SCCharts Kompilierung für eingebettete Systeme mit limitierten Ressourcen

Jonas Busse

Bachelorarbeit
September 2016

Arbeitsgruppe Echtzeitsysteme und Eingebettete Systeme
Prof. Dr. Reinhard von Hanxleden
Institut für Informatik
Christian-Albrechts-Universität zu Kiel

Betreut durch
Dipl.-Inf. Ass. jur. Insa Fuhrmann
Dipl.-Inf. Steven Smyth

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Kiel,

Vorwort

Im Bereich der eingebetteten Systeme wird häufig mit sowohl physikalisch als auch leistungstechnisch kleinen Systemen gearbeitet. Dies hängt mit dem eingebetteten Kontext zusammen. Häufig sind die Systeme so minimal gestaltet, dass Funktionalitäten wie Netzwerkkommunikation oder eine Bildschirmausgabe entfernt werden, um weniger Ressourcen zu verwenden oder ganze Hardwarekomponenten nicht verbauen zu müssen. Es wird nur verbaut, was für den Anwendungszweck benötigt wird. Für Entwickler eingebetteter Systeme ergibt sich in diesem Kontext die Schwierigkeit, dass die zu entwickelnden Programme schnell an eine harte Speicherbegrenzung stoßen. Hierzu optimieren die meisten Kompiler den erzeugten Maschinencode bereits.

Diese Arbeit behandelt einen leicht abgewandelten Ansatz. Auf Basis der von von Hanxleden et al. [HDM+14] entwickelten Sequentially Constructive Charts (SCCharts) wird heute bereits im Kiel Integrated Environment for Layout Eclipse Rich Client (KIELER)[1] mittels Transformationen ausführbarer Code erzeugt. Während dieser Transformationen setzt die Implementierung dieser Arbeit an und optimiert die Modelle bereits vor der Codegenerierung. Dies hat zum Vorteil, dass die optimierten Modelle in der KIELER-Umgebung zusätzlich visualisiert werden können.

Zudem bietet diese Ausarbeitung einen Einblick in die für eingebettete Systeme nutzbaren High-Level-Konstrukte in SCCharts. Unter High-Level-Konstrukten verstehen sich erweiterte Programmierkonstrukte mit komplexer Funktionalität, welche auf Basiskonstrukte der Sprache abgebildet werden können.

[1] https://rtsys.informatik.uni-kiel.de/confluence/display/KIELER/

Inhaltsverzeichnis

1 Einleitung **1**

1.1 SCCharts 1

1.2 Problemstellung 4

1.3 Beitrag der Arbeit 7

1.4 Aufbau 8

2 Verwandte Arbeiten **11**

2.1 Kompileroptimierung 11

2.1.1 Konstantenpropagation 12

2.1.2 Kopierpropagation 13

2.1.3 Registerwiederverwendung 15

2.1.4 Eliminierung toten / nicht erreichbaren Codes 16

2.1.5 Eliminierung nicht verwendeten Codes 18

2.1.6 Fallunterscheidungen optimieren 20

2.2 Hardwaresynthese 23

3 Verwendete Technologien **25**

3.1 KIELER 25

3.1.1 SCCharts 26

3.1.2 KIELER Compiler 29

3.1.3 Single-Pass Language-Driven Incremental Compilation (SLIC) 30

3.1.4 TestRunner 30

3.1.5 KIELER Execution Manager 31

3.2 SonarQube 32

4 Konzept **35**

4.1 Analyse Quellcode 35

4.2 Kopierpropagation 42

4.3 Konstantenpropagation 43

4.4 Variablen wiederverwenden 44

4.5 Eliminierung ungenutzten Codes 45

5 Implementierung Quellcodeoptimierung **47**
5.1 Kopierpropagation 48
5.2 Variablenwiederverwendung 51

6 Evaluation Quellcodeoptimierung **55**
6.1 Ziel 56
6.2 Quantitative Auswertung realer Modelle 58
6.3 Ergebnisse ausgewählter Beispiele 75
6.4 Quantitative Auswertung automatisch generierter Modelle 81
6.5 Analyse der High-Level-Transformationen 84
6.6 Analyse der Optimierungen im Kontext der Hardwaresynthese 88

7 Zusammenfassung **93**
7.1 Ausblick 93
7.2 Fazit 94

Bibliography **95**

Abbildungsverzeichnis

1.1 Core & Extended SCCharts - Übersicht 2
1.2 Core & Extended SCCharts — Tabellarische Auflistung 3
1.3 Übersicht SCG / SCL Instruktionen 3
1.4 BarcodeReader Modell mit zusammengefassten Makrozuständen . . . 5
1.5 ABO Modell 7
1.6 ABO Java-Code 7

2.1 Konstantenpropagation - Beispiel [Gue99, S. 262] 12
2.2 Positivbeispiel Kopierpropagation nach Muchnick [Muc97, S. 357], dargestellt als SCCharts 14
2.3 Negativbeispiel Kopierpropagation nach Muchnick [Muc97, S. 362], dargestellt in SCCharts 15
2.4 Registerwiederverwendung - Beispiel 16
2.5 Eliminierung toten Codes - Beispiel [Gue99, S. 266] 17
2.6 Nicht verwendeter Code durch Optimierung 17
2.7 Nicht verwendeter Code durch Optimierung 19
2.8 T4f Transitionsübersetzung Beispiel - Codegenerierung für sequenzielle Konstruktivität, [Smy13, S. 48] 22

3.1 Abhängigkeitsgraph Extended SCCharts Transformationen [HDM+14] 27
3.2 Kompilerkette SCCharts nach Motika [MSH+13] 28
3.3 KiCo Compilerchain - Übersicht 29
3.4 Zufällig generierte ESO-Datei (2 Traces) des ABO-Modells (Spur zwei manuell um Ausgaben erweitert) 32

4.1 ABO Modell - Seq. SCG 36
4.2 Mehrfachberechnung Negation - Beispiel 37
4.3 ABO Modell mit Markierungen - Seq. SCG (Erste Markierung: Initiale Zuweisung, Weitere Markierungen: Verwendung) 38
4.4 ABO Modell Optimierung mittels Kopierpropagation - Seq. SCG . . . 39
4.5 Analyse ABO Modell Variable g9 39
4.6 ABO Modell Optimierung mittels Variablenwiederverwendung - Seq. SCG 40
4.7 ABO-Modell Kopierpropagation & Variablenwiederverwendung . . . 41

4.8 Kopierpropagation Erweiterung 42
4.9 Kopierpropagation Einschränkung 43
4.10 Kopierpropagation Einschränkung - Beispiel 43
4.11 Konstantenpropagation - Beispiel 44
4.12 Wiederverwendung von Variablen - Beispiel - Zulässige (a) und unzulässige (b) Wiederverwendung 45

5.1 AbstractProductionTransformation (Implementierung in KIELER durch Schulz-Rosengarten, A.) - UML 47
5.2 CopyPropagation - UML 48
5.3 Kopierpropagation Algorithmus 49
5.4 Relevante Variablen in dem ABO-Modell - Kopierpropagation 50
5.5 ReuseVariables - UML 51
5.6 Variablenwiederverwendung Algorithmus 52

6.1 Vergleich Knotenanzahl nach Optimierung - (Die Kurven Variablenwiederverwendung und Variablenwiederverwendung → Kopierpropagation, sowie die Kurven Kopierpropagation und Kopierpropagation → Variablenwiederverwendung überschneiden sich.) 59
6.2 Vergleich Knotenanzahl (bis 50) nach Optimierung - (Die Kurven Kopierpropagation und Kopierpropagation → Variablenwiederverwendung überschneiden sich.) 60
6.3 ABO Modell Optimierung durch Variablenwiederverwendung → Kopierpropagation 61
6.4 Minimum, Maximum, Durchschnitt und Median der prozentualen Optimierung der Knotenanzahl aus Abbildung 6.1 62
6.5 Vergleich Zuweisungsanzahl nach Optimierung - (Die Kurven Variablenwiederverwendung und Variablenwiederverwendung → Kopierpropagation, sowie die Kurven Kopierpropagation und Kopierpropagation → Variablenwiederverwendung überschneiden sich.) 63
6.6 Minimum, Maximum, Durchschnitt und Median der prozentualen Optimierung der Zuweisungsanzahl aus Abbildung 6.5 64
6.7 Vergleich Zuweisungsanzahl (bis 50) nach Optimierung - (Überschneidung der Kurven Kopierpropagation und Kopierpropagation → Variablenwiederverwendung) 65
6.8 Minimum, Maximum, Durchschnitt und Median der prozentualen Optimierung der Deklarationsanzahl aus Abbildung 6.9 66

6.9 Vergleich Deklarationsanzahl nach Optimierung - (Die Kurven Variablenwiederverwendung und Variablenwiederverwendung → Kopierpropagation überschneiden sich.) 67
6.10 Vergleich Deklarationsanzahl (bis 50) nach Optimierung 68
6.11 Vergleich Java-Bytecodegröße nach Optimierung 70
6.12 Vergleich Java-Bytecodegröße nach Optimierung bis 1500 Byte Ausgangsgröße 71
6.13 Minimum, Maximum, Durchschnitt und Median der prozentualen Optimierung der Bytecodegröße aus Abbildung 6.11 72
6.14 Vergleich Ausführungszeiten realer Modelle - (Prozentualer Unterschied der Ausführungszeit des optimierten SCGs zu dem unoptimierten SCG - KP: Kopierpropagation, VW: Variablenwiederverwendung) 73
6.15 Vergleich Ausführungszeiten realer Modelle (Kopierpropagation ohne negierte Terme) - (Prozentualer Unterschied der Ausführungszeit des optimierten SCGs zu dem unoptimierten SCG - KP: Kopierpropagation, VW: Variablenwiederverwendung) 74
6.16 Beispiel 1 - SCChart 75
6.17 Beispiel 1 - Kopierpropagation → Variablenwiederverwendung 76
6.18 Beispiel 2 - SCChart 77
6.19 Beispiel 3 - SCT-Modell 78
6.20 Beispiel 3 - seq. SCG 79
6.21 Beispiel 3 - Kopierpropagation 79
6.22 Beispiel 4 - SCT-Modell 79
6.23 Beispiel 4 - Variablenwiederverwendung 80
6.24 SCTGenerator - Einstellungen 82
6.25 Vergleich prozentuale Optimierung automatisch generierter Daten - (Prozentualer Unterschied der Modellgröße des optimierten SCGs zu dem unoptimierten SCG - KP: Kopierpropagation, VW: Variablenwiederverwendung) 83
6.26 Transition with Trigger - SCChart 85
6.27 Transition with Trigger - seq. SCG 85
6.28 High-Level-Transformationen Auswertung - Ressourcenschonende Konstrukte 86
6.29 High-Level-Transformationen Auswertung - Übergangsbereich zwischen Ressourcenschonenden und stark expandierenden Konstrukten 87
6.30 High-Level-Transformationen Auswertung - Stark expandierende Konstrukte 88
6.31 ABO-Modell - Hardwaresynthese 89
6.32 ThreeRegionsNormalTermination-Modell - Hardwaresynthese 90

6.33 ABO-Modell optimiert - Hardwaresynthese 91
6.34 ThreeRegionsNormalTermination-Modell optimiert - Hardwaresynthese 91

Abkürzungsverzeichnis

DTO Duplicate Transition Optimization

EMF Eclipse Modeling Framework

Ext. SCCharts Extended Sequentially Constructive Charts

KiCo KIELER Compiler

KIELER Kiel Integrated Environment for Layout Eclipse Rich Client

KIEM KIELER Execution Manager

M2M Modell-zu-Modell

SCCharts Sequentially Constructive Charts

SLIC Single-Pass Language-Driven Incremental Compilation

SSA Static Single Assignment

UML Unified Modeling Language

WTOTO Write-Thing-Once Transition Optimization

Kapitel 1

Einleitung

Bei der Entwicklung von eingebetteten Echtzeitsystemen stehen für die Ausführung eines Programms typischerweise limitierte Ressourcen zur Verfügung. Werden diese Systeme mit graphischen Programmiersprachen entwickelt, kann ein optimierter Codegenerierungsprozess dazu beitragen, größere Modelle unter diesen Bedingungen lauffähig zu machen. In diesem Zusammenhang steht die vorliegende Arbeit. Allgemein lässt sich die Arbeit in einen Optimierungskontext einordnen. Genauer werden Optimierungen und Verwendungsvorschläge für Programmierkonstrukte in Bezug auf die Verwendung in eingebetteten Systemen betrachtet.

Im weiteren Verlauf dieses Kapitels werden zuerst SCCharts genauer erläutert. Die zugrundeliegende Problemstellung der Ausarbeitung, sowie die sich daraus ergebenden Aufgaben folgen im Anschluss. Abgeschlossen wird das Kapitel mit der Erläuterung des weiteren Aufbaus der Ausarbeitung.

1.1 SCCharts

Die Arbeitsumgebung ist in diesem Fall eine Eclipse-Modellierungswerkzeug namens KIELER. Der Kiel Integrated Environment for Layout Eclipse Rich Client[1] wird von der Arbeitsgruppe Echtzeitsysteme und Eingebettete Systeme der Christian-Albrechts-Universität zu Kiel[2] entwickelt. KIELER ist zudem ein Projekt mit dem Ziel der Erstellung von ausführbaren Programmen mit Hilfe graphischer Darstellung als Automatengraph. Innerhalb der KIELER-Umgebung wird unter anderem die Sprache SCCharts, welche von von Hanxleden et al. [HDM+14] vorgestellt wurde, verwendet und graphisch dargestellt. Aus dieser Darstellung kann mit Hilfe von Umwandlungsschritten eine Art Kontrollflussgraph erstellt werden, auf dem die Optimierungen dieser Ausarbeitung funktionieren. Wie genau die KIELER-Umgebung aufgebaut ist wird in Teilkapitel 3.1 erläutert. SCCharts steht für Sequentially Constructive Charts. Diese werden im Anwendungskontext der Modellierung für sicherheitskritische reaktive Systeme entwickelt. Aus SCCharts Modellen werden später ausführbare

[1] http://rtsys.informatik.uni-kiel.de/confluence/display/KIELER/Overview
[2] http://www.rtsys.informatik.uni-kiel.de/en

Programme generiert. Der aktuelle Stand[3] erlaubt es C-, Java- und Arduino-Code zu generieren.

SCCharts basiert auf einer Kernsprache namens Core SCCharts. Auf SCCharts-Modellebene können neben den Core SCCharts Befehlen auch erweiterte Strukturen, sogenannte Extended SCCharts, verwendet werden. Diese bündeln Core SCCharts Befehle zu komplexen Ausdrücken und machen diese einfach verwendbar. Während der Kompilierung werden die erweiterten Strukturen mittels semantik-erhaltener Modell-zu-Modell (M2M)-Transformationen auf Core SCCharts Befehle zurückgeführt. Hierbei wird von Expansion gesprochen. In Abbildung 1.1 sind die verfügbaren Sprachkonstrukte der Core und Extended SCCharts[4] zu sehen.

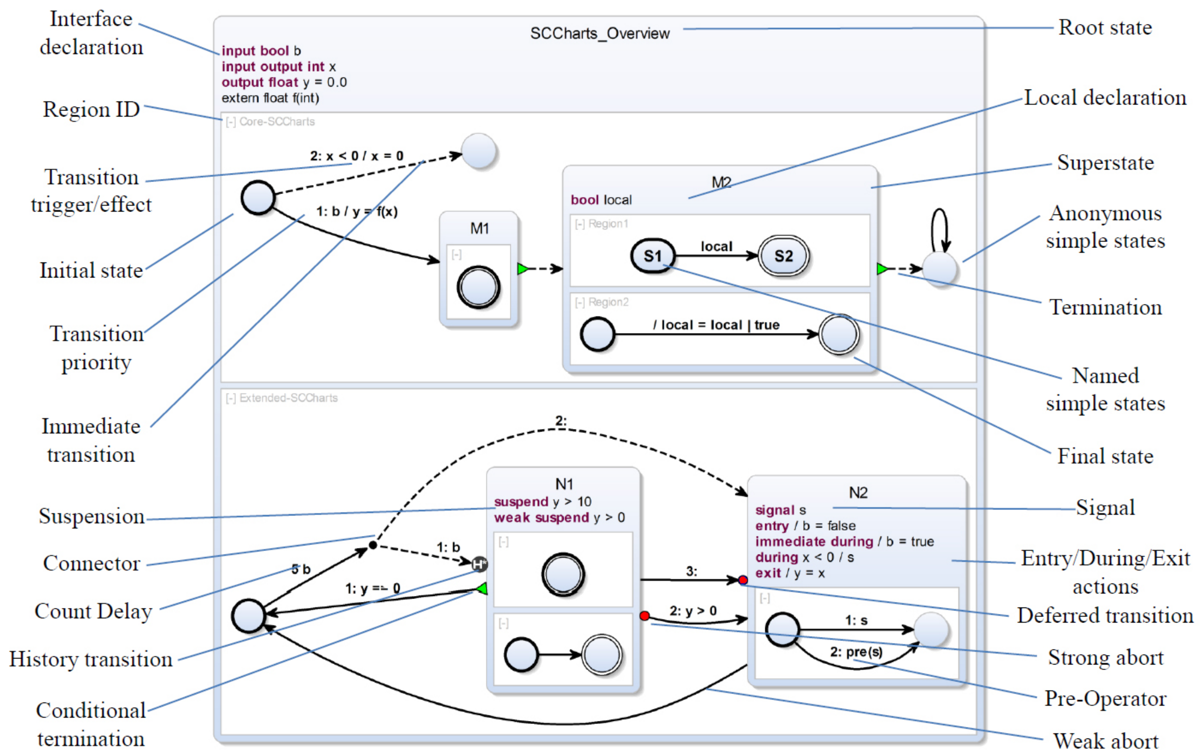


Abbildung 1.1. Core & Extended SCCharts - Übersicht

In der Tabelle 1.2 wird dargestellt, welche Befehle den Core SCCharts und welche den Extended SCCharts zugeordnet werden. Zu nennen ist hier auch, dass alle Core SCCharts Befehle in Extended SCCharts verwendet werden können. Im späteren Verlauf dieser Abhandlung wird dargestellt, welche der High-Level-Konstrukte für

[3]Git-Commit ea3e3ff7fd0 vom 24. Mai 2016

[4]http://rtsys.informatik.uni-kiel.de/confluence/display/KIELER/SCCharts

die Programmierung von eingebetteten Systemen mit sehr knappem Speicherplatz bevorzugt verwendet werden können oder vermieden werden sollten.

Core SCCharts	Extended SCCharts
Hierarchical States (Composite States)	Core SCCharts
Immediate Transitions	Complex Final State
Interface Declaration	Conditional Termination
Normal Termination	Connector (Choice)
Regions	Count Delay
Simple States	Deferred Transition
Transitions (Weak Abort)	During Action (Inner Action)
Variables (Primitive Types)	Entry Action
	Exit Action
	History Transition
	Initialization
	Pre
	Signal
	Strong Abort
	Suspension
	Weak Abort

Abbildung 1.2. Core & Extended SCCharts — Tabellarische Auflistung

Weitere semantik-erhaltende M2M-Transformationen erzeugen, in einer Kompilerkette zusammengeschaltet, aus einem Core SCChart den für diese Arbeit sehr wichtigen sequenzialisierten SCG. Bei dem seq. SCG handelt es sich um eine Art Kontrollflussgraph ohne Zyklen. Dieser stellt ein sequenziell ablauffähiges Modell dar und kann, aufgrund der fehlenden Nebenläufigkeit, einfacher optimiert werden als beispielsweise ein SCCharts-Modell. Die Abbildung 1.3 zeigt die möglichen Instruktionen eines SCG-Modells oder eines SCL-Programmes. SCL steht für Sequentially Constructive Language und ist die textuelle Darstellung eines SCGs.

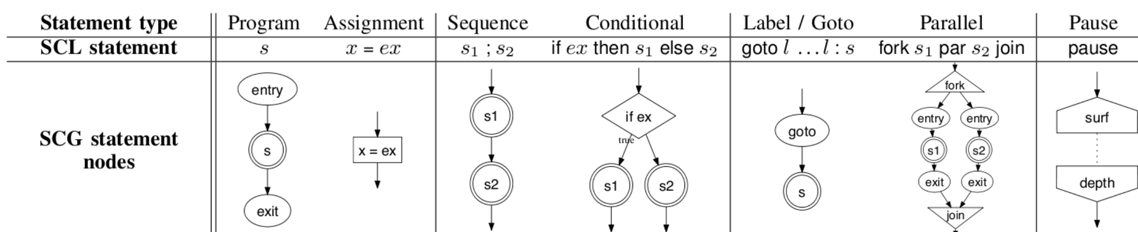


Abbildung 1.3. Übersicht SCG / SCL Instruktionen

Es ist zu erkennen, dass innerhalb eines SCGs sieben verschiedene Konstrukte verwendet werden können. SCG-Modelle enthalten immer einen Entry- und einen Exit-Knoten. Diese markieren Start und Ende eines Modells. Zuweisungen werden als rechteckige Knoten dargestellt. Die sequenzielle Ausführung von Programmteilen wird mittels runder Knoten mit gerichteten Kanten dargestellt. Für die nebenläufige Ausführung wird zusätzlich ein Fork-Join-Bereich benötigt. Programmsprünge werden durch ovale Knoten, sogenannte Bezeichner, gekennzeichnet. Um eine Operation einen Tick später auszuführen, wird die Pause verwendet.

