 := *sc.LastChallenge@3991; *packet.Response@12749 := *packet.Response@3992;
50 *sc.state@12750 := *sc.state@3993; *sc.AmIGG@12751 := *sc.AmIGG@3994;
51 *sc.LastChallenge@12752 := *sc.LastChallenge@3995; *pMErr.Betrag@12753 := *pMErr.Betrag@3996;
52 *sc.TransactionNo@12754 := *sc.TransactionNo@3997; *sha1.Length_Low@12755 := *sha1.Length_Low@3998;
53 *sha1.Length_High@12756 := *sha1.Length_High@3999;
54 *sha1.Message_Block_Index@12757 := *sha1.Message_Block_Index@4000;
55 *sha1.Computed@12758 := *sha1.Computed@4001; *sha1.Corrupted@12759 := *sha1.Corrupted@4002;
56 *sha1.Message_Block_Index@12760 := *sha1.Message_Block_Index@4003;
57 (HandleData, PH_Control.c)
58
59 <OR>  ...
60 <OR>
61
62 NOT [r][deref]ctrl@6741.AmIGG:cu@3632
63 (20:ilu@4512 ==:i@4511 [r][deref]pCBParms:p@5575.rxLen:iu@6720)
64 [r][deref]ctrl@6741.State:iu@4506 == 1
65 [f]StrCompare:i@4861([r][deref]pCBParms:p@5575.rxBuff:p@6717, :p@4863)
66 [r][deref]ctrl@6741.confirm:cu@4588
67 ([f]DecryptStr:p@12806(crypto:s@12807, data:p@12686, len:iu@12687,
68 ...
69 *sha1.Corrupted@12834, *sha1.Message_Block_Index@12835) ==:i@12855 0:i@12857)
70
71 sc:p@12685 := [r][deref]ctrl@6741.SmartCardPtr:p@4886; data:p@12686 := [r][deref]pCBParms:p@5575.rxBuff:p@6717;
72 ...
73 (HandleData, PH_Control.c)
```

Abbildung 8.10: Pfadbedingung für PalME 1 in Rohform

Kapitel 9

VALSOFT

Alle in den vorherigen Kapiteln präsentierten Techniken sind im VALSOFT-System implementiert. VALSOFT stellt ein Analyseframework für ANSI-C-Programme dar, das vielfältigste Analysen ermöglicht, die dem Programmverstehen und der Validierung dienen. Für die Beschreibung derjenigen Komponenten, die keinen direkten Bezug zu Pfadbedingungen haben, wird auf die Ausführungen in [Kri03] verwiesen.

9.1 Constraint-Solving

Pfadbedingungen werden soweit wie möglich während ihrer Berechnung mittels BDDs und expliziter aussagenlogischer Vereinfachung (Quine/McCluskey, Minimalpolynome) minimiert. Für eine weitergehende prädikatenlogische Vereinfachung, die zudem Arithmetik und verschiedene Variablentypen erlaubt, können externe Constraint-Solver an den Pfadbedingungsgenerator angeschlossen werden. Ein Constraint stellt dabei für Pfadbedingungen eine Relation zwischen Zahlen und Variablen dar[1]. Die Anzahl der möglichen Variablen ist grundsätzlich nicht beschränkt ist, es handelt es sich also um n-äre Constraints, deren Domäne von der Plattform vorgegeben wird, auf der die analysierte Software ausgeführt werden soll. Wenn die Constraints erfüllt sind, existiert mindestens eine eindeutige Belegung für alle Variablen, mit der alle Constraints zur selben Zeit erfüllbar werden. Durch die Definition von sog. Eingabevariablen aus dem Variablenpool der Pfadbedingung können Zeugen über Beispielbelegungen generiert werden, die zur Programmlaufzeit den Informationsfluss reproduzieren können.

[1] Funktionsaufrufe in Pfadbedingungen repräsentieren deren Rückgabewerte und werden daher i.d.R. wie Variablen modelliert.

Die Variablen einer Pfadbedingung sind grundsätzlich existenziell quantifiziert, da sie nur dann Bestandteil einer Pfadbedingung sein können, wenn sie im Programm real vorkommen. Hierdurch lassen sich auch Constraint-Solver einsetzen, die auf Quantorenelimination [Wei97b, Wei99] basieren, wie z.B. Redlog [DS97, SW96, DSW98, DS].

Redlog ist auf reelle Zahlen beschränkt, so dass zusätzlich das Constraint-Logic-Programming-System ECLiPSe [IP02] für natürliche und reelle Zahlen zum Einsatz kommt. Constraint-Logic-Programming (CLP,[FA97, Stu02]) kombiniert logische Programmiersprachen, wie z.B. Prolog, mit Constraint-Solvern.

9.1.1 Reduce/Redlog

Redlog steht für „Reduce Logic“ und stellt eine Erweiterung des Computer-Algebra-Systems Reduce um ein Computer-Logic-System dar. Es basiert auf symbolischen Algorithmen für die Prädikatenlogik erster Stufe und arbeitet für verschiedene Sprachen und Theorien. Derzeit stehen die folgenden drei Kontexte zur Auswahl: Ordered Fields, Discretely Valued Fields und Algebraically Closed Fields.

Der für Pfadbedingungen sinnvolle Kontext ist Ordered Fields, da die Vereinfachung auf geordneten reellen Zahlen basiert. Constraint-Solving-Verfahren sind jedoch nicht universell übergreifend einsetzbar, sondern müssen für ein bestimmtes Problem und spezielle Erfordernisse gezielt ausgewählt werden [BC93, MS98].

Beispiel 9.1 (Beispiel für das Auflösen mittels Redlog)
Sei $x + b \geq 0 \wedge ax + c \leq 0$ ein Monom der Pfadbedingung. Durch die existenzielle Quantifizierung ist der Ausdruck gleichwertig zu $\exists a, b, c, x : x + b \geq 0 \wedge ax + c \leq 0$. Seien nun a, b und c die Eingabevariablen des Programms, die nicht eliminiert werden sollen, dann berechnet Redlog ausschließlich für x Belegungen, die nur noch von a, b und c bzw. konstanten Werten abhängen. In diesem Fall erzeugt Redlog die Lösungen:

- Elimination: $x := \frac{-c}{a}$, Beispielbelegung: $a^2b - ac \geq 0 \wedge a \neq 0$
- Elimination: $x := \infty$, Beispielbelegung: $a < 0 \vee (a = 0 \wedge c \leq 0)$

Redlog garantiert immer die Äquivalenz und Erfüllbarkeit der Beispielbelegungen, so dass der Anwender z.B. mit $a = 2$, $b = 1$ und $c = 1$ bereits einen Zeugen für einen möglichen Informationsfluss bilden könnte. Jedoch lassen sich nicht alle Lösungen direkt umsetzen, wie der zweite Lösungsvorschlag mit $x = \infty$ zeigt, der in ANSI-C-Programmen mit eingeschränktem Wertebereich für die Variablen nicht möglich ist. I.d.R. bedarf jede Beispielzeugenberechnung der manuellen Überprüfung.

Für die Pfadbedingung $PC(y,x) \equiv \exists i,j : (i+3 = 2*j-42) \wedge (i > 10)$ aus Abschnitt 4 liefert Redlog für Eingabevariable j die Beispielbelegung $2j > 55$ und eliminiert i zu $i := 2j-45$. Ohne Angabe einer Eingabevariablen wertet Redlog die Pfadbedingung direkt zu *true* aus. Generell nimmt Redlog zur Erfüllung von Formeln immer die größtmöglichen Werte (z.B. ∞) an. Die zweite Pfadbedingung $PC(y,x) \equiv \exists i,j : (i+3 = 2*j-42) \wedge (i > 10) \wedge (j < 5)$ wertet Redlog korrekt zu *false* aus, da es keine Lösungen geben kann.

Aufgrund der konservativen Approximation der Abhängigkeitsanalyse, des Program-Slicings und Pfadbedingungen kann es auch „falsche Alarme" geben, da Pfadbedingungen nur notwendige Bedingungen darstellen, die nicht immer zugleich hinreichend sein müssen. Für die Softwaresicherheitsanalyse ist dies jedoch ohne Bedeutung, da die Beweislast bei Gutachten immer bei den Softwareherstellern liegt.

9.1.2 ECLiPSe

ECLiPSe basiert im Gegensatz zu Reduce/Redlog nicht auf einem Computer-Algebra-System. Es handelt sich um eine Constraint-Programmiersprache, die den vollen Umfang von Prolog umfasst. Ein CLP-System arbeitet mit logischen Kalkülen, die aus Atomen, Zielen, Klauseln und Programmen bestehen. Über ein Zustandsübergangssystem wird versucht, einen Endzustand *abzuleiten* (engl. deduction).

Die zu lösenden Constraints der Pfadbedingung müssen in eine Klauselform überführt werden, die nicht mehr der natürlichen Darstellung von Pfadbedingungen, wie z.B. bei Redlog, entspricht.

Der Einsatz von ECLiPSe bringt den Vorteil, ein CLP(BNR)-System zu erhalten, das sowohl für boolesche und reelle Werte (siehe Redlog) Lösungen liefert, als auch für natürliche Zahlen. Die verwendete „ic"-Bibliothek (interval constraint library) ermöglicht das Lösen über abgeschlossene Wertebereiche von natürlichen Zahlen und über ununterbrochene Intervalle von reellen Zahlen.

9.1.3 Constraint-Solving in VALSOFT

Die Ausgangsform der Pfadbedingung für Constraint-Solver ist die sog. Rohform in DNF, die in Abbildung 8.10 bereits gezeigt wurde. Sie enthält zur Unterscheidung von gleichlautenden Variablen die Knotennummer und sowohl den Operanden-, als auch den Operatortyp. Aus der Rohform wird jedes Monom in eine Meta-Repräsentation konvertiert, die vom jeweiligen Constraint-Solver gelöst wird.

Die Gleichungen und Ungleichungen können beliebige arithmetische Ausdrücke enthalten, wobei nur die Grundoperatoren des ANSI-C-Sprachstandards $+, -, *, /, \%, =, \neq, <, \leq, >, \geq, \wedge, \vee, \neg$ bedeutsam sind. Sollten Operatoren vorkommen, die von den Constraint-Solvern nicht direkt lösbar sind, wie z.B. Bitoperationen, werden neue Variablen gebildet, die für den nicht-auflösbaren Term stehen.

Die einzelnen Komponenten des Systems bestehen aus: 1. einem über reguläre Ausdrücke konfigurierbaren Parser für die Pfadbedingungs-Rohform, 2. einem Zerlegungssystem, das die in Disjunktiver Normalform vorkommende Pfadbedingung in einzelne Monome und Literale zerlegt, 3. einem Umformer, der aus der bisherigen Meta-Repräsentation syntaktisch und in Abhängigkeit von der Funktionalität der externen Constraint-Solver semantisch korrekte Ausdrücke generiert, 4. einem Anreicherungsystem, das die aufzulösenden Variablen der Pfadbedingung um exakte Wertebereiche anreichert und 5. einem Ausgabesystem, das die Ergebnisse der Auflösung allgemein verständlich als Text oder in LaTeX ausgibt.

Constraint-Solver wie Redlog eliminieren Variablen besonders häufig unter der Annahme, dass diese Werte wie ∞ und $-\infty$ annehmen, die außerhalb des Wertebereichs von ANSI-C-Variablen liegen. Daher wird jede Variable in Abhängigkeit ihres Typs durch zwei Bedingungen nach oben und unten beschränkt. Diese Beschränkung ist von der Zielmaschine abhängig und wird z.B. durch die Datei „limits.h“ unter Linux festgelegt.

Der Anschluss der Constraint-Solver wurde von Elisabeth Angerer-Grubmüller [AG04] implementiert.

9.1.4 Fazit

Constraint-Solving von beliebigen arithmetischen ANSI-C-Ausdrücken verursacht eine Reihe von Problemen. Zum einen realisieren die Constraint-Solver nicht exakt die Gleitpunktarithmetik des ANSI-C-Standards, so dass es zu Rundungsfehlern kommen kann. Zum anderen enthält der Standard keine Definition für ein exaktes Verhalten bei Wertebereichsüberläufen, so dass dies nur abhängig vom verwendeten Compiler ist. Daher stellen die resultierenden Variablenbelegungen eine Abschätzung dar, unter der ein Informationsfluss zur Laufzeit sichtbar wird. Nicht jede gelieferte Variablenbelegung kann in der Realität durchgeführt werden, und Auswertungen zu *false* müssen mit der Pfadbedingung auf Plausibilität überprüft werden.

Die Fallstudien in Kapitel 8 haben gezeigt, dass aufgrund eines arithmetischen Minimalismus und der guten Lesbarkeit von Pfadbedingungen ein Constraint-Solving selten notwendig ist, um Relationen zwischen Literalen

zu erkennen. Meistens lassen sich die Wertebereiche von Variablen sogar direkt ablesen.

9.2 Visualisierung von PDGs mittels Overlaying

Im bisherigen VALSOFT-System wurden Programmabhängigkeitsgraphen in einer grafischen Form mittels spezieller Layoutingverfahren präsentiert. Grafische Darstellungen haben den Vorteil, dass sie sehr intuitiv sind. Für Program-Slices hat sich jedoch gezeigt, dass eine textuelle Darstellung der grafischen überlegen ist.

Für die Analyse selbst, z.B. der Auswahl und Traversierung von Kanten und Knoten, führt kein Weg an einer grafischen Darstellung vorbei, da rein textuelle Repräsentationen keine Abhängigkeiten darstellen.

Die Abbildung 9.1 auf Seite 158 zeigt die Funktion „IrHandler“ des Programms „PalME“. Hierbei wird schnell deutlich, dass feingranulare PDGs schon bei mittelgroßen Programmen extrem groß werden können und zudem eine unüberschaubare Anzahl von Abhängigkeitskanten enthalten. Die größte gemessene Knoten-Kanten-Ratio in den Fallstudien liegt bei 1:50. Die gerade noch sichtbare vertikale Knotenkette inmitten des Bildes stellt einen einzigen Funktionsaufruf dar. Links davon befinden sich ankommende Datenabhängigkeitskanten, rechts davon Summarykanten.

Um dieses Problem zu lösen und nicht nur auf eine textuelle Repräsentation auszuweichen, wurde eine neue Sicht auf Programmabhängigkeitsgraphen entwickelt, die Skalierungsprobleme weitgehend umgeht. Der Quelltext bleibt als textuelle Repräsentation bestehen und wird, mittels Overlaying-Techniken, von Abhängigkeitskanten des PDGs überlagert. Über Parameter werden die Abstände der Kanten zueinander konfiguriert, um sie optimal auf Schriftgrößen und Graphkomplexität abzustimmen. Ein virtueller Cursor stellt für jeden PDG-Knoten einen direkten Bezug zum dargestellten Quelltext her. Einem Prüfingenieur müssen zum Verfolgen von Abhängigkeiten nicht immer *alle* möglichen Kanten präsentiert werden, sondern nur diejenigen, die entlang des gerade beobachteten Pfadknotens liegen, also alle ankommenden und abgehenden Kanten an der Cursorposition.

Die Abbildung 9.2 auf Seite 159 zeigt das neue Visualisierungswerkzeug für Abhängigkeitsgraphen. In Zeile 331 ist derselbe Knoten markiert, der in Abb. 9.1 als schwarzer Knoten in der oberen rechten Ecke zu sehen ist.

An jeder Cursorposition wird der Quelltext mit farblich unterschiedenen ankommenden und abgehenden Kanten überlagert. Zudem werden Listen dieser Kanten in separaten Fenstern gezeigt, die einen Hyperlink an entsprechende Quelltextpositionen ermöglichen. Die Gesamtübersicht über alle

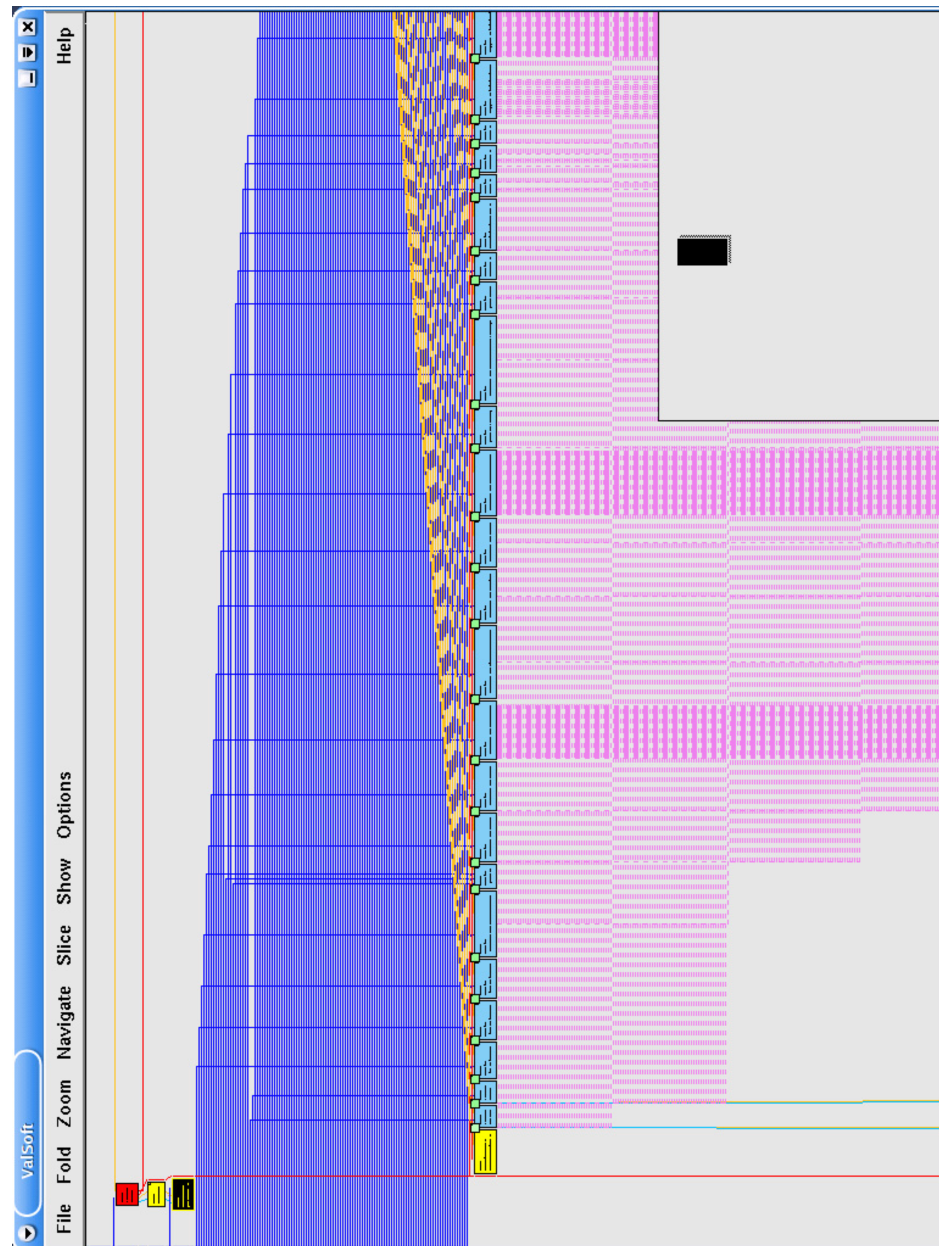


Abbildung 9.1: Darstellung von „PalME“ mittels grafischer Repräsentation

definierten Funktionen wird in einem weiteren Fenster, und die Übersicht der Dateien wird als Reiter präsentiert. Die Berechnung von Pfadbedingungen erfolgt über ein Kontrollfeld, in dem Start- und Endknoten-Listen verwaltet und die Berechnung in einer Queue ausgeführt wird. Sobald die Ergebnisse vorliegen, können diese in einem separaten Fenster überprüft werden.

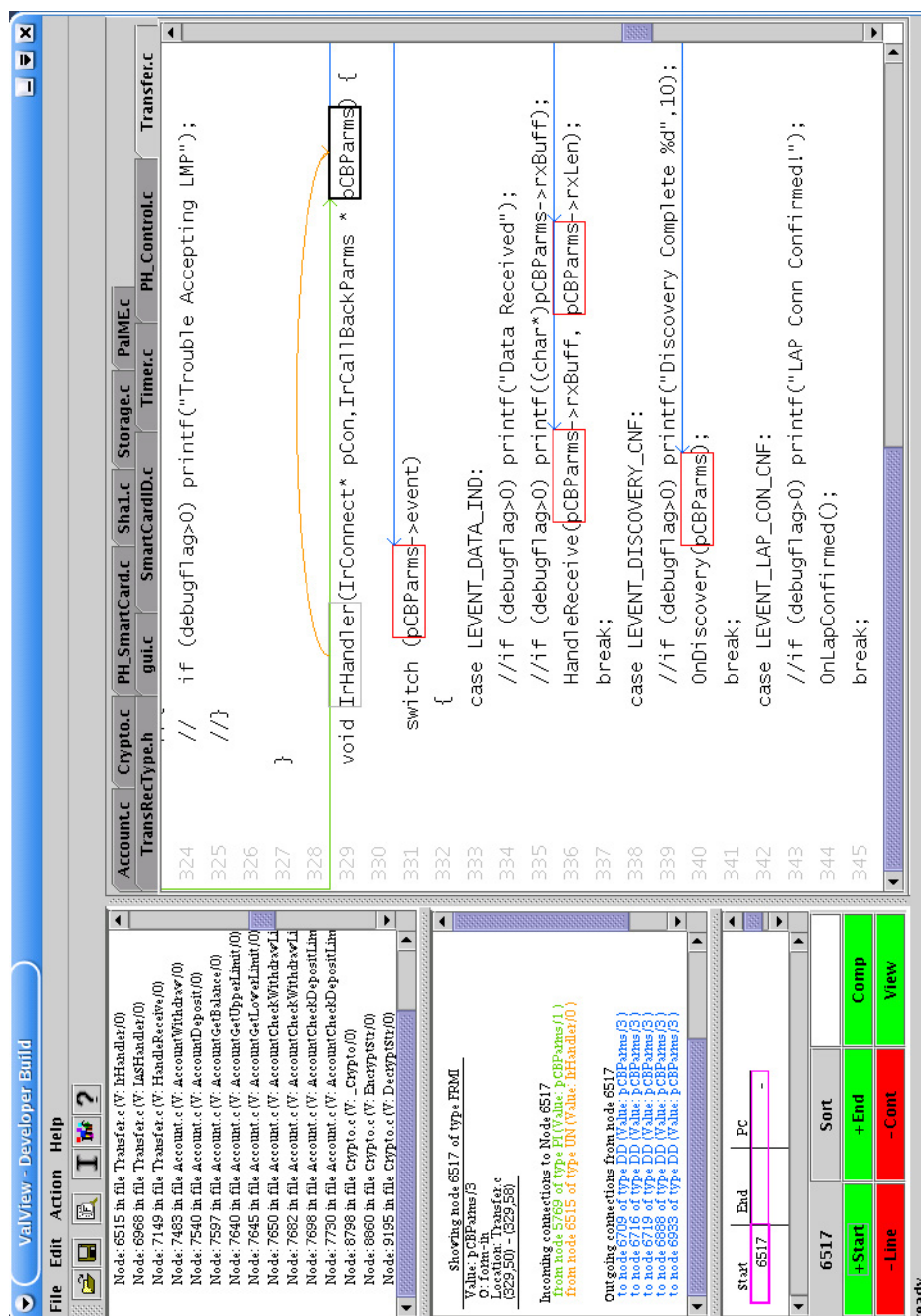


Abbildung 9.2: Darstellung von „PalME“ mittels Overlaying-Techniken

Die Implementierung wurde hauptsächlich von Stephan Bösebeck [Bös02] durchgeführt.

	Komponenten des Systems	LOC
	BDD Paket „BuDDy“	13293
	Constraint-Solving	12270
	Visualisierung	9516
S	Ausführungsbedingungen	792
O	Pfadbedingungen	7445
L	Zyklenzerlegungen	2134
V	Minimierung	2718
E	Ausgabe	1327
R	Infrastruktur	14731
		64226

Tabelle 9.1: Komponenten zur Pfadbedingungsberechnung

9.3 Der Pfadbedingungsgenerator „Solver“

Der Pfadbedingungsgenerator „Solver“ verwendet von der VALSOFT-Infrastruktur die PDG-Bibliothek zur Traversierung des Graphen und den Program-Slicer, um einen minimalen Teilgraphen zwischen zwei Knoten zu generieren. Der „Solver“ ist ein Kommandozeilen-Werkzeug, das als Ergebnis die minimierte Pfadbedingung in der bekannten Rohform liefert, die alle Daten enthält, um sowohl typisiertes Constraint-Solving durchzuführen, als auch anhand der Pfadbedingung direkte Rückschlüsse in den Quelltext zuzulassen. Als Nebenprodukt wird zu jeder Pfadbedingung der BDD erzeugt und eine Reportdatei verfasst, die den Bezug von BDD-Knoten zu Literalen herstellt.

Der „Solver“ besteht aus den obigen Komponenten in Tabelle 9.1 und berechnet Pfadbedingungen entsprechend der Techniken der bisherigen Kapitel nach folgendem Ablauf:

1. Nach dem Einlesen des PDG werden die Φ-Bedingungen berechnet und der Prozedur-Aufrufgraph erstellt. Mittels dieses Graphen werden später sowohl rekursive Aufrufe überprüft, als auch Tests vorgenommen, ob Zielknoten über Funktionsabstiege erreichbar sind.

2. Für den Chop werden die Starken Zusammenhangskomponenten ermittelt, die später in eine Hierarchie von geschachtelten Zyklen zerlegt werden.

3. Die Starken Zusammenhangskomponenten werden entsprechend des gewählten Zerlegungsverfahrens virtuell zerlegt, ohne den Programmabhängigkeitsgraphen zu ändern. Dabei entsteht eine Liste von Zyklenhierarchien mit entsprechenden Ein- und Austrittsknoten.