1.2 Problemstellung

Die ausschlaggebende Motivation für diese Arbeit war die Variablenobergrenze der leJOS Java VM[5] auf den Lego Mindstorms[6] NXT Systemen. Das Lego Mindstorms NXT System, hergestellt von der Firma Lego[7] mit Hauptsitz in Billund (Dänemark), ist ein programmierbarer Roboter mit Sensorein- und Motorausgängen. Das NXT System ist ein Beispiel für ein eingebettetes System und kann mithilfe der leJOS[8]-Umgebung mit Java programmiert werden. Wird die Variablenobergrenze überschritten, so kann das Programm nicht auf das NXT System übertragen und ausgeführt werden. Die Begrenzung der Variablenanzahl auf 255 hängt mit der Implementierung der Java Virtual Machine (VM) in leJOS zusammen. Ziel der leJOS Java VM ist es möglichst wenig Speicher zu verbrauchen und einen schnellen Start der Programme sicherzustellen. Um dies zu gewährleisten wurden softwareseitig einige Limitierungen[9] implementiert. Zu diesen Limitierungen gehört auch die Begrenzung der Felder (MAX_FIELDS) eines Objektes auf 255.

SCCharts wurden an der CAU Kiel in einem Lehrmodul bei der Bearbeitung von Aufgaben eingesetzt. Einige dieser Aufgaben wurden auf dem NXT System umgesetzt. Hierbei wurde des öfteren die besprochene Variablenobergrenze erreicht, was eine Übertragung und Ausführung der Modelle auf dem NXT System fehlschlagen ließ. Ziel einer der Aufgaben war es einen Barcode-Scanner mit Hilfe der KIELER-Umgebung zu entwickeln und auf dem genannten System zu testen. Die zur Lösung dieses Problems erstellten Modelle werden von KIELER zu Java-Quelltext transformiert und im Anschluss mit der leJOS-Umgebung kompiliert. Bei recht simplen Modellen zur Lösung der Aufgabenstellung kam es bereits zu Platzproblemen auf dem NXT.

[5] https://sourceforge.net/projects/lejos/
[6] http://www.lego.com/en-us/mindstorms
[7] http://www.lego.com/de-de
[8] http://www.lejos.org
[9] https://sourceforge.net/p/lejos/wiki-nxt/Virtual%20Machine%20Issues/

Ein Modell, welches die Variablenobergrenze von 255 Variablen überschreitet, ist in Abbildung 1.4 zu sehen. Die Aufgabe des Modells ist das Lesen von Barcodes mithilfe von Sensoren und Motoren gesteuert durch ein NXT System. Der sequenzialisierte SCG dieses Modells beinhaltet 256 Variablen.

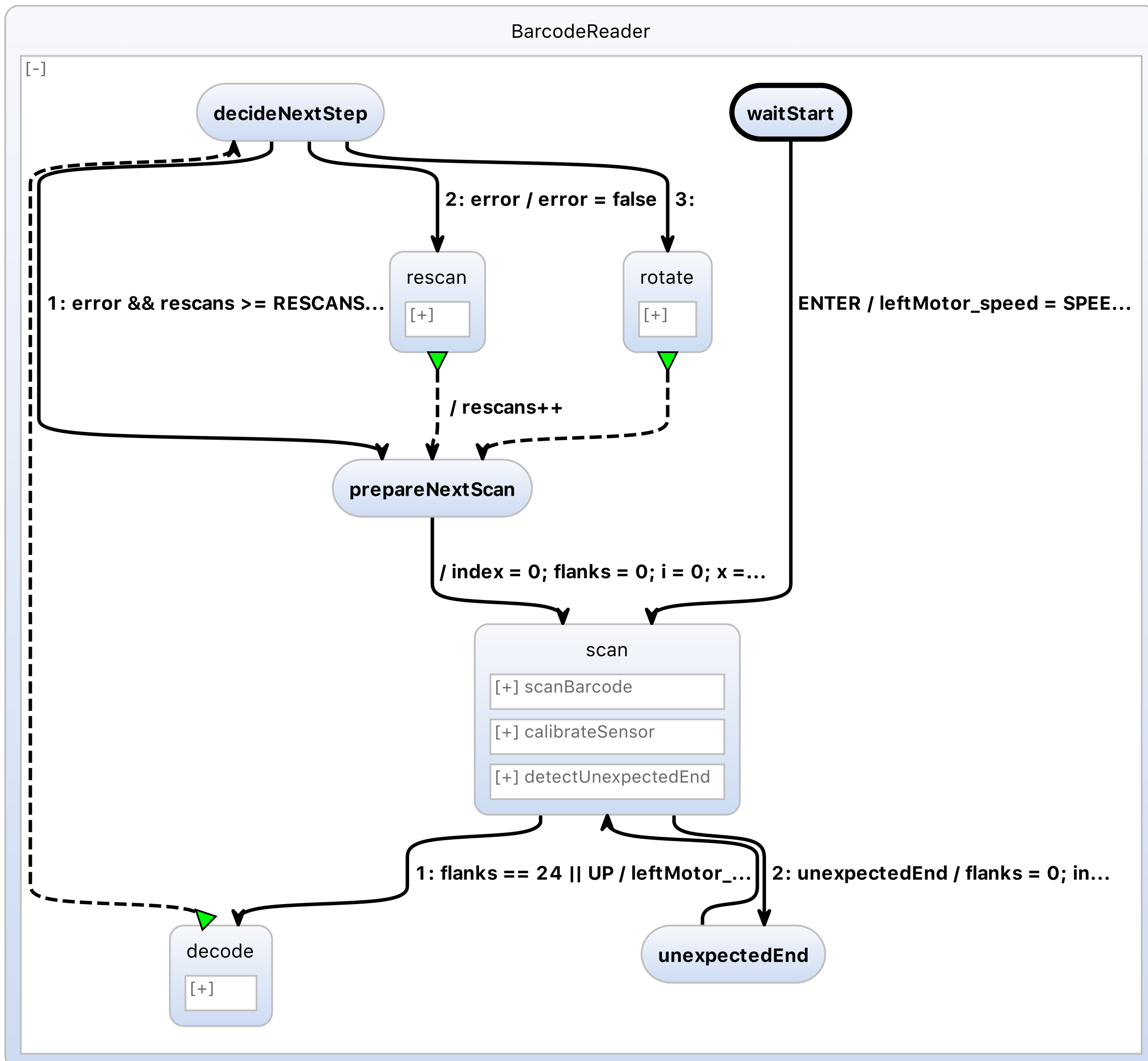


Abbildung 1.4. BarcodeReader Modell mit zusammengefassten Makrozuständen

Der BarcodeReader startet in dem Initialzustand waitStart. Nachdem mit ENTER der Start bestätigt wurde, wechselt das Modell in den Zustand scan. Dieser ist in drei nebenläufige Regionen unterteilt. Die Region scanBarcode hat die Erkennung des zu lesenden Barcodes zur Aufgabe und arbeitet eng mit der Region calibrateSensor

zusammen. Diese Region kalibriert während des Lesens den Sensor anhand der bereits gelesenen Werte. Der Erkennung fehlerhafter Barcodes widmet sich die Region `detectUnexpectedEnd`. Wird ein defekter Barcode erkannt, so wechselt der Automat in den Zustand `unexpectedEnd`. Dieser geht per Transition zurück in den `scan` Zustand und setzt eine Errorfahne. Sobald 24 Flanken, ein kompletter Barcode, gelesen wurden wechselt der Automat in den `decode` Zustand. Dieser dient der Dekodierung der gelesenen Hell-Dunkel-Werte des Barcodes in eine natürliche Zahl. Wenn der Barcode nicht dekodiert werden kann, so wird ebenfalls eine Errorfahne gesetzt. Der Zustand `decideNextStep` entscheidet anhand der Errorfahne und der Anzahl an Scanversuchen, des aktuellen Barcodes, welcher Zustand als nächstes durchgeführt wird. Liegt ein Fehler vor und ist die maximale Anzahl an Scanläufen überschritten, so wird in den Zustand `prepareNextScan` gewechselt. Liegt lediglich ein Fehler aber keine Überschreitung der max. Scanläufe vor, so wird in den Zustand `rescan` gewechselt. Dieser setzt den Scanvorgang zurück und wechselt danach ebenfalls in den Zustand `prepareNextScan`. Der letzte Fall deckt den korrekt gelesenen Barcode ab. Es wird um die Gradzahl, welche im Barcode kodiert ist, gedreht und in den Zustand `prepareNextScan` gewechselt. In `prepareNextScan` werden beim Wechsel in den Zustand `scan` alle für den Scan notwendigen Variablen zurückgesetzt. Ein neuer Scan beginnt.

Zum Ende dieser Ausarbeitung wird dieses Modell ein weiteres Mal aufgegriffen und mittels der entwickelten Optimierungen auf eine Variablenanzahl, welche für die leJOS Java VM auswertbar ist, reduziert.

Insgesamt konnten zwei Problemstellen bei Modellen wie dem BarcodeReader ausfindig gemacht werden. Programme, die mit Hilfe von SCCharts aus mittelgroßen Modellen generiert werden, können für kleine eingebettete Systeme (NXT) schnell zu groß sein, um übertragen oder ausgeführt werden zu können. Darüber hinaus gibt es Operationen auf höchster Modellebene, die den generierten Quellcode stark expandieren lassen.

Im Folgenden wird die Problemstellung anhand des ABO-Modells verdeutlicht. Das Modell (Abb. 1.5) hat zwei Eingänge, A und B, sowie einen Ausgang O. Nach dem Start des Modells wird bei dem sofortigen Übergang von init nach WaitAandB der Ausgang O auf false gesetzt. Wenn nun A oder B in einem der nachfolgenden Ticks als Eingabe true erhalten, so geht die jeweilige Region in ihren Endzustand. Sobald beide Regionen sich in ihrem Endzustand befinden sorgt eine sofortige Transition dafür, dass O auf true gesetzt wird und das gesamte Modell in den Endzustand done wechselt. Widmet man sich nun dem generierten Java-Code aus Abbildung 1.6 so fallen einige Besonderheiten auf. Zum einen wird g0 lediglich in der ersten Fallunterscheidung verwendet. Danach wird die Variable _GO direkt verwendet. Des Weiteren tauchen häufig Zuweisungen der Form gX = (pre_gY) auf. X und Y stehen für positive natürliche Zahlen. Diese Zuweisungen erleichtern das Verstehen des

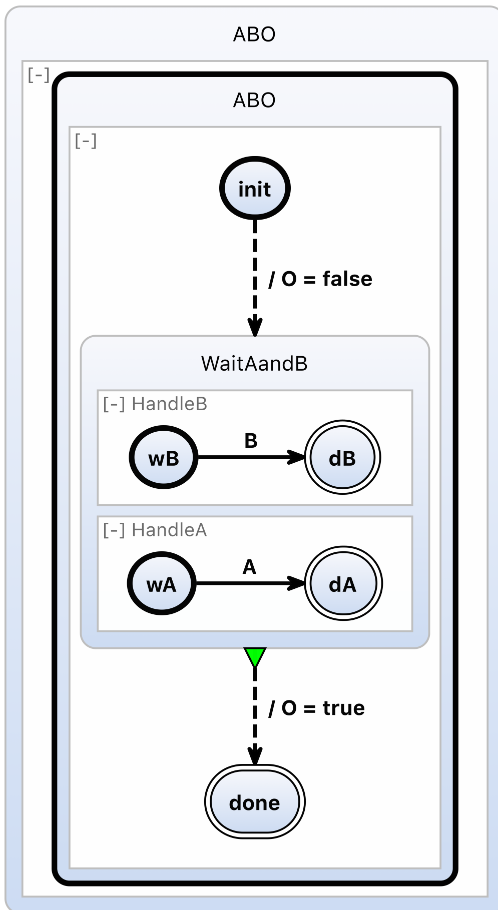


Abbildung 1.5. ABO Modell

```
1 public void tick() {
2  boolean g0;
3  boolean g2;
4  boolean g3;
5  boolean g4;
6  boolean g6;
7  boolean g7;
8  boolean g8;
9  boolean g9;
10 boolean g4_e1;
11 boolean g8_e2;
12 {
13     g0 = _GO;
14     if (g0) {
15         O = false;
16     }
17     g3 = PRE_g2;
18     g2 = _GO||(g3&&!A);
19     g4 = g3&&A;
20     g7 = PRE_g6;
21     g6 = _GO||(g7&&!B);
22     g8 = g7&&B;
23     g4_e1 = !g3;
24     g8_e2 = !g7;
25     g9 = (g4_e1||g4)&&(g8_e2||g8)&&(g4||g8);
26     if (g9) {
27         O = true;
28     }
29 }
30 PRE_g2 = g2;
31 PRE_g6 = g6;
32 _GO = false;
33 return;
34 }
```

Abbildung 1.6. ABO Java-Code

Quelltextes und können Rückschlüsse auf das zugrundeliegende Modell geben. Wird das M2M-Tracing von Schulz-Rosengarten [Sch14] verwendet, so kann ermittelt werden aus welchem Modellteil welche Codezeile generiert wurde. Für die Ausführung auf einem eingebetteten System hingegen sind die der Übersichtlichkeit dienenden Codezeilen nicht vonnöten. Es besteht Optimierungspotential.

1.3 Beitrag der Arbeit

Die Ausarbeitung trägt folgenden Lösungen für die beschriebene Problemstellung bei. Zum einen wird der Quellcode auf Optimierungspotential in Hinsicht auf Reduktion der Codegröße und Variablenanzahl untersucht. Im Nachgang werden diese Optimie-

rungen in KIELER implementiert und nutzbar gemacht. Diese Optimierungen werden in KIELER als zusätzliche Transformationen eingebunden. Dies hat zum Vorteil, dass der Benutzer selbstständig bestimmen kann, welche Optimierungen verwendet werden sollen. Ebenfalls ordnen sich so die Optimierungen programmatisch als auch optisch in die später in Kapitel 3.1.2 erläuterte Kompilerkette ein. Zum anderen werden die High-Level-Konstrukte auf die Verwendbarkeit in ressourcenbegrenzten eingebetteten Systemen untersucht.

Darüber hinaus bietet diese Arbeit einen Einblick in die Optimierung mittels Konstanten- und Kopierpropagation auf sequenzialisierten SCGs. Ebenfalls wird die Idee der Registerwiederverwendung auf die Variablenwiederverwendung übertragen. Aufgezeigt wird auch, wie effizient die Optimierungen arbeiten und ob neben der Reduzierung der Quellcodegröße auch ein Speedup in der Ausführung des Quellcodes als C-Programm erreicht werden kann. Neben der Auflistung der Expansionsgröße der Extended SCCharts Befehle zu ihren semantisch äquivalenten SCGs wird auch eine Verwendungsempfehlung der Ext. SCCharts Konstrukte im Kontext der eingebetteten Systeme gegeben.

1.4 Aufbau

Die Arbeit handelt zuerst in Kapitel 2 verwandte Arbeiten im Bereich der Kompilierreoptimierung (Teilkapitel 2.1) und Hardwaresynthese, in Teilkapitel 2.2, ab. Die Hardwaresynthese ist für die Implementierung interessant, da es hier zu Problemen mit einigen Optimierungsoptionen aus der Kompileroptimierung kommen kann. Anschließend werden die verwendeten Technologien in Kapitel 3 erläutert. Vorgestellt werden die relevanten Teilmodule, wie SCCharts, Kieler Compiler, SLIC, TestRunner, KIELER Execution Manager, der KIELER-Umgebung und das statische Analysetool SonarQube, welches für die Erstanalyse des generierten Java-Codes verwendet wird. Darauf folgt das Konzept in Kapitel 4 dieser Ausarbeitung. Es wird dargelegt, wie die Phasen geplant und durchgeführt werden. Zudem wird die Motivation für die Teilbereiche erläutert.

Im Anschluss an die Konzeptionierung wird die Implementierung der Quellcodeoptimierung in Kapitel 5 vorgestellt und in ihren Einzelheiten erklärt. Hier werden sowohl Themen aus bereits vorgestellten Arbeiten aus Kapitel 2 (Verwandte Arbeiten), als auch neue Ansätze vorgestellt und umgesetzt. Stichwörter sind Kopierpropagation, Variablenwiederverwendung und Eliminierung toten Codes. Angeschlossen daran folgt die Evaluation der Quellcodeoptimierung in Kapitel 6. Zuerst werden ausgewählte Beispiele vorgestellt und auf die Effektivität der Optimierungen untersucht. Darauf folgend werden quantitative Auswertungen über real verwendete

Modelle vorgestellt. Hierbei wird auf die Größe vor und nach den Optimierungen, als auch auf die Ausführungsgeschwindigkeiten der Modelle eingegangen. Im Anschluss soll eine große Menge an automatisch generierten Modellen untersucht werden. Auf diesen Modellen wird lediglich eine Evaluation über die Modellgrößen vor und nach den Optimierungen angefertigt. Abgeschlossen wird die Evaluation mit der Analyse der High-Level-Transformationen in Hinsicht auf die Verwendbarkeit in kleinen eingebetteten Systemen mit begrenzten Ressourcen. Die Arbeit wird in Kapitel 7 mit einem Fazit sowie einem Ausblick geschlossen.

Kapitel 2

Verwandte Arbeiten

Dieses Kapitel umfasst die für diese These essentiellen wissenschaftlichen Arbeiten im Umfeld der Kompilereroptimierung sowie die Hardwaresynthese als auch die Optimierung von Fallunterscheidungen in KIELER. Beginnend mit den Kompileroptimierungen wie Konstanten- und Kopierpropagation, Registerwiederverwendung, Eliminierung von nicht erreichbaren oder nicht verwendeten Codes und der Optimierung von Fallunterscheidungen werden wichtige Themen und Herangehensweisen zuerst abstrakt erläutert und dann an Beispielen verdeutlicht. Der zweite Teil dieses Kapitels beschränkt sich dann auf die Hardwaresynthese, da diese im Konflikt mit einer der späteren Optimierungen steht.

2.1 Kompileroptimierung

Das Thema der Kompileroptimierung ist seit längerem ein Forschungsgebiet der Informatik. Die Zielsetzung ist häufig die Verkleinerung oder Beschleunigung des Maschinencodes, welcher aus dem Quelltext generiert wird. Für die Ausarbeitung interessant ist der Part der Verkleinerung. Ein Speedup der Ausführung ist allerdings ein angenehmer Nebeneffekt zu der Reduktion. Bungo [Bun08] hat sich hierzu mit verschiedenen Kombinationen der verfügbaren Kompileroptionen eines ARM-Kompilers angesehen. Die Kombinationen wurden auf die Größe und Ausführungsgeschwindigkeit des generierten Kompilats untersucht. Schneck [Sch73] hat in seinem Artikel die Kompileroptimierungen der Konstantenpropagation (Kapitel 2.1.1) und der Eliminierung toten Codes (Kapitel 2.1.4) zusammengefasst. Ziel des Artikels ist es die Kompileroptimierungen für Parallel- als auch Vektorprozessoren vorzustellen und exemplarisch nutzbar zu machen.

Allgemeine Themen der Kompileroptimierung handelt das Buch von Aho et al. [ASU86], auch Dragonbook genannt, ab. Angefangen mit der Struktur von Kompilern und Sprachprozessoren über einfache Übersetzer bis hinzu komplexen Analysen auf Quellcode werden Techniken und Prinzipien im Bereich des Kompilerbaus als auch der Kompileroptimierung erklärt. Ebenfalls werden maschinenunabhängige Analysen wie die Konstanten- und Kopierpropagation besprochen.

Die weiteren Quellen zur Kompileroptimierung werden in den nachfolgenden Unterkapiteln strukturiert dargestellt.

2.1.1 Konstantenpropagation

Bei der sogenannten Konstantenpropagation werden konstante Werte zur Kompilierzeit an ihrem Verwendungspunkt eingesetzt. Es erfolgt eine Ersetzung der verwendeten Variable mit dem konstanten Wert der Variable. Sollte danach ein auswertbarer Term entstehen, so wird dieser ausgewertet und an Stelle des Terms eingefügt. Das Papier von Kildall [Kil73] stellt einen einheitlichen Ansatz der Programmoptimierung mittels Konstantenpropagation vor. Hierzu wurde ein Algorithmus auf Kontrollflussgraphen definiert. Anders als der von Aho et al. [ASU70] vorgestellte Algorithmus, welcher lediglich auf sequenziell ablaufendem Code arbeitet, kann der Algorithmus von Kildall Programmcode global optimieren ohne auf die sequenzielle Eigenschaft zurückzugreifen. Im Hinblick auf die Codekomprimierung haben sich Debray et al. [DEM+00] näher mit der Konstantenpropagation auseinander gesetzt. Eine Erkenntnis war, dass die Konstantenpropagation fundamental für ein Kompilersystem ist, da viele Kontroll- und Datenflussanalysen auf dem Wissen der im Programm errechneten konstanten Adressen beruhen.

Konstantenpropagation auf konditionalen Verzweigungen wird in der Veröffentlichung von Wegman und Zadeck [WZ91] betrachtet. Das Papier liefert Algorithmen der Konstantenpropagation im konservativen Stil. Die vorgestellten Algorithmen erkennen somit nicht alle konstanten Werte, allerdings alle konstanten Werte die über die gesamte Ausführungszeit konstant sind. Zusätzlich wird auf die Eliminierung nicht verwendeten Codes, der nach der Konstantenpropagation auftreten kann, gesprochen. Eine genauere Ausführung zu der Eliminierung nicht verwendeten Codes folgt im Verlauf dieses Kapitels.