4. Für jede Hierarchie werden die Ein- und Austrittsknoten entlang

größer werdender Zyklen propagiert und die reale Reduzibilität bzw. Irreduzibilität bestimmt.

5. Die Pfadaufzählung basiert auf einer rekursiven Tiefensuche, die Informationen über den aktuellen Chop/Zyklus hat und für die Suchbaumbegrenzung die aktuelle und bisherige schwächste Bedingung entlang des Pfades enthält. Jede Bedingung wird dabei als BDD verwaltet.
6. Beim Erreichen eines Zyklus am aktuellen Knoten wird so lange entlang der Zyklenhierarchie abgestiegen, wie der Knoten einen Eintrittsknoten darstellt. Der Aufbau der Pfadbedingung erfolgt innerhalb des gegenwärtigen Zyklus sowie von den Austrittsknoten zu aktuellen Endknoten der Pfadbedingung. Jede Hierarchie wird bottom-up abgearbeitet.
7. Während der Pfadaufzählung wird für jeden traversierten Knoten einmalig seine Ausführungsbedingung als BDD im Kontrollabhängigkeitsgraphen berechnet. Jede Ausführungsbedingung besteht dabei aus wiederverwendbaren atomaren Bedingungen eines gemeinsamen Pools, der sich aus den Prädikaten des Programms zusammensetzt. Die atomaren Bedingungen bestehen aus einer Baumstruktur und werden über ein Mapping innerhalb von BDDs als Nummern verwaltet. Eine automatische Auswertung von einfachen Prädikaten ($=,\neq$) mit typgleichen Operanden (z.B. für Array-Indizes) findet auf Basis des GNU C-Compilers „gcc“ statt.
8. Jede Kante im Chop wird entsprechend ihres Typs nach einer festgelegten Reihenfolge behandelt: 1. intraprozedurale Abhängigkeitskanten, 2. Summarykanten, 3. Aufrufkanten, 4. Parameterkanten. (Beispiel: Liegt der aktuelle Endknoten der Pfadbedingung in derselben Funktion wie der Ausgangsknoten einer Summarykante, können die abgehenden Parameterkanten bei nicht-rekursiven Aufrufen ignoriert werden, da sie die Funktion verlassen und den Endknoten nicht erreichen würden).
9. Datenabhängigkeitskanten generieren je nach Wahl spezielle Bedingungen über Arrays oder Abstrakte Datentypen.
10. Die vollständige Pfadbedingung als BDD wird in eine Baumstruktur übertragen und in Disjunktive Normalform übersetzt. Während dieser Transformation wird eine Minimierung boolescher Funktionen mehrfach durchgeführt, um ein exponentielles Wachstum der Baumstruktur zu begrenzen. Die Minimierung boolescher Funktionen basiert neben Quine/McCluskey [Qui55, Qui52, McC56] auch auf der Berechnung von Minimalpolynomen [Weg89].
11. Die resultierende Pfadbedingung in minimaler Disjunktiver Normal-

form wird gelayoutet, anhand der PDG-Informationen mit Typen angereichert, eindeutige Substitutionen (z.B. Φ-Bedingungen) vorgenommen und die Pfadbedingung ausgegeben.

12. Die Pfadbedingung kann anschließend manuell überprüft werden oder mittels Constraint-Solver nach Eingabevariablen aufgelöst werden. Sie belegt in Form eines „Zeugen“ den Informationsfluss zwischen gegebenen Programmpunkten.

9.4 Symbolischer Ablauf einer Softwareanalyse

Die Abbildung 9.3 stellt für diese Arbeit abschließend und unkommentiert den Ablauf einer Softwareanalyse zwischen Abnehmer und Aktuatoren dar.

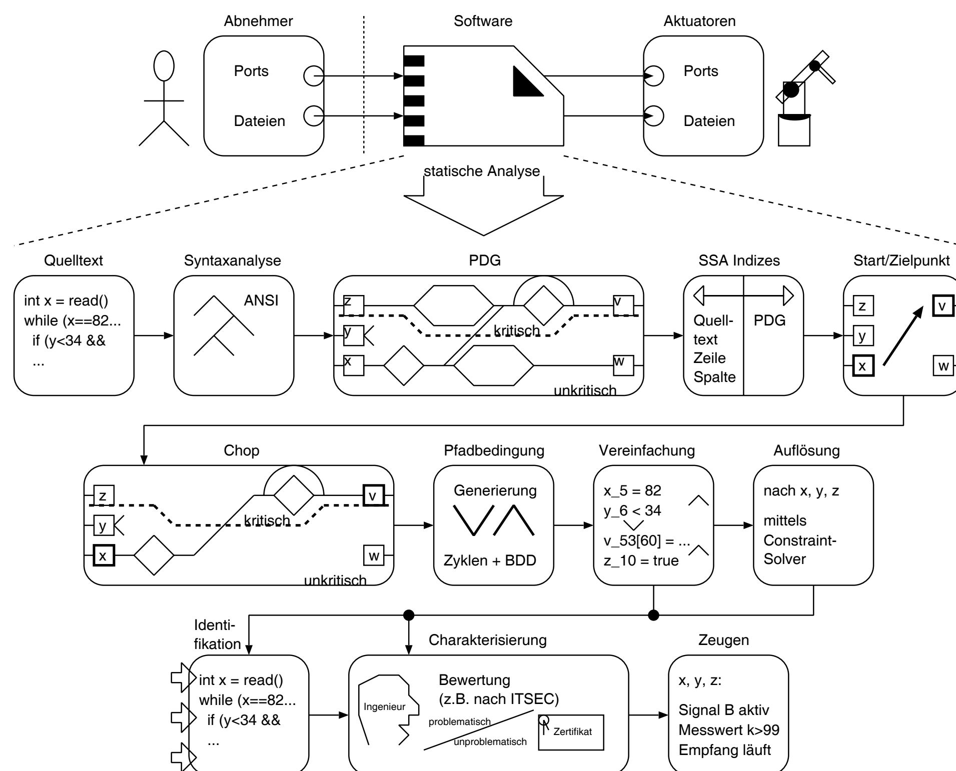


Abbildung 9.3: Ablaufplan einer Softwareanalyse mittels VALSOFT

Kapitel 10

Verwandte Arbeiten

Auf dem Gebiet der statischen Softwareanalyse gibt es derzeit keine Arbeiten, die Pfadbedingungen in Abhängigkeitsgraphen berechnen und sich auf reale interprozedurale Programme anwenden lassen. Es gibt jedoch Arbeiten in ähnlicher Richtung, die auf Kontrollflussgraphen basieren bzw. unter Aufgabe einer konservativen Berechnung arbeiten.

Debugging und Testdatengenerierung

Im Bereich Debugging gibt es die Arbeiten von [GBR98, GMS98, GWZ94, MO91], bei denen constraint-basierte Testdaten in Kontrollflussgraphen berechnet werden. Mittels der berechneten Testdaten kann der Kontrollfluss des Programms gezielt gesteuert werden. Im Unterschied zur vorliegenden Arbeit können hierbei keine Informationsflüsse, für die neben dem Kontrollfluss auch Kontroll- und Datenabhängigkeiten betrachtet werden müssen, berechnet werden. Daher lassen sich keine Bedingungen für Informationsflüsse aufstellen, so dass die genannten Arbeiten für die Manipulationsprüfung in Abhängigkeitsgraphen nicht geeignet sind.

Den umgekehrten Weg versuchen [TRC93, PC90] mit ihrem Relay-Modell. Es ist im Ansatz ähnlich zu Pfadbedingungen und berechnet Transfer-Bedingungen in Kontrollflussgraphen unter Einbeziehung des Datenflusses, unter denen Fehler im Pogramm entlang der möglichen Pfade weiterpropagiert werden. Das Modell beruht auf dem Wissen über korrekte Variableninhalte und die automatische Auswertung der Bedingungen. Durch die exponentielle Komplexität ist das Modell für reine Fallunterscheidungen geeignet, das Vorhandensein von Schleifen setzt schon gegebene Abstiegsfunktionen voraus. Ergebnisse über die Analyse realer Programme liegen nicht vor.

Bei den oben aufgeführten Arbeiten liegt der Schwerpunkt weder auf Sicherheitsanalyse noch auf den Einsatz in real existierenden Systemen. So wird nicht immer die konservative Approximation eingehalten und der Einsatz nur an kleinen Programmen gezeigt, die keinen Aufschluss auf echte Skalierung geben. Der VALSOFT-Pfadbedingungsgenerator wurde zudem unabhängig von speziellen Domänen gehalten, so dass er universell einsetzbar ist und durch den Anschluss von domänenspezifischen Constraint-Solvern beliebig erweitert werden kann.

Ein weiteres System ist PREfix [BPS00b, BPS00a], das typische Programmierfehler, wie z.B. Speicherlecks und Zugriff auf Nullpointer aufdecken kann, indem es den Kontrollfluss analysiert und erreichbare Pfade in Funktionen mittels einer virtuellen Maschine stichprobenartig simuliert. Das Verfahren wird bei Microsoft eingesetzt und ist aus Performancegründen nicht konservativ. Es liefert nur einfache Pfadbedingungen, basierend auf dem Kontrollfluss.

Zu einer ähnlichen Kategorie gehört die Shape-Analyse [NRS98, Nor99, NRS00], mit der Speichergraphen als Grundlage einer abstrakten Repräsentation erstellt werden, um Speicherzugriffsfehler zu finden. Das Verfahren ist im Gegensatz zu PREfix präzise, jedoch nur für die Überprüfung von linearen Listen realisiert. Das System ESC/Modula3 [LN98] verfolgt ein ähnliches Ziel, erfordert jedoch die Annotation des Programms mit Assertions und setzt zur Auswertung echte Verifikation ein. Für Java existiert ein ähnliches System [FLL+02].

Im Unterschied zu den genannten Verfahren basiert der VALSOFT-Pfadbedingungsgenerator nicht auf dem Auffinden von möglichen Programmierfehlern, sondern auf der Entdeckung und Beschreibung unerwünschter Informationsflüsse.

CQual [FTA02] ist ein typ-basiertes System, mit dem z.B. Locking-Fehler im Linux-Kernel gefunden wurden. Durch Annotationen des Programms und Kodierung der Flussinformationen in Typen stellen illegale Flüsse registrierbare Typfehler dar. Im Kontrast dazu liefert der VALSOFT-Pfadbedingungsgenerator ohne notwendige Annotationen direkte Zeugen für ermittelte Informationsflüsse, die i.d.R. aussagekräftiger als reine Typfehler sind und nicht von der Vollständigkeit der Annotationen abhängen.

Program-Slicing

Program-Slicing basiert nicht immer auf Kontroll- und Datenabhängigkeiten innerhalb von Graphen. [FRT95] berechnet mit parametrisierbarem Program-Slicing statische und dynamische Slices, die abhängig von gegebenen Constraints über den Eingabevariablen des Programms sind (ähnlich

Abschnitt 6.4). Die Slices werden dadurch kleiner und präziser als unbedingte Slices, sind jedoch nur für Programmläufe innerhalb der Constraints gültig. Über Rewriting werden die Slices ermittelt, indem die Semantik der Sprache in Rewrite-Regeln angegeben und dann durch Constraints angereichert wird. [CCD98] verfolgt einen ähnlichen Ansatz mit bedingtem Slicing. Beide Verfahren sind jedoch nicht auf real existierende Programme angewendet worden und liefern im Gegensatz zu Pfadbedingungen nur binäre Informationen.

Arrays und Parallelisierung

Präzise Abhängigkeiten zwischen Arrayfeldern werden z.B. für die automatische Parallelisierung von Schleifen benötigt. Das Omega-Projekt [PW98] löst Constraints über diese Array-Abhängigkeiten mittels Presburger Arithmetik. Der VALSOFT-Ansatz in Abschnitt 6.2 macht Pfadbedingungen präziser, ist jedoch nicht so weitgehend wie der Omega-Ansatz und setzt auch keine Presburger Arithmetik ein, da Arrays nur einen kleinen Teil von Pfadbedingungen ausmachen. Jedoch ist es möglich, den Pfadbedingungsgenerator um speziellere Techniken zur Array-Abhängigkeitsbehandlung zu erweitern.

Abstrakte Datentypen

Der Pfadbedingungsgenerator verwendet Spezifikationen über abstrakte Datentypen, um Pfadbedingungen zu präzisieren. Reps hat in [Rep00] gezeigt, dass Gleichungen für abstrakte Datentypen im interprozeduralen Fall unentscheidbar sind. Der im Abschnitt 6.3 vorgestellte Ansatz basiert auf Term-Rewriting modulo Datenabhängigkeiten (statt Abhängigkeitsanalyse) und ist für die intraprozedurale Verwendung von Gleichungen über abstrakte Datentypen vorgesehen. Dadurch ist dieser Ansatz nicht von Entscheidbarkeitsproblemen betroffen und als universelles Plugin unabhängig von der Pfadbedingungsberechnung.

Informationsflusskontrolle

Pfadbedingungen sind ein ideales Werkzeug um den präzisen Informationsfluss zwischen Sicherheitsdomänen zu untersuchen. Ansätze zur Informationsflusskontrolle gibt es von Smith/Volpano [SV98], die Bell/La Padula-Bedingungen mit einem speziellen Typsystem für imperative Sprachen nach Denning [DD77] überprüfen können. Denning hat Sicherheitsdomänen als Verband dargestellt und definiert, wie für eine einfache Sprache festgestellt

werden kann, ob Informationsflüsse zwischen unterschiedlichen Sicherheitsebenen auftreten. Der Ansatz von Smith/Volpano und Denning ist nur flussinsensitiv, so dass die Präzision nicht derjenigen des VALSOFT-Slicer und Pfadbedingungsgenerators entspricht. Da definierte Eichpfade schon als Sicherheitsbereiche angesehen werden können, lassen sich mit VALSOFT jetzt schon Informationsflüsse zwischen grundlegenden Sicherheitsebenen bestimmen.

Weitere typ-basierte Arbeiten sind in [SA03] aufgeführt, die auf Programmanalyse basieren, jedoch in keiner Weise so weitreichend wie Pfadbedingungen sind, auch wenn einige davon Datenflussanalyse und Program-Slicing einsetzen.

Generatoren für Programmanalysen

Generatoren bieten Frameworks für verschiedene Analysen an. Mit PAG [Mar98] oder TVLA [LAS00] lassen sich statische Programmanalysen realisieren. Die Entscheidung, ein eigenes Framework wie VALSOFT zu entwickeln lag an der Tatsache, dass keines dieser Systeme die volle Semantik von ANSI-C abdeckt und Pfadbedingungen feingranulare Programmabhängigkeitsgraphen benötigen. Zudem ließen sich die Skalierungstechniken (siehe Abschnitt 7) nur mit eigenen Datenstrukturen effizient lösen.

Model Checking

Sicherheitseigenschaften in Form von temporal-logischen Formeln lassen sich für Hard- und Softwaresysteme mittels Model Checking überprüfen. Das Bandera-Projekt [CDHR00] und SLAM [BR00, BR01] generieren endliche Modelle aus Software, die gegen eine vorgegebene Spezifikation in temporaler Logik getestet werden können. Infolge des hohen Abstraktionsgrads der Modelle ist nicht gewährleistet, dass alle notwendigen Informationen zur Entdeckung von illegalen Informationsflüssen enthalten sind oder die Modelle eine noch genügende Präzision haben. Da Abhängigkeitsgraphen auch als endliche Modelle angesehen werden können, lassen sie sich ebenso temporal-logisch überprüfen, jedoch ist die Semantik von Abhängigkeiten zu Kontroll- und Datenflüssen grundverschieden.

Kapitel 11

Zusammenfassung

In dieser Arbeit wurde gezeigt, dass Pfadbedingungen ein sinnvolles Werkzeug zur Beeinflussungsanalyse sicherheitskritischer Programme sind. Außerdem stellen sie ein gutes Werkzeug zum Programmverstehen dar, zumal sie die Ursachen von Informationsflüssen in beliebigen Programmen aufzeigen können. Die Arbeit hat effiziente Techniken für interprozedurale Pfadbedingungen präsentiert, Skalierungsmethoden gezeigt und in umfangreichen empirischen Untersuchungen die Anwendbarkeit auf reale Programme belegt.

11.1 Pfadbedingungen und ihre Anwendung in der Softwaresicherheitstechnik

Die Eigenschaften von Pfadbedingungen können mit den Techniken dieser Arbeit zusammenfassend wie folgt präsentiert werden:

- Pfadbedingungen können als Verbesserung von Program-Slicing nicht nur binäre Informationen berechnen, sondern insbesondere informative Bedingungen, unter denen Informationsflüsse stattfinden. Mithilfe dieser Bedingungen können zudem die Ursachen der Informationsflüsse leichter bestimmt werden.
- Pfadbedingungen können Program-Slices reduzieren und damit präziser machen. Sie zeigen, dass manche Slices in der Realität unmöglich sind.
- Pfadbedingungen bleiben exponentiell, jedoch lassen sie sich durch den Einsatz intelligenter Techniken, wie
  - Intervallanalyse für Zyklen,

  - einer interprozeduralen Analyse mit wiederverwendbaren Pfadbedingungen,
  - Binäre Entscheidungsgraphen,
  - Verfahren aus der Spieletheorie und
  - einer Pfadauswertungsbegrenzung für unzerlegbare Zyklen

  trotzdem effizient für reale Programme berechnen. Hierbei zeigte sich, dass unterschiedliche Zyklenzerlegungsverfahren je nach Fallstudie stark variierende Performancegewinne liefern.

- Pfadbedingungen bieten die Möglichkeit der Anreicherung um Zusatzbedingungen, die spezielle Datenstrukturen oder Hardwarerestriktionen beschreiben.
- Pfadbedingungen lassen sich um dedizierte Constraint-Solver erweitern und können domänespezifische Zeugen für illegale Informationsflüsse oder andere Abhängigkeitseigenschaften liefern.
- Pfadbedingungen basieren auf Abhängigkeitsgraphen mit Quelltextinformationen. Sie lassen sich daher im Gegensatz zu anderen Verfahren sowohl von Prüfingenieuren als auch von Programmierern einsetzen und sind selbst für Nicht-Experten lesbar und verständlich.
- Pfadbedingungen haben ihre Anwendbarkeit durch die Entdeckung einer Sicherheitsverletzung in einem größeren Programm mit sicherheitskritischem Bezug demonstiert.
- Pfadbedingungen sind prinzipiell sprachunabhängig, da sie nur auf der abstrakten Form von Abhängigkeitsgraphen basieren. Der Einsatz für ANSI-C-Programme zeigt daher nur die erste praktische Anwendbarkeit.

11.2 Ausblick

Die Aufgabe dieser Arbeit war die Realisierung eines Systems mit neuen Techniken, mit dem sich für reale Programme Pfadbedingungen in Abhängigkeitsgraphen berechnen lassen. Das System ist für die Analyse von ANSI-C-Programme realisiert und hat seinen Nutzen gezeigt. Zukünftige Arbeiten können sowohl von Pfadbedingungen als auch von den eingesetzten Techniken profitieren. Im Folgenden sind Vorschläge für darauf aufbauende Arbeiten angeführt:

- Java ist eine Programmiersprache, die in eingebetteten Systemen wie Smartcards (wenn auch nur in Form einer reduzierten Version)

zum Einsatz kommt. Pfadbedingungen sollten daher für Java-basierte Abhängigkeitsgraphen angewendet werden. Dies erfordert jedoch eine Modellierung von Objekten und der statischen Approximation der dynamischen Bindung und stellt daher eine Erweiterung imperativer Abhängigkeitsgraphen dar. Wichtige Schritte wurden bereits in [HS04] durchgeführt.

- Pfadbedingungen liefern Zeugen für Informationsflüsse, können jedoch bei hoher algorithmischer Komplexität der Bedingungen nicht manuell, sondern nur mit Constraint-Solvern ausgewertet werden, um reale Variablenbelegungen zu erhalten. Der Einsatz von Redlog [DS97], Maple [Map] und ECLiPSe [IP02] wurde bereits untersucht. Andere Systeme wie Mathematica [Res], Omega-Test (Pugh) [PW98] oder weitere Constraint-Logic-Verfahren liefern je nach Einsatz von Pfadbedingungen möglicherweise noch bessere Ergebnisse. Pfadbedingungen können zudem nicht nur nach Typen, sondern auch nach ihrer Struktur klassifiziert und dem jeweiligen optimalen Solver zum Auflösen geschickt werden.
- Pfadbedingungen wurden in dieser Arbeit derzeit für Programme bis 25 kLOC untersucht. Die Arbeit hat gezeigt, dass die Kantenzahl hierbei eine bedeutende Rolle bei der Laufzeit spielt. Verbesserungen können nicht nur neue Zerlegungsverfahren bringen, sondern eine Änderung der Struktur von Abhägigkeitsgraphen, die speziell auf eine Reduzierung von Knoten und Kanten optimiert ist, indem teilweise der feingranulare Ansatz durch eine Bündelung von Abhängigkeiten reduziert wird.
- Pfadbedingungen der statischen Analyse können durch dynamische Informationen erweitert und präzisiert werden. Wenn z.B. bestimmte Programmabläufe fest vorgegeben sind, können sie über dynamische Traces zur Laufzeit aufgezeichnet werden und bieten damit die Möglichkeit, Pfadbedingungen nur für diese Traces und unter Berücksichtigung mehrfacher Schleifeniterationen mit konkreten Variablenwerten zu berechnen. Dies schränkt den statischen Programmabhängigkeitsgraphen ein und reduziert die Berechnungskomplexität.

Besonders wünschenswert wäre eine Anwendung des Pfadbedingungsgenerators auf weitverbreitete kommerzielle sicherheitskritische Software, um echte – bisher unentdeckte oder verschwiegene – Sicherheitslöcher zu finden. Man muss aber davon ausgehen, dass ein tatsächlicher Sucherfolg mit der Pfadbedingungstechnologie sehr diskret behandelt und korrigiert werden dürfte.

Anhang A

Weitere BDDs

Binäre Entscheidungsgraphen stellen ebenso wie die daraus generierten Pfadbedingungen Ergebnisse von Informationsflüssen dar. Sie können unabhängig einer Disjunktiven Normalform genutzt werden. Für die in Abschnitt 8 durchgeführten Fallstudien werden die entsprechenden BDDs zur Veranschaulichung der Größe und Komplexität angegeben.

A.1 BDDs zur Fallstudie „Wackeltisch“

Für die Erkennung einer Sicherheitsverletzung in der zweiten Wackeltisch-Fallstudie (Abschnitt. 8.1.3 auf Seite 139) erfolgt zudem die Beschreibung der BDD-Knotennummern in folgendem Report, der sich auf den BDD in Abbildung A.2 bezieht:

```
1:    ziel_nicht_erreicht@9361
2:    ([f]abs@9393([r]platte@9394.y@9395) >@9391 250@9397)
3:    ([f]abs@9386([r]platte@9387.x@9388) >@9384 250@9390)
4:    ([f]dspkommEmpfangeDoublePunkt@9365(...) ==@9375 0@9377)
5:    [f]pfadNaechsterZielpunkt@9423(...)
19:   (anzahl_neuronen_in_schicht@5298[0@5415] ==@5414 2@5418)
24:   ([f]schichtVon@3591(...) ==@3589 0@3597)
25:   ((*ping@3499 &@3637 128@3640) >@3636 0@3641)   (Sicherheitsverletzung)
26:   (i@4530 <@4529 anzahl_neuronen@4484)
27:   ([f]schichtVon@4167(...) ==@4175 0@4177)
30:   neuron@4109 := i@4535;... (berechneAusgangVon, neuronal.c)
31:   cpu_port@4483 := cpu_port@5468;... (berechneTDNNAusgang, neuronal.c)
32:   cpu_port@4483 := cpu_port@5609;... (berechneTDNNAusgang, neuronal.c)
33:   (anzahl_neuronen_in_schicht@5298[0@5694] ==@5693 3@5697)
34:   cpu_port@4483 := cpu_port@5770;... (berechneTDNNAusgang, neuronal.c)
35:   cpu_port@4483 := cpu_port@5934;... (berechneTDNNAusgang, neuronal.c)
36:   (anzahl_neuronen_in_schicht@5298[0@6019] ==@6018 8@6022)
37:   cpu_port@4483 := cpu_port@6218;... (berechneTDNNAusgang, neuronal.c)
38:   abstand_x@5283 := [r][deref][f]calloc@9195(1@9196, 8@9197).x@9498;...
        (reglerBerechneMotorschritte, regler.c)
41:   (#v_inhalt@5309 <@7309 v_laenge@6404)
42:   ([r]v_endpunkt@7285.y@7286 !=@7284 [r]aktueller_zielpunkt@7287.y@7288)
```

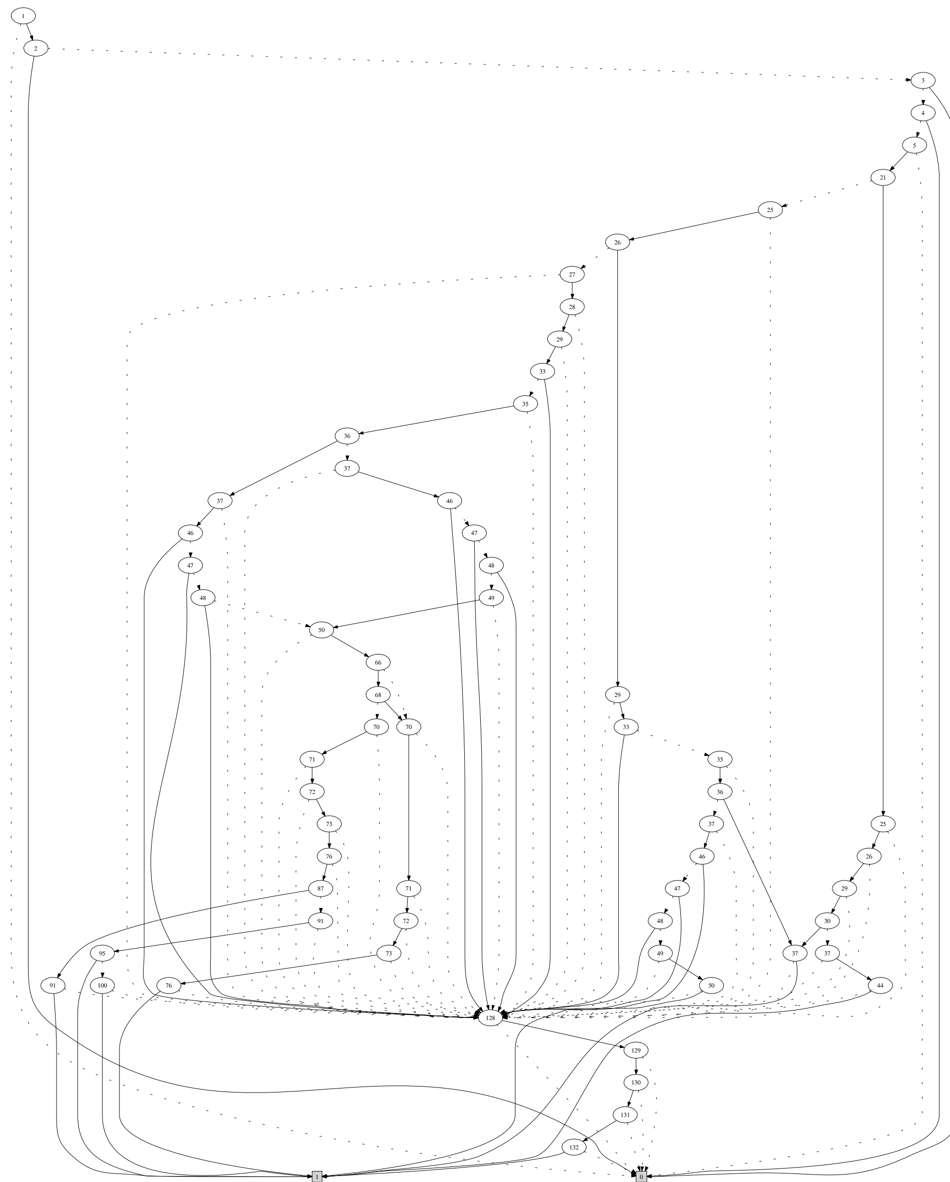


Abbildung A.1: Der BDD zur Fallstudie „Wackeltisch 1“ (siehe Abschnitt 8.1.1, Seite 136)

```
43:   ([r]v_endpunkt@7280.x@7281 !=@7279 [r]aktueller_zielpunkt@7282.x@7283)
44:   cpu_port@4483 := cpu_port@7575;... (berechneTDNNAusgang, neuronal.c)
45:   ping@7179 := ping@6387;...
        (trainiereFuzzyUndPlatteErweitertNetz2in1, neuronal.c)
82:   wert@3496 := geschwindigkeit_x@5284;... (skaliere, neuronal.c)
83:   wert@3496 := geschwindigkeit_y@5287;... (skaliere, neuronal.c)
84 :  wert@3496 := ausgang@5619[[f]erstesNeuronIn@5373(...)][0@5627];...
        (skaliere, neuronal.c)
```

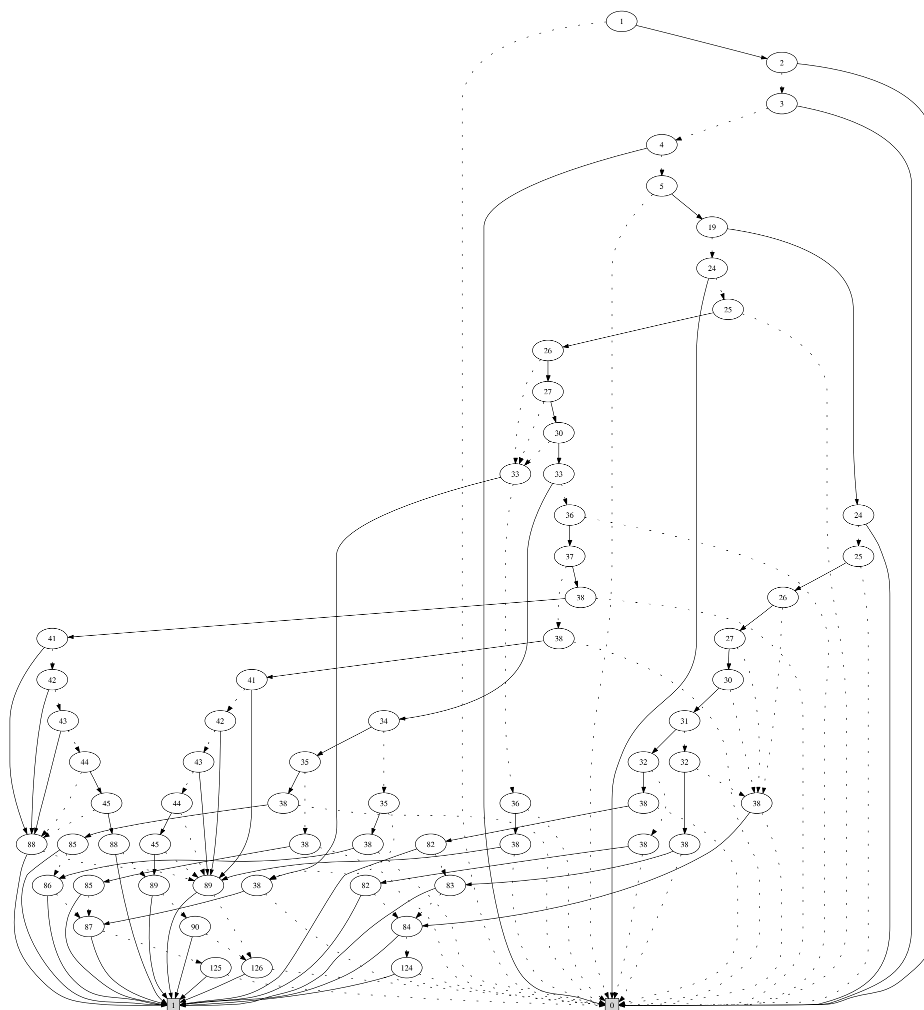


Abbildung A.2: Der BDD zur Fallstudie „Wackeltisch 2“ (siehe Abschnitt 8.1.3, Seite 139)

```
85:   wert@3496 := plattenstellung_x@5285;... (skaliere, neuronal.c)
86:   wert@3496 := plattenstellung_y@5288;... (skaliere, neuronal.c)
87:   wert@3496 := ausgang@5944[[f]erstesNeuronIn@5373(...)][0@5952];...
        (skaliere, neuronal.c)
88:   wert@3496 := ([r]lotfusspunkt@6193.y@6194
        -@6192 [r]aktueller_punkt@.y@6196);... (skaliere, nnal.c)
89:   wert@3496 := ausgang@6228[[f]erstesNeuronIn@5373(...][0@6236];...
        (skaliere, neuronal.c)
90:   wert@3496 := vergangenheit@6382[((v_position@5312 +@6297 1@6299)
        %@6296 v_laenge@5308)][...][...][...];... (skaliere, neuronal.c)
124:  wert@3496 := ausgang@5478[...][0@5486];... (skaliere, neuronal.c)

125:  wert@3496 := ausgang@5780[...][0@5788];... (skaliere, neuronal.c)
126:  wert@3496 := ausgang@6228[([f]erstesNeuronIn@5373(...)
        +@6267 1@6269)][0@6266];...  (skaliere, neuronal.c)
```

Die Ursache der Sicherheitsverletzung ist als Bedingung über *ping* in Knoten 25 zu sehen. Ein Blick auf den zugehörenden BDD zeigt Knoten 25 an zwei Stellen (rechter Rand und 8. Knoten von oben). Die Bedingung existiert auf *jedem* Pfad zum finalen BDD-Knoten 1 (*true*) und ist somit erwartungsgemäß Bestandteil aller Monome der Pfadbedingung.

A.2 BDDs zur Fallstudie „PalME“

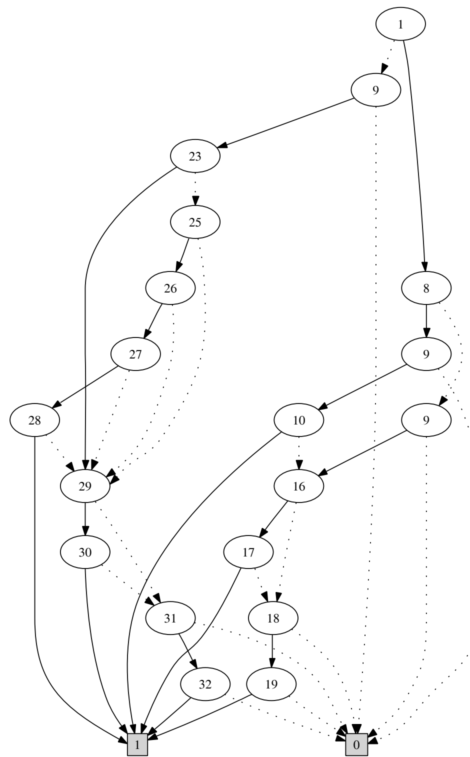


Abbildung A.3: Der BDD zur Fallstudie „PalME 1“ (siehe Abbildung 8.6, Seite 147)

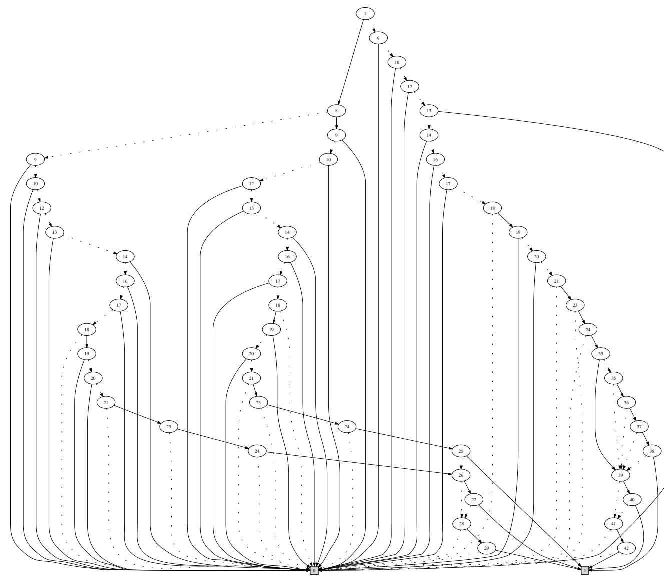


Abbildung A.4: Der BDD zur Fallstudie „PalME 2“ (siehe Abbildung 8.7, Seite 147)

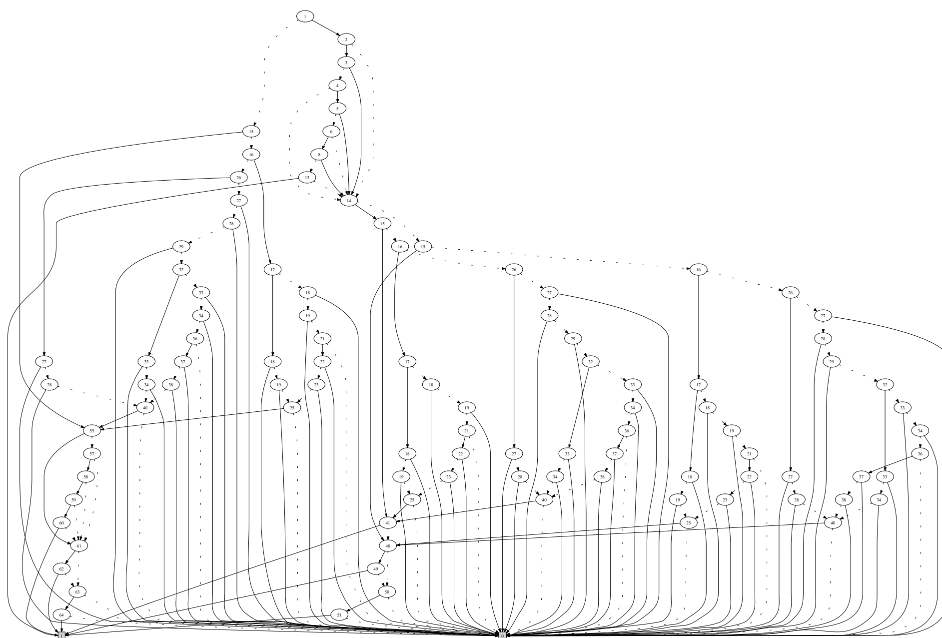


Abbildung A.5: Der BDD zur Fallstudie „PalME 3“ (siehe Abbildung 8.8, Seite 148)

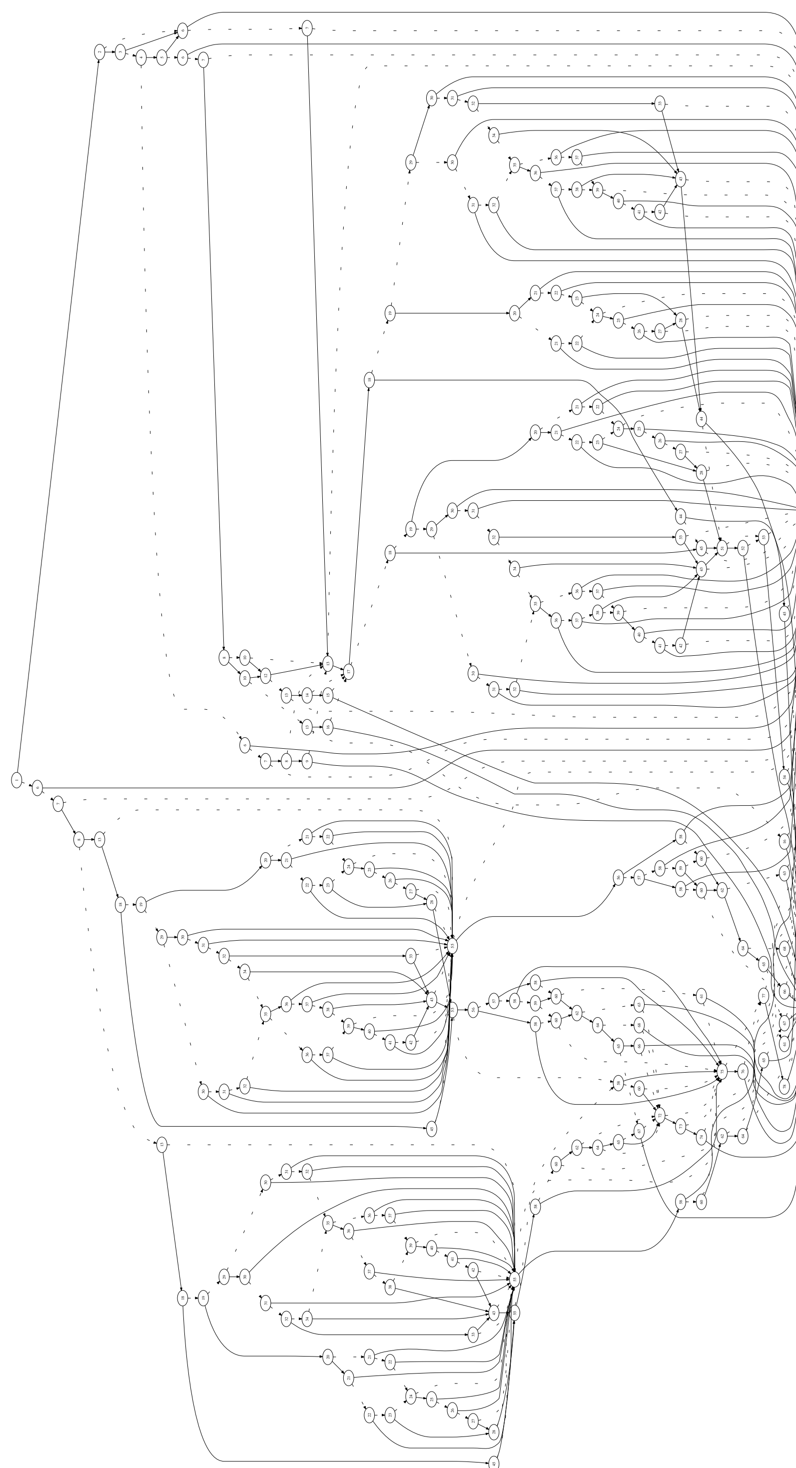


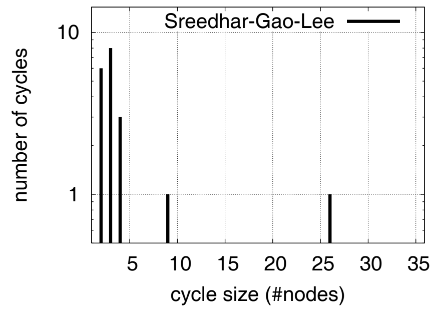
Abbildung A.6: Der BDD zur Fallstudie „PalME 4“ (siehe Abbildung 8.9, Seite 149)

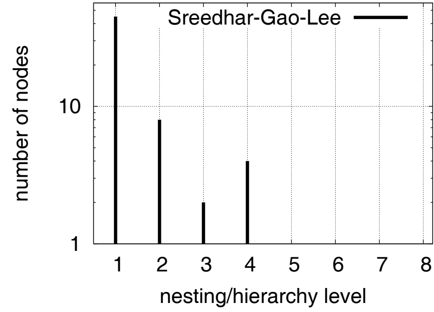
Anhang B

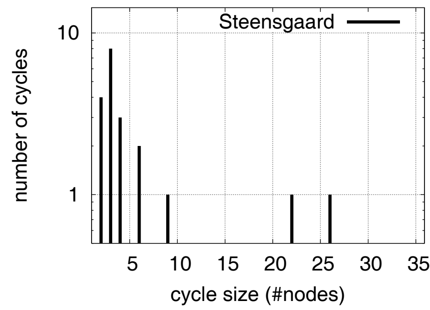
Zyklengröße und Zerlegungstiefe aller Fallstudien

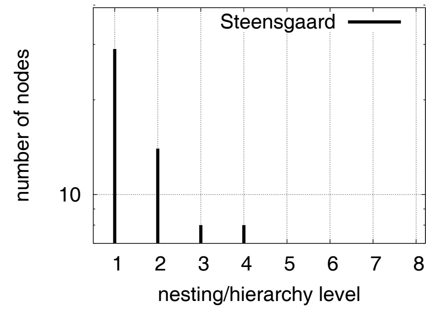
Die Grafiken auf den folgenden Seiten zeigen für alle Fallstudien Daten über die Zyklengröße und Zerlegungstiefe je Zerlegungsverfahren. Neben der schon bekannten Anzahl von reduziblen und irreduziblen geschachtelten Zyklen und ihrer maximalen Knoten- und Kantenzahl (siehe Abschnitt 7.5.10 auf Seite 109) zeigen die Grafiken detaillierter die Charakteristika der hierarchischen Zerlegung. Die linke Seite zeigt, wieviele Zyklen (number of cycles) je angegebener Zyklengröße (cycle size) existieren. Die vertikale Skala ist dabei logarithmisch. Die rechte Seite zeigt, wie stark die hierarchische Zerlegung ist. Für jede Hierarchie-Ebene (nesting level) wird die Gesamtzahl der Knoten (number of nodes) dargestellt. Auch hier ist die vertikale Skala logarithmisch.