Das Beispiel in Abbildung 2.1 zeigt die Anwendung der Konstantenpropagation auf Pseudocode.

Original	Konstantenpropagation	Auswertung
`x := 3`	`x := 3`	`x := 3`
`y := 4`	`y := 4`	`y := 4`
`z := x*y`	`z := 3*4`	`z := 12`

Abbildung 2.1. Konstantenpropagation - Beispiel [Gue99, S. 262]

In dem Beispiel ist zu sehen, wie zuerst auf dem originalen Pseudocode die Konstantenpropagation und später die Auswertung der neu entstandenen Terme

angewendet wird. Zudem ist zu sehen, dass die nun überflüssigen Variablen x und y weiterhin im Programmcode enthalten sind. Die Entfernung überflüssiger Variablen wird durch die Eliminierung ungenutzten Codes durchgeführt.

Nachteil einer auf Quellcode durchgeführten Konstantenpropagation ist die Entstehung von sogenannten magischen Zahlen. Es handelt sich hierbei um konstante Werte, welche ohne Bedeutung (Variablenname), auftauchen. Deshalb wird die Konstantenpropagation üblicherweise während der Kompilierung in Richtung Maschinencode verwendet. Die Konstantenpropagation wird in dieser These in der gleichen Transformation wie die Kopierpropagation implementiert.

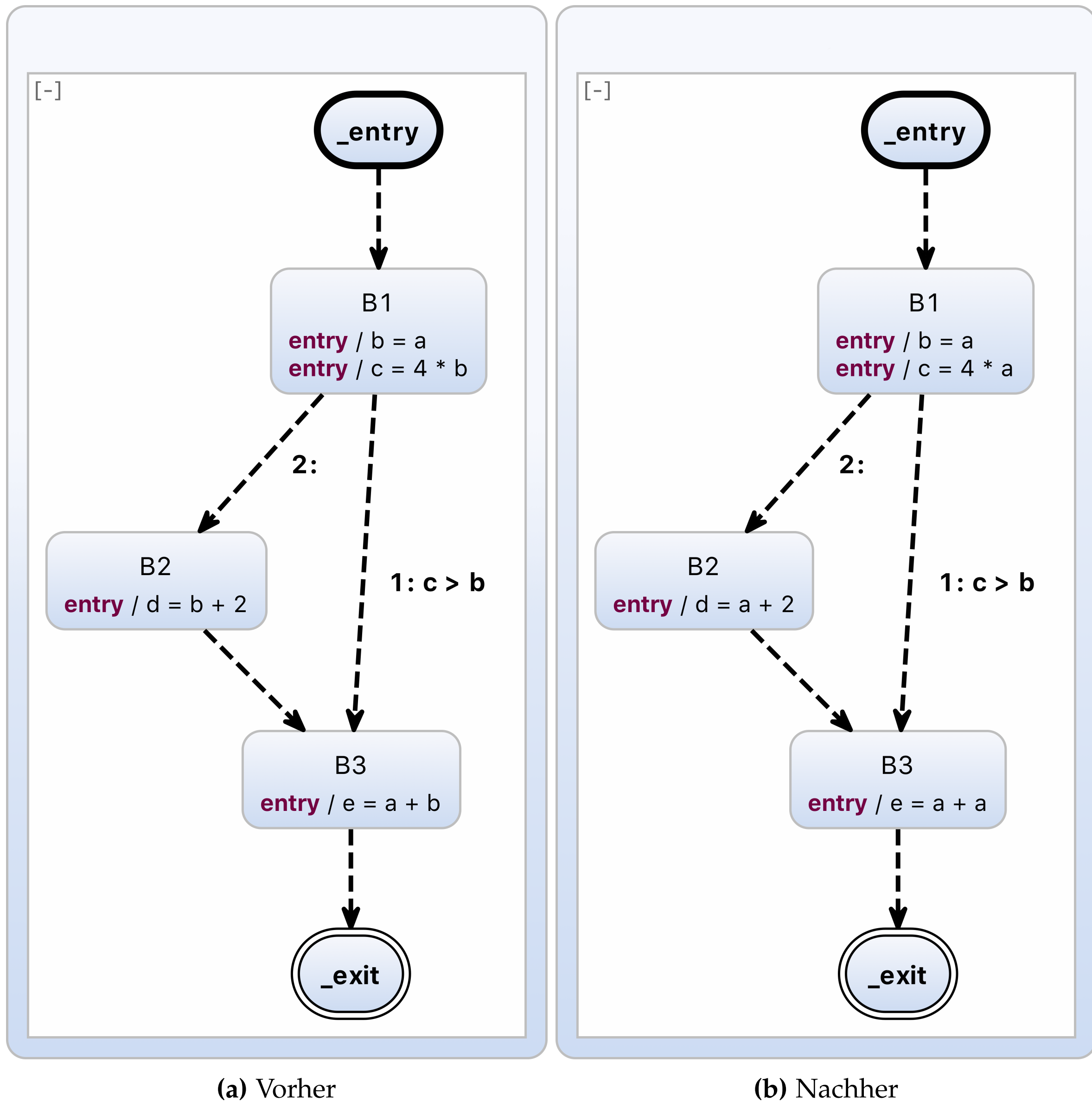
2.1.2 Kopierpropagation

Neben dem in der Einleitung genannten Buch von Aho et al. [ASU86] beschreibt auch das Buch von Muchnick [Muc97] die Kopierpropagation. Es werden Algorithmen der Kopierpropagation vorgestellt und Beispiele für erkennbare und nicht erkennbare kopierbare Strukturen entwickelt. Bereits 1986 hat Kikuchi ein Patent [Kik89] zum Thema Kopierpropagation eingereicht. Beschrieben wird sowohl die Propagation von direkt zugewiesenen Variablen als auch die Propagation von Termen. Die Propagation von ganzen Termen wird im Konzeptkapitel dieser Ausarbeitung wieder aufgegriffen. Im Vorfeld haben bereits Aho et al. [ASU70] auch die Kopierpropagation auf sequenziellem Quellcode vorgestellt.

Um die Kopierpropagation zu verdeutlichen werden in Abbildung 2.2 ein Positiv- und in Abbildung 2.3 ein Negativbeispiel dargestellt.

Das Positivbeispiel zeigt in (a) den Zustand vor und in (b) nach der Kopierpropagation. Das dargestellte Programm setzt zu Beginn in B1 `b = a` und `c = 4 * b`. Anschließend folgt eine Fallunterscheidung mit `c > b`. Wertet die Fallunterscheidung zu wahr aus, so wird die Operation B3 mit `e = a + b` ausgeführt. Bei einer Auswertung zu falsch, wird vor der Operation in B3 noch die Operation B2 mit `d = b + 2` ausgeführt. Das Beispiel propagiert nun die Zuweisung `b = a`. Es werden somit vorkommen von `b` durch `a` ersetzt. Das Ergebnis der Ersetzung ist in (b) zu sehen. Die Semantik des Programms hat sich nicht geändert.

Neben der Propagation von lediglich initial zugewiesenen Variablen ist die Propagation von sequenziell geänderten Variablen ebenfalls möglich. Angenommen eine Variable X wird mit Y initialisiert (X = Y). Darauffolgend wird der Wert von X der Variable Z zugewiesen (Z = X). Nach dieser Zuweisung wird der Wert von X auf K geändert (X = K). Wird nun die angepasste Kopierpropagation ausgeführt, so wird in der Zuweisung Z = X die Variable X durch Y ersetzt. Nachdem die Zuweisung X = K sequenziell gilt, wird für die Variable X die Variable K propagiert.



(a) Vorher **(b)** Nachher

Abbildung 2.2. Positivbeispiel Kopierpropagation nach Muchnick [Muc97, S. 357], dargestellt als SCCharts

Eine nicht durch Kopierpropagation erkennbare Zuweisungen ist in dem Negativbeispiel dargestellt. Das Programm führt zu Beginn eine Fallunterscheidung auf der Variable z durch. Ist z > 0 wahr, so wird B2 ausgeführt. Wird die Fallunterscheidung zu falsch ausgewertet, so wird B3 bearbeitet. B2 und B3 enthalten den gleichen Code und haben somit die gleiche Funktionalität. Im Anschluss an B2 oder B3 wird B4

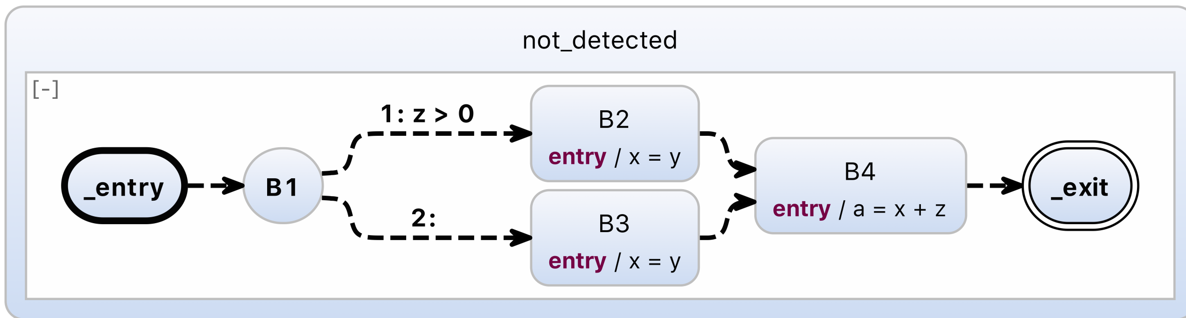


Abbildung 2.3. Negativbeispiel Kopierpropagation nach Muchnick [Muc97, S. 362], dargestellt in SCCharts

angesteuert. B4 wertet a = x + z aus. Danach terminiert das Programm. Von einer Kopierpropagation würde nun erwartet werden, dass in B4 das Vorkommen von x durch y ersetzt würde. Nun können allerdings keine zu propagierenden Variablen in Zweigen von Fallunterscheidungen gefunden werden, da zur Optimierungszeit nicht bekannt ist, welcher Zweig ausgeführt wird. Erst eine vorher ausgeführte Redundanzentfernung, also das Entfernen gleichen Quellcodes in Zweigen einer Fallunterscheidung (in dem genannten Beispiel B2 und B3), ermöglicht es der Kopierpropagation zu erkennen, dass x = y ein propagierfähiger Term ist. Ein Nebeneffekt der Redundanzentfernung ist zudem, dass einige Fallunterscheidungen komplett durch den Code eines Zweiges ersetzt werden können.

Die Kopierpropagation hat ähnliche Vor- und Nachteile wie die Konstantenpropagation. Zusammengefasst ist dies eine generelle Quellcodereduktion und Verminderung der Variablenanzahl auf Kosten von Kontextinformationen wie Variablennamen. Eine Implementierung der einfachen Kopierpropagation findet in dieser Arbeit statt. Die Erweiterung auf sequenziell veränderte Variablen wird nicht implementiert, da im sequenzialisierten SCG sequenziell aufeinander folgende Zuweisungen der gleichen Variable nicht stattfinden.

2.1.3 Registerwiederverwendung

Häufig ist das Wiederverwenden von Registern auf Assembler-Ebene ein Ansatz um den Maschinencode zu verkleinern, die Ausführungsgeschwindigkeit zu erhöhen und mit der beschränkten Registerzahl zu arbeiten. Das Papier von Goodman et al. [GH88] betrachtet die Kompileroptimierung in Hinsicht auf die Reduzierung von Pipelineverzögerungen und die Minimierung von Registerverwendungen. Eine gute Beschreibung, wie die Anordnung von Programmcode und die somit gesparten Register zusammenhängen, bietet das Buch von Güting und Erwig [Gue99]. Um zu

erkennen, ob ein Register wiederverwendet werden kann, muss zunächst geprüft werden, welche Register sequenziell an welchem Programmpunkt beschrieben und zuletzt verwendet werden. Nach der letzten Verwendung, dies kann ein Lese- oder Schreibzugriff sein, kann das Register als nicht mehr verwendet angesehen werden und ist somit in einem anderen Kontext erneut beschreibbar. Abbildung 2.4 verdeutlicht die Methode anhand eines Beispiels.

Es ist zu sehen, dass Register 3 nicht mehr genutzt wird und Register 1 an die Stelle tritt. Dies ist möglich, da Register 1 zum letzten Mal vor der Zuweisung der Addition der Werte aus Register 1 und 2 verwendet wird. Register 1 wird somit nach der Additionsoperation nicht erneut im alten Kontext verwendet und die Wiederverwendung ist valide.

```
Reg1 := 1
Reg2 := 2
Reg3 := Reg1 + Reg2
```

Original

```
Reg1 := 1
Reg2 := 2
Reg1 := Reg1 + Reg2
```

Wiederverwendung

Abbildung 2.4. Registerwiederverwendung - Beispiel

2.1.4 Eliminierung toten / nicht erreichbaren Codes

Bei der Eliminierung von 'totem' oder unerreichbarem Code geht es, wie der Name bereits vermuten lässt, um das Entfernen von nicht vom Programmfluss durchlaufenen Codestellen. Dies können Zeilen nach dem Aufrufen des Rückgabebefehls einer Funktion sein oder auch Fallunterscheidungen bei denen der Else-Zweig nicht erreicht werden kann, da zur Kompilierzeit bereits bekannt ist, dass die If-Bedingung immer zu wahr ausgewertet wird. Das Buch von Srikant und Shankar [SS07] befasst sich ebenfalls mit der Definition toten Codes und ihrer Entfernung. Es wird dargelegt, dass der ursprüngliche Quellcode unerreichbare Codestellen haben kann. Die Wahrscheinlichkeit, dass durch andere Optimierungen toter Code entsteht, sei allerdings höher. Ebenfalls wird aufgegriffen, dass anderes als bei der Eliminierung nicht verwendeten Codes bei der Eliminierung nicht erreichbaren Codes keine Berechnung der Codestellen erfolgt. Somit kann lediglich die Größe des Kompilats verringert werden, allerdings kein Speedup der Ausführung auftreten. Ein unerreichbarer Codeblock der in von Hand geschriebenen Programmen auftreten kann, wird in Abbildung 2.5 gezeigt.

```
const debug := false
if debug then begin // unerreichbar, da debug = false
        writeln(...)
        [...]
end
```

Abbildung 2.5. Eliminierung toten Codes - Beispiel [Gue99, S. 266]

Der Programmierer will mit der Konstanten debug die Debug-Ausgaben des Programmes regeln. Sobald der Code allerdings mit dem Wert debug = false kompiliert wird, wird ein solcher Debugausgabeblock niemals ausgeführt werden. Bei dem Beispiel ist klar, dass zum Zeitpunkt der Kompilierung die if-Bedingung zu falsch ausgewertet wird. Somit kann der komplette if-Block als nichtexistent betrachtet werden. Der Block wird somit nicht mit kompiliert.

Das Beispiel in Abbildung 2.6 zeigt nicht verwendeten Code, der erst durch eine andere Optimierung erkennbar wird. Der dargestellte Quelltext zeigt eine einstellige Funktion `dosomething`. Diese setzt zu Beginn die Variable `x` auf den Wert `true`. Danach wird das Ergebnis der boolschen Termes `x && y` der Variablen `z` zugewiesen. Die Bedingung der Fallunterscheidung wertet daraufhin den Term `z || (z || y)` aus. Wird zu wahr ausgewertet so wird `y` zurückgegeben, andernfalls `x`. Weiterhin ist zu erkennen, dass der Term der Bedingung der Fallunterscheidung immer zu wahr auswertet. Eine boolesche Optimierung, wie sie ebenfalls in der Abbildung zu erkennen ist, ersetzt den Term der Bedingung durch den statischen Wert `true`. Die Eliminierung nicht erreichbaren Codes entfernt im nächsten Schritt die 'toten' Codestellen.

Die Eliminierung nicht erreichbaren Codes wird dieser Ausarbeitung nicht implementiert.

```
fun dosomething(y) {
  x := true
  z := x && y
  if (z || (!z || y)) {
    return y
  }
  return x
}
```

Original

```
fun dosomething(y) {
  x := true
  z := x && y
  if (true) {
    return y
  }
  return x
}
```

Boolesche Optimierung

```
fun dosomething(y) {
 x := true
  z := x && y
  if (true) {
    return y
  }

}
```

Dead Code Elimination

Abbildung 2.6. Nicht verwendeter Code durch Optimierung

2.1.5 Eliminierung nicht verwendeten Codes

Wie im vorhergehenden Kapitel bereits beschrieben, haben sich Srikant und Shankar [SS07] in ihrem Buch mit der Eliminierung 'toten' Codes beschäftigt. Das Entfernen von nicht verwendeten Codes ist eine Sonderform des 'toten' Codes. Die Besonderheit liegt darin, dass der Code im Programmverlauf erreichbar ist und ausgeführt wird, aber keinen Effekt auf das Ergebnis hat. Ebenfalls wird angesprochen, dass durch die Entfernung von nicht verwendeten Code nicht nur die Kompilatgröße verringert wird, sondern auch ein Speedup der Ausführung erzielt werden kann, da die entfernten Codestellen nicht mehr ausgewertet werden müssen. Je nach Komplexität der Codestellen kann dieser Speedup signifikant sein. Ein Beispiel für eine Codestelle hohen zeitlichen Aufwands ist eine Datenbankabfrage gegen ein entferntes System. Sowohl die Netzwerkkommunikation als auch der Query kosten Zeit. Bei ressourcenbegrenzten eingebetteten System können allerdings schon vermiedene Schreibzugriffe auf den Hauptspeicher einen großen Unterschied machen. Der Artikel von Midkiff [MP90] betrachtet die Eliminierung toten Codes in parallelen Programmen. Bei parallelen Programmen ist es wichtig Strukturen, die von mehreren Teilprozessen oder Threads verwendet werden, korrekt zu betrachten. Programmstellen, die durch die Eliminierung toten Codes entfernt werden würden, könnten von anderen Threads benötigt werden, da nur so bestimmte Daten in den Speicher geschrieben werden. Ein Beispiel dieser Besonderheit folgt im Verlauf dieses Unterkapitels.

Um nicht verwendeten Code zu entfernen muss zuerst erkannt werden, wann Code nicht verwendet wird. Dies kann ähnlich zu der Eliminierung von nicht erreichbaren Codes erfolgen. Es können somit nicht verwendete Codeblöcke erkannt werden. Funktionen die nicht verwendet werden können dadurch erkannt werden, dass sie nicht aufgerufen werden. Hierbei muss allerdings beachtet werden, dass die Programmiersprache keine dynamischen Funktionsaufrufe unterstützt. In der Sprache C können beispielsweise Funktionspointer für den Zugriff auf Codestellen verwendet werden. Die Verwendung von Funktionspointern wird durch Kerninghan und Ritchie in ihrem Buch [Ker88] neben anderen Spracheigenschaften detailliert beschrieben. In Java und C# sind wird dieses Konzept durch sogenannte Reflections umgesetzt. Wie trotz Reflections eine statische Analyse, so wie sie auch bei Eliminierung nicht verwendeten Codes notwendig ist, möglich ist, beschreiben Livshits et al. in ihrem Buch [LWL05]. Ihr Ansatz ist es einen Aufrufgraphen mittels potentiell aufrufbaren Methoden eines bereits referenzierten Objekts zu erstellen. Kann so die dazugehörige Klasse oder Methode nicht oder nicht vollständig aufgelöst werden, muss der Benutzer eingreifen und den Aufruf genauer spezifizieren. SCCharts erbt mit der Möglichkeit der Hostcodeaufrufe die genannten Schwierigkeiten der statischen Analyse. Da Hostcodeaufrufe von den in dieser Arbeit getätigten Implementierungen

allerdings nicht verändert werden, können diese Probleme ignoriert werden.

Ähnlich ist es bei der Erkennung von Variablen ohne Verwendung. Grob gesehen werden alle Variablen, deren Wert nicht gelesen wird, nicht verwendet. Bei näherer Betrachtung müssen ein paar Unterscheidungen gemacht werden. Sind Variablen als 'jederzeit veränderbar' (volatile) gekennzeichnet, so kann ihr Wert über den aktuellen Programmcode hinweg verändert und somit auch gelesen werden. Dies kann durch Nebenläufigkeit mit geteiltem Speicher erzielt werden. Ein klassisches Beispiel für einen nur lesenden und einen nur schreibenden Programmcode stellt das Consumer-Producer-Problem dar. Hierbei produziert der eine Thread oder Prozess Daten die von einem anderen Thread oder Prozess gelesen werden können, da beide Programme auf den gleichen Speicherbereich zugreifen. An diesen Stellen wäre es fatal wenn eine Optimierung die schreibenden oder lesenden Zugriffe auf den Speicherbereich entfernen würde. Ähnlich verhält es sich bei Programmiersprachen die dynamische Variablen unterstützen. Hierzu zählt beispielsweise die Skriptsprache PHP. Schlossnagle beschreibt in seinem Lehrbuch [Sch06] sowohl wie dynamisch Klassenvariablen angelegt als auch per Reflection mit einem String als Bezeichner (ähnlich wie in Java oder C#) angesprochen werden können. Dynamische Variablen machen die statische Analyse schwierig.

Wird zudem das in Kapitel 2.1.4 erwähnte Beispiel in Abbildung 2.6 mit der Eliminierung nicht verwendeten Codes optimiert, so ist Abbildung 2.7 das Ergebnis. Es ist zu erkennen, dass die Variable z entfernt wurde. Grund dafür ist der aus der Bedingung der Fallunterscheidung entfernte Lesezugriff. Die Variable x bleibt erhalten, da mit return x ein lesender Zugriff auf die Variable besteht, auch wenn dieser Codeabschnitt nie erreicht werden kann.

```
fun dosomething(y) {
  x = true
  if (true) {
    return y
  }
  return x
}
```

Abbildung 2.7. Nicht verwendeter Code durch Optimierung

Ein Beispiel mit Bezug auf eine andere Optimierung kann in dem vorhergehenden Kapitel 2.1.2 zur Kopierpropagation beobachtet werden. In der Abbildung 2.2 kann nach der Kopierpropagation die Variable x entfernt werden, sodass lediglich y und z als Variablen verbleiben. Erst durch diese Eliminierung kann die Länge des

Quelltextes nach einer Konstanten- oder Kopierpropagation verringert werden. Die Ausarbeitung wird die Eliminierung nicht verwendeten Codes implementieren.

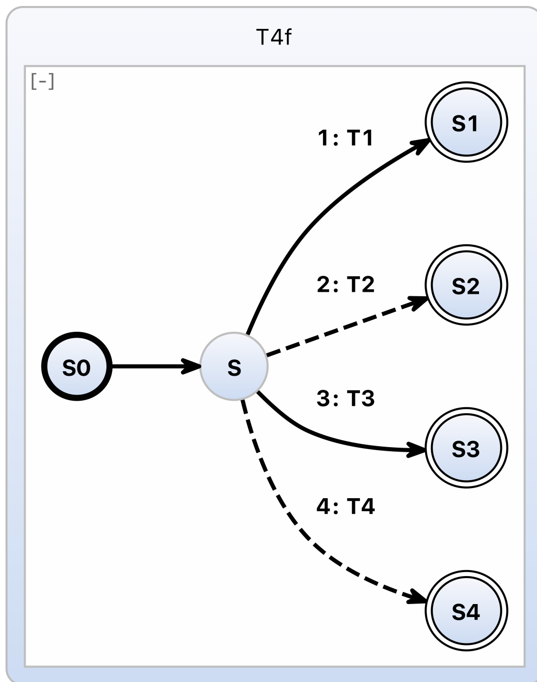
2.1.6 Fallunterscheidungen optimieren

In SCCharts werden Transitionen mit Trigger in Fallunterscheidungen umgewandelt und ihrer Priorität nach angeordnet. Wenn nun sowohl sofortige (immediate) als auch normale (delayed) Transitionen verwendet werden, kann es vorkommen, dass Transitionsbedingungen doppelt geprüft werden müssen. Eine Darstellung dieser Doppelung ist in Abbildung 2.8 (b) zu erkennen. Die doppelte Überprüfung von Bedingungen kostet nicht nur Ausführungszeit, sondern führt auch zu einem längeren Quelltext, der unter Umständen auch zu einem größeren Kompilat führt.

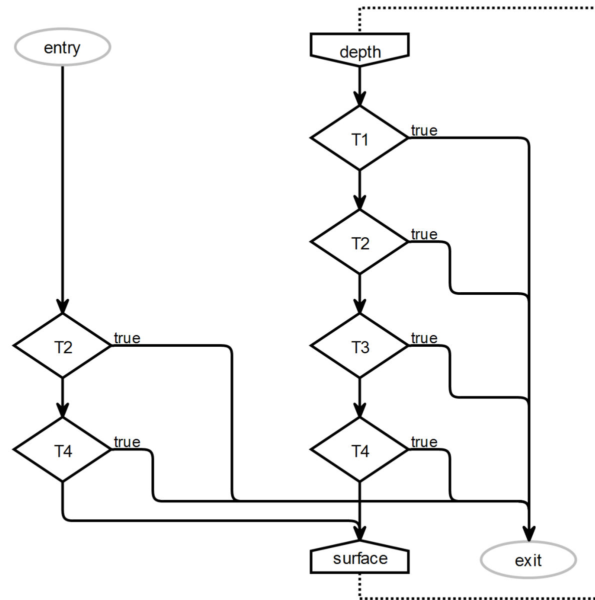
Einige Lösungen dieses Problems werden in der Diplomarbeit von Smyth [Smy13] bereits vorgestellt. Die verschiedenen Herangehensweisen werden in Abbildung 2.8 dargestellt. Abgebildet sind vier Modelle, ein SCCharts-Modell und drei SCG-Modelle. In (a) ist das Ausgangsmodell als SCCharts-Modell aufgeführt. Das Modell startet im Zustand `S0`. Nach einem Tick wird in den Zustand `S` gewechselt. Wenn bereits in dem Tick, in dem zu `S` gewechselt wurde, `T2` oder `T4` den Wert wahr angenommen haben, so wechselt der Automat in den Endzustand `S2` bzw. `S4`. Dies hängt mit der Semantik der sofortigen Transitionen zusammen. Erst im nächsten Tick werden auch die Transitionsbedingungen der normalen Transitionen, in diesem Fall `T1` und `T3`, mit geprüft. Die Überprüfung erfolgt dann in der Reihenfolge der Prioritäten. Die anderen drei Modelle stellen Übersetzungen von (a) in SCG-Modelle vor. Die in (b) gezeigte Übersetzung wird als naive Übersetzung bezeichnet. Zu sehen ist, dass zuerst die Transitionensbedingungen `T2` und `T4` der sofortigen Transitionen von `S` nach `S2` bzw. `S4` als Fallunterscheidung im SCG dargestellt werden. Die Semantik, dass sofortige Transitionen vor normalen Transitionen geprüft werden, bleibt somit bestehen. Den Prioritäten ist es geschuldet, dass `T2` vor `T4` eingeplant wird.

Sowohl (c) als auch (d) sind Optimierungen der Übersetzung aus (b). Die Duplicate Transition Optimization (DTO) Übersetzung zeigt, dass die zweite Fallunterscheidung über `T4` entfernt wurde. Der Knoten der Fallunterscheidung über `T3` verweist nun auf die erste Prüfung von `T4`.

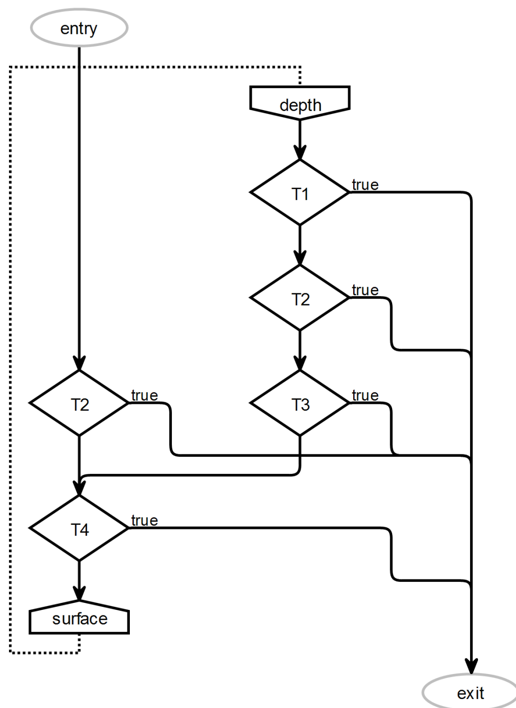
Dies ist möglich, da die Transition mit Trigger `T4` die niedrigste Priorität hat und somit sowohl bei den sofortigen als auch normalen Transitionen als letzte Überprüfung eingeplant wird. Mittels der Write-Thing-Once Transition Optimization (WTOTO) Übersetzung aus (d) ist es möglich mit nur vier Fallunterscheidungen zu arbeiten. Hierzu wird die Variable `dep` eingeführt. Diese dient als Indikator, ob für den Zustand `S` der Initialtick oder einer der Nachfolgeticks gilt. Es ist zu sehen, dass `dep` zuerst den Wert falsch annimmt.



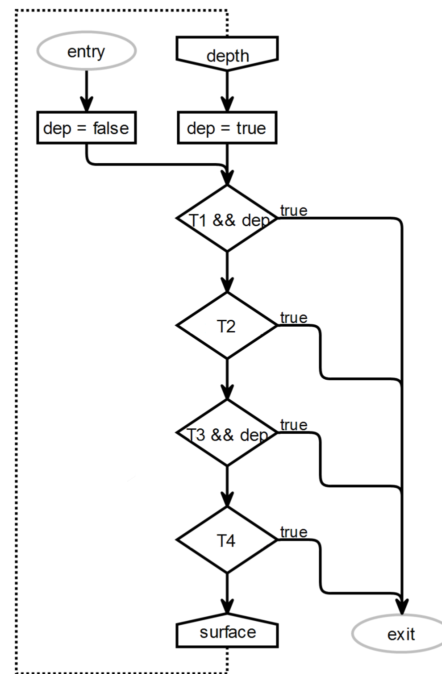
(a) T4f SCChart



(b) Naive Translation



(c) DTO Translation



(d) WTOTO Translation

Abbildung 2.8. T4f Transitionsübersetzung Beispiel - Codegenerierung für sequenzielle Konstruktivität, [Smy13, S. 48]

Dies hat zur Folge, dass die Fallunterscheidungen über `T1` und `T3` mittels Konjunktion mit `dep` ebenfalls zu falsch auswerten. Lediglich die Fallunterscheidungen über `T2` und `T4` können so zu wahr auswerten. Erst im nächsten Tick wird die Variable `dep` auf wahr gesetzt. Die Überprüfung der Fallunterscheidungen erfolgt dann anhand der Prioritäten.

Die WTOTO Übersetzung wurde seiner Zeit prototypisch implementiert, war allerdings schon für mittelgroße Modelle zu langsam um einen schnellen Modell-Entwicklungszyklus zu erlauben. Aktuell ist in KIELER eine Form der DTO Übersetzung aktiv. Diese Optimierung wird vor den Optimierungen dieser Arbeit durchgeführt, sodass die Implementierungen dieser Arbeit profitieren.

2.2 Hardwaresynthese

Für die Implementierung der Optimierungen dieser Ausarbeitung ist es zudem interessant, wie sich die Umsetzung mit bereits verfügbaren Erweiterungen innerhalb der KIELER Umgebung verhält.

Die kürzlich in der Bachelorarbeit von Rybicki [Ryb16] vorgestellte Hardwaresynthese gehört zu diesen Erweiterungen. Ziel der Arbeit war es eine Transformation ausgehend von SCChart Modellen zu einer Hardwaredarstellung zu entwerfen.

Hier ist im Verlauf der Arbeit darauf zu achten, dass bei der Hardwaresynthese die verwendeten Variablen in Hardware als Leitung angesehen werden. Dies kann zu Problemen mit der Wiederverwendung von Variablen führen. Auf einer Leitung kann in Hardware nicht mehr als ein Signal zurzeit übertragen werden. Für die Hardwaresynthese ist im einfachsten Ansatz jede Variable eine eigene Leitung. Wenn nun das Variablenrecycling mehrfach dieselbe Variable in unterschiedlichen Kontexten verwendete, würde dies zu Problemen führen. Der Hardwaresynthese ist allerdings eine Static Single Assignment (SSA)-Transformation vorgeschaltet. SSA führt für jeden neuen Schreibzugriff auf eine Variable eine neue Version der Variablen ein. Um sequenziell die aktuelle Version der Variablen zu lesen wird eine Phi-Funktion verwendet. Da durch SSA genau so viele Variablen versioniert werden müssen, wie mit dem Variablenrecycling eingespart werden, wird der Effekt aufgehoben. In diesem speziellen Fall hat das Wiederverwenden von Variablen keine Auswirkung auf die Modell-Größe.

Kapitel 3

Verwendete Technologien

Im Verlauf dieses Kapitels werden die genutzten Technologien vorgestellt und ihre Bedeutung für die Arbeit erläutert. Angefangen mit der am Lehrstuhl von Herr Prof. Dr. von Hanxleden entwickelten Software KIELER und den für diese Arbeit wichtigen Teilkomponenten von SonarQube, einem statischen Code-Analysetool, werden Verwendung in der Ausarbeitung und eine Kurzbeschreibung angegeben. Ziel ist dieses Abschnitts ist es ein besseres Verständnis der Lösung und dessen Implementierung zu ermöglichen.

3.1 KIELER

Als Basis verwendet KIELER das Eclipse Modeling Framework (EMF)[1]. Das EMF bietet einen Stamm an Funktionalitäten zur Erstellung von Modellen und der Generierung von Quellcode aus diesen Modellen. Dies ermöglicht es eigene Applikationen im Hinblick auf die Verwendung von Modellen strukturierter Daten zu erstellen. Um eine Erweiterung von Eclipse-Applicationen zu erreichen, sind sogenannte Eclipse Plugins[2] erstellbar. Erweiterungen werden an Extension-Points in Eclipse geladen und ausgeführt. Zudem ist es möglich eigene Extension-Points zu generieren, um beispielsweise andere Erweiterungen in einer Erweiterung zu laden.

Ziel des KIELER-Projektes ist die Entwicklung neuer Methoden für den modellbasierten Entwurf von Programmen und Systemen. Diese Modelle werden zudem graphisch dargestellt. Grob gesehen kann das Projekt in zwei Bereiche aufgespalten werden, den Pragmatics- und den Semantics-Bereich. Unter dem Pragmatics-Teil verstehen sich die praktischen Aspekte der modellgetriebenen Entwicklung. Wichtige Stichworte aus diesem Bereich sind die automatische Erstellung von Diagrammen, das strukturbasierte Editieren und Layoutalgorithmen. Der Sematics-Teil beschäftigt sich unter anderem mit der Kompilierung und Simulation von Modellen. Es wird somit die Semantik der Ausführung eines Modells definiert.

Diese Ausarbeitung lässt sich dem Bereich der Semantics unterordnen. In den

[1] http://www.eclipse.org/modeling/emf/
[2] http://www.eclipse.org/pde/

nächsten beiden Unterkapiteln werden die für diese Arbeit wichtigen Teilkomponenten des KIELER-Projektes KiCo, KIEM und die TestRunner vorgestellt. Die Bedeutung der Teilkomponenten wird in der später dargestellten Implementierung und Evaluation zum Tragen kommen.

3.1.1 SCCharts

Konzeptionell ist die von Hanxleden et al. [HDM+14] entwickelte visuelle Programmiersprache SCCharts grafisch an die Statecharts von Harel [Har87] angelehnt. Es werden zudem Notationen aus SyncCharts verwendet. SyncCharts ist eine von André [And96] eingeführte synchrone Programmiersprache und stellt die von Berry [BC84] vorgestellte Programmiersprache Esterel graphisch dar. Bei der Programmierung mit SCCharts ist es möglich komplexe Problemstellungen mit Hilfe von Modellierung zu lösen und gleichzeitig die Vorteile der deterministischen Nebenläufigkeit des sequenziell konstruktiven Berechnungsmodells zu nutzen. Die folgende Abbildung 3.1 legt dar, welche erweiterten Funktionalitäten innerhalb Extended Sequentially Constructive Charts (Ext. SCCharts) nutzbar sind und welche Abhängigkeiten diese haben. Bei genauer Betrachtung der Abbildung 3.1 fällt auf, dass die Extended SCCharts auf einer Sammlung von Operationen aus den drei Programmiersprachen SCADE / QUARTZ / Esterel v7, Statecharts und SyncCharts basieren. Diese Sprachen sind in das Repertoire von SCCharts aufgenommen worden. Dies ist in der Abbildung 1.1 zu erkennen. Die Operationen aus Extended SCCharts werden in Core SCCharts übersetzt. Diese Übersetzungsarbeit wird Modelltransformation genannt. Die Implementierung der Modelltransformationen in KIELER wird im nächsten Teilkapitel beschrieben. In der Abbildung 3.2 ist eine grobe Übersicht des Kompilierungsprozesses zu sehen. Die blauen Pfeile stellen sowohl einzelne als auch ganze Sammlungen von Transformationen dar.

Abbildung 3.1 (1) zeigt ein Beispiel eines SCCharts Modells. Das in der Einleitung erwähnte ABO-Modell gehört zu dieser Gruppe. In (2) sind bereits Core SCCharts vorhanden. Es werden somit durch die Modelltransformationen Extended SCCharts in Basiskonstrukte der Core SCCharts übersetzt. (3) stellt eine normalisierte Form der Core SCCharts dar. Diese Darstellung ist für die Übersetzung in die, für diese Arbeit interessante, sequenzialisierte SCG-Form wichtig. Der seq. SCG stellt einen zyklenfreien Kontrollflussgraphen dar. Diese Eigenschaft erleichtert Optimierungen erheblich, da nicht auf Rücksprünge geachtet werden muss. Jeder Knoten wird pro Durchlauf maximal ein einziges Mal ausgeführt. Sowohl (4) als auch (5) sind bereits in SCG-Form, wobei (5) sequenzialisiert vorliegt und somit keine ausgezeichnete Nebenläufigkeit enthält. Die Sequenzialisierungsoperation von (4) zu (5) ordnet die nebenläufigen Operationen ihrer Abhängigkeiten nach an. Hierbei wird das Initialize-

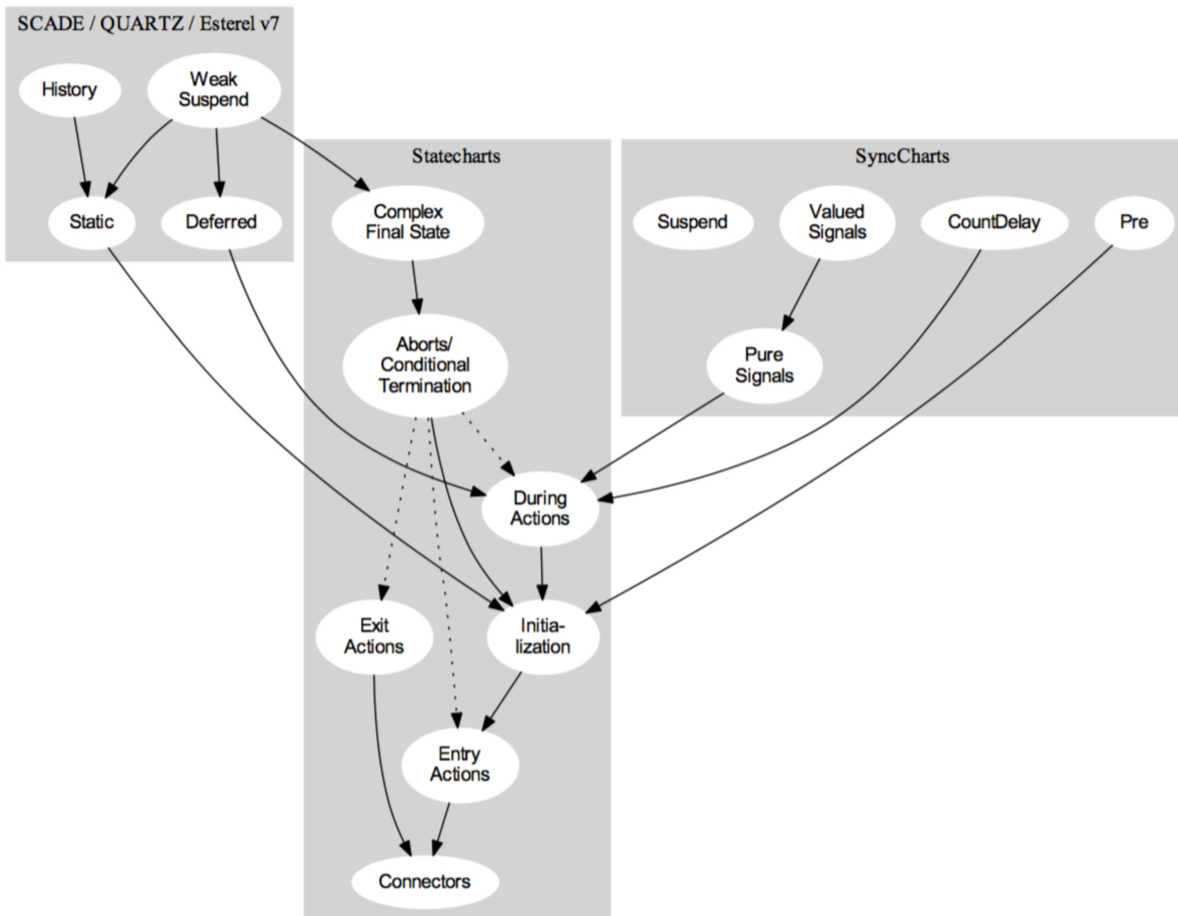
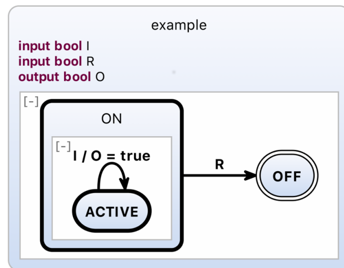
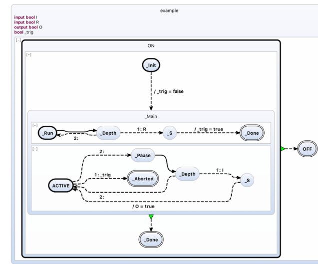


Abbildung 3.1. Abhängigkeitsgraph Extended SCCharts Transformationen [HDM+14]

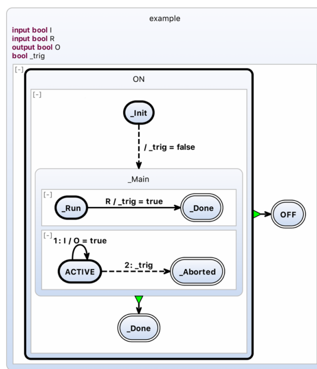
Update-Read Protokoll von Hanxleden et al. [HMA+14] verwendet. Unter dem Initialize-Update-Read Protokoll versteht sich ein Verfahren zur Sequenzialisierung nebenläufiger Modelle. Zuerst werden Initialisierungen aufgenommen, da die Initialisierung vor der Verwendung (Schreib-/Lesezugriff) stattfinden muss. Als Nächstes werden alle Modellteile aufgenommen, die einen aktualisierenden Schreibzugriff auf die Variable ausführen. Die Schreibzugriffe werden in nebenläufigen Systemen vor den Lesezugriffen durchgeführt um inkonsistente Zustände durch Race-Conditions (Ein Thread überholt einen Anderen) zu vermeiden. Auf die Aufnahme der schreibenden Zugriffe folgt die Aufnahme der Lesezugriffe. Es wird der aktualisierte Wert ausgelesen. Sollten write-write-Konflikte auftreten, so ist das Modell nicht sequenzialisiertbar. Das Protokoll kann confluent-write-Konflikte an diesem Punkt nicht feststellen, da nicht bekannt ist welcher Schreibzugriff zuerst ausgeführt werden soll.