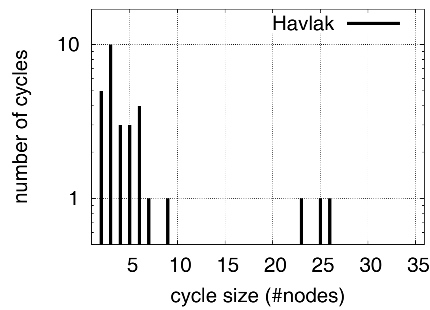
„mergesort 1“

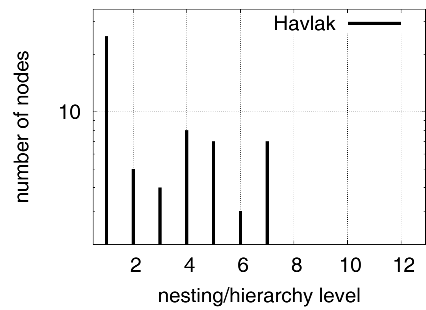


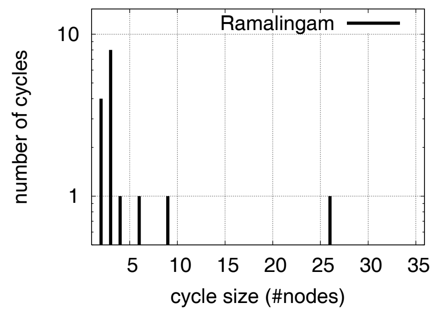


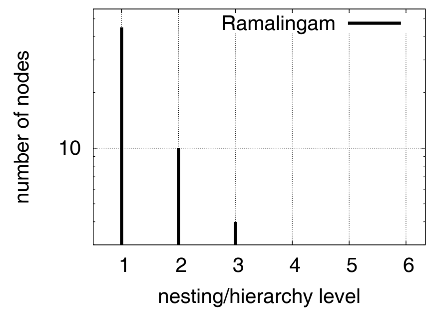




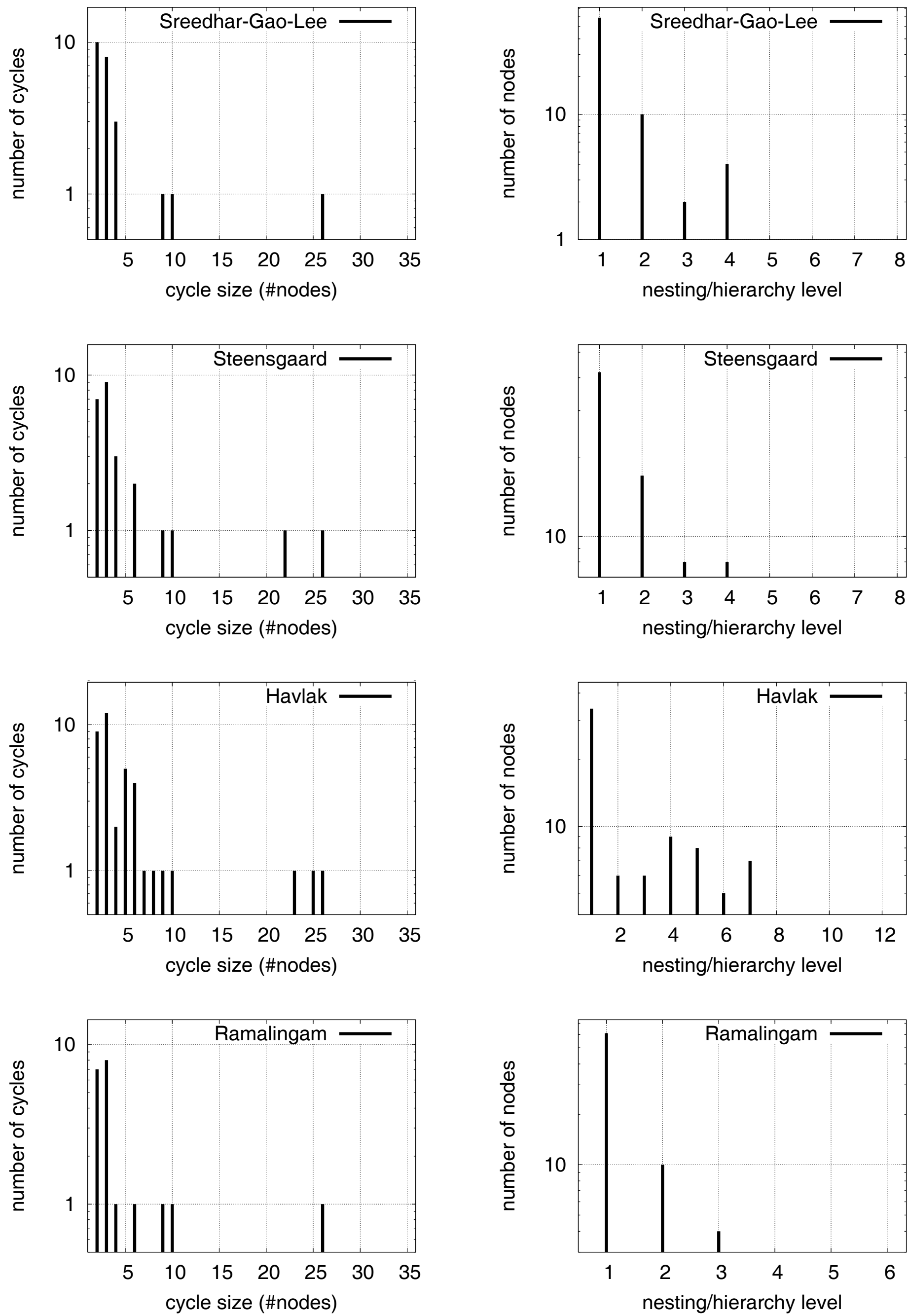




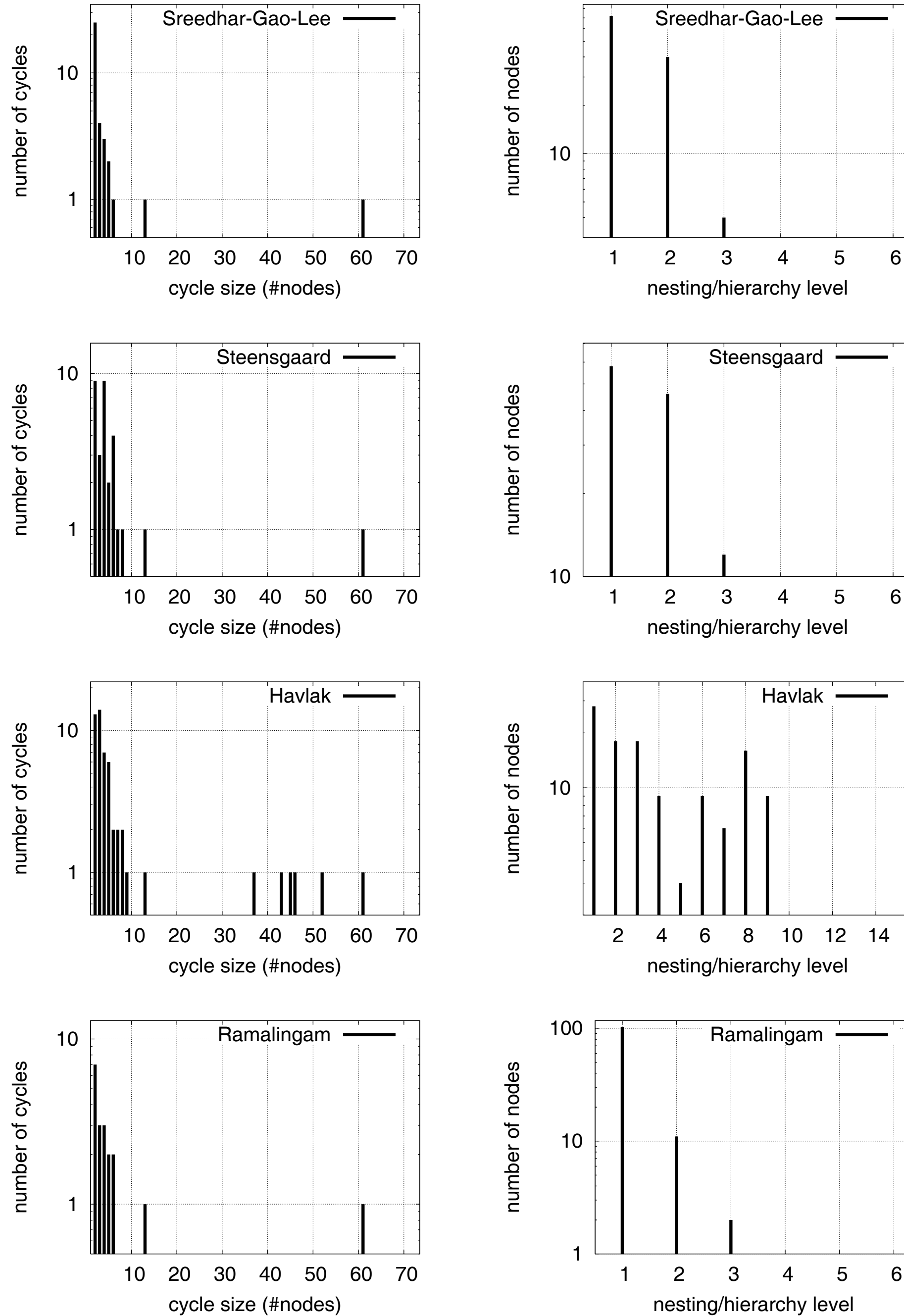




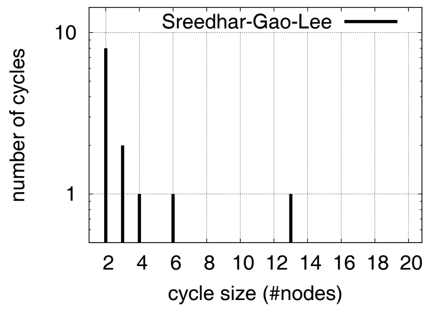
„mergesort 2“

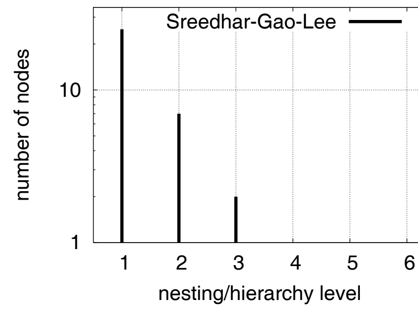


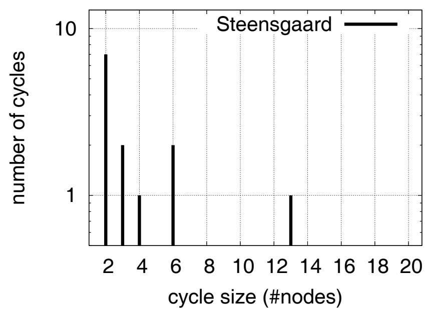
„calculator 1“

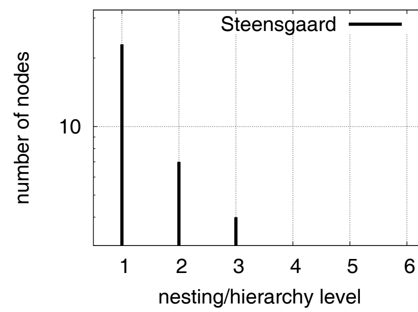


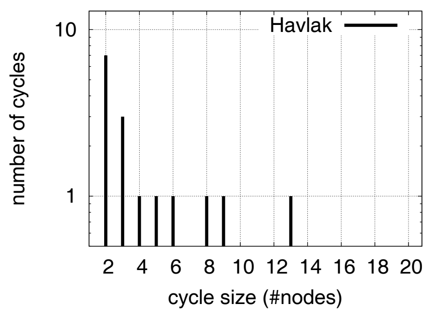
„calculator 2“

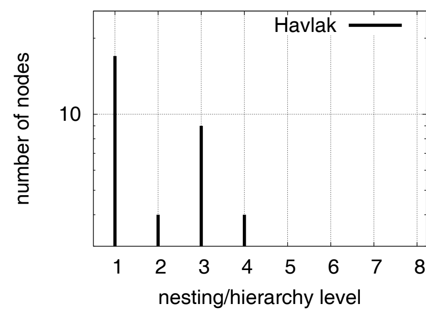


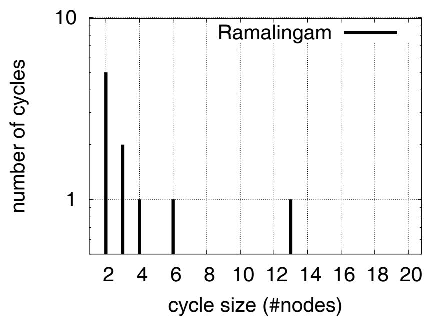


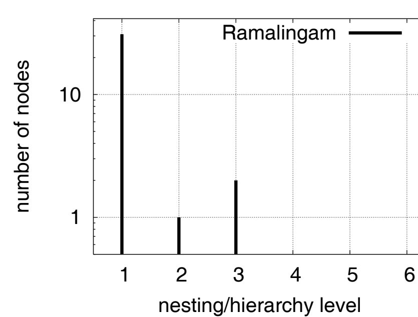




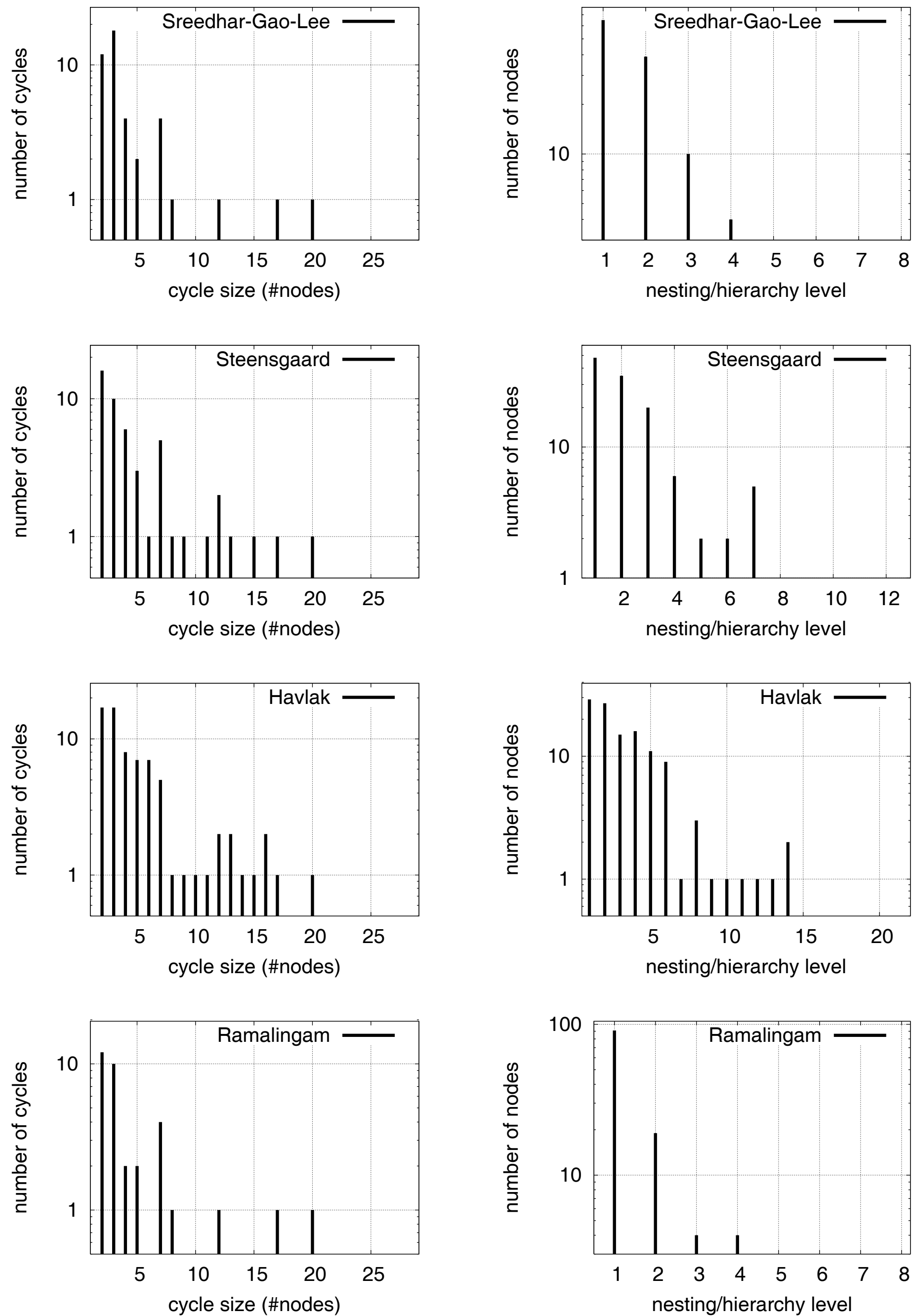




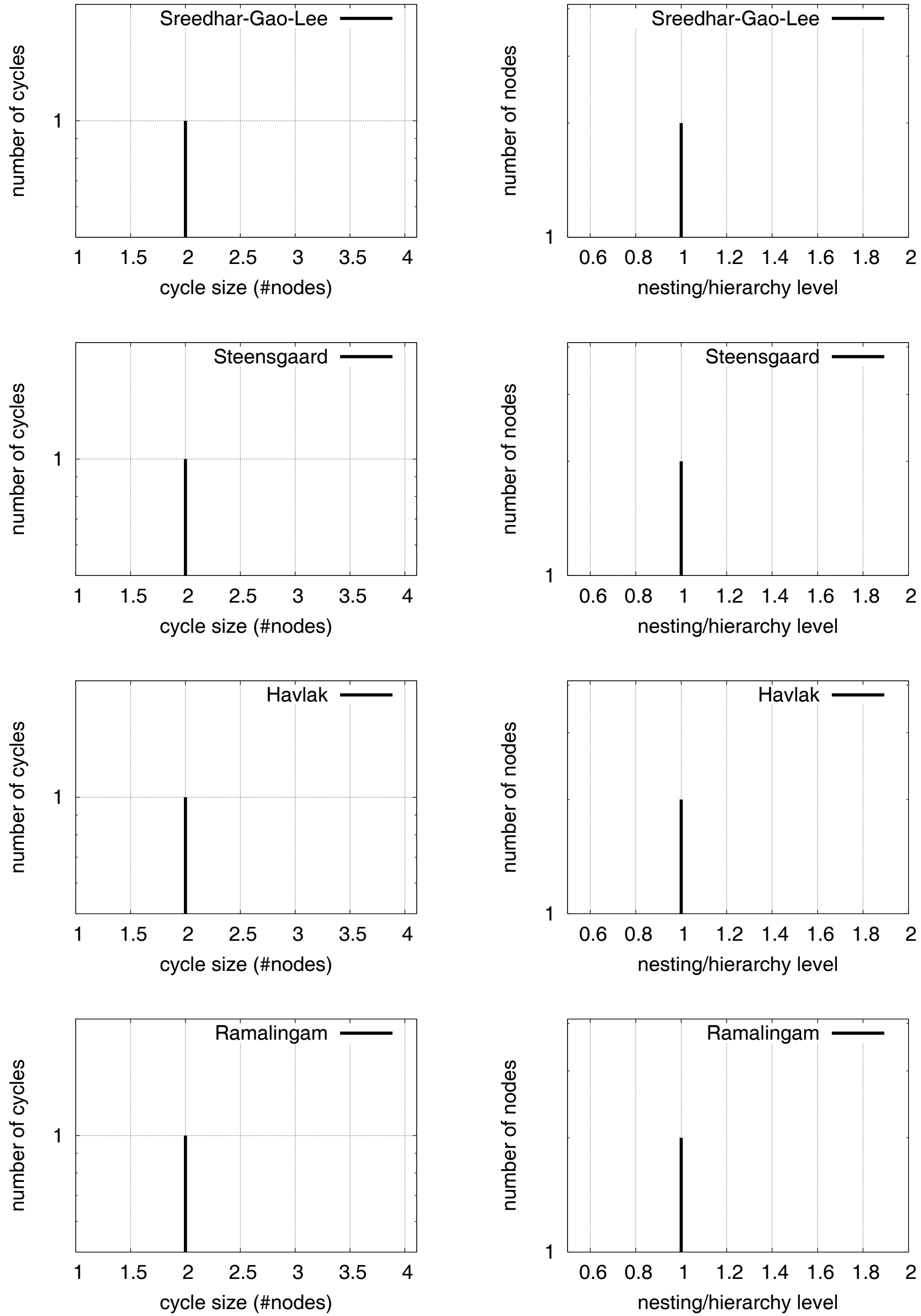




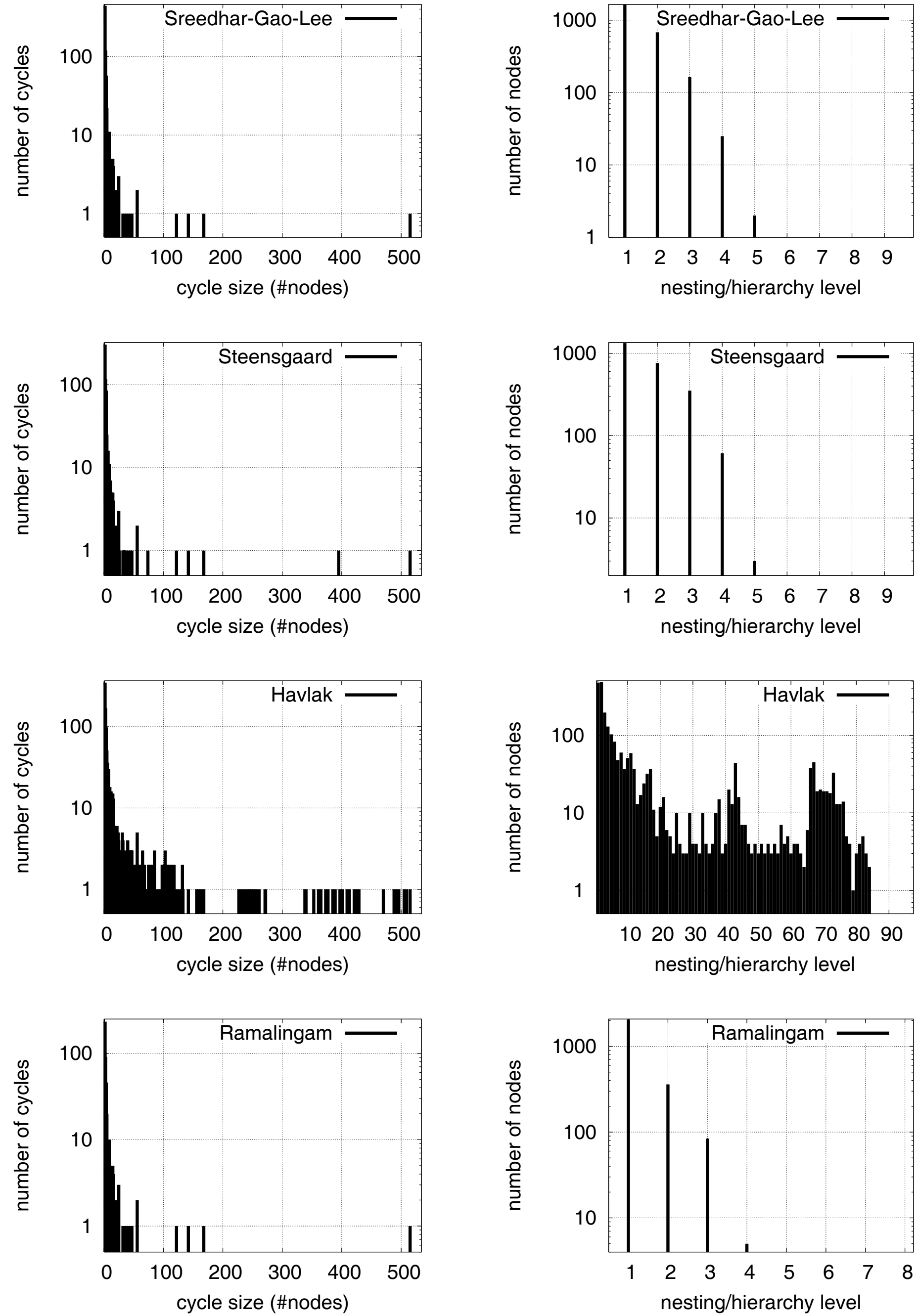
„triple des 1“



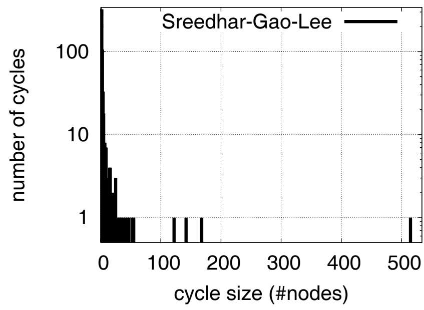
„triple des 2“

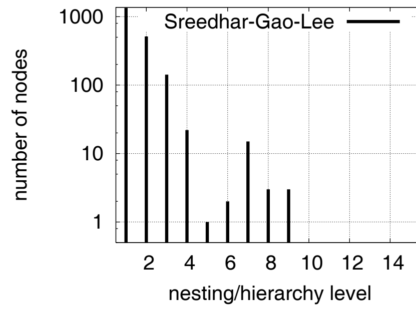


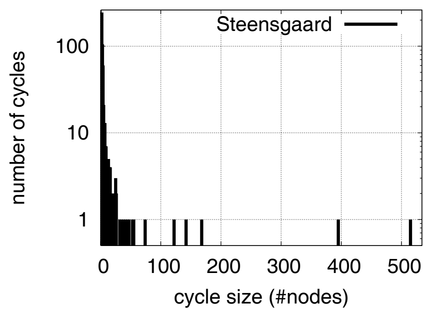
„ctags 1“

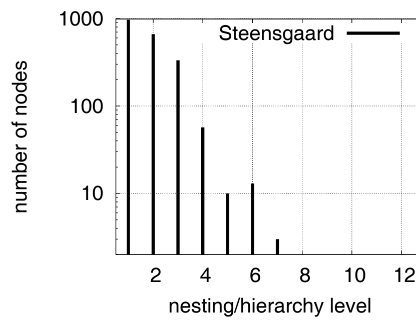


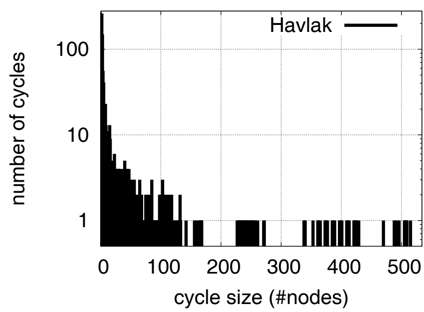
„ctags 2“

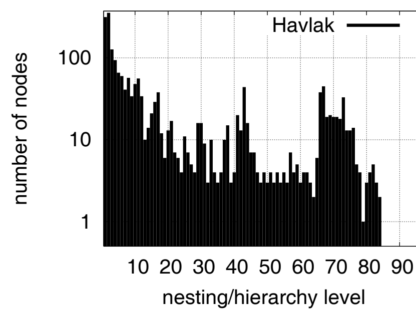


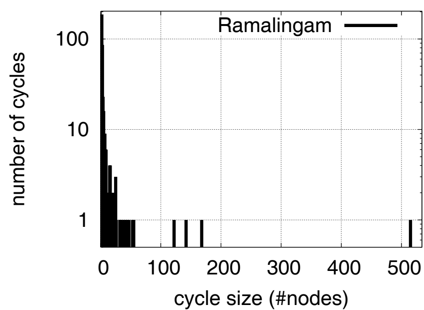


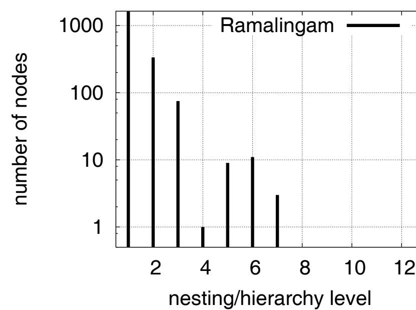




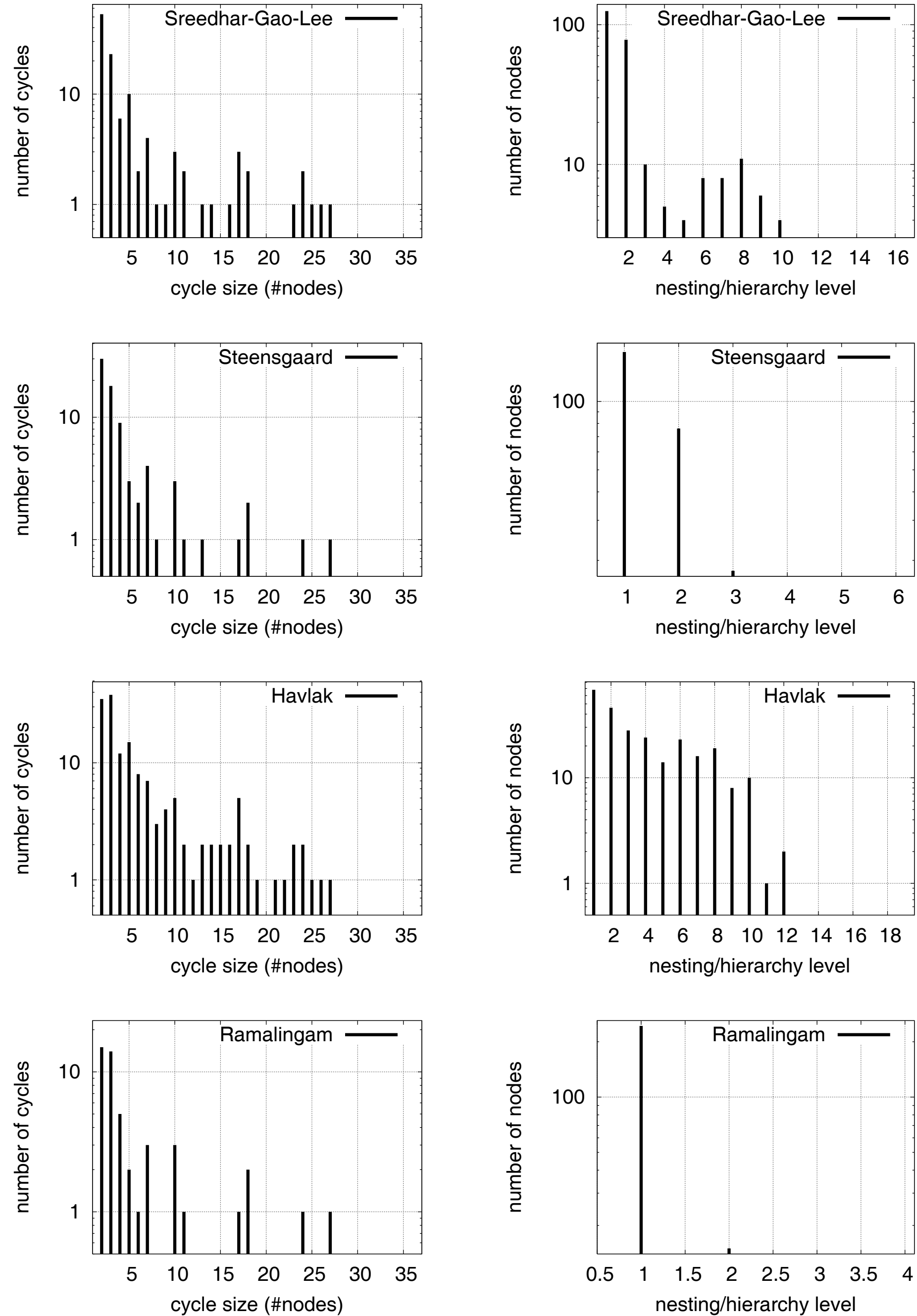




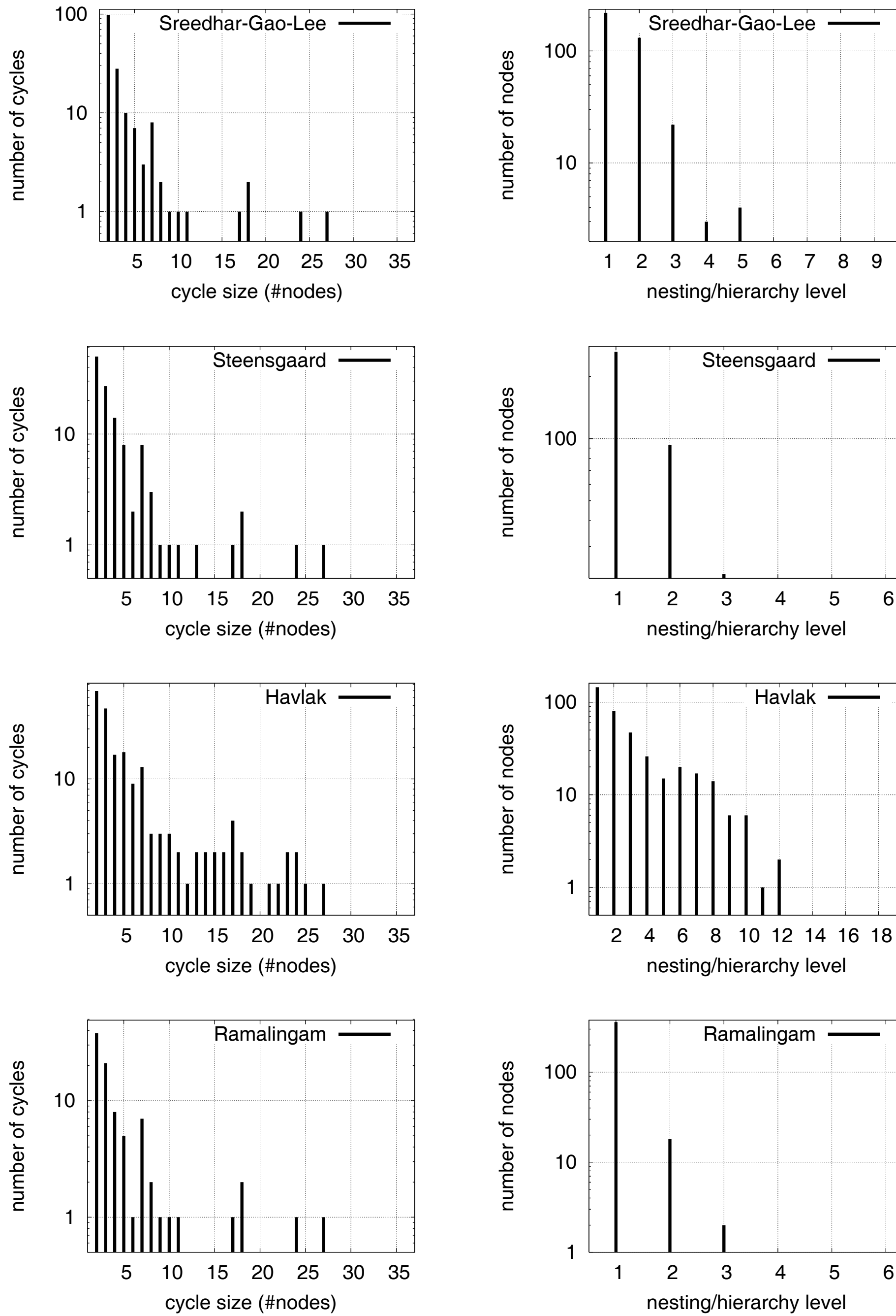




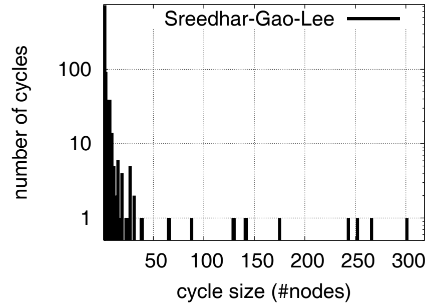
„assembler 1“

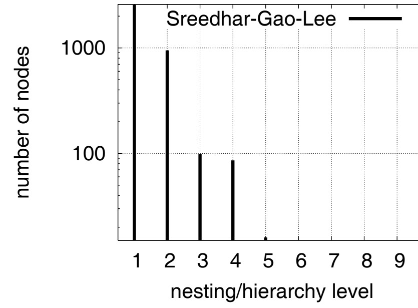


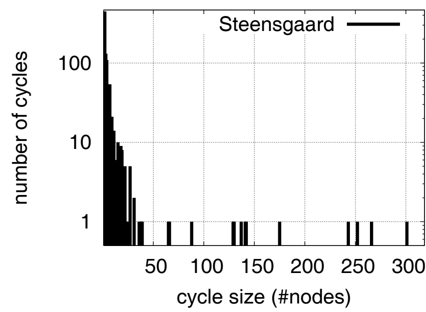
„assembler 2“

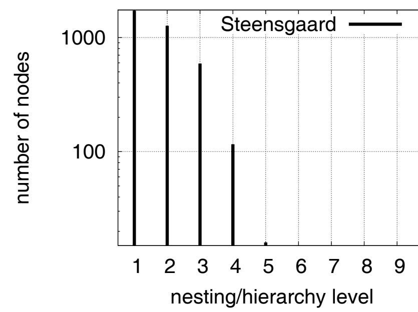


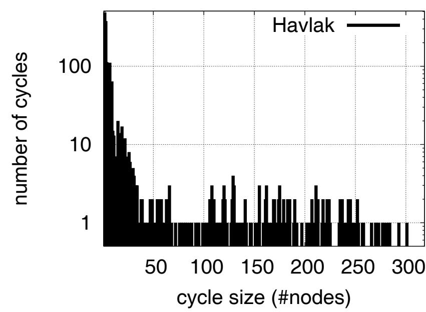
„gnugo 1“

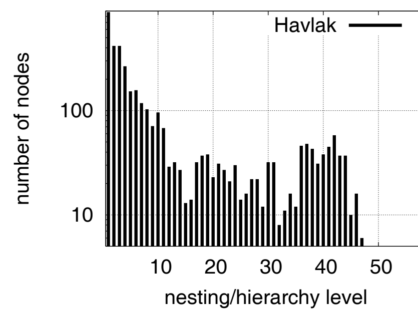


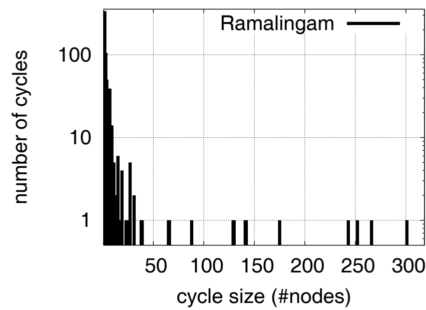


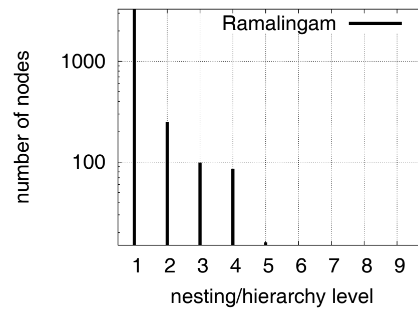




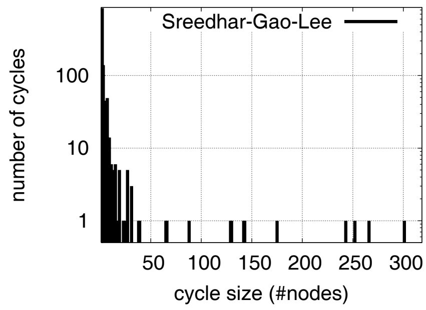


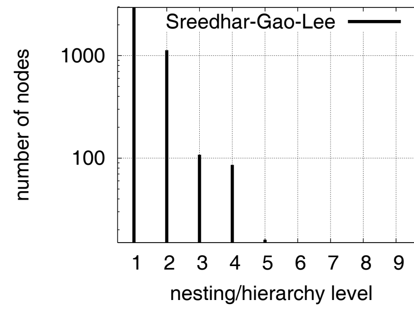


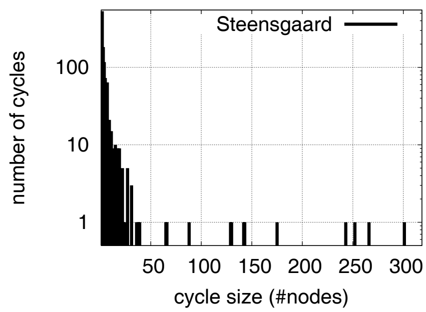


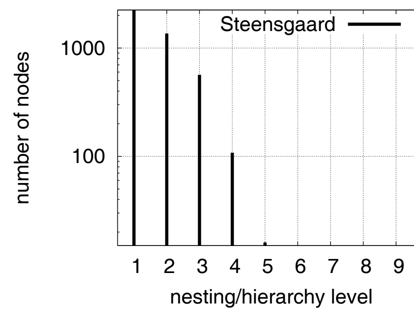


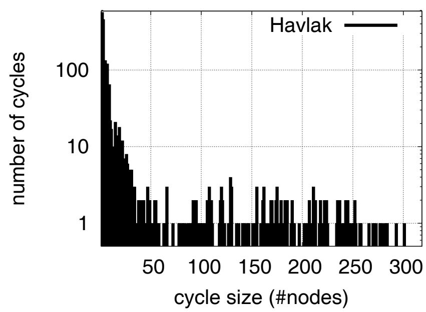
„gnugo 2“

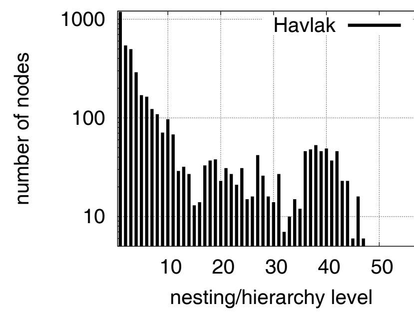


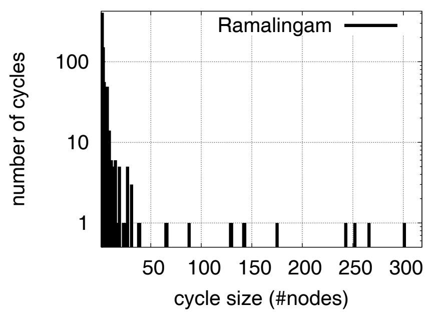


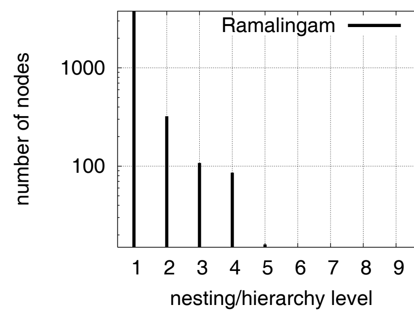




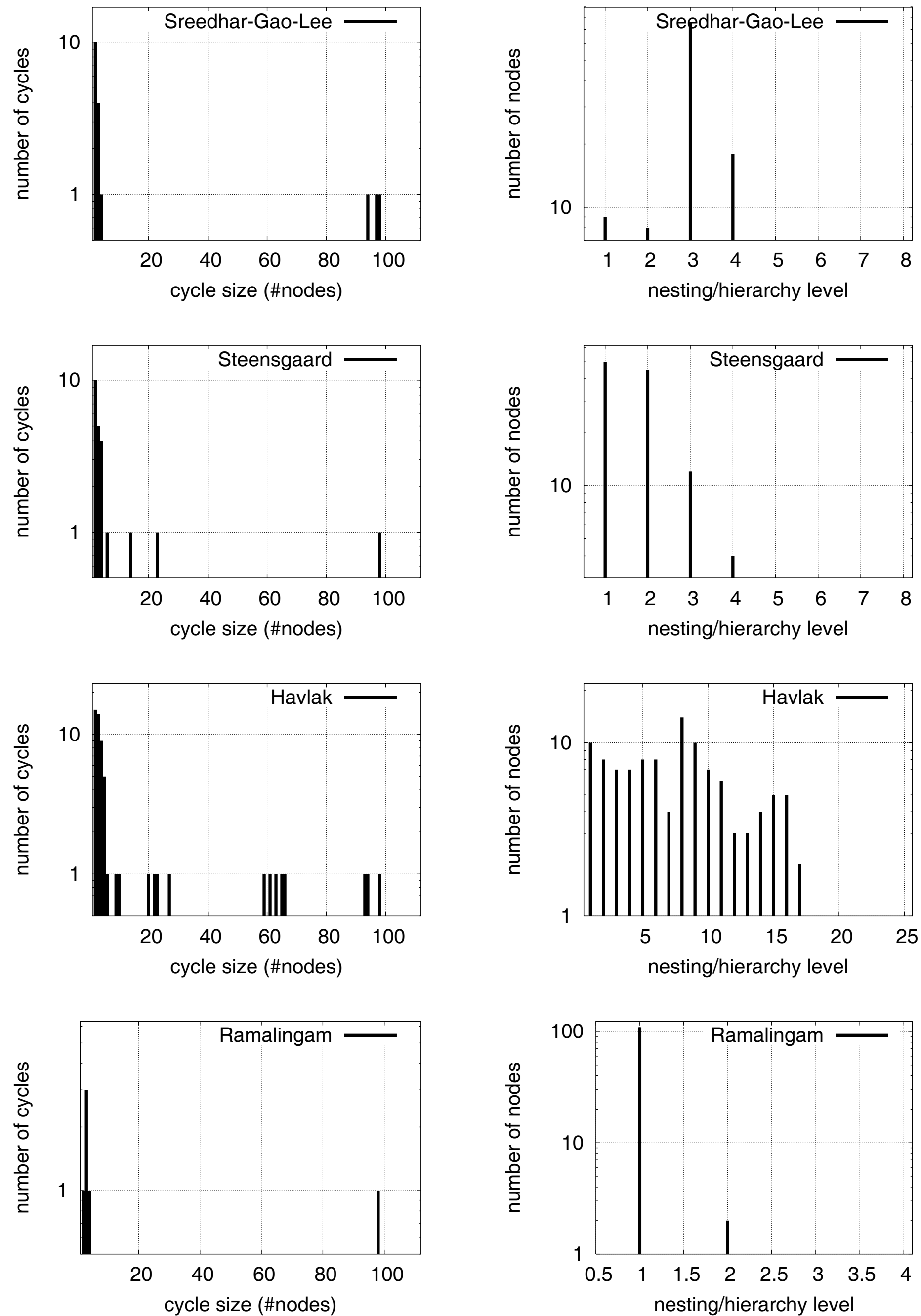




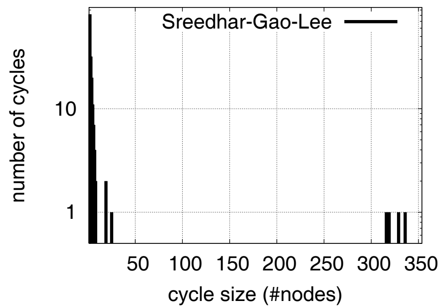


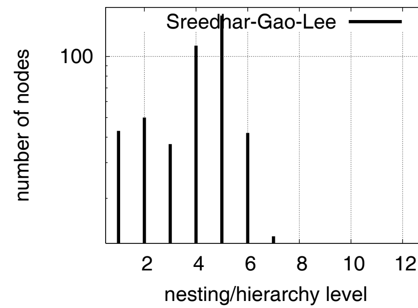


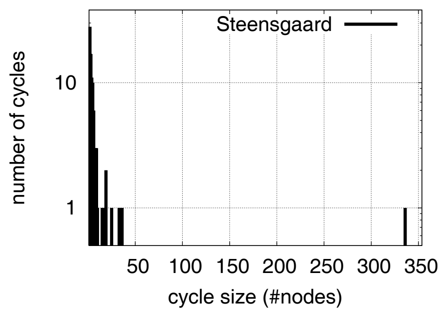
„agrep 1“

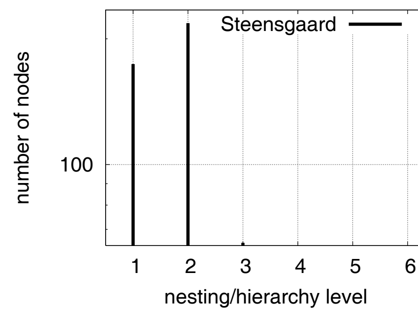


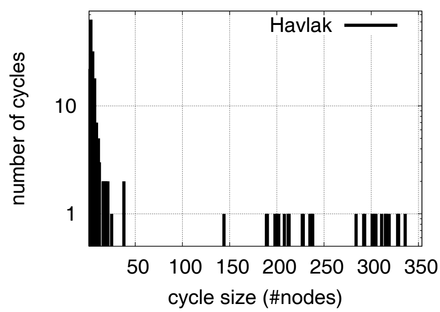
„agrep 2“

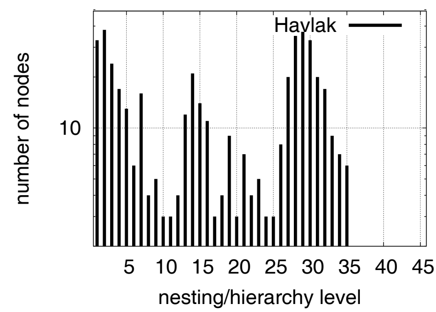


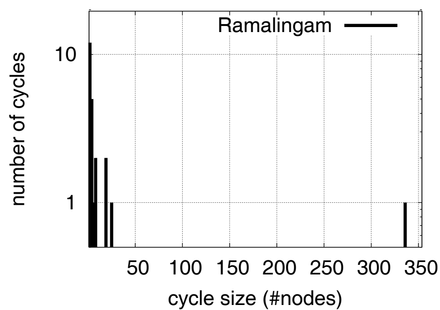


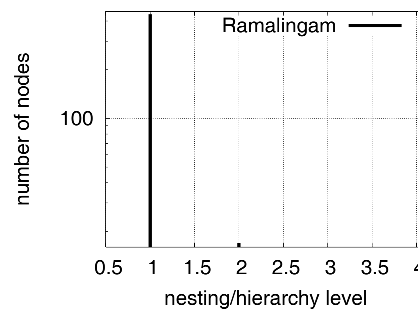




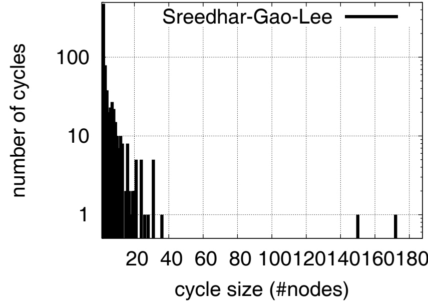


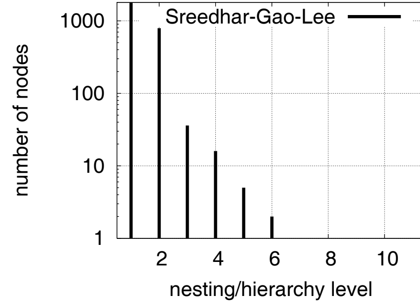


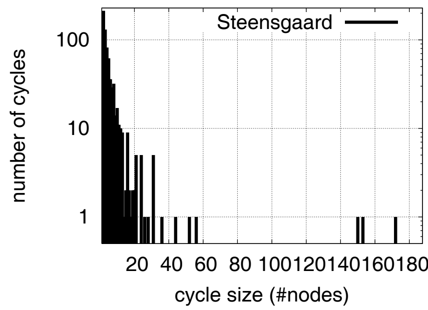


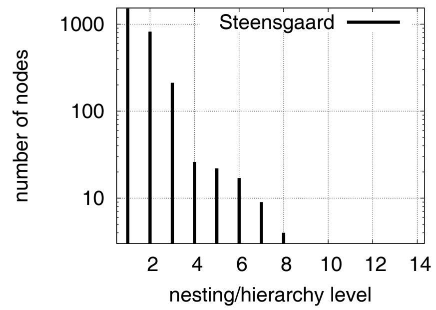


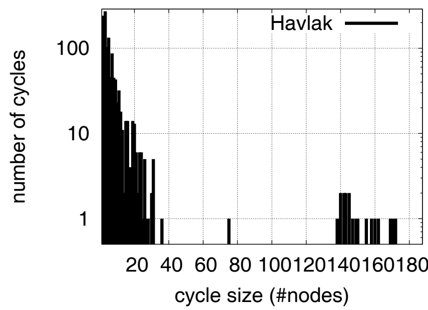
„wackeltisch 1“

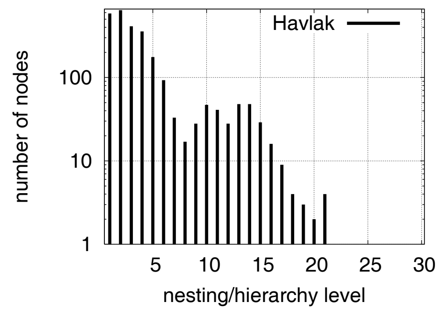


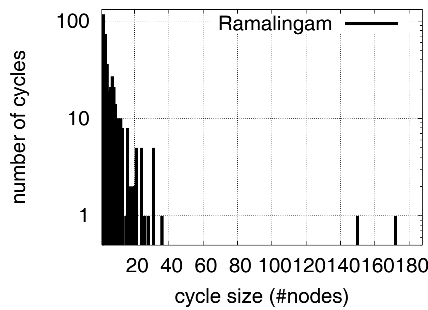