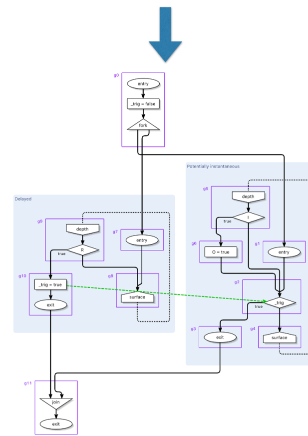
(1) Extended SCCharts



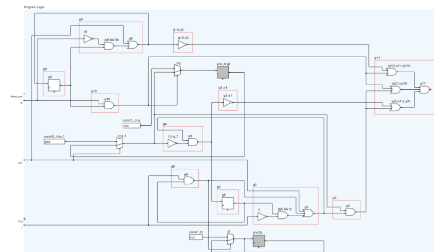
(3) SCG-normalized Core SCCharts



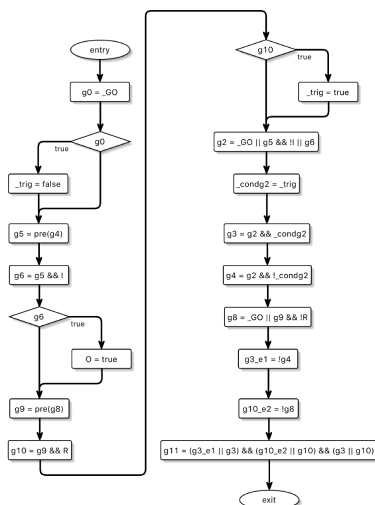
(2) Core SCCharts



(4) SCG & Dependency-Analysis & Basic Blocks



(6b) Hardware (SSA / Netlist)



(5) Sequentialised SCG

```
public void tick(){
    boolean g0;
    boolean g2;
    boolean g3;
    boolean g4;
    boolean g5;
    boolean g6;
    boolean g8;
    boolean g9;
    boolean g10;
    boolean g11;
    boolean _condg2;
    boolean g3_e1;
    boolean g10_e2;
    {
        g0 = _GO;
        if(g0){
            _trig = false;
        }
        g5 =(PRE_g4);
        g6 =(g5&&I);
        if(g6){
            O = true;
        }
        g9 =(PRE_g8);
        g10 =(g9&&R);
        if(g10){
            _trig = true;
        }
        g2 =(_GO||g6||(g5&&(!(I))));
        _condg2 = _trig;
        g3 =(g2&&_condg2);
        g4 =(g2&&(!(_condg2)));
        g8 =(_GO||(g9&&(!(R))));
        g3_e1 =(!(g4));
        g10_e2 =(!(g8));
        g11 =((g3_e1||g3)&&(g10_e2||g10)&&(g3||g10));
    }
    PRE_g4 = g4;
    PRE_g8 = g8;
    _GO = false;
    return;
}
```

(6a) Software (Java Program)

Abbildung 3.2. Kompilerkette SCCharts nach Motika [MSH+13]

Nachdem der sequenzialisierte SCG vorliegt, sind zwei Wege möglich, zum einen die Software- und zum anderen die Hardwaresynthese. Für diese Ausarbeitung interessant ist die Quellcodegenerierung.

Ziel der Kompilierung ist die Umwandlung der Extended SCCharts in Core SCCharts bis hin zur Sequenzialisierung der Nebenläufigkeit. Während der Kompilierung kommt es typischerweise zu einer Expansion der Modellgröße. Dies hängt mit der Übersetzung in Basiskonstrukte zusammen. Im Gegensatz zu High-Level-Konstrukten ermöglichen Basiskonstrukte die Übersetzung in andere Modelltypen wie zum Beispiel SCGs.

3.1.2 KIELER Compiler

Der Kieler Compiler[3], kurz KiCo, entwickelt von Motika, Smyth und von Hanxleden [MSH14], dient der Transformation von High-Level-Modellen zu Zwischenprodukten bis hin zu ausführbarem Code. Primär enthält der Kompiler eine Menge von Transformationen aus anderen Eclipse Plugins. Diese Transformationen werden an Extension Points innerhalb von KiCo eingehängt und verrichten an diesen Stellen ihre Arbeit. Grundlegende Funktionalität dieser Transformationen ist die Umwandlung eines Modells in ein Anderes.

Ergebnisse von Transformationen werden an den Kompiler zurückgeliefert und können danach an weitere Transformationen übergeben werden. Aktuell wird dieses Weitergeben von Modellen durch die KiCo-Kette realisiert. Die bei KiCo registrierten Transformationen übergeben zum Start die akzeptierten Eingangs- und die produzierten Ausgangsmodellformate. So können Transformationen ihrer Abhängigkeiten nach angeordnet werden. Optional können auch Gruppen von Transformationen gebildet werden, um die Übersichtlichkeit in der Entwicklungsumgebung zu wahren. Zudem werden Transformationen einer Gruppe beim Ausführen der Gruppe ihrer Abhängigkeiten nach hintereinander ausgeführt.

In der KIELER-Umgebung wird die KiCo-Kette wie in Abbildung 3.3 dargestellt. Die Abbildung zeigt die oberste Ebene der Kompilerkette. Die Knoten können durch Klicks weiter expandiert werden und legen so die tiefer liegenden Transformationsschritte frei.

Abbildung 3.3. KiCo Compilerchain - Übersicht

[3] http://rtsys.informatik.uni-kiel.de/confluence/display/KIELER/Kieler+Compiler

Knoten in der Kette können angeklickt werden. Es werden alle Abhängigkeiten für die ausgewählte Transformation ausgeführt und das Ergebnis ausgegeben. Während der Implementierungsphase dieser Ausarbeitung wurde die Kompilerkette um eigene Transformationen erweitert.

3.1.3 Single-Pass Language-Driven Incremental Compilation (SLIC)

Das Prinzip der Single-Pass Language-Driven Incremental Compilation (SLIC), entwickelt von Motika, Smyth und von Hanxleden [MSH14], ermöglicht es, dass nach jeder Transformation wieder ein gültiges Modell generiert wird. Das Verfahren basiert auf dem Ansatz, mehrere Kompilerkomponenten und eine Reihenfolge dieser vorzuhalten. Jede Kompilerkomponente erfüllt eine bestimmte Funktion innerhalb des Kompilationsprozesses. Ziel des Single-Pass-Ansatzes ist es nun jede Kompilerkomponente nur ein einziges Mal zu durchlaufen. Durch die Modularisierung ist es zudem möglich lediglich bis zu einem bestimmten Schritt in der Kompilation zu gehen und dort ein semantisch korrektes Modell zu erhalten. Je nachdem welches Zwischenmodell vorliegt, müssen nicht alle vorhergehenden Kompilationsschritte wiederholt werden. Hierbei wird von inkrementeller Kompilation gesprochen. Ziel hierbei ist es lediglich die Kompilerschritte auszuführen, die noch nicht ausgeführt wurden. Bei Änderung des Modells müssen alle vorherigen Schritte wiederholt werden. Das Wiederholen kommt dem Clean Build Ansatz herkömmlicher Kompiler nahe. Werden Transformationen durch KiCo ausgeführt, so wird hierbei nach dem SLIC-Ansatz verfahren.

3.1.4 TestRunner

Zu Eclipse Plugins lassen sich auch JUnit[4] Plugin Tests anlegen. Diese Tests dienen der Überprüfung der von dem Plugin zur Verfügung gestellten Funktionen.

Um nun auf einer Menge von SCChart-Modellen die gleichen Tests ausführen zu können, wurden von Schneider und Smyth sogenannte TestRunner entwickelt. Diese wenden geschriebene Unit-Test auf jedes einzelne Modell innerhalb eines spezifizierten Ordners an und geben das Ergebnis aus. Genauer wird für jede Test-Modell-Kombination ein eigener Unit-Test erstellt und ausgeführt.

Für diese Arbeit wird der TestRunner nicht primär zum Testen eingesetzt, sondern um Auswertungen über eine große Menge von Modellen zu führen. Hierzu wurden die TestRunner mit KiCo kombiniert. Die durch den TestRunner geladenen Modelle werden mittels KiCo und einer vorgegebenen Kompilerkette kompiliert und vermes-

[4] http://junit.org/

sen. Die Ergebnisse der einzelnen Modelltests werden danach in eine Datenhaltung geschrieben. Aufgrund der sequenziellen Struktur von Unittests und der alphanumerischen Sortierung werden die Ergebnisse einer Menge von Modellen immer in der gleichen Reihenfolge in die Datenhaltung geschrieben. Dies vereinfacht die Analyse der Daten stark.

3.1.5 KIELER Execution Manager

Der KIELER Execution Manager (KIEM) dient der Ausführung von Modellen innerhalb von KIELER. Aktuell ist in KIEM eine Ausführung ein Durchlauf der tick-Funktion des durch KiCo generierten C-Codes. Um eine Vermessung der Ausführung zu ermöglichen, wird vor der Kompilation durch einen C-Kompiler zusätzlicher C-Code in den generierten C-Code eingefügt. Dieser dient zum Beispiel der Ermittlung der benötigten Ausführungszeit der tick-Funktion. Zusätzlich können Observer und Producer hinzugefügt werden, welche Daten in eine Ausführung einbringen oder Daten auslesen. Wenn beispielsweise ein Producer aufgezeichnete Eingaben einer vorherigen Ausführung einspielt, so kann eine Ausführung mehrfach hintereinander gleich ablaufen. Parallel kann durch einen Observer eine Überprüfung dieser Ausführung stattfinden. Je nachdem welche Daten geprüft werden kann somit sowohl die Ausführungszeit als auch die Korrektheit, bei Vergleich der Ausgaben des Modells nach jeder Ausführung, als Kennzahl verwendet werden.

Für diese Arbeit wird KIEM in Verbindung mit den vorher genannten TestRunnern für die Ausführungszeitanalyse verwendet. Hierzu wird das von KIEM bereitgestellte Benchmark-Modul und ein eigens angefertigter Observer verwendet. Dieser Aufbau ermöglicht es für einen Ordner von Modellen und dazugehörige ESO-Dateien, eine Art Trace-Datei, wiederholt Analysen durchzuführen.

Abbildung 3.4 zeigt eine zufällig generierte ESO-Datei für das aus Abbildung 1.5 bereits bekannte ABO-Modell. Die Beispieldatei enthält zwei Spuren (Traces) mit je fünf Ticks sowie dazugehörige Ein- und Ausgaben. Beide Spuren beginnen mit '`! reset;`'. Dieser Befehl setzt alle Variablen eines Modells zurück. Die einzelnen Ticks sind immer eine Abfolge von gesetzten Eingaben, erwarteten Ausgaben und dem Semikolon als Trennzeichen. Bei automatisch generierten ESO-Dateien werden aktuell keine erwarteten Ausgaben definiert. Die in (2) zu sehenden Ausgaben sind manuell simuliert anhand der Eingaben der Spur. Es ist zu erkennen, dass in Tick drei mittels der Eingabe `A(true) B(true)` das ABO-Modell die Transition zum Endzustand mit der Ausgabe `O(true)` ausführt. Der Automat verweilt danach im Endzustand, weshalb sich die Ausgabe unabhängig von den Eingaben nicht weiter ändert.

```
! reset;                        ! reset;
A(true) B(true)                 A(false) B(false)
% Output:                       % Output: O(false)
;                               ;
A(false) B(false)               A(false) B(false)
% Output:                       % Output: O(false)
;                               ;
A(true) B(true)                 A(true) B(true)
% Output:                       % Output: O(true)
;                               ;
A(true) B(true)                 A(false) B(false)
% Output:                       % Output: O(true)
;                               ;
A(false) B(false)               A(true) B(true)
% Output:                       % Output: O(true)
;                               ;
```

Spur 1 Spur 2

Abbildung 3.4. Zufällig generierte ESO-Datei (2 Traces) des ABO-Modells (Spur zwei manuell um Ausgaben erweitert)

3.2 SonarQube

Statische Codeanalyse auf Quellcode hilft frühzeitig Fehler oder optimierbare Stellen zu finden. Zu diesem Zweck wurde auch SonarQube[5] entwickelt. Die Verwendung von Sonarqube beschreibt Campbell [AP13] ausführlich. SonarQube besteht aus zwei verschiedenen Komponenten. Die erste Komponente ist der SonarQube Server und hat die Analyse und Speicherung der Projektdaten zur Aufgabe. Die Projektdaten werden von der zweiten Komponente, dem SonarRunner, an den Server übertragen. Mit Hilfe einer im Wurzelverzeichnis des Projektes liegenden XML-Datei werden dem SonarRunner Parameter wie zu analysierende Ordner, Projektname, Projektstatus oder Zielserver übergeben. Anhand dieser Daten wird das Projekt nach Quellcodedateien durchsucht und an den Server übertragen. Klassischerweise kann nach der Analyse das Ergebnis mit dem Webinterface des Servers eingesehen werden. Innerhalb dieser Ausarbeitung wird SonarQube in Verbindung mit dem Java-Plugin[6] genutzt. Ziel ist

[5]http://www.sonarqube.org
[6]http://docs.sonarqube.org/display/PLUG/Java+Plugin

es den aus KIELER generierten Java-Code zu analysieren. Hierzu bringt das Plugin verschiedenste Regeln für die Sprache mit. Darunter sind zum einen Regel für die eigentliche Analyse der Syntax und zum anderen die Regeln für den statischen Codecheck, wie zum Beispiel typische Strukturen die eine Null-Pointer-Exception auslösen können, mit Templates für Verbesserungsvorschläge. Die Liste der statischen Codechecks ist bei dem Java-Plugin recht umfangreich.

Für diese Ausarbeitung wurde SonarQube verwendet, da mit dieser Software bereits Erfahrungen im PHP-, C++- und C#-Umfeld gesammelt wurde. Somit war keine Einarbeitungszeit für ein neues Tool erforderlich. Ebenfalls spricht für Sonar-Qube der OpenSource-Ansatz, welcher es erlaubt die Software zu erweitern und den persönlichen Präferenzen nach anzupassen.

Kapitel 4

Konzept

Aktuell wird in KIELER die Kompilation in Richtung Quellcode aus der sequenzialisierten Form eines SCG generiert. Diese SCGs werden aus Modellen höherer Ebenen generiert. Bei der Übersetzung zu Quellcode werden die im SCG enthaltenen Knoten in Codezeilen umgewandelt. Es findet somit eine Eins-zu-Eins-Transformation statt. Vorteil dieser Herangehensweise ist, dass noch gut nachverfolgt werden kann, wo die Anweisungen ihren Ursprung haben und wie Knoten des SCGs zu Stande gekommen sind. Ein Nachteil ist allerdings, dass so mehr Variablen verwendet werden als für das eigentliche Programm notwendig wären. Um nun Ansatzpunkte für eine spätere Optimierung zu finden, wird der Quellcode und auch der sequenzialisierte SCG einiger Modelle analysiert. Auf Basis der Analyse werden Optimierungsmöglichkeiten diskutiert und ihrem Potenzial nach eingeordnet.

4.1 Analyse Quellcode

Um das Optimierungspotenzial von KIELER generierten Quellcodes zu ermitteln, soll sowohl der Java-Quellcode als auch der sequenzialisierte SCG einiger Beispielmodelle analysiert werden. Der sequenzialisierte SCG wird im Lauf dieses Kapitels von Hand analysiert. Hierzu wird ein Beispielmodell in Richtung SCG transformiert und mittels KIELER grafisch dargestellt. Der Vorteil der grafischen Darstellung gegenüber der textuellen Darstellung des SCGs ist, dass mit dem bloßen Auge schneller optimierbare Strukturen im Sinne der Fragestellung erfasst werden können. Dazu zählen sowohl wann Variablen zugewiesen und verwendet werden, als auch wie komplex die Zuweisungen ausfallen. Für den Java-Quellcode wird SonarQube als Hilfsmittel verwendet. Die statische Codeanalyse dient größtenteils der Prüfung von Programmierrichtlinien anhand von Regeln. Auf die Analyse mit SonarQube wird am Ende des Kapitels eingegangen.

Als erstes Beispiel soll das bereits in der Einleitung genannte ABO-Modell (Abb. 1.5) dienen.

Bei erster Betrachtung der Abbildung 4.1 ist ein entry- und ein exit-Knoten zu sehen. Diese markieren Start bzw. Ende eines Threads. Die Eingänge A und B sowie

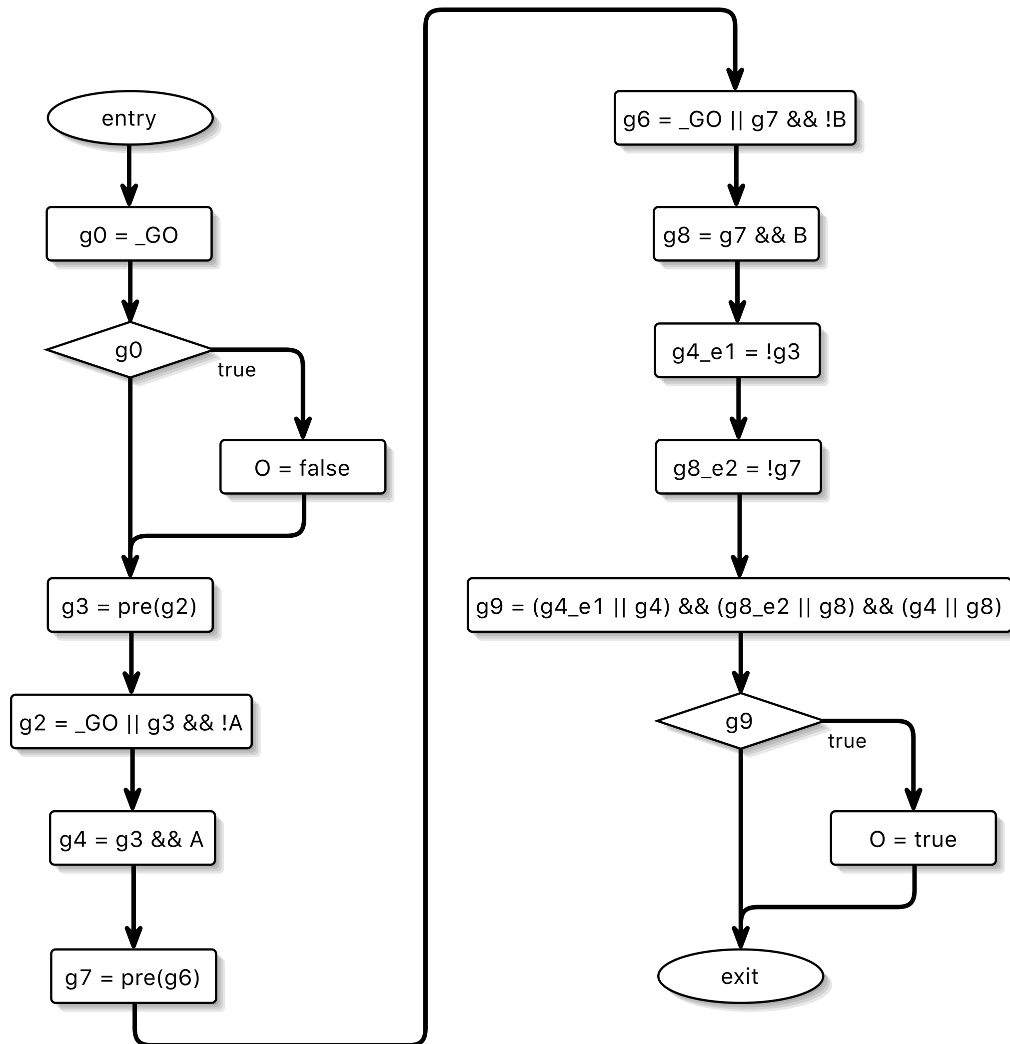


Abbildung 4.1. ABO Modell - Seq. SCG

der Ausgang O sind ebenfalls wiederzufinden. Es ist allerdings nicht mehr genau spezifiziert, ob es sich hierbei um Ein- oder Ausgänge handelt. Ihre Verwendung ähnelt der einer Variablen.