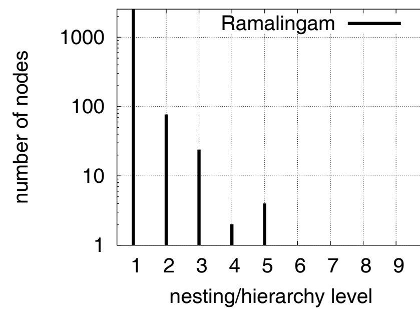




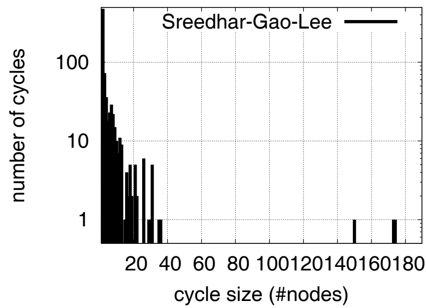


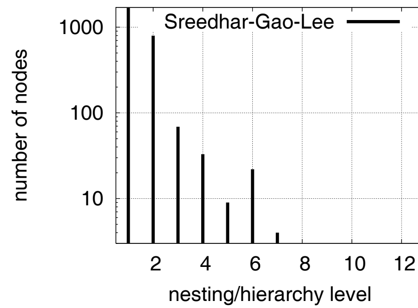


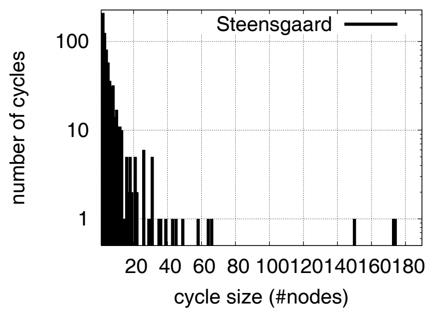


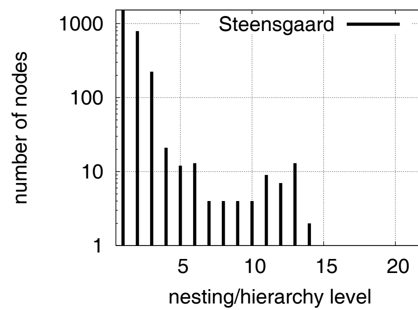


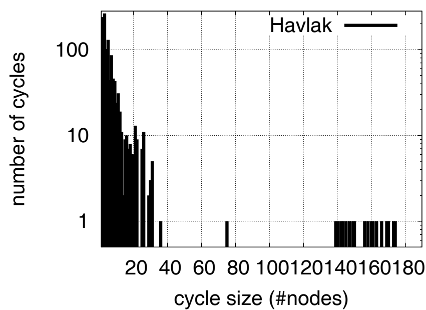
„wackeltisch 2“

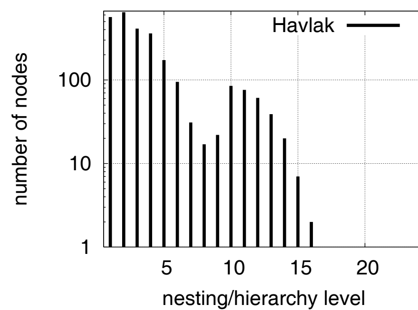


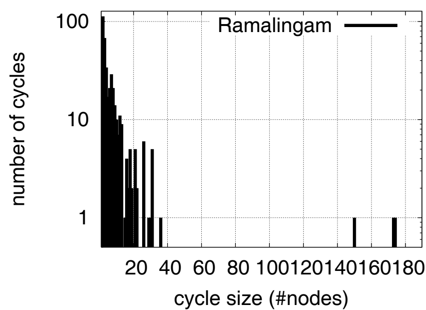


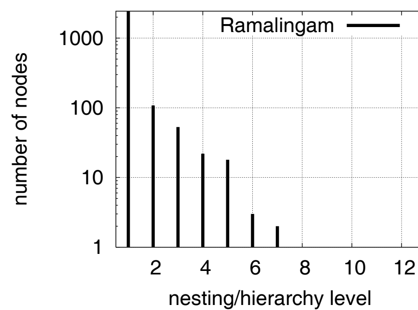




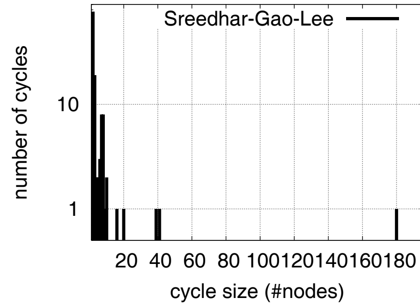


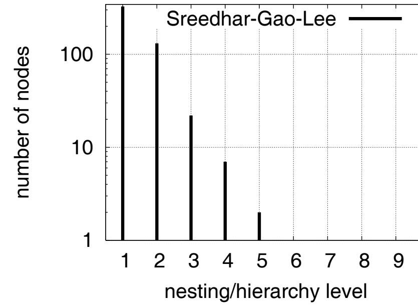


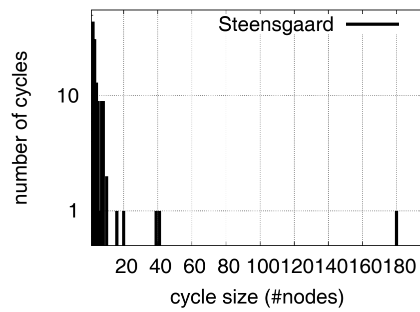


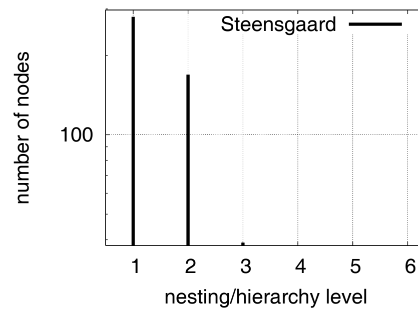


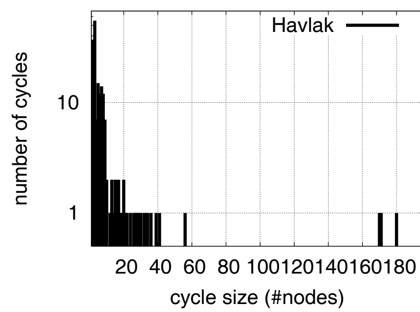
„flex 1“

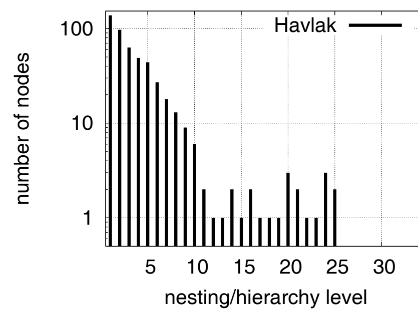


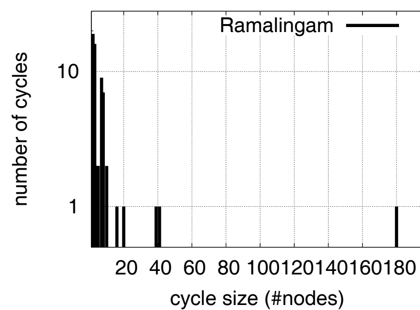


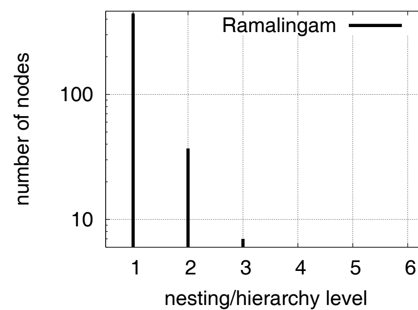




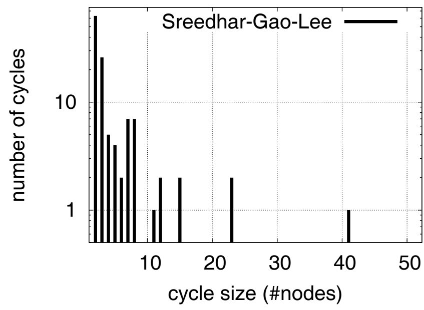


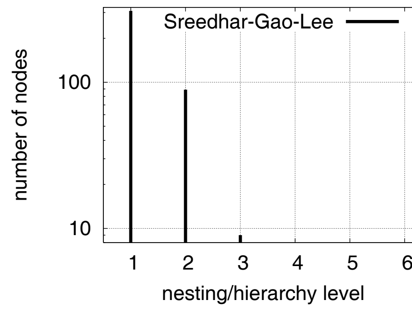


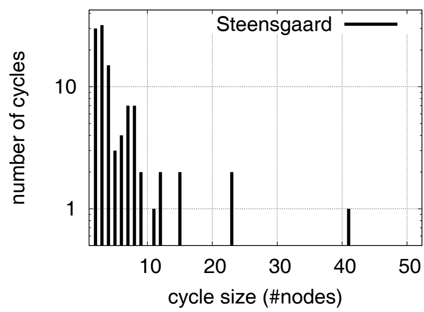


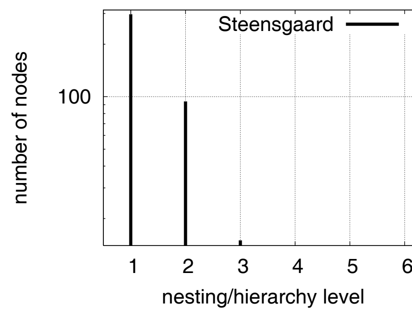


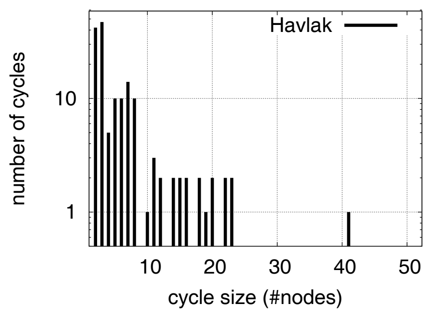
„flex 2“

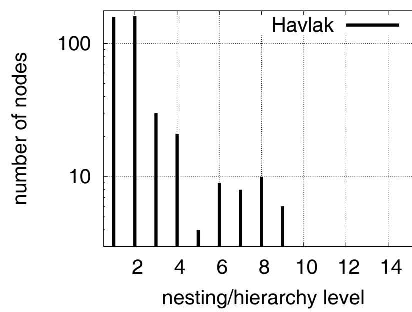


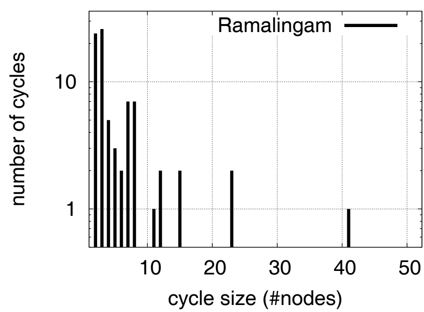


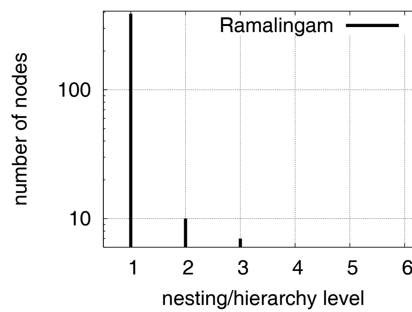




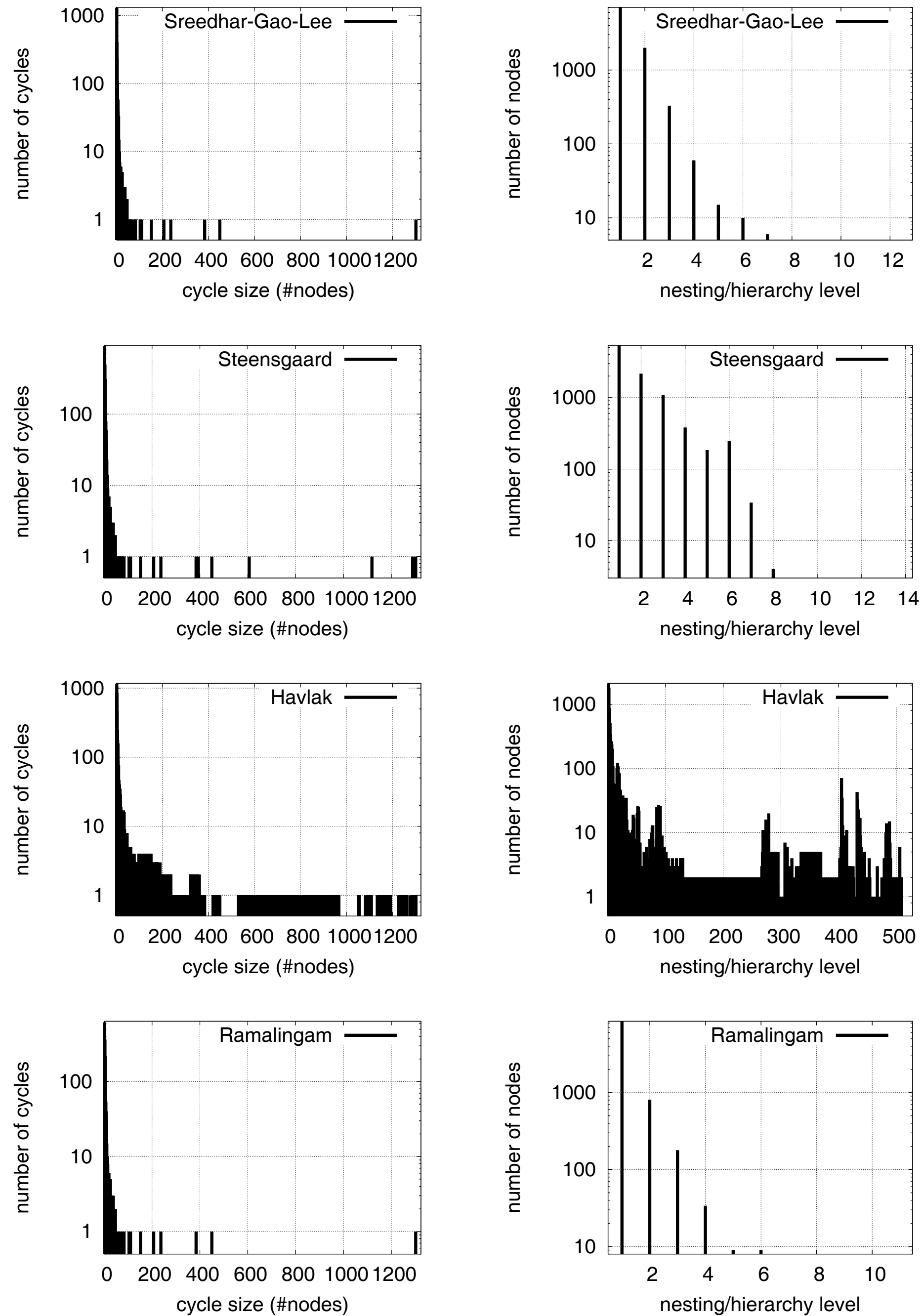




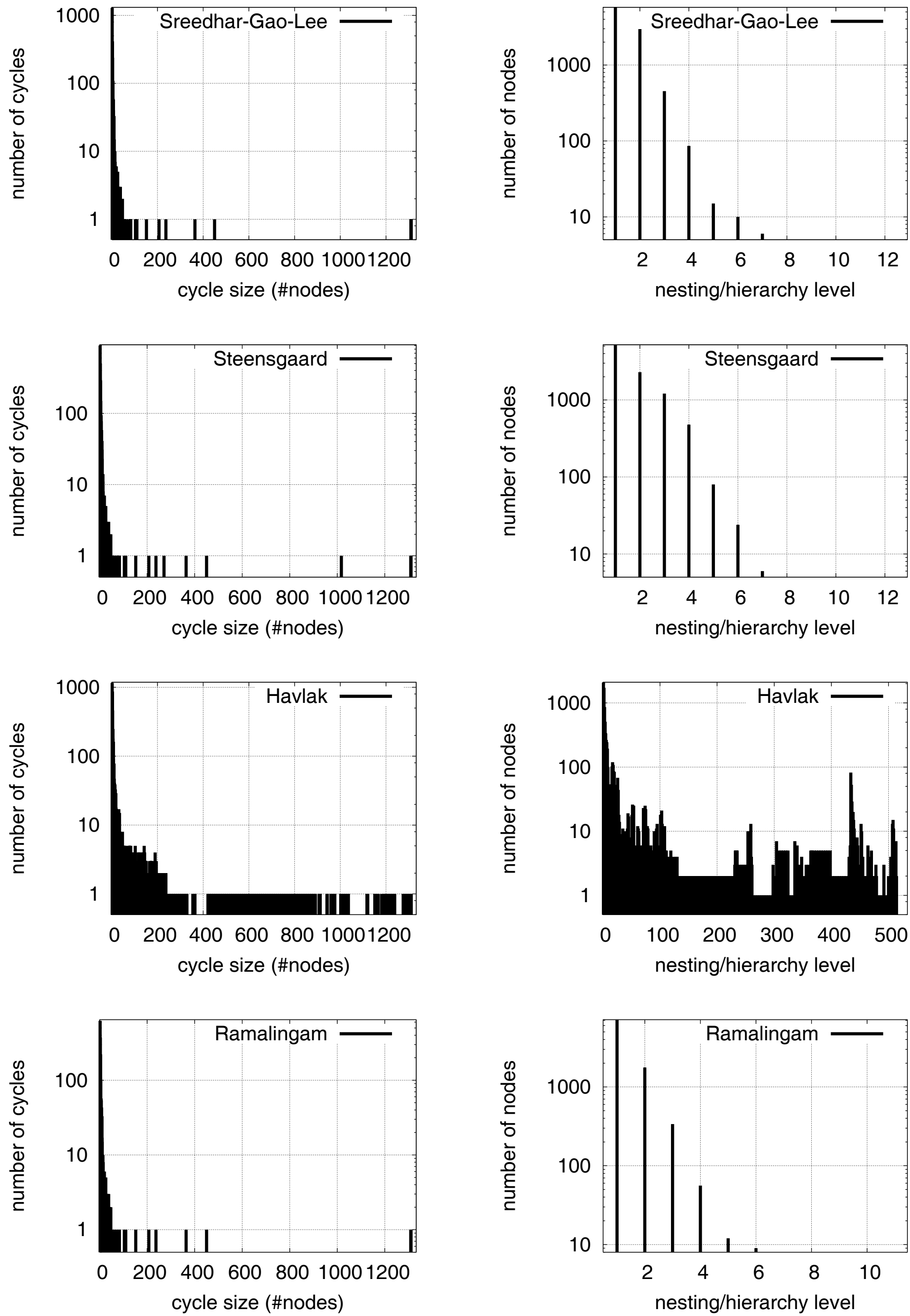




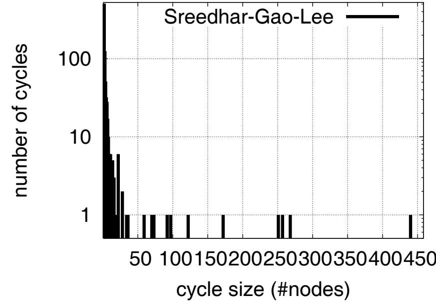
„patch 1“

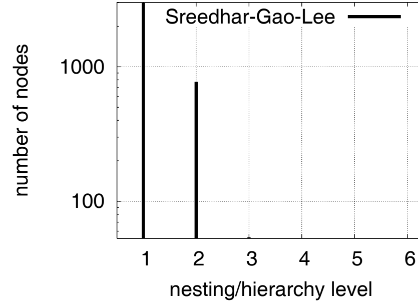


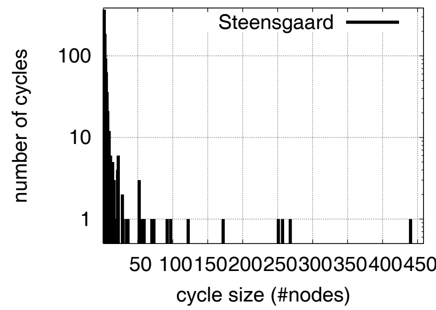
„patch 2“

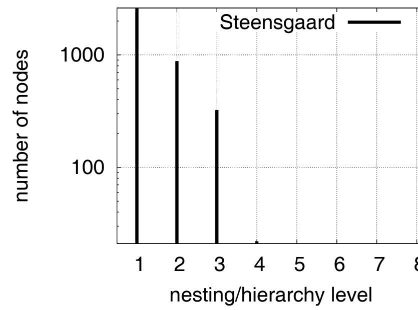