Im Hinblick auf Optimierung des Quellcodes ist zu erkennen, dass viele Variablen lediglich einen Wert einer anderen Variablen aufnehmen und keine Auswertung vor der Zuweisung stattfindet. Da im späteren Verlauf auch pre-Funktionen zu Variablen übersetzt werden, können diese zu den Eins-zu-Eins-Zuweisungen gezählt werden. Zusätzlich können verneinte Aussagen ebenfalls hinzugezogen werden um ein höheres Optimierungspotenzial in Hinsicht auf die Größe zu erzielen.

Durch das Hinzuziehen der Negationen kann es bei der späteren Ausführung

Original	Mehrfachberechnung
`g0 = _GO;`	`g0 = _GO;`
`g1 = !go;`	`g1 = !_GO; // !g0`
`g2 = g1;`	`g2 = !_GO; // g1`
`g3 = !g2;`	`g3 = !(!_GO); // !g2`
`if (g3) {`	`if (!(!_GO)) {`
`  O = true;`	`  O = true;`
`}`	`}`
`if (!g3) {`	`if( !(!(!_GO))) {`
`  O = false;`	`  O = false;`
`}`	`}`

Abbildung 4.2. Mehrfachberechnung Negation - Beispiel

zu einer Verlangsamung kommen, da die Negation mehrfach berechnet werden muss. Hierzu betrachte man den Beispielcode in Abbildung 4.2. Es sind zwei Codeausschnitte dargestellt. Der original Code zeigt ein Programm, welches einige Negationen auf Variablen berechnet. Je nach Wert der Variable `_GO` wird entweder `O` auf wahr oder falsch gesetzt. Die Sinnhaftigkeit des Programmes ist an dieser Stelle nicht wichtig. Der rechte Codeausschnitt zeigt die Optimierung durch die Kopierpropagation auf Negationen von Variablen. Es ist zu erkennen, dass sowohl in der ersten als auch in der zweiten Bedingung Fallunterscheidungen mehrfach die Negation berechnet wird. Insgesamt erfolgen fünf Negationsberechnungen innerhalb der beiden If-Bedingungen. Das Original enthält insgesamt, über den vollständigen Codeausschnitt, nur drei Negationen. Es müssen somit mehr Berechnungen durchgeführt werden. Eine genauere Betrachtung der Ausführungszeiten der späteren Implementierung erfolgt in Kapitel 6.2.

Werden nun all diese Vorkommen in dem SCG markiert, so ist folgende Abbildung 4.3 das Ergebnis. Die erste Markierung einer Farbe ist die initiale Zuweisung der markierten Variable. Jede weitere Markierung der Farbe kennzeichnet eine schreibende oder lesende Verwendung der Variable. Bei genauerer Betrachtung fällt auf, dass Abhängigkeiten der gewählten Variablen auftreten können. Beispielsweise wird g7 in g8_e2 verwendet. Eine Kopierpropagation an diesen beiden Stellen hätte somit einen doppelten Nutzen. Eine mögliche Optimierung ist somit die Kopierpropagation. Eine Optimierung mittels Kopierpropagation (inklusive Eliminierung nicht verwendeten Codes) ist in Abbildung 4.4 zu finden.

Es ist zu erkennen, dass die vorher farbig markierten initialen Zuweisungen nicht weiter in dem Modell existent sind. Der Term der initialen Zuweisung wurde an der Verwendungsstelle einsetzt. Die ersetzten Variablen sind `g0`, `g3`, `g7`, `g4_e1` und `g8_e2`.

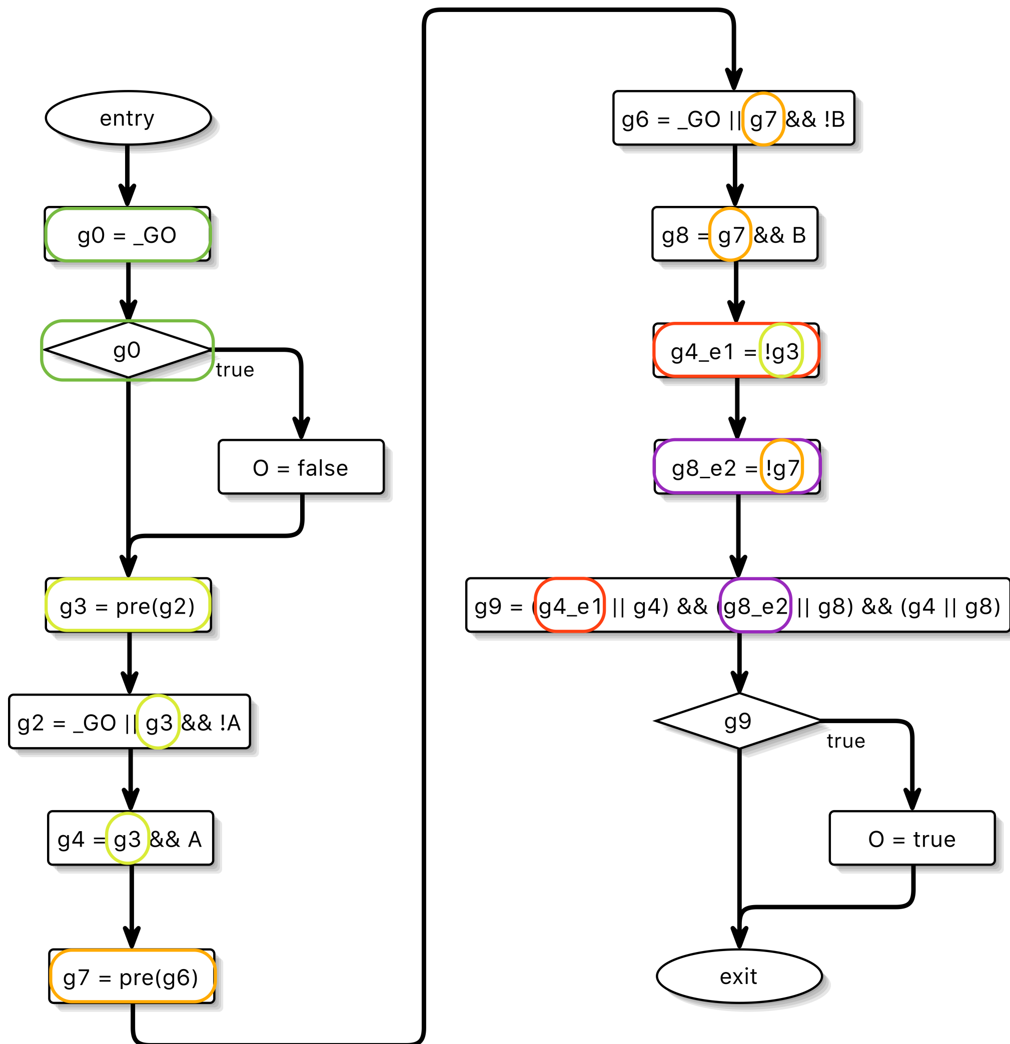


Abbildung 4.3. ABO Modell mit Markierungen - Seq. SCG (Erste Markierung: Initiale Zuweisung, Weitere Markierungen: Verwendung)

Durch die Substitutionen von beispielsweise `g3` und `g4_e1` entstehen zudem zusammengesetzte Terme wie `!pre(g2)`. Dies geschieht durch die Komposition der Substitution von `g3` in `g4_e1 = !g3` und `g4_e1` in `g9 = (g4_e1 || g4) && [...]`.

Bei weiterer Betrachtung einer der beiden vorhergehenden Abbildungen 4.3 ist zudem zu sehen, dass nur wenige Variablen bis zum Ende des Programmes verwendet werden. Häufig ist es sogar so, dass unmittelbar nach der Variablenzuweisung die einzige Abfrage der Variablen stattfindet. Dies bietet ein großes Potenzial für die Wiederverwendung von Variablen. Somit ist eine weitere Optimierungsmöglichkeit die Variablenwiederverwendung.

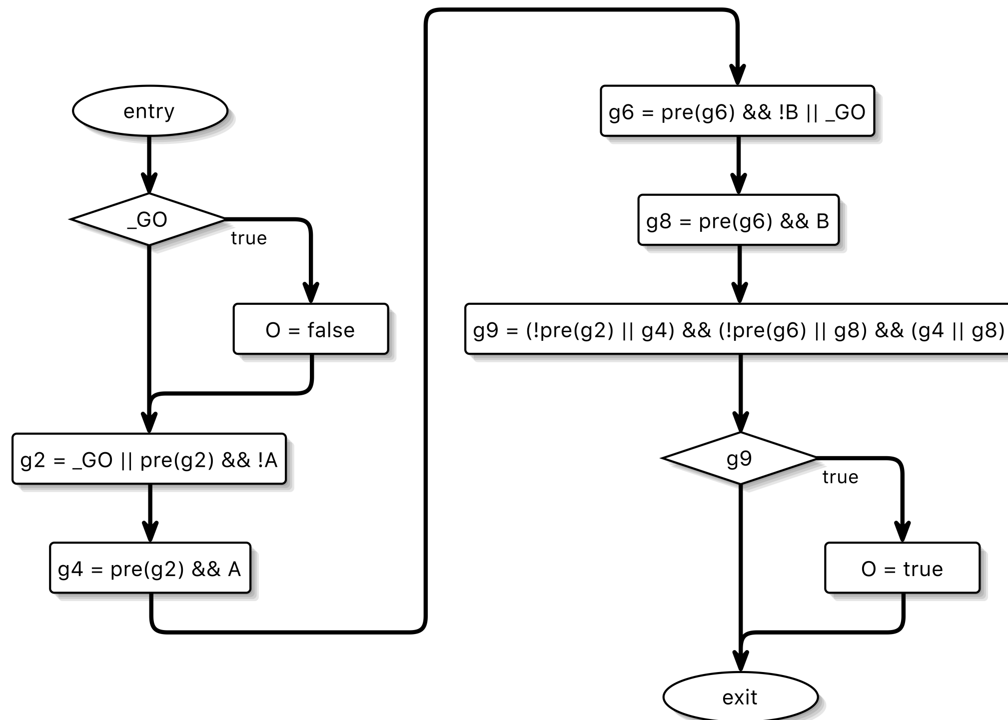


Abbildung 4.4. ABO Modell Optimierung mittels Kopierpropagation - Seq. SCG

Eine Optimierung mittels Variablenwiederverwendung ist in Abbildung 4.6 zu finden. Es ist zu erkennen, dass die Variablen `g3` und `g9` nicht weiter auftreten. `g3` wurde in `g0` umbenannt, da `g0` nach der Prüfung in der Bedingung der Fallunterscheidung über `g0` nicht weiter verwendet wird. Ähnlich verhält es sich mit `g9`. Diese wird in `g7` umbenannt, da `g7` in der Zeile `g8_e2 = !g7` die letzte Verwendung aufweist.

Wird nun ein Blick auf die Variable g9 in Abbildung 4.5 geworfen, so fällt eine weitere Optimierungsmöglichkeit auf. Die letzte Konjunktion von Disjunktionen lässt sich mittels Distributivgesetz vereinfachen. (g8_e2 $\vee$ g8) $\wedge$ (g4 $\vee$ g8) ist äquivalent zu (g8_e2 $\wedge$ g4) $\vee$ g8.

```
g9 = (g4_e1 || g4) && (g8_e2 || g8) && (g4 || g8)
```

Abbildung 4.5. Analyse ABO Modell Variable g9

Somit kann eine weitere Optimierungsmöglichkeit auch die Optimierung von booleschen Ausdrücken sein. Bei weiteren Analysen ist allerdings aufgefallen, dass diese optimierbaren Terme selten auftreten und, wenn sie auftreten, eine sehr geringe Auswirkung auf die Größe des generierten Codes haben. Zudem werden die meisten

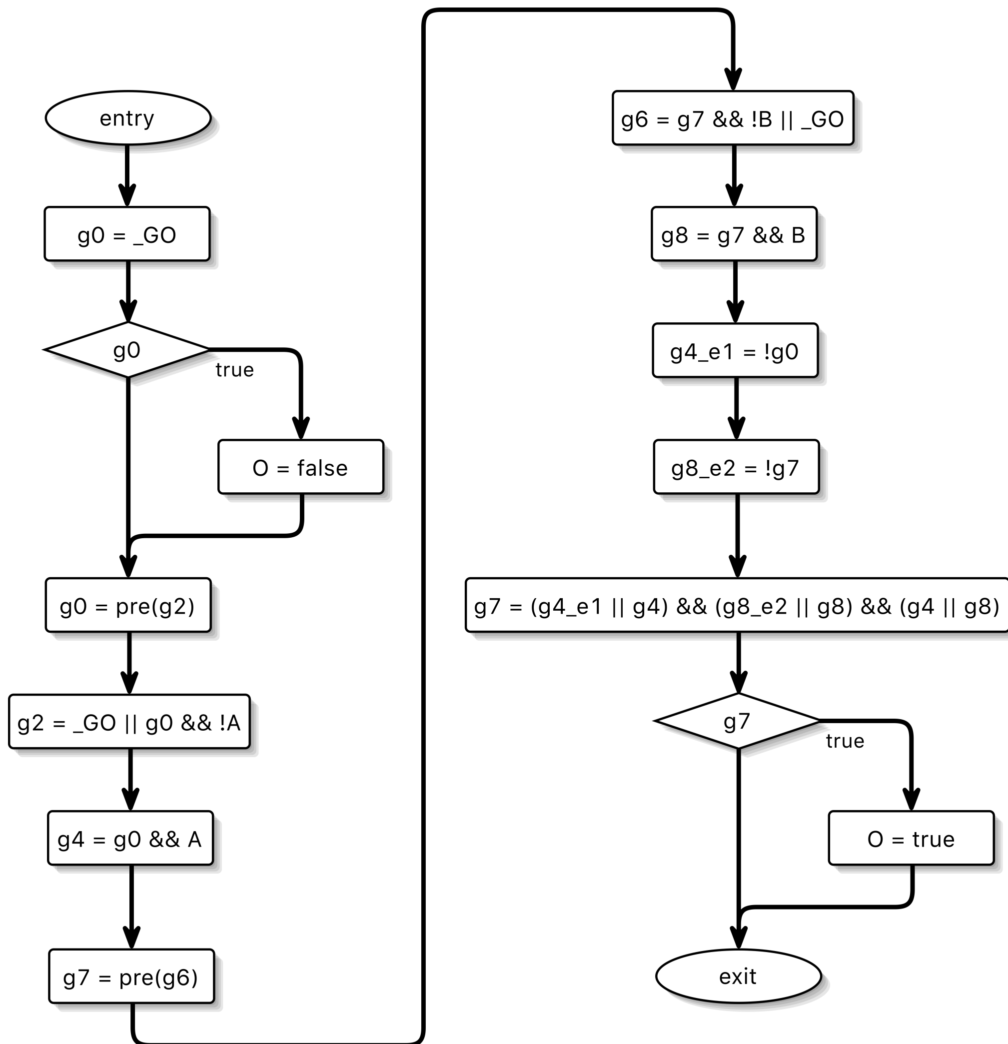


Abbildung 4.6. ABO Modell Optimierung mittels Variablenwiederverwendung - Seq. SCG

booleschen Terme bereits durch die Kompiler für die ausführbaren Kompilate vereinfacht. Angesichts dieses geringen Optimierungspotenzials wird auf die Optimerung von booleschen Termen verzichtet.

Während der Analyse ist aufgefallen, dass keine konstanten Werte generiert werden. Um für spätere Anwendungszwecke dennoch konstante Werte, ähnlich wie bei der Konstantenpropagation zu optimieren, wird ein zusätzlicher Selektor in die Kopierpropagation aufgenommen. Dieser sorgt dafür, dass Zuweisungen der Art `X := 42` oder `Y := true` als Werte an der Stelle ihrer Verwendung ersetzt werden.

Als nächstes wird das ABO-Modell zu Java-Code transformiert und in die SonarQube-Umgebung einspielt. Das Ergebnis der Analyse hat keine der vorher genannten

Optimierungspunkte erkannt. Dies macht auch Sinn, da eine statische Codeanalyse lediglich bei der Wartbarkeit von Quellcode helfen soll und keine Kontextinformationen entfernen möchte. Es werden allerdings andere Verbesserungsvorschläge aufgedeckt. Als bedeutende Fehler werden zum einen die überflüssigen Klammerungen der booleschen Ausdrücke und die Anzahl der Variablen pro booleschen Term angemerkt. Die überflüssigen Klammern werden bei der Kompilierung entfernt und wirken sich nicht negativ auf die Codegröße aus. Ebenso verhält es sich mit der Anzahl an Variablen innerhalb eines booleschen Terms. Dies mag für einen Programmierer unübersichtlich und verwirrend sein. Da der Quellcode an dieser Stelle allerdings von computergenerierter Form ist, ist dies ebenfalls keiner Optimierung wert.

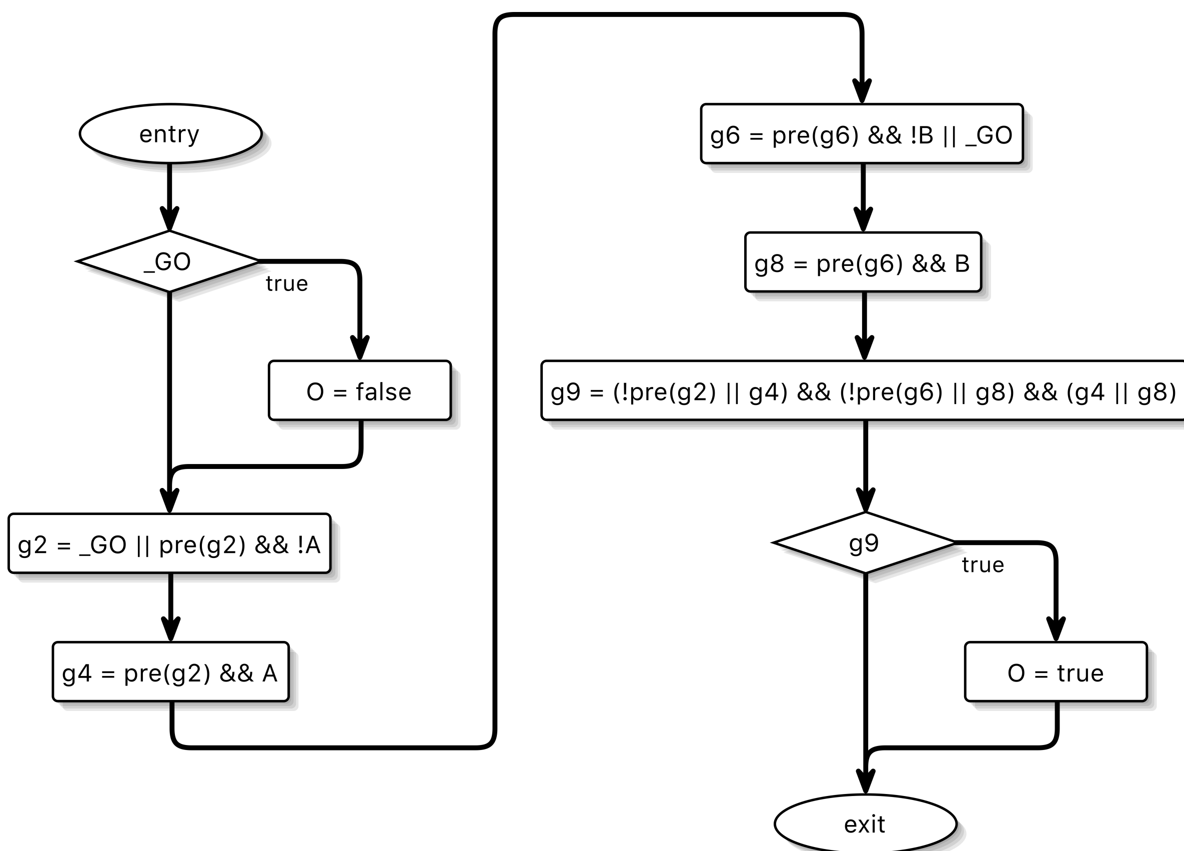


Abbildung 4.7. ABO-Modell Kopierpropagation & Variablenwiederverwendung

Zusammengefasst gibt es zwei Optimierungen welche genügend Potential haben, um für eine Implementierung herangezogen zu werden. Hinzu gesellt sich die Eliminierung von ungenutztem Code. Diese ist nötig, um die durch die anderen Optimierungen eingesparten Codezeilen zu entfernen.

Wird das ABO-Modell aus Abbildung 1.5 (SCCharts-Modell) und 4.1 (seq. SCG) mittels Kopierpropagation und Variablenrecycling optimiert, so ist das SCG-Modell in Abbildung 4.7 das Ergebnis. Im Vergleich zu Abbildung 4.1 ist eine deutliche Reduktion der Knoten zu erkennen. Die Knotenersparnis liegt bei 31,25%. Ebenfalls wird die Variablenanzahl der g-Variablen von 10 auf 5 reduziert.