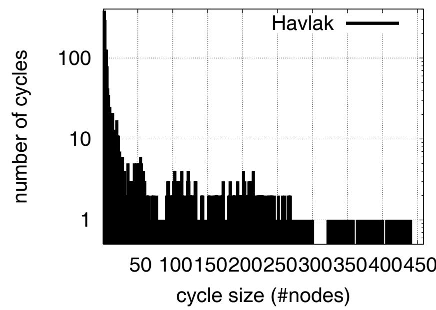
„bison 1"

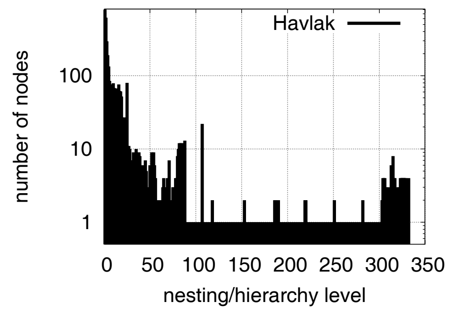


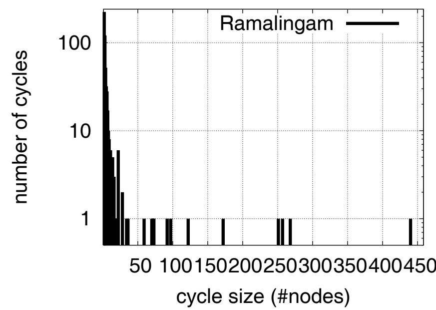


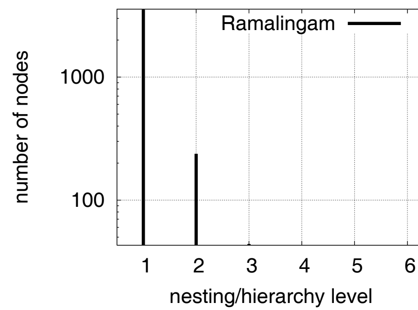




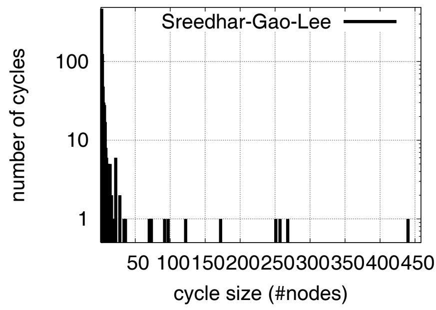


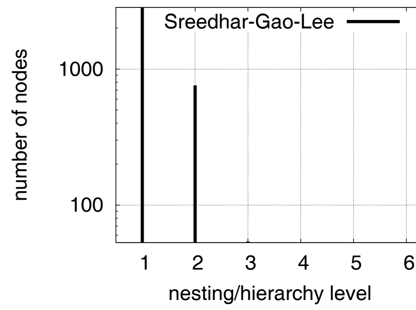


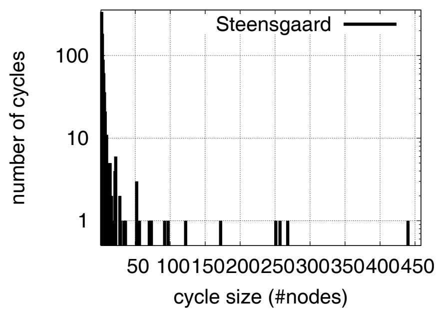


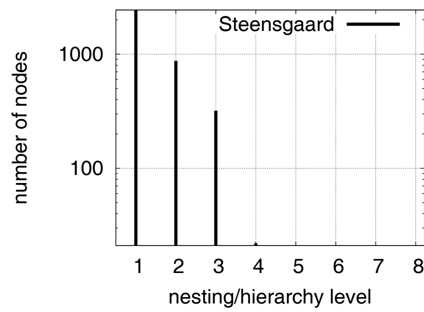


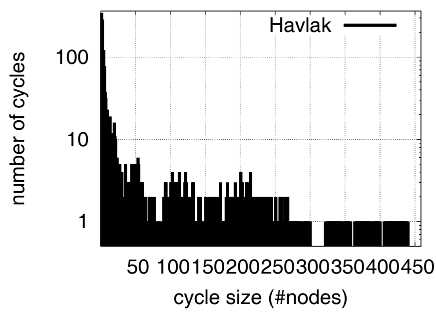
„bison 2“

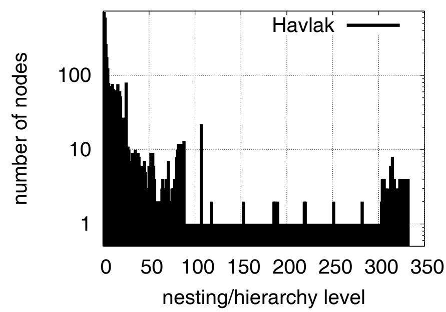


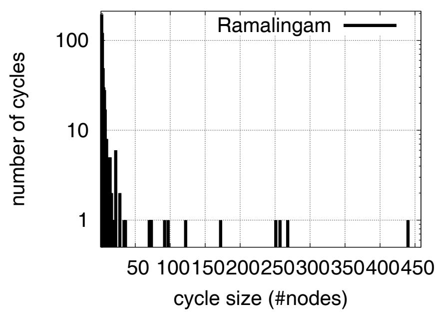


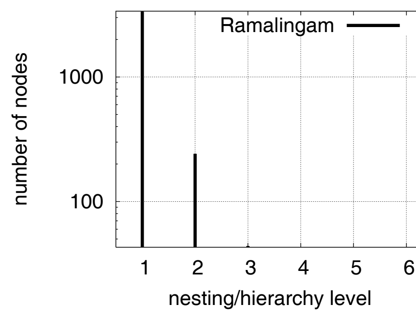




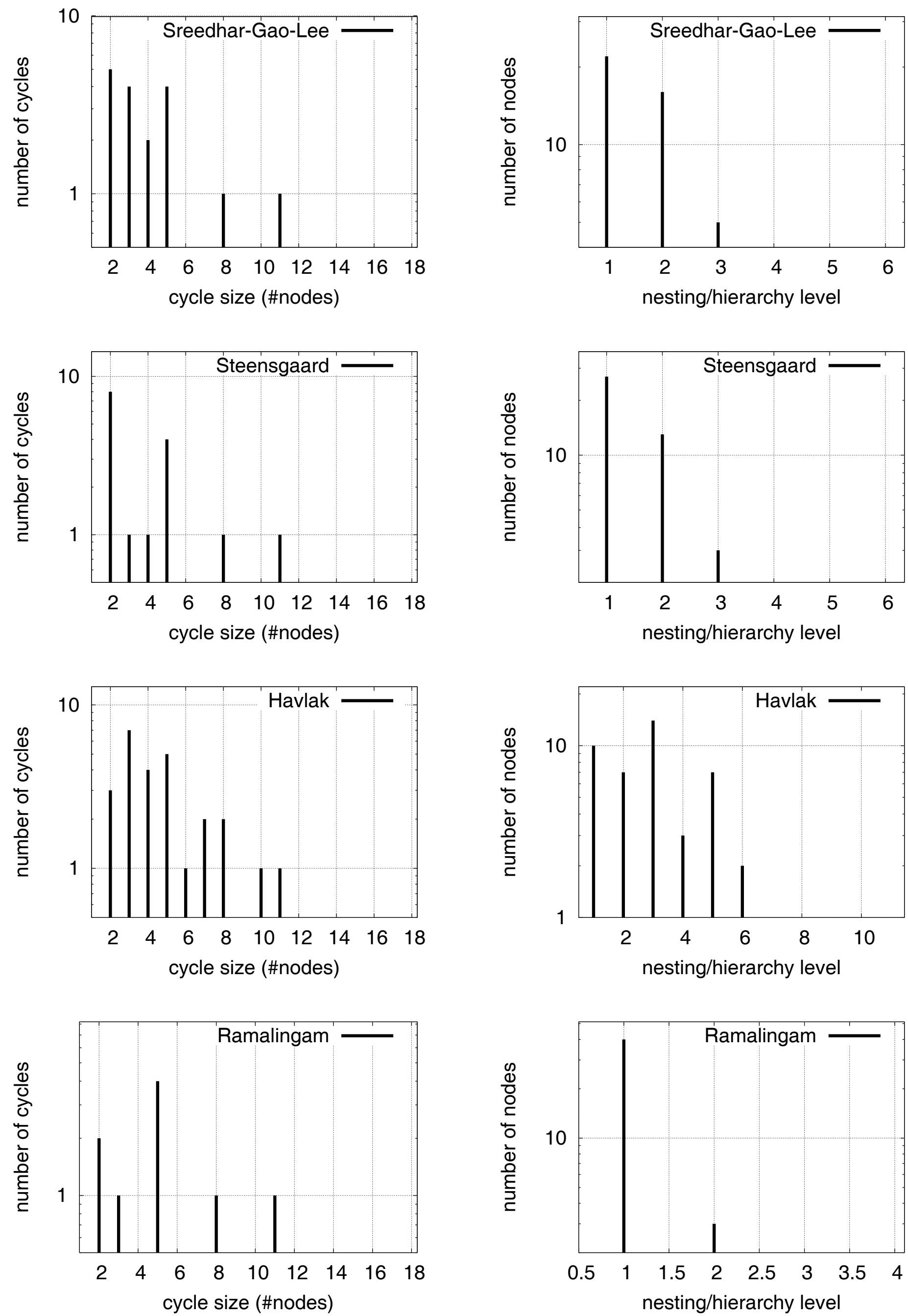




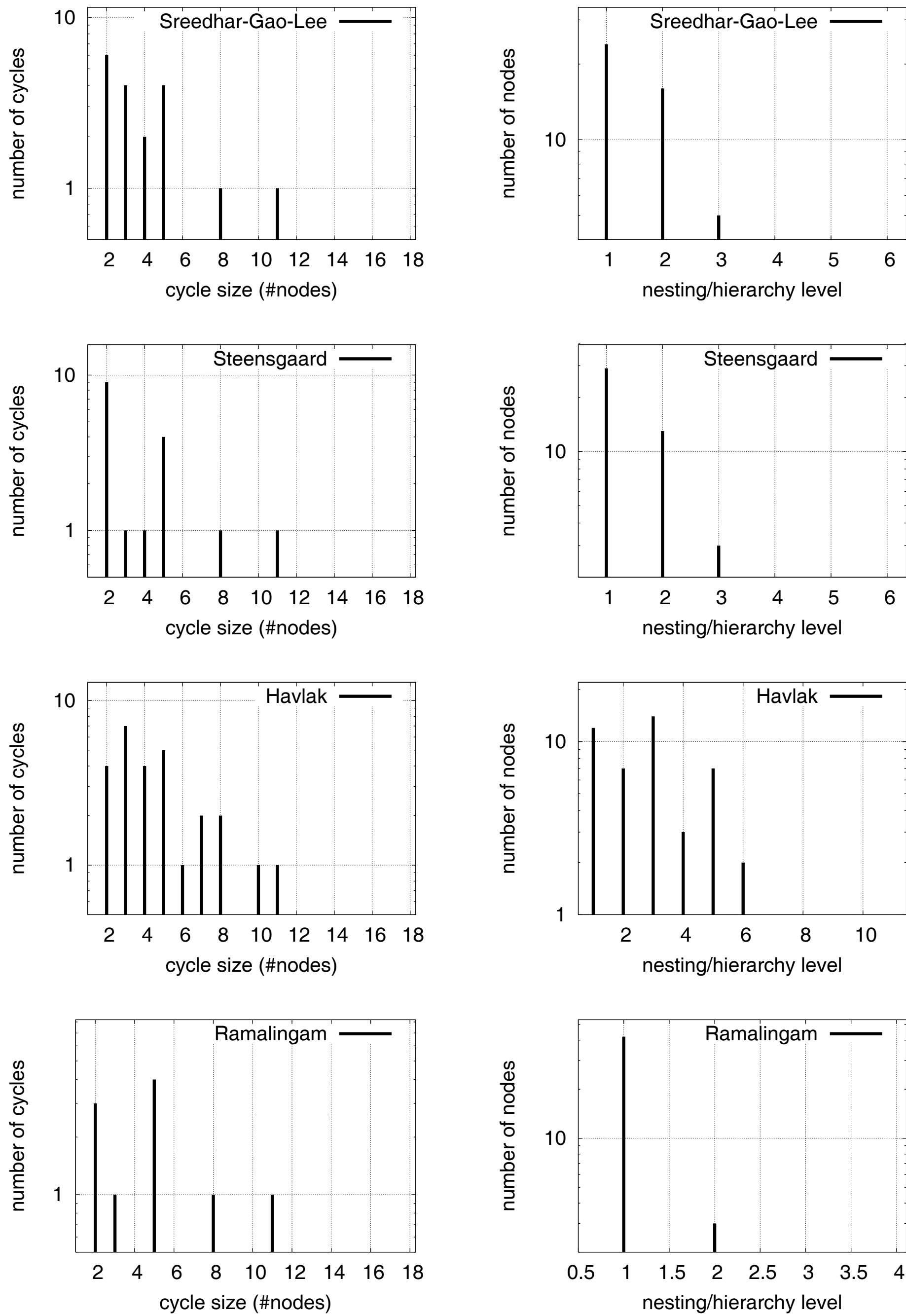




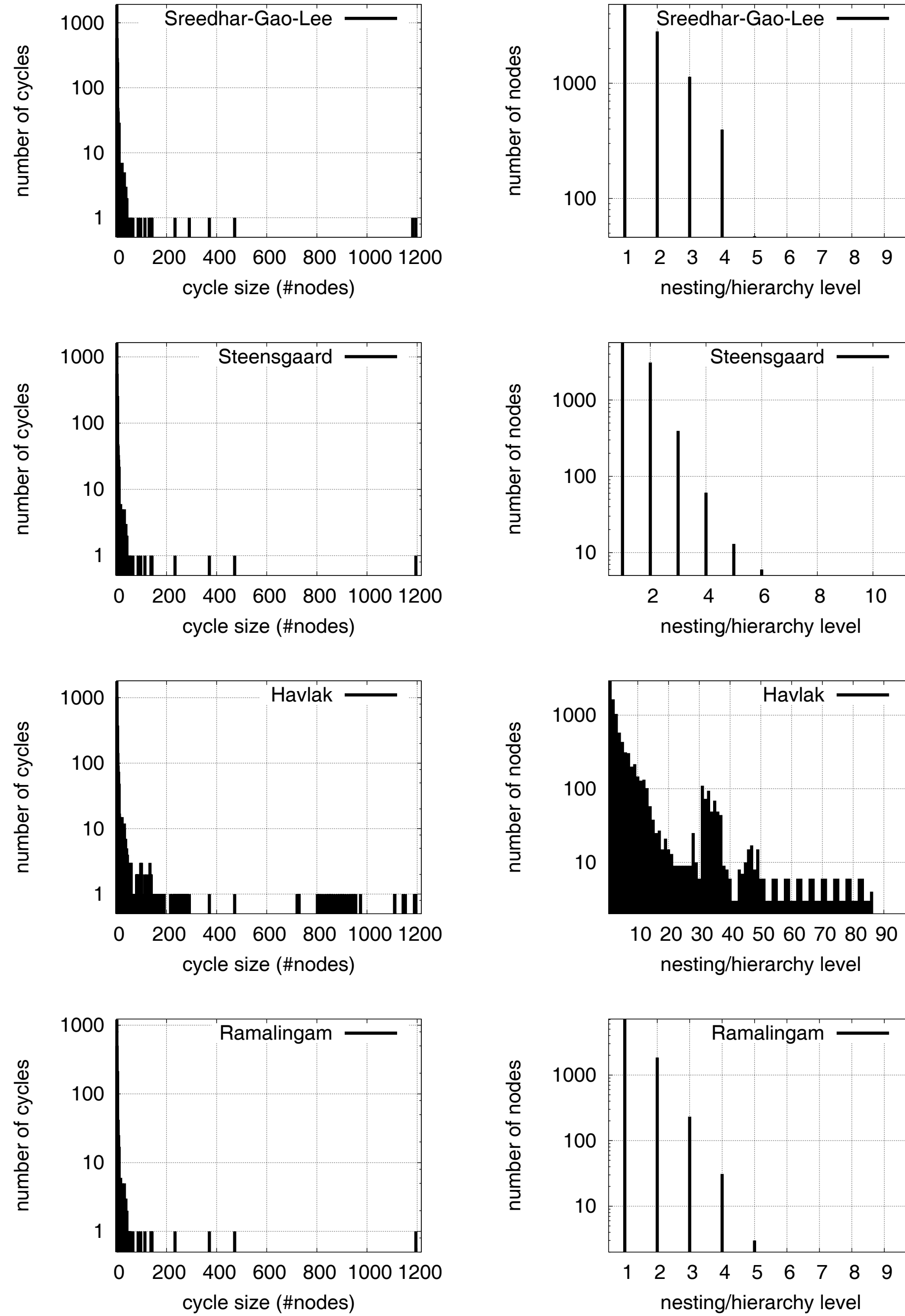
„larn 1“



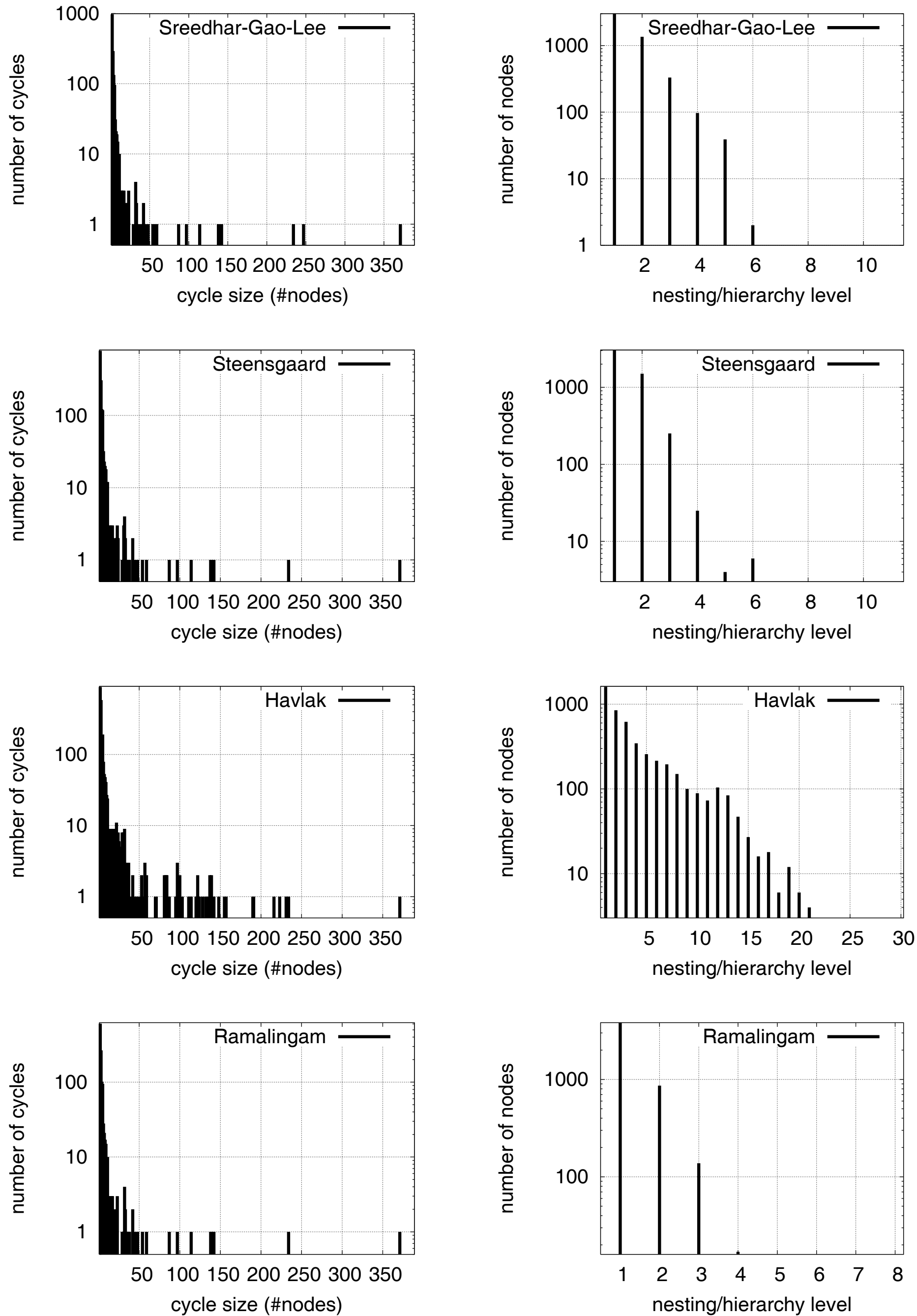
„larn 2“



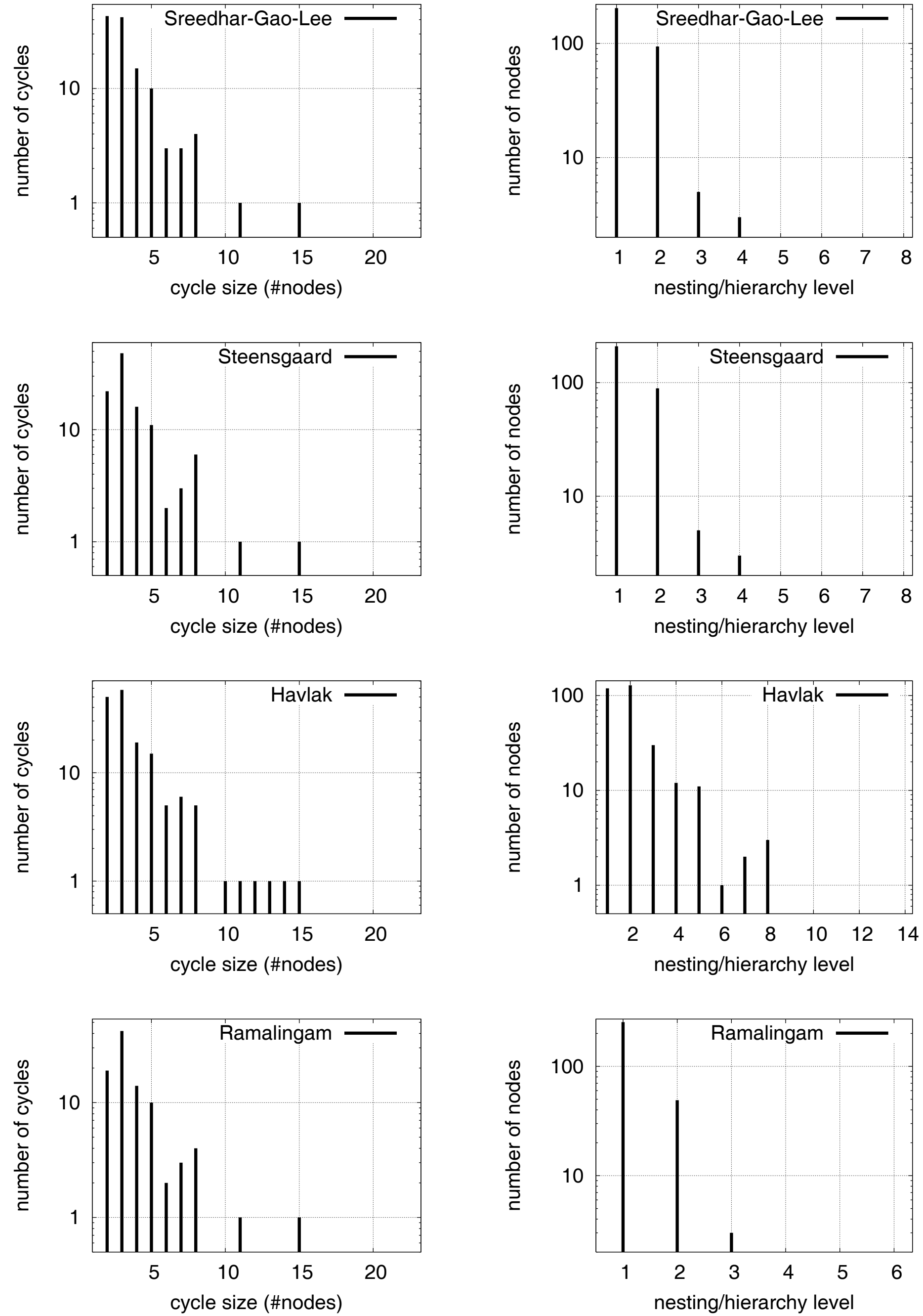
„moria 1"



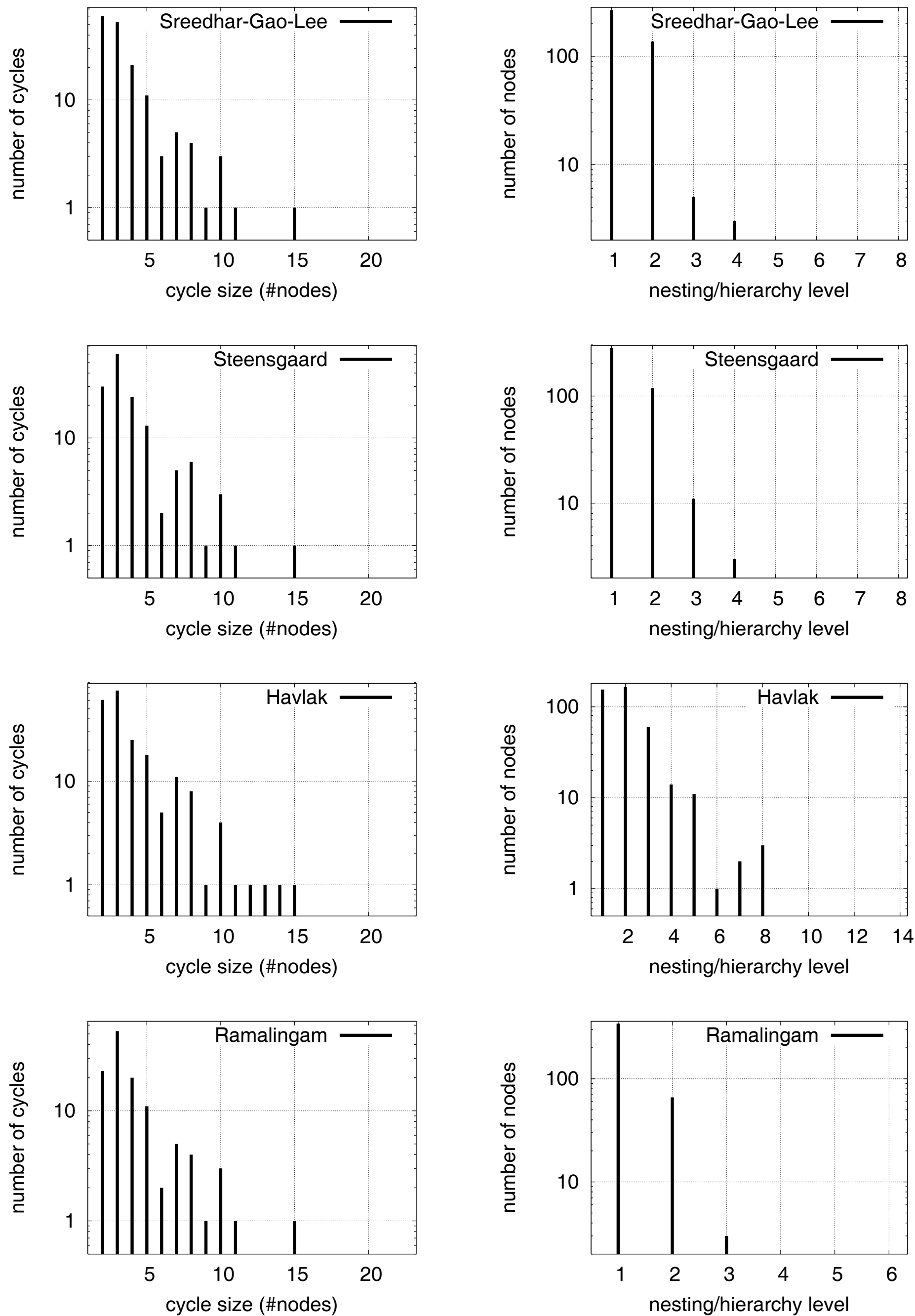
„moria 2“



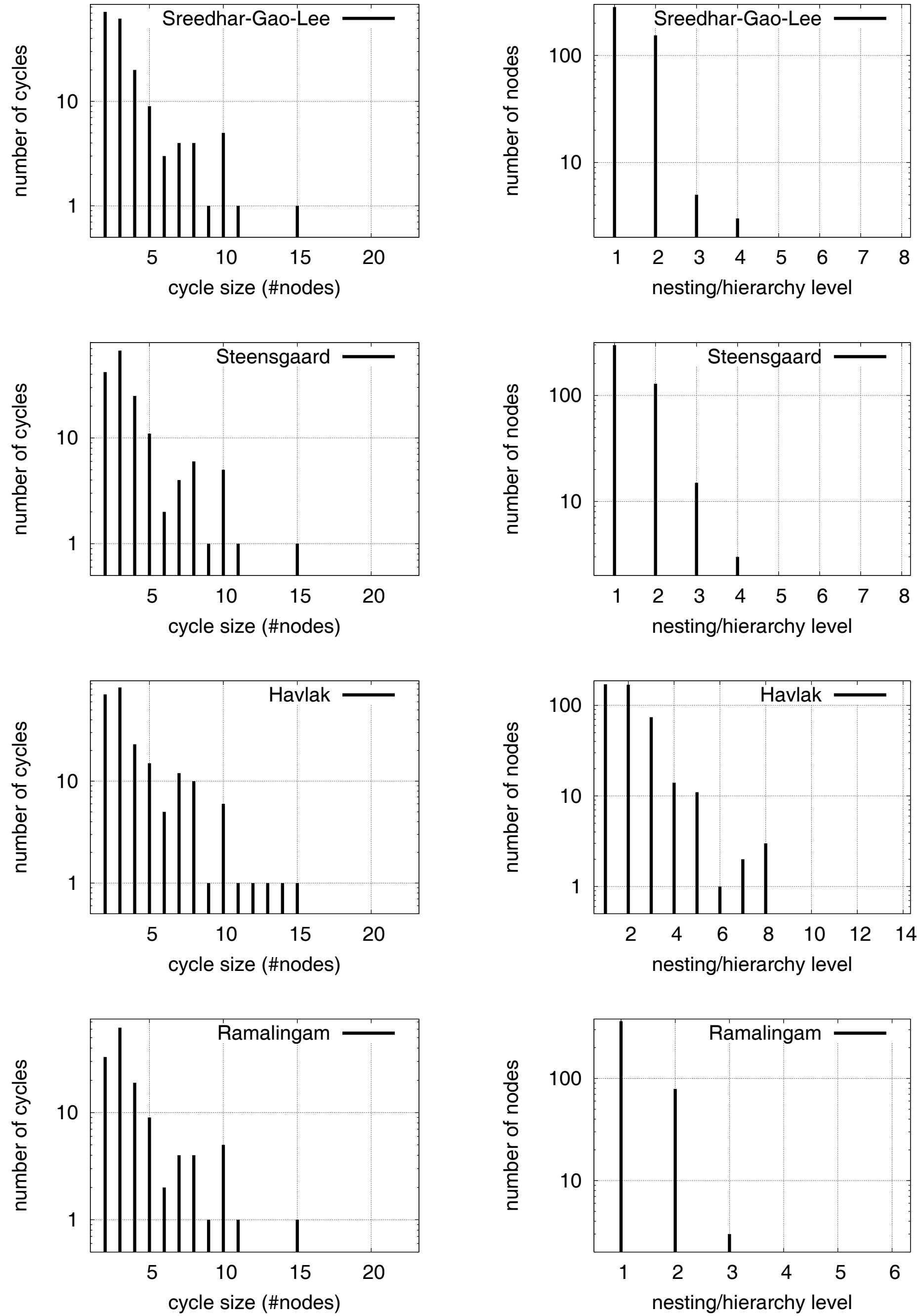
„palme 1“



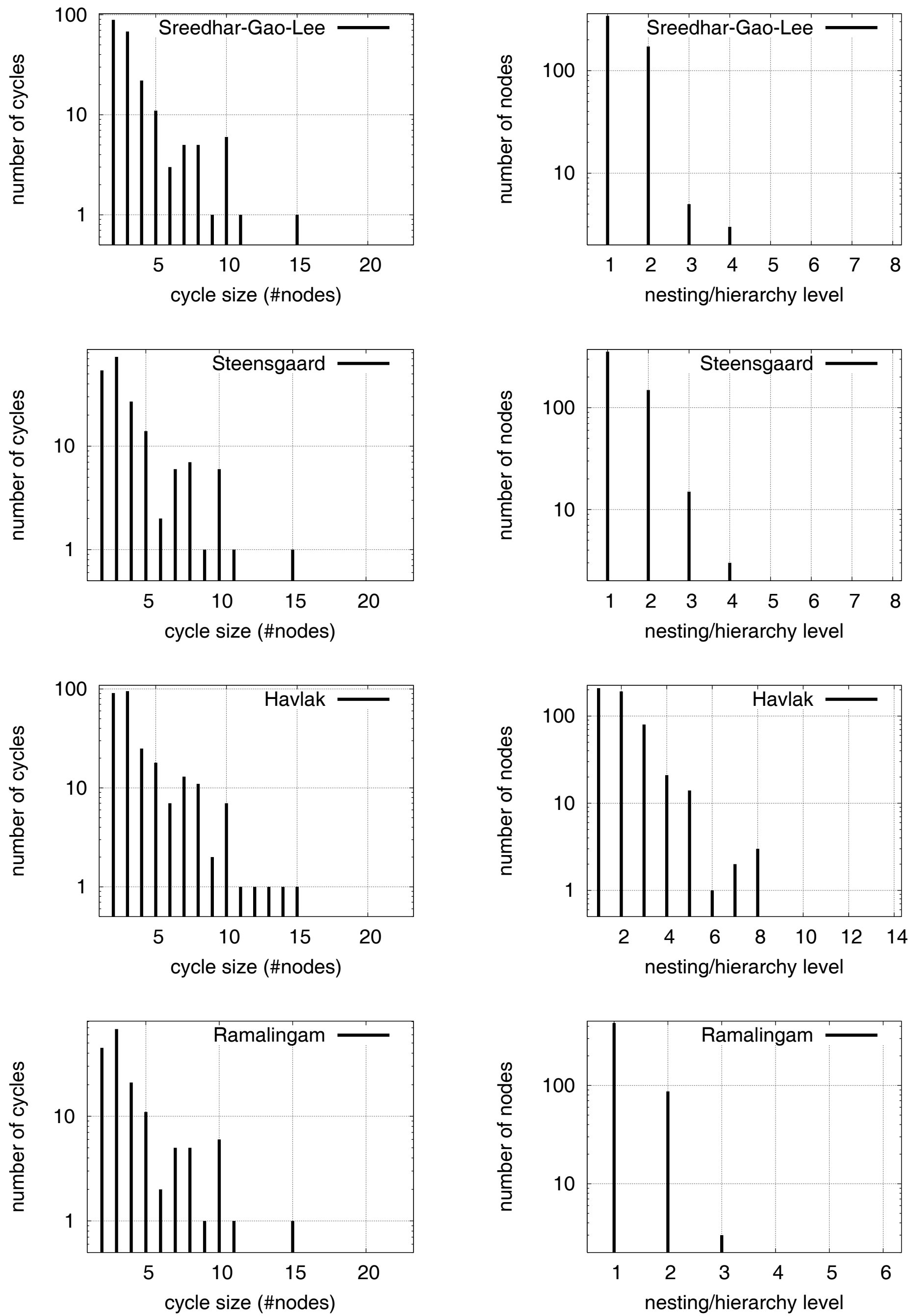
„palme 2"



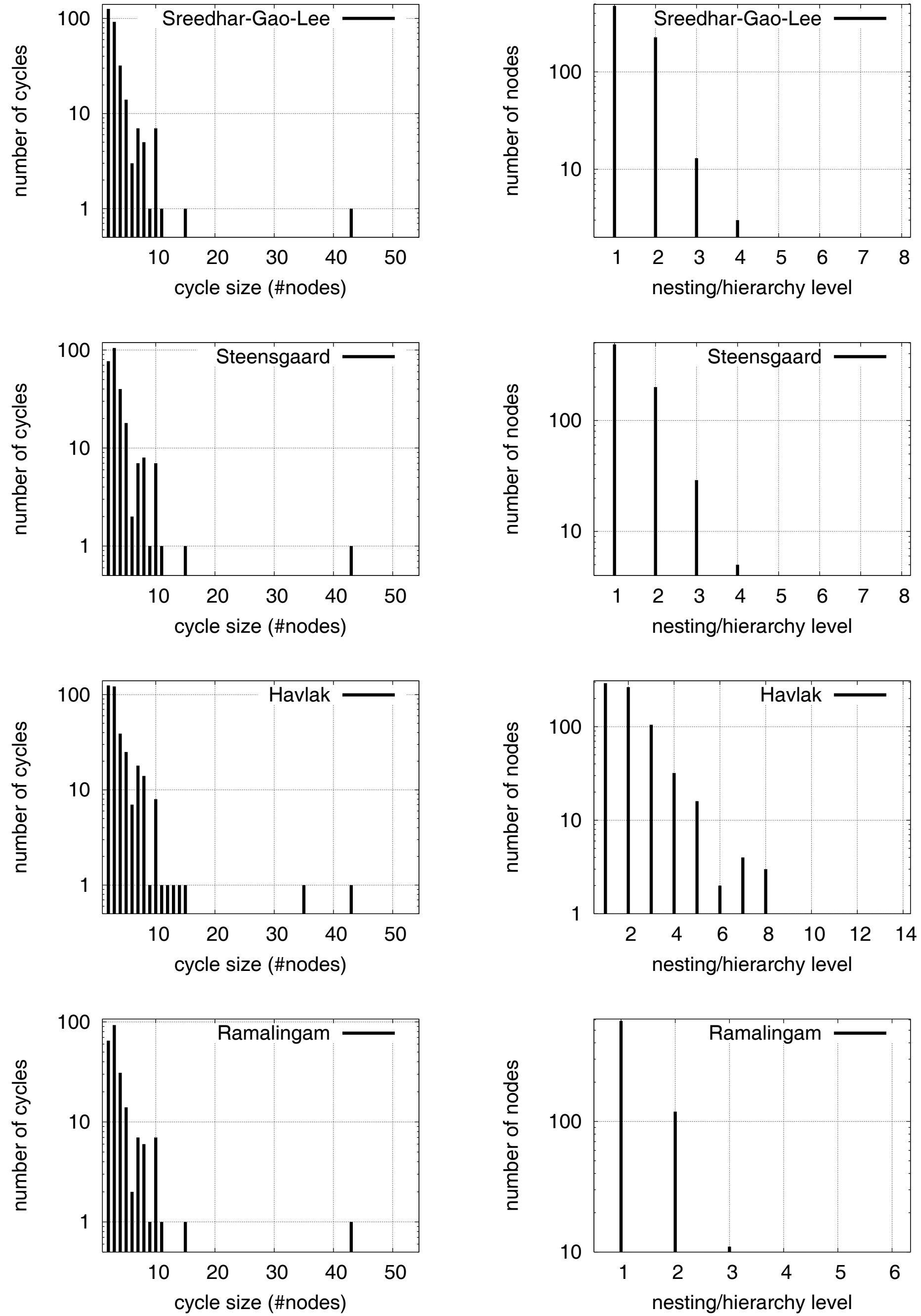
„palme 3"



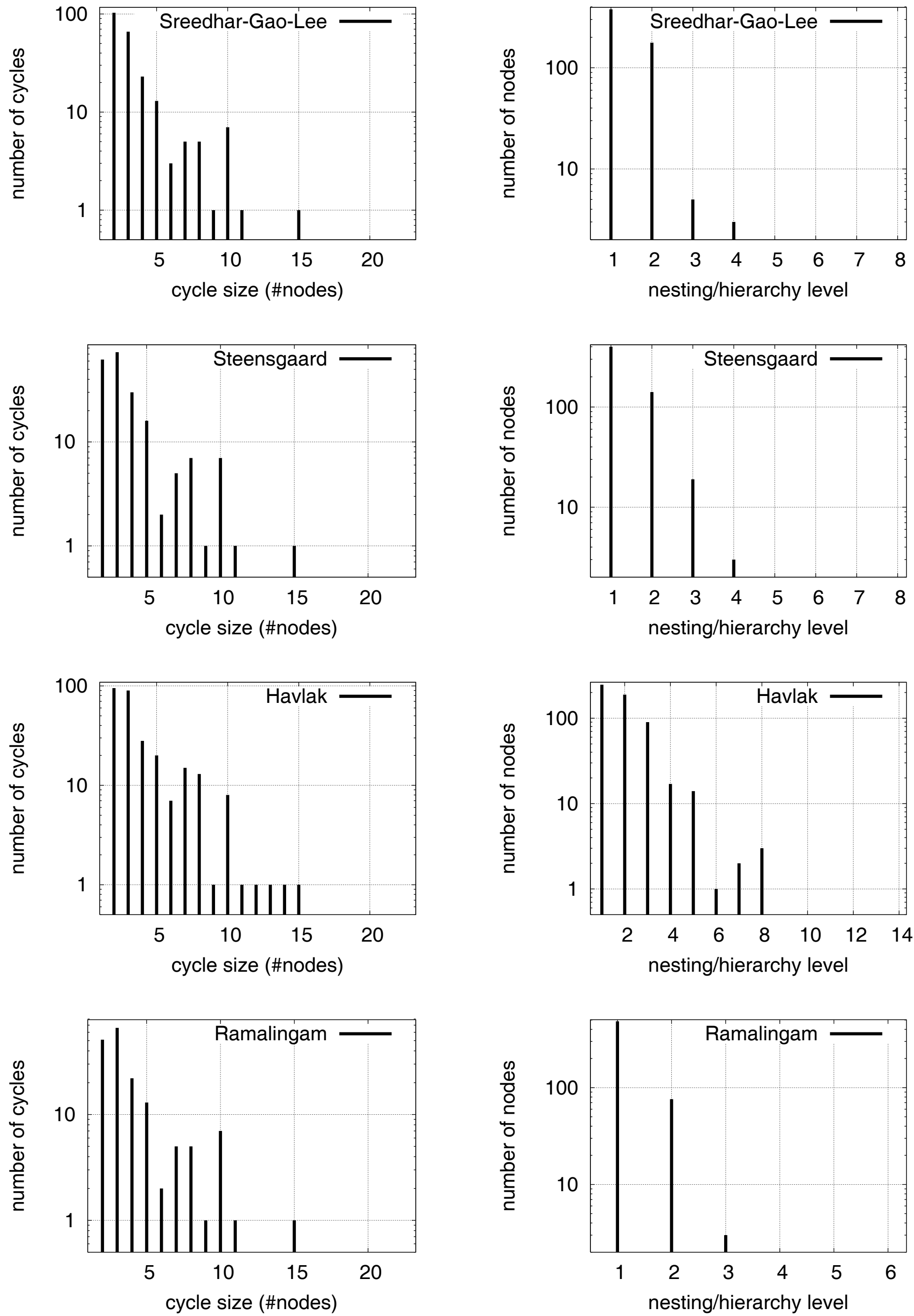
„palme 4“



„palme 5“



„palme 6“



Literaturverzeichnis

[ADS91] Hiralal Agrawal, Richard A. DeMillo, and Eugene H. Spafford. Dynamic slicing in the presence of pointers, arrays and records. In *Proceedings of the ACM Fourth Symposium on Testing, Analysis and Verification (TAV4)*, pages 60–73, New York, October 1991. ACM Press.

[AG04] Elisabeth Angerer-Grubmüller. Constraint-Solving von Pfadbedingungen zur Validierung sicherheitskritischer Systeme. 2004.

[BAG00] Leeann Bent, Darren C. Atkinson, and William G. Griswold. A comparative study of two whole program slicers for C. Technical Report CS2000-0643, University of California, San Diego, Computer Science and Engineering, 2000.

[Bal98] Helmut Balzert. *Lehrbuch der Software-Technik*. Spektrum Akademischer Verlag, 1998.

[BC93] Frederic Benhamou and Alain Colmerauer. *Constraint Logic Programming: Selected Research*. MIT Press, 1993.

[BHK89] J.A. Bergstra, J. Heering, and P. Klint. *Algebraic specifications*. ACM Press/Addison Wesley, 1989.

[Bib77] K. Biba. Integrity considerations for secure computer systems. *MITRE Technical Report 3153*, 1977.

[BL73] D. Bell and L. La Padula. Secure computer systems: mathematical foundations. *MITRE Technical Report 2547*, March 1973.

[BL76] D.E. Bell and L. La Padula. Secure computer systems: Unified expositions and multics interpretation. *MITRE Technical Report 2997*, March 1976.

[BM92] Robert A. Ballance and Arthur B. Maccabe. Program dependence graphs for the rest of us. Technical Report TR 92-10, Department of Computer Science, The University of New Mexico, October 1992.

[Bös02] Stephan Bösebeck. Visualisierung von Programmabhängigkeitsgraphen in C-Quelltexten mittels Overlaying. 2002.

[BPS00a] William Bush, Jonathan Pincus, and David Sielaff. Analysis is necessary, but far from sufficient: Experiences building and deploying successful tools for developers and testers. *International Symposium of Software Testing and Analysis*, 2000. Opening talk.

[BPS00b] William Bush, Jonathan Pincus, and David Sielaff. A static analyzer for finding dynamic programming errors. *Software-Practice and Experience*, 30:775–802, 2000.

[BR00] Thomas Ball and Sriram K. Rajamani. Bebop: A symbolic model checker for boolean programs. In *Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification*, pages 113–130. Springer-Verlag, 2000.

[BR01] Thomas Ball and Sriram K. Rajamani. Bebop: a path-sensitive interprocedural dataflow engine. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 97–103. ACM Press, 2001.

[Bry86] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, pages 677–691, 1986.

[BSI03a] BSI. *Bundesamt für Sicherheit in der Informationstechnik: IT-Sicherheitskriterien*, 2003.
http://www.bsi.de/zertifiz/itkrit/itkrit.htm.

[BSI03b] BSI. *Bundesamt für Sicherheit in der Informationstechnik: Zertifizierung-/IT-Sicherheitskriterien*, 2003.
http://www.bsi.de/zertifiz/zert/index.htm.

[BSI03c] BSI. *Leitfaden für die Erstellung und Prüfung formaler Sicherheitsmodelle im Rahmen von ITSEC und Common Criteria*, 2003.
http://www.bsi.com.

[Bun97] Physikalisch-Technische Bundesanstalt. *Merkblatt für Anträge auf Bauartzulassung von Meßgeräten*, 1997.
http://www.ptb.de.

[CCD98] G. Canfora, A. Cimitile, and A. De Lucia. Parametric program slicing. In *Harman, M., Gallagher, K. (Eds.), Information and Software Technology Special Issue on Program Slicing*, volume 40, pages 595–607. Elsevier Science, 1998.

[CDHR00] James C. Corbett, Matthew B. Dwyer, John Hatcliff, and Roby. Bandera: a source-level interface for model checking java programs. In *Proceedings of the 22nd international conference on Software engineering*, pages 762–765. ACM Press, 2000.

[CFR+91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, pages 451–490, October 1991.

[CL94] Yoonsik Cheon and Gary T. Leavens. A quick overview of larch/c++. *Journal of Object-Oriented Programming*, 7(6):39–49, 1994.

[CLR01] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press/McGraw-Hill, 2001.

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2001.

[Com99] Common Criteria Project Sponsoring Organizations. Common criteria for information technology security evaluation. *ISO/IEC 15408*, 1999.

[DB98] Rolf Drechsler and Bernd Becker. *Graphenbasierte Funktionsdarstellung: Boolesche und Pseudo-Boolesche Funktionen*. Teubner, 1998.

[DD77] Dorothy Denning and Peter Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, 1977.

[DS] Andreas Dolzmann and Thomas Sturm. Redlog user manual.

[DS97] Andreas Dolzmann and Thomas Sturm. Redlog: Computer algebra meets computer logic. *ACM SIGSAM Bulletin*, 31(2):2–9, 1997.

[DSW98] Andreas Dolzmann, Thomas Sturm, and Volker Weispfenning. Real quantifier elimination in practice. in B.H. Matzat, G.-M. Greul, and G. Hiss, editors. *Algorithmic Algebra and Number Theory*, pages 221–247, 1998.

[EC94] Model Checking Group Edmund M. Clarke. The BDD library. Technical report, Carnegie Mellon University, 1994. http://www-2.cs.cmu.edu/afs/cs/project/modck/pub/www/bdd.html.

[EGHT94] David Evans, John Guttag, James Horning, and Yang Meng Tan. LCLint: A tool for using specifications to check code. In *Proceedings of the ACM SIGSOFT '94 Symposium on the Foundations of Software Engineering*, pages 87–96, 1994.

[Eva94] D. Evans. Using specifications to check source code. Technical Report MIT/LCS/TR-628, 1994.

[FA97] T. Frühwirth and S. Abdennadher. *Constraint-Programmierung.* Springer, 1997.

[FLL+02] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 234–245. ACM Press, 2002.

[FOW87] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[FRT95] John Field, G. Ramalingam, and Frank Tip. Parametric program slicing. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 379–392. ACM Press, 1995.

[FTA02] Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. Flow-sensitive type qualifiers. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 1–12. ACM Press, 2002.

[GBR98] Arnaud Gotlieb, Bernard Botella, and Michael Rueher. Automatic test data generation using constraint solving techniques. In *Proc. International Symposium on Software Testing and Analysis*, pages 53–62. ACM, 1998.

[GMS98] Neelam Gupta, Aditya Mathur, and Mary Lou Soffa. Automated test data generation using an iterative relaxation model. In *Proc. International Symposium on Foundations of Software Engineering*, pages 231–244. ACM, 1998.

[GWZ94] Allen Goldberg, T. C. Wang, and David Zimmerman. Applications of feasible path analysis to program testing. In *Proceedings of the 1994 international symposium on Software testing and analysis*, pages 80–94. ACM Press, 1994.

[Hav97] Paul Havlak. Nesting of reducible and irreducible loops. *ACM Transactions on Programming Languages and Systems*, pages 557–567, July 1997.

[HRB90] Susan B. Horwitz, Thomas W. Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, January 1990.

[HS96] G. Hachtel and F. Somenzi. *Logic Synthesis and Verification Algorithms*. Kluwer Academic Publisher, 1996.

[HS04] C. Hammer and G. Snelting. An Improved Slicer for Java. *Proc. ACM SIGPLAN/SIGFSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'04)*, 2004.

[Hun03] Thomas Hungenberg. *Online-Lexikon der Informationssicherheit*, 2003.
http://www.demonium.de/th/home/sicherheit/lexikon/index.phtml.

[IP02] IC-PARC. *The ECLiPSe Constraint Programming System*, 2002.
http://www-icparc.doc.ic.ac.uk/eclipse/features.html.

[JG84] J. Meseguer J. A. Goguen. Unwinding and inference control. In *Proc. Symposium on Security and Privacy*, pages 75–86. IEEE, 1984.

[Kri03] Jens Krinke. *Advanced Slicing of Sequential and Concurrent Programs*. Universität Passau, April 2003.

[KS98] Jens Krinke and Gregor Snelting. Validation of measurement software as an application of slicing and constraint solving. *Information and Software Technology*, pages 661–675, November/December 1998. Special issue on Program Slicing.

[LAS00] Tal Lev-Ami and Shmuel Sagiv. Tvla: A system for implementing static analyses. In *Proceedings of the 7th International Symposium on Static Analysis*, pages 280–301. Springer-Verlag, 2000.

[Lea96] Gary T. Leavens. An overview of Larch/C++: Behavioral specifications for C++ modules. In Haim Kilov and William Harvey, editors, *Specification of Behavioral Semantics in Object-Oriented Information Modeling*, pages 121–142. Kluwer Academic Publishers, Boston, 1996.

[LEW96] J. Loeckx, H-D. Ehrich, and M. Wolf. *Specification of Abstract Data Types*. Wiley-Teubner, 1996.

[LN98] K. Rustan M. Leino and Greg Nelson. An extended static checker for Modula-3. In Kai Koskimies, editor, *Proceedings of the 7th International Conference on Compiler Construction, CC'98*, volume 1383 of *Lecture Notes in Computer Science*, pages 302–305. Springer, April 1998.

[LN03] Jorn Lind-Nielsen. BuDDy - a binary decision diagram package. Technical report, University of Copenhagen, 2003. http://www.itu.dk/research/buddy/.

[LR04] AT&T Labs-Research. Graphviz. Technical report, AT&T, 2004. http://www.research.att.com/sw/tools/graphviz/.

[LSS01] *PalMe – Secure Palm-based Money Exchange.* Lehrstuhl für Software und Systems Engineering, Technische Universität München, 2001. http://www4.in.tum.de/∼palme/.

[LT79] Thomas Lengauer and Robert Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Transaction on Programming Languages and Systems*, pages 121–141, July 1979.

[Map] Maplesoft. *Maple.* http://www.maplesoft.com.

[Mar98] Florian Martin. PAG – an efficient program analyzer generator. *International Journal on Software Tools for Technology Transfer*, 2(1):46–67, 1998.

[McC56] E.J. McCluskey. Minimization of Boolean Functions. *Bell Sys. Tech. Journal*, 35(5):1417–1444, 1956.

[Met] MetroWorks. *MetroWorks CodeWarrior for PalmOS.* http://www.metroworks.com.

[MO91] R.A. de Millo and A.J. Offut. Constraint-based automatic test data generation. *IEEE Transactions in Software Engineering*, pages 900–910, September 1991.

[MS98] Kim Marriott and Peter Stuckey. *Programming with Constraints.* MIT Press, 1998.

[NNH99] F. Nielson, H. Nielson, and C. Hankin. *Principles of Program Analysis.* Springer Verlag, 1999.

[Nor99] D. Nor. Detecting memory errors via static pointer analysis. 1999. M.Sc. Thesis.

[NRS98] D. Nor, M. Rodeh, and M. Sagiv. Detecting memory errors via static pointer analysis. *Proc. ACM SIGPLAN/SIGFSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'98)*, 1998.

[NRS00] D. Nor, M. Rodeh, and M. Sagiv. Checking cleanness in linked lists. *Static Analysis; Third International Symposium, SAS'00*, 2000.

[OO84] Karl J. Ottenstein and Linda M. Ottenstein. The program dependence graph in a software development environment. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, volume 19(5) of *ACM SIGPLAN Notices*, pages 177–184, 1984.

[Pal04a] PalmSource. *PalmOS Simulator*, 2004. http://www.palmos.com/dev/tools/simulator/.

[Pal04b] PalmSource. *PalmOS Software Development Kit*, 2004. http://www.palmos.com/dev/tools/sdk/.

[PC90] Andy Podgurski and Lori A. Clarke. A formal model of program dependences and it's implications for software testing. *IEEE Transactions on Software Engineering*, 16(9):965–979, 1990.

[PCCW93] M.C. Paulk, B. Curtis, M.B. Chrissis, and C. Weber. Capability maturity model for software, version 1.1. 1993. http://www.sei.cmu.edu/cmm/cmm.html.

[PRC03] *PRC-Tools*, 2003. http://prc-tools.sourceforge.net.

[PW98] William Pugh and David Wonnacott. Constraint-based array dependency analysis. *ACM Transaction on Programming Languages and Systems*, pages 1248–1278, May 1998.

[Qui52] W.V. Quine. The Problem of Simplifying Truth Functions. *American Mathematical Monthly*, 59(8):521–531, 1952.

[Qui55] W.V. Quine. A Way to Simplify Truth Functions. *American Mathematical Monthly*, 62:627–631, 1955.

[Ram99] G. Ramalingam. Identifying loops in almost linear time. *ACM Transactions on Programming Languages and Systems*, 21(2):175–188, 1999.

[Ram02] G. Ramalingam. On loops, dominators, and dominance frontiers. *ACM Transactions on Programming Languages and Systems*, 24(5):455–490, 2002.

[Ran98] Rajeev Ranjan. CAL BDD package. Technical report, University of California, Berkeley, 1998. http://www-2.cs.cmu.edu/afs/cs/project/modck/pub/www/bdd.html.

[Rep98] Tom Reps. Program analysis via graph reachability. *Information and Software Technology*, pages 701–726, November/December 1998. Special issue on program slicing.

[Rep00] Thomas Reps. Undecideability of context-sensitive data-dependence analysis. *ACM Transactions on Programming Languages and Systems*, pages 162–186, January 2000.

[Res] Wolfram Research. *Mathematica*. http://www.wolfram.com.

[RHSR94a] Thomas Reps, Susan Horwitz, Mooly Sagiv, and Genevieve Rosay. Speeding up slicing. In *Proc. Foundations of Software Engineering*, pages 11–20. ACM, 1994.

[RHSR94b] Thomas W. Reps, Susan B. Horwitz, Mooly Sagiv, and Genevieve Rosay. Speeding up slicing. In *SIGSOFT'94: Proceedings of the Second ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 11–20, New York, December 1994. ACM Press. 19(5).

[RR95] Thomas W. Reps and Genevieve Rosay. Precise interprocedural chopping. In *SIGSOFT'95: Proceedings of the Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Washington, DC, October 1995.

[SA03] A. Sabelfeld and A.C. Myers. Language-Based Information-Flow Security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.

[SAF90] B. Simons, D. Alpern, and J. Ferrante. A foundation for sequentializing parallel code. In *Proceedings of the second annual ACM symposium on Parallel algorithms and architectures*, pages 350–359. ACM Press, 1990.

[SCFY96] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, 1996.

[Sch00] Uwe Schöning. *Logik für Informatiker*. Spektrum Akademischer Verlag, 2000.

[SF93] Barbara Simons and Jeanne Ferrante. An efficient algorithm for constructing a control flow graph for parallel code. Tech-

nical Report TR 03.465, Santa Teresa Laboratory, San Jose, California, February 1993.

[SGL96] Vugranam C. Sreedhar, Guang R. Gao, and Yong-Fong Lee. Identifying loops using DJ graphs. *ACM Transactions on Programming Languages and Systems*, pages 649–658, November 1996.

[SKRS98] B. Sick, M. Keidl, M. Ramsauer, and S. Seltzsam. A comparison of traditional and soft-computing methods in a real-time control application. In *L. Niklasson, M. Boden, T. Ziemke (Hrsg.); ICANN '98 (Proceedings of the 8th International Conference on Artificial Neural Networks)*. Springer Verlag London Ltd., 1998. Series: Perspectives in Neural Computing; Vol. 2, S. 725 - 730; Skövde, 2. - 4.; ISBN 3-540-76263-9.

[SL02] Stephane Lussier. *Part of Your Complete Breakfast – Code Review is a Source of Essential Vitamins and Minerals*, 2002. http://www.macadamian.com/column/.

[Sne96] Gregor Snelting. Combining slicing and constraint solving for validation of measurement software. In *Proc. Static Analysis Symposium*, volume 1145 of *LNCS*, pages 332–348, 1996.

[Som04] Fabio Somenzi. CUDD - CU decision diagram package. Technical report, University of Colorado at Boulder, 2004. http://vlsi.colorado.edu/ fabio/CUDD/.

[Sou03a] Dr. Amie Souter. *Compiler Construction I*, 2003. http://www.cs.drexel.edu/∼souter/cs761.

[Sou03b] Dr. Amie Souter. *Compiler Construction II*, 2003. http://www.cs.drexel.edu/∼souter/cs762.

[SRK03] Gregor Snelting, Torsten Robschink, and Jens Krinke. Efficient Path Conditions in Dependence Graphs for Software Safety Analysis. *Submitted for publication*, 2003.

[Ste93] Bjarne Steensgaard. Sequentializing program dependence graphs for irreducible programs. Technical Report MSR-TR-93-14, Redmond, WA, 1993.

[Stu02] Thomas Sturm. Quantifier Elimination-Based Constraint Logic Programming. *FMI, University of Passau, Germany*, January 2002.

[SV98] Geoffrey Smith and Dennis Volpano. Secure information flow in a multi-threaded imperative language. In *Proceedings of the*

Twenty-Fifth ACM Symposium on Principles of Programming Languages, pages 355–364, San Diego, CA, January 1998.

[SW96] Thomas Sturm and Volker Weispfenning. Computational geometry problems in REDLOG. In *Automated Deduction in Geometry*, pages 58–86, 1996.

[Tar74] Robert Endre Tarjan. Testing flow graph reducibility. *Journal of Computer and System Sciences*, 9:355–365, 1974.

[TCA] The Countryside Agency. *The Countryside Agency.* http://www.countryside.gov.uk.

[TRC93] Margaret C. Thompson, Debra J. Richardson, and Lori A. Clarke. An information flow model of fault detection. In *Proceedings of the International Symposium of Software Testing and Analysis*, pages 182–192. ACM Press, June 1993.

[Weg89] Ingo Wegener. *Effiziente Algorithmen für grundlegende Funktionen.* B.G. Teubner, Stuttgart, 1989.

[Wei79] Mark Weiser. *Program slices: formal, psychological, and practical investigations of an automatic program abstraction method.* PhD thesis, University of Michigan, Ann Arbor, 1979.

[Wei82] Mark Weiser. Programmers use slices when debugging. *Communications of the ACM*, 25(7):446–452, 1982.

[Wei84] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, July 1984.

[Wei97a] Weispfenning. Complexity and uniformity of elimination in presburger arithmetic. In *ISSAC: Proceedings of the ACM SIGSAM International Symposium on Symbolic and Algebraic Computation (formerly SYMSAM, SYMSAC, EUROSAM, EUROCAL) (also sometimes in cooperation with the Symbolic and Algebraic Manipulation Groupe in Europe (SAME))*, 1997.

[Wei97b] Volker Weispfenning. Simulation and optimization by quantifier elimination. *Journal of Symbolic Computation*, 24(2):189–208, 1997.

[Wei99] Volker Weispfenning. Mixed real-integer linear quantifier elimination. In *ISSAC: Proceedings of the ACM SIGSAM International Symposium on Symbolic and Algebraic Computation*, pages 129–136, 1999.