In der Einleitung wird zudem das Barcode-Modell (Abbildung 1.4) ein erstes Mal angeführt. Ohne Optimierungen enthält das Modelle 256 Variablen. Weiter schlüsselt sich das Modell in 159 Zuweisungen und 406 Knoten auf. Mit den Optimierungen Kopierpropagation und Variablenwiederverwendung sinken diese Kennzahlen ab. Das optimierte Modell setzt sich aus 364 Knoten, 117 Zuweisungen und 173 Variablen zusammen. Es werden 32,4% der Variablen eingespart. Somit kann das Modell für die leJOS-Umgebung des Lego Mindstorms NXT kompiliert werden.

Als Nächstes soll nun die Kopierpropagation genauer vorgestellt werden. Darauf folgend werden sowohl die Variablenwiederverwendung und als auch die Eliminierung ungenutzten Codes genauer erläutert.

4.2 Kopierpropagation

Die Abbildung 4.8 stellt die vorgeschlagenen Optionen der Kopierpropagation vor. Wie in den verwandten Arbeiten beschrieben, sollen Eins-zu-Eins-Zuweisungen ersetzt werden (a). Um eine höhere Effektivität zu erreichen werden weitere Strukturen hinzugenommen. Es sollen Variablen ersetzt werden, die lediglich die Negation einer Variablen beinhalten (b). Ebenso sollen Variablen, die Werte einer ausgewerteten pre-Funktion enthalten, ersetzt werden (c). In der Quellcodegenerierung werden pre-Funktionen des SCG zu Variablen übersetzt, daher handelt es sich bei einer Zuweisung einer pre-Funktion um eine Zuweisung der in (a) beschriebenen Form.

```
(a) X = Y; (b) X = !Y; (c) X = pre(Y)
```

Abbildung 4.8. Kopierpropagation Erweiterung

Allerdings ergeben sich auch einige Schwierigkeiten bei der Ersetzung von Variablen innerhalb eines sequenzialisierten SCGs. Genauer betrifft dies die Verwendung der pre-Funktion. Es können keine Variablen ersetzt werden, die innerhalb einer pre-Funktion verwendet werden. Eine Ersetzung von `X` in Term `pre(X)` aus Abbildung 4.9 ist somit nicht zulässig.

```
X = Y; Z = pre(X);
```

Abbildung 4.9. Kopierpropagation Einschränkung

Ohne diese Einschränkung können Verkettungen von pre-Funktionen entstehen, welche so nicht vorgesehen sind. Hierzu soll die Abbildung 4.10 ein Beispiel sein. Es ist zu erkennen, dass aus der originalen Zeile `g2 = pre(g1)` die Zeile `g2 = pre(pre(_GO))` entstanten ist. Dies hängt wieder mit der Komposition der Substitution zusammen. Durch die Ersetzung von `g0` in `g1 = pre(g0)` entsteht der Term `g1 = pre(_GO)`. Die Substitution von `g1` in `g2 = pre(g1)` resultiert darauf in dem dargestellten Term `g2 = pre(pre(_GO))`.

```
g0 = _GO;
g1 = pre(g0);
if (g1) {
  g2 = g1;
}
if (!g1) {
  g2 = pre(g1);
}
```

Original

```
g0 = _GO;
g1 = pre(_GO);
if (pre(_GO)) {
  g2 = pre(_GO);
}
if (!(pre(_GO))) {
  g2 = pre(pre(_GO));
}
```

pre-Verkettung

Abbildung 4.10. Kopierpropagation Einschränkung - Beispiel

Nach der Anwendung der Kopierpropagation verbleiben einige ungenutzte Codestellen. Diese entstehen durch die nunmehr ungenutzten Variablen. Das Entfernen dieser Stellen wird der Eliminierung ungenutzten Codes im späteren Verlauf dieses Kapitels überlassen.

4.3 Konstantenpropagation

Im Analysekapitel dieser Ausarbeitung wurde erwähnt, dass keine Stellen konstanter Werte in den analysierten Modellen gefunden wurden. Um an dieser Stelle zukunftssicher zu bleiben, werden die aktuell in KIELER implementierten statischen Wertedeklarationen für Variablen mit in einen Selektor aufgenommen. Hierdurch wird keine Auswertung der entstehenden Terme durchgeführt und eine allgemeine Einsparung an Variablen erreicht. In Abbildung 4.11 ist ein entsprechendes Beispiel der Konstantenpropagation auf SCG-Ebene zu sehen.

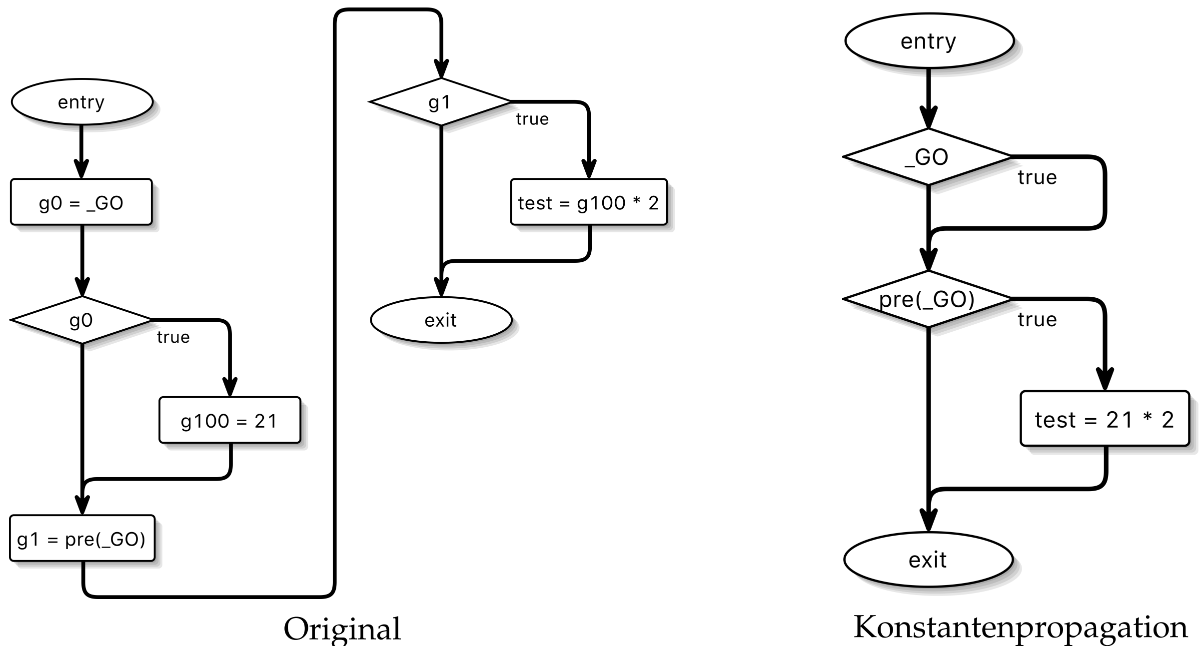


Abbildung 4.11. Konstantenpropagation - Beispiel

Das konstruierte Beispiel zeigt einen Vorher-Nachher-Vergleich. Das Original zeigt insbesondere die Variable `g100`. Diese wird im ersten Tick auf die Zahl 21 initialisiert. Im zweiten Tick wird mit zwei multipliziert in der Variablen test zugewiesen. Wird nun die Kopierpropagation mit Konstantenpropagation-Erweiterung auf das Modell angewendet, so lässt sich die typischen Ersetzungen der Kopierpropagation erkennen. Zusätzlich wurde in dem Term `test = g100 * 2` die Variable `g100` durch ihren Wert substituiert, sodass der Term `test = 21 * 2` das Ergebnis ist.

4.4 Variablen wiederverwenden

Die Wiederverwendung von Variablen lehnt sich an die Wiederverwendung von Registern (Kapitel 2.1.3) an. Hierbei sollen bereits genutzte Variablen nach ihrer letzten Verwendung wiederverwendet werden können.

Der Ablauf der Wiederverwendung beschränkt sich auf drei Schritte. Zuerst werden für alle Variablen die erste Zuweisung und die letzte Verwendung ermittelt. Wenn nun eine Variable A den Namen einer anderen Variable B annehmen soll, so muss überprüft werden, ob die Zuweisung von A nach der letzten Verwendung von B liegt. Ebenso darf vor der Zuweisung von B keine Verwendung von A stattgefunden haben. Wenn diese Kriterien erfüllt sind können alle Vorkommen von A durch B ersetzt werden. Zur Verdeutlichung dient Abbildung 4.12.

```
A = true;
B = !A;
C = B;
---------
A = true;
A = !A;
C = A;
```

(a)

```
A = true;
B = !A;
D = A;
---------
A = true;
A = !A; // Kontext!
D = A;
```

(b)

Abbildung 4.12. Wiederverwendung von Variablen - Beispiel - Zulässige (a) und unzulässige (b) Wiederverwendung

In Teilabbildung 4.12 (a) ist zu sehen wie in der zweiten Zeile die Variable B durch die Variable A ersetzt wird. Ebenfalls muss dann in der dritten Zeile die Verwendung von B durch die Verwendung von A ersetzt werden. Dies ist eine korrekte Anwendung der Variablenwiederverwendung.

Teilabbildung 4.12 (b) hingegen zeigt ein Negativbeispiel. Es wird ebenfalls in Zeile zwei die Variable B ersetzt. Da hier in Zeile drei die Variable A jedoch in ihrem alten Kontext wiederverwendet wird, ist die Ersetzung in Zeile zwei unzulässig.

Ähnlich wie bei der Kopierpropagation entsteht durch die Variablenwiederverwendung ungenutzter Code. Dieser liegt in Form von Variablendeklarationen vor. Die Entfernung dieser Codestellen wird im nächsten Teilkapitel besprochen.

4.5 Eliminierung ungenutzten Codes

Aus der Kopierpropagation und der Variablenwiederverwendung werden die ungenutzten Codestellen an die Eliminierung ungenutzten Codes übergeben. Bei Variablendeklarationen wird diese aus der Liste der Deklarationen entfernt. Bei Zuweisungen wird der entsprechende Knoten innerhalb des SCGs entfernt. Dabei ist darauf zu achten, dass der vorhergehende Knoten nach der Entfernung mit dem nachfolgenden Knoten verbunden wird. Da die zu entfernenden Stellen direkt aus den vorhergehenden Optimierungen übergeben werden, müssen keine zusätzlichen Vorkehrungen oder Überprüfungen durchgeführt werden.

Wird die Eliminierung ungenutzten Codes modular von den anderen Optimierungen getrennt, so wird eine Erfassung der zu entfernenden Variablen notwendig. Konzeptuell werden alle im seq. SCG enthaltenen initialen Zuweisungen von Variablen auf eine Verwendung in den nachfolgenden erreichbaren Knoten überprüft. Da keine Rücksprünge in einem seq. SCG möglich sind, ist es ausreichend lediglich die

nachfolgenden Knoten zu betrachten. Ist die Verwendungsanzahl der Variable gleich null, so wird diese als nicht verwendet angesehen und entfernt.

Erst mithilfe dieser Optimierung kann die Codegröße effektiv reduziert werden.

Kapitel 5

Implementierung Quellcodeoptimierung

Die Quellcodeoptimierungen sind in KIELER in einem eigens erstelltem Plugin gekapselt. Dieses Plugin wird über vordefinierte Erweiterungspunkte in andere KIELER-Plugins eingehängt und von dort aus angesteuert. Das Plugin steuert zudem die visuelle Kapselung der einzelnen Optimierungen. In der Kompilerkette erstellt das Plugin eine, der Sequenzialisierung des SCGs nachgelagerte, Gruppe von Transformationen. Jede Transformation stellt dabei eine in sich geschlossene Optimierung dar.

Beide Optimierungen sind selbst innerhalb einer Klasse beheimatet. Als wichtigste Komponente ist die transform-Methode zu verstehen. Diese nimmt als Eingabe jeweils einen sequenzialisierten SCG an und transformiert diesen in einen optimierten sequenzialisierten SCG. Die Ausgabe kann danach wie ein sequenzialisierter SCG weiterverarbeitet werden.

Die Implementierungen erben ihre Rahmenfunktionalitäten von einer AbstractProductionTransformation. Die Darstellung in Unified Modeling Language (UML) ist in Abbildung 5.1 zu sehen.

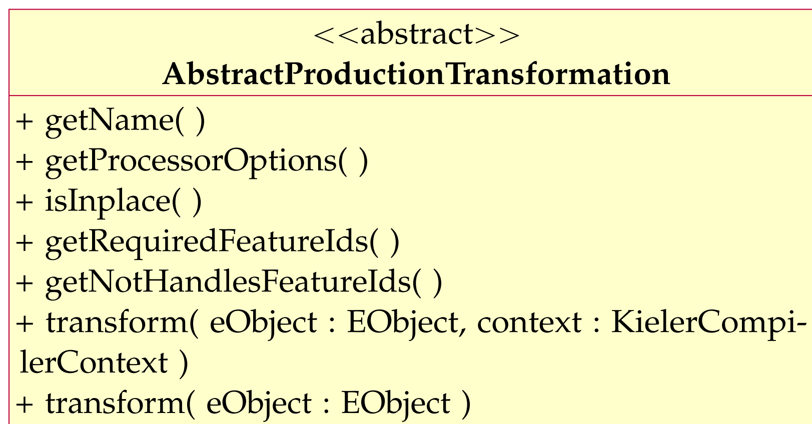


Abbildung 5.1. AbstractProductionTransformation (Implementierung in KIELER durch Schulz-Rosengarten, A.) - UML

Methoden denen ein get voransteht dienen der Beschreibung einer Production-Transformation. Der Rückgabewert von getName() legt den Namen innerhalb der Kompilerkette fest. Weiter sind die Rückgabewerte von getRequiredFeaturesIds und getNotHandlesFeaturesIds wichtig für die nachfolgenden Implementierungen. Sie legen fest, welche Typen an Eingabewerten für die transform-Methoden verwendet werden können und welche nicht. Die Methode transform nimmt ein Modell auf und liefert ein neues Modell. Eine Überladung der transform-Methode erlaubt es zusätzlich einen gegebenen Kontext der Kompilerumgebung zu übergeben. Die übrig gebliebenen Methoden isInplace und getProcessorOptions werden an dieser Stelle nicht behandelt, da sie nicht wichtig für den weiteren Verlauf der Arbeit sind.

Die genaue Arbeitsweise der einzelnen Implementierungen wird im Nachfolgenden erklärt.

5.1 Kopierpropagation

Für die Implementierung der Kopierpropagation und die Erweiterung durch die Konstantenpropagation wird AbstractProductionTransformation aus Abbildung 5.1 wie folgt erweitert:

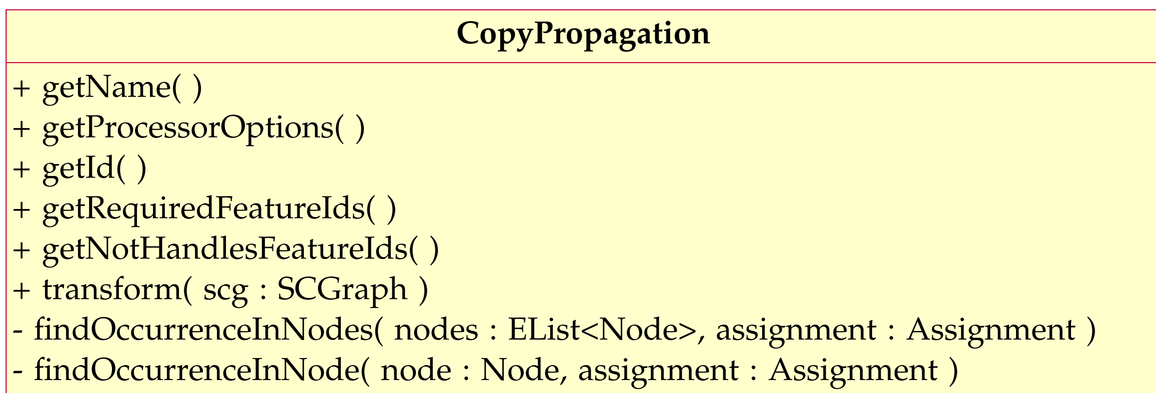


Abbildung 5.2. CopyPropagation - UML

Neben den bereits in der Einleitung dieses Kapitel beschriebenen Methoden existieren in dem UML-Modell zusätzlich getId, findOccurrenceInNodes und findOccurrenceInNode. GetId liefert ähnlich wie getName einen Bezeichner für die Kompilerkette. FindOccurrenceInNodes und findOccurrenceInNode sind die wichtigsten Methoden für die Kopierpropagation. Sie beinhalten die Substitution der Variablen.

```
Data: sequentialized SCG {Assignments, Nodes, Declarations}
Result: sequentialized SCG
1  Get direct assignments from Assignments;
2  Get PRE-function assignments from Assignments;
3  Get negated assignments from Assignments;
4  Merge direct, pre and negated assignments to mergedAssignments;
5  if assignments contains duplicates then
6      Remove all duplicates and the remaining assignment from
       mergedAssignments;
7  Remove all assignments from mergedAssignments if the assignment is used in a
   PRE-function;
8  forall the node in Nodes do
9      forall the assignment in mergedAssignments do
10         if node is Assignment then
11             if node.ValuedObject = assignment.ValuedObject then
12                 Replace assignment in node with assignment.Value;
13         if node is Conditional then
14             if assignment in node.Condition then
15                 Replace assignment in node.Condition with assignment.Value;
16         if node is OperatorExpression then
17             if assignment in node.Expression then
18                 Replace assignment in node.Expression with assignment.Value;
19 Remove replaced variables from Declarations;
```

Abbildung 5.3. Kopierpropagation Algorithmus

Die Implementierung der Kopierpropagation arbeitet wie im Konzept beschrieben und ist in Abbildung 5.3 als Pseudocode dargestellt. Zu Anfang werden die benötigten Daten gesammelt. Dazu gehören die Eins-zu-Eins-Zuweisungen, pre-Zuweisungen und Zuweisungen von Negationen von Variablen. Ebenfalls werden die statischen Werte für die Konstantenpropagation gesammelt. Diese Zuweisungen werden zusammengefasst und der eigentlichen Kopierpropagation übergeben.

Nun wird der Reihenfolge des Auftretens im Quellcode nach die Liste der Zuweisungen durchlaufen und jeweils am Punkt der Benutzung der gerade gewählten Variable ihr Term eingesetzt. Genauer gibt es eine Fallunterscheidung mit drei Fällen. Der erste Fall beläuft sich auf die Ersetzung einer Variablen innerhalb einer Zuweisung

(Assignment-Interface). In diesem Fall kann der vorherige Term der Zuweisungen einfach durch den neuen Term substituiert werden. Der zweite Fall beschränkt sich auf die Fallunterscheidungen (Conditional-Interface). Erfolgt eine Ersetzung innerhalb einer Fallunterscheidung kann die Bedingung (Condition) ersetzt werden. Der dritte und letzte Fall behandelt die Operatorausdrücke (OperatorExpression-Interface). Hierbei handelt es sich um zusammengesetzte Ausdrücke mit Verschachtelung. Diese Operatorausdrücke können sowohl in Fallunterscheidungen als auch in normalen Zuweisungen auftreten. Unabhängig von der Elternstruktur kann hier bei einer Ersetzung der alte Term aus der Liste der Subausdrücke entfernt und der neue Term eingesetzt werden. Da pre-Zuweisungen als Operatorausdrücke dargestellt werden, muss lediglich bei der Substitution innerhalb solcher Ausdrücke auf die Verwendung von pre-Variablen geachtet werden. Soll eine Variable innerhalb eines pre-Ausdrucks ersetzt werden, so unterdrückt die Implementierung dies, um eine Verschachtelung dieser Ausdrücke zu verhindern. Konstante Werte können in dieser Implementierung wie Terme verwendet werden. Die Ersetzung in den drei genannten Fällen funktioniert analog. Der Algorithmus hat eine Worst-Case-Laufzeit von $\mathcal{O}(RN)$, wobei N die Anzahl der Knoten und R die Anzahl der relevanten Variablen bezeichnet.

Der Verdeutlichung der Funktionalität soll das aus Abbildung 4.1 bekannte ABO-Modell dienen. Die Analyse der relevanten Variablen ergibt auf das Modell angewendet die Tabelle 5.4. Es ist zu erkennen, dass alle Variablen aufgenommen wurden, die auch in Abbildung 4.3 markiert wurden. Im nächsten Schritt erfolgt die Substitution,

1-zu-1	Pre	Negation
g0 = _GO	g3 = pre(g2) g7 = pre(g6)	g4_e1 = !g3 g8_e2 = !g7

Abbildung 5.4. Relevante Variablen in dem ABO-Modell - Kopierpropagation

indem jeder Knoten des seq. SCGs auf die Vorkommen der in der Tabelle aufgelisteten Variablen geprüft wird. Bei einem Fund wird die Ersetzung wie oben bereits erklärt durchgeführt. Sollte kein Fund vorliegen, so wird der Knoten nicht verändert.

Nachdem alle Variablen der Zuweisungsliste durchlaufen sind, erfolgt eine Entfernung der nicht mehr genutzten Variablen. Hierzu wird eine Verwendungsmatrix der ersetzten Variablen erstellt. Im nächsten Schritt werden nun alle Variablen aus der Variablenliste gestrichen, die eine oder mehr Verwendungen haben. Diese Verwendungen können durch den Substitutionsabbruch bei pre-Ausdrücken übrig geblieben sein. Das Ergebnis nach dem Streichen ist eine Liste aller nicht verwendeten Variablen. In einem nächsten Schritt werden diese Variablen nun aus dem Modell entfernt. Hierzu wird zuerst der Vorgänger- mit dem Nachfolgeknoten verbunden und danach die Variablendeklaration aus der Deklarationsliste des Modells entfernt. Die Eliminierung

hat eine Worst-Case-Laufzeit von $\mathcal{O}(R)$, wobei R die Anzahl der relevanten Variablen darstellt.

5.2 Variablenwiederverwendung

Um die Variablenwiederverwendung ebenfalls als eine ProductionTransformation, wie sie in Abbildung 5.1 zu sehen ist, darzustellen wird ebenfalls eine Erweiterung implementiert.

ReuseVariables
- visited : ArrayList<Node> - assignments : Iterable<AssignmentImpl> - MAX_SEARCH_DEPTH : Integer
+ getName( ) + getProcessorOptions( ) + getId( ) + getRequiredFeatureIds( ) + getNotHandlesFeatureIds( ) + transform(scg : SCGraph) - InNextPointerChain(needle : Node, nextP : Node, hard : boolean) - InNextPointerChain(needle : Node, nextP : Node, hard : boolean, hops : int) - GetSecondElem(map : TreeMap<String,Pair<Node,Node», firstEntry : Entry<String,Pair<Node,Node»)

Abbildung 5.5. ReuseVariables - UML

Die ersten sechs Methoden der in Abbildung 5.5 dargestellten Klassen haben die Funktionalität, die bereits in den vorhergehenden Kapiteln beschrieben wird. Die beiden InNextPointerChain Methoden überprüfen, ob der Knoten needle ein Nachfolgeknoten von nextP ist. Hierbei ist hard ein Schalter, der kontrolliert, ob der Knoten nextP bereits dieser Nachfolgeknoten sein kann oder nicht. Der Integerwert hops wird für den rekursiven Aufruf verwendet und zählt die bereits durchlaufenen Knoten. Hierbei kommt auch das Attribut visited ins Spiel. Innerhalb der Liste werden alle bereits durchlaufenen Knoten aufgelistet. GetSecondElem basiert auf der gerade beschriebenen Methode und sucht zu einem firstEntry aus einer Liste map eine geeignete Substitutionsvariable.

```
   Data: sequentialized SCG {Assignments, Nodes, Declarations}
   Result: sequentialized SCG
 1 Get first assignment node positions for all variables;
 2 Get last use node positions for all variables;
 3 Merge first assignment and last use node positions into varPositions;
 4 Remove all assignments from mergedAssignments if the assignment is used in a
   PRE-function;
 5 forall the relevantVar in varPositions do
 6 |   Remove relevantVar from varPositions;
 7 |   Find substitutionCandidate in varPositions with criteria:
   |   substitutionCandidate.LastUsePos < relevantVar.FirstAssignmentPos;
 8 |   if substitutionCandidate found then
 9 |   |   Change all occurrence of relevantVar to substitutionCandidate;
10 |   |   Remove substitutionCandidate from varPositions;
11 Remove replaced variables from Declarations;
```

Abbildung 5.6. Variablenwiederverwendung Algorithmus

Die Implementierung arbeitet, wie die Kopierpropagation, in drei Schritten und ist als Pseudocode in Abbildung 5.6 dargestellt. Der erste Schritt analysiert den übergebenen SCG und generiert die benötigten Datenstrukturen. Hierzu zählen die erste Zuweisung und die letzte Verwendung jeder Variablen. Ebenfalls werden Variablen, die innerhalb von pre-Funktionen verwendet werden, aus den Listen entfernt. Durch die Kenntnis der Verwendungsbereiche der Variablen kann ein Algorithmus entwickelt werden, der zu zwei gegebenen Knoten erkennt, ob der zweite Knoten ein Nachfolger des ersten Knotens ist. Dieser Algorithmus startet mit dem ersten Knoten und überprüft, ob der Knoten gleich mit dem zweiten Knoten ist. Sollte dies der Fall sein, wird wahr zurückgegeben und terminiert. Sind die Knoten ungleich, gibt es zwei Möglichkeiten weiter zu verfahren. Möglichkeit eins wird ausgeführt, wenn der erste Knoten ein normaler Knoten ist. Dann wird der Algorithmus rekursiv mit dem Nachfolgeknoten und dem zweiten Knoten aufgerufen. Möglichkeit zwei wird ausgeführt, wenn der erste Knoten eine Fallunterscheidung enthält. Der Algorithmus wird dann rekursiv sowohl mit dem If- als auch mit dem Then-Fall aufgerufen. Hierbei wird der Nachfolgeknoten wieder als erster Parameter, als erster Knoten, übergeben. Wird der letzte Knoten, also ein Knoten ohne Nachfolger, erreicht, so wird falsch zurückgeliefert und terminiert. Jeder Knoten wird in einer Liste vermerkt, um eine Doppelprüfung nach einer Fallunterscheidung zu verhindern. Knoten die bereits besucht wurden, werden nicht ein weiteres Mal geprüft.

Es wird dann falsch zurückgegeben und terminiert. Der Algorithmus unterstützt die Findung eines Substitutionskandidaten. Im zweiten Schritt wird die Liste der Variablen abgearbeitet und für jede Variable eine geeignete Ersetzung gesucht. Das Kriterium für den Substitutionskandidaten ist, dass die letzte Verwendung der zu ersetzenden Variable innerhalb eines Nachfolgeknotens des Kandidaten ist. Dies kann mit dem vorher genannten Algorithmus überprüft werden. Nachdem ein Kandidat gefunden ist, wird die Substitution durchgeführt. Kann kein Kandidat gefunden werden, wird die Variable aus der Liste entfernt. Als Abbruchkriterium wird die Größe der Liste der noch zu ersetzenden Variablen verwendet. Sobald diese leer ist endet das Suchen und Ersetzen. Der Algorithmus hat eine Worst-Case-Laufzeit von $\mathcal{O}(N^2)$, wobei N die Anzahl der Knoten darstellt. Im dritten und letzten Schritt werden die nicht mehr genutzten Variablen entfernt. Die Funktionalität ist gleich mit der Eliminierung nicht verwendeten Codes am Ende der Kopierpropagation.

Kapitel 6

Evaluation Quellcodeoptimierung

Im Laufe dieses Kapitels werden zuerst die Ziele dieser Evaluation genauer erläutert. Nachfolgend wird die Auswertung der Ergebnisse aus Modellen, die im Models-Repository[1] zu finden sind, durchgeführt. Das Models-Repository sammelt verschiedenste Modelle, praktischer sowie theoretischer Natur, an einem zentralen Punkt. Den Anschluss wird eine Diskussion über besonders gut oder schlecht optimierbare Modelle machen. Hier werden Beispiele aus den Ergebnissen des darauf folgenden Kapitels vorweg gegriffen und ihrer Besonderheiten nach analysiert und evaluiert. Danach wird eine größere, automatisch generierte Testmenge ins Auge gefasst, um das Optimierungspotential noch genauer einschätzen zu können. Ziel ist es festzustellen, ob die getätigten Optimierungen Modelle verkleinern und wie viel Größe eingespart wird. Zudem wird überprüft, ob es einen Speedup der Ausführung gibt.

Als Vergleichsdaten sollen im Ausgangsmodell die Anzahl der Deklarationen und Zuweisungen verwendet werden. Die Anzahl der Zustände wird ebenfalls durch den TestRunner aufgezeichnet, findet in der Auswertung allerdings keine Verwendung. Die Messwerte des Ausgangsmodells werden für die Analyse der High-Level-Konstrukte herangezogen. Ebenfalls interessant ist die Abhängigkeit zwischen den einzelnen Optimierungen und der Ausführungszeit. Nach der Sequenzialisierung wird die Anzahl der Knoten und ebenfalls die der Deklarationen und Zuweisungen herangezogen. Diese Rahmendaten eines Modells wurden sowohl vor als auch nach den einzelnen Optimierungen erfasst. Im Ausgangsmodell sind in den Deklarationen und Zuweisungen lediglich durch den Benutzer festgelegte Variablen in Verwendung. Während der Kompilierung zum sequenzialisierten SCG kommen die generierten Variablen hinzu.

Für die Evaluation werden nicht nur die einzelnen Optimierungen, Kopierpropagation und Variablenwiederverwendung, sondern auch Kombinationen aus beiden verwendet. Es soll ebenfalls überprüft werden, ob die Kombinationen der Optimierungen einen kleineren Quellcode erzeugen oder schneller in der Ausführung sind als nur mit einer Optimierung behandelte Modelle. Im Nachgang werden dann die absolut generierten Knoten, Deklarationen und Zuweisungen mit einer Metrik ku-

[1] http://git.rtsys.informatik.uni-kiel.de/projects/KIELER/repos/models/browse/sccharts

muliert und ihrem Wert nach geordnet. Die besagte Metrik addiert die Werte aus Knoten-, Deklarations- und Zuweisungsanzahl und dient lediglich der Vergleichbarkeit. Die Vergleichbarkeit ist durch die Metrik aus mehreren Gründen gegeben. Durch die aufgenommene Knotenanzahl wird die Komplexität des SCGs und somit auch die Komplexität des späteren Quellcodes dargestellt. Die Anzahl der Deklarationen gibt zudem eine Aussage über die Menge der enthaltenen Variablen. Da die Variablenwiederverwendung nur die Deklarationen verringern kann, wird mittels dieser Komponente die Optimierung durch die Variablenwiederverwendung erst vergleichbar. Da die Kopierpropagation auf den Zuweisungen arbeitet, ist eine Aufnahme der Komponente für die Vergleichbarkeit mit der Kopierpropagation notwendig.

Die Evaluation der Optimierungen arbeitet auf relativen Werten. Interessant ist die prozentuale Einsparung an Knoten, Deklarationen und Zuweisungen, aber auch Trends abhängig von der Modellgröße. Die gemessenen Ausführungszeiten sollen zudem Klarheit schaffen, ob die Optimierungen neben der Minimierung der Quellcodegröße auch eine beschleunigte Ausführung aufweisen. Modelle die stark von einem Trend abweichen, werden genauer untersucht.

Eine genauere Betrachtung der High-Level-Konstrukte mit der stärksten Expansion soll Klarheit schaffen, ob zu einer Verwendung in eingebetteten Systemen geraten wird. Analysiert werden lediglich minimale Modelle zu den einzelnen Extended Features. Die Modelle werden als minimal bezeichnet, da sie nur die für die Verwendung des jeweiligen Extended Features benötigten Elemente enthalten. In einigen Modellen wurde ein Feature mehrmals verwendet. Wenn dies der Fall ist, so sind mehrere Modelle eines Features mit verschiedener Quantität vorhanden. Hierbei muss zusätzlich auf die von der Transformation geleistete Funktionalität geachtet werden. Je schwieriger die Funktion zu ersetzen ist, desto schwieriger ist es diese nicht zu verwenden oder einen Workaround, der eine geringere Graphgröße hat, zu entwickeln. Die Analyse der High-Level-Konstrukte arbeitet auf absoluten Werten. Ziel ist es eine Handlungsempfehlung in der Verwendung von High-Level-Konstrukten im eingebetteten Kontext zu geben.

6.1 Ziel

Die zugrundeliegende Fragestellung der Evaluation ist, ob die Optimierungen Kopierpropagation und Variablenwiederverwendung sowie die Kombination aus den beiden Optimierungen verschiedenste Modelle optimieren können. Dabei ist wichtig herauszufinden, wie stark die Modelle optimiert werden können und ob auch die Ausführungszeit beschleunigt werden kann. Die Größe der Modelle wird anhand dreier Werte bestimmt, die Knoten-, Deklarations- und Zuweisungsanzahl. Die

Ausführungszeit wird mit Hilfe der Funktionalitäten von KIEM ermittelt.

Wie bereits beschrieben geschieht dies für real genutzte Modelle und eine große Menge an automatisch generierten Modellen aus einem von Smyth entwickelten Modellgenerator[2].

Die Daten für die Evaluation werden mit Hilfe von zwei TestRunnern erfasst. Der erste TestRunner diente der Erfassung der Modellgrößen vor und nach den einzelnen Optimierungen. Genauer wird zuerst die Modellgröße des SCCharts-Modells gemessen. Weiterführend findet eine Transformation in die Form des sequenzialisierten SCGs statt. Anschließend wird ebenfalls die Größe des Modells vermessen. Da die Optimierungen auf dem seq. SCG arbeiten, wird in den nächsten Schritten der seq. SCG mit Hilfe einer Optimierung transformiert und die Größe ein weiteres Mal ermittelt. Das Ergebnis des TestRunners ist eine Tabelle von Modellen und die zugehörigen Werte (Knoten-, Deklarations-, und Zuweisungsanzahl) des Ausgangsmodells, des seq. SCG und der einzelnen Optimierungen. Der zweite TestRunner erbt bereits die Funktionalität des KIEM TestRunners. Es ist somit möglich Modelle mit Hilfe einer Spur-Datei wiederholt gleich ablaufen zu lassen. Zu diesem Zweck werden für den seq. SCG als auch für die vier Optimierungen Execution-Dateien angelegt. Diese Dateien beinhalten alle Rahmenparameter für die Ausführung von Modellen. Die Spur(Trace)-Dateien können mit Hilfe von KIELER automatisch generiert werden. Es ist auch möglich für einen Ordner von Modellen Spurdateien zu generieren. Wichtige Zusatzinformation an dieser Stelle ist, dass die Spuren zufällig erzeugt werden. Hierzu zählt aktuell, dass für ein Modell auf Basis der möglichen Eingaben zufällige Eingabewerte generiert und in eine Trace-Datei geschrieben werden. Die Erzeugung der Ausführungszeitdaten ähnelt der der Größendaten. Der TestRunner wird mit einer Execution-Datei auf einem Ordner von Modellen gestartet. KIEM kompiliert nach dem Start der Reihenfolge nach die Modelle und führt diese zusammen mit der Trace-Datei den Rahmenparametern nach aus. Als Ergebnis wird die BenchTime, die Ausführungszeit in Millisekunden, an einen Observer übergeben und gespeichert. Dies passiert für alle Ticks der Spur-Datei. Pro Modell werden 50 Iterationen der jeweiligen Spur-Datei durchgeführt, um ein möglichst belastbares Ergebnis zu erhalten.

In einem nachfolgenden Schritt soll zudem eine Handlungsempfehlung für die Verwendung von Extended Features zu geben. Hierzu wird ebenfalls der erste genannte TestRunner verwendet um die absoluten Werte der minimalen Modelle zu generieren.

[2]Git-Commit: 83c1941 vom 10.06.2016

6.2 Quantitative Auswertung realer Modelle

Die quantitative Auswertung realer Modelle beschränkt sich auf Modelle aus den Modell-Repositories[3] des Lehrstuhls. Insgesamt werden ca. 250 Modelle unterschiedlicher Größen (5 - 1400 Knoten) betrachtet. Vor der Vermessung werden die Modelle in die Form des sequenzialisierten SCGs transformiert. Für die abschließende Ausführungszeitanalyse werden ca. 75 Modelle verwendet. Dies hängt zum einen mit der Einschränkung auf Modelle ohne Hostcode und zum anderen mit der sehr langen Simulationszeit zusammen.

Die Auswertung beginnt mit einem Vergleich der Knotenanzahl. Die Knoten werden in der Quellcodegenerierung direkt in Codezeilen übersetzt. Somit sind sie ein guter Anhaltspunkt für die Länge des generierten Quellcodes und somit auch die Effektivität der genutzten Optimierungen. Da die Einführung der Variablen nicht innerhalb der Knoten des SCGs geschieht, sondern vor der Generierung der Quellcodelogik aus den Variablendeklarationen des SCGs, kann anhand der Abbildung 6.1 nicht ermittelt werden, wie viele Variablen eingespart worden sind. Eine Aufstellung über die eingesparten Variablen ist in Abbildung 6.9 im weiteren Verlauf dieses Kapitels zu finden. Lediglich eine Aussage über die Länge der tick-Funktion lässt sich ableiten.

Die folgende Abbildung 6.1 trägt die Knotenanzahl des seq. SCG gegen die Knotenanzahl der optimierten Form auf. Es sind vier Kurven zu erkennen von denen sich jeweils zwei überlagern. Dies ist bei den Kurven Variablenwiederverwendung und Variablenwiederverwendung → Kopierpropagation, sowie Kopierpropagation und Kopierpropagation → Variablenwiederverwendung der Fall. Die Kurve der Variablenwiederverwendung liegt exakt auf der Ursprungsgeraden. Dies hängt mit der Variablenwiederverwendung zusammen. Sowohl die Variablenwiederverwendung als auch die Kombination aus Variablenwiederverwendung und nachfolgender Kopierpropagation haben keine Auswirkung auf die Anzahl der Knoten innerhalb des SCGs. Grund hierfür ist, dass die Variablenwiederverwendung zum einen keine Knoten entfernt, sondern lediglich Deklarationen. Zum anderen sorgt sie der Kopierpropagation vorgeschaltet für eine schlechtere Optimierung durch diese. Durch die Wiederverwendung der Variablenbezeichner wird der Kopierpropagation vermittelt, dass die Variablen nicht mehr dem Schema entsprechen. Demnach werden sie nicht in die Menge der optimierbaren Variablen aufgenommen. Diese Variablen werden nicht durch die Kopierpropagation optimiert und somit auch keine Knoten entfernt.

[3] http://git.rtsys.informatik.uni-kiel.de/projects/KIELER/repos/models/browse/sccharts

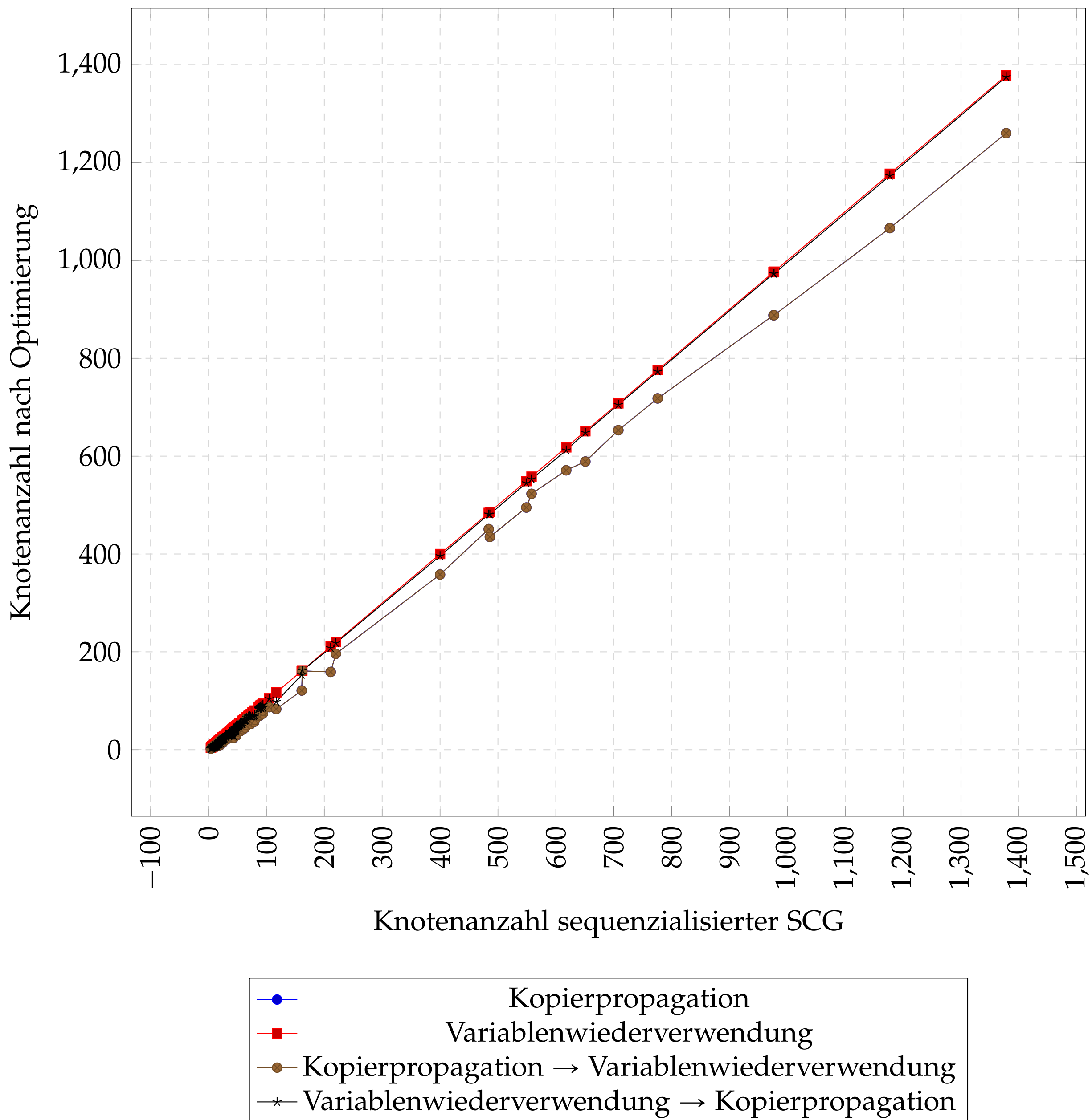


Abbildung 6.1. Vergleich Knotenanzahl nach Optimierung - (Die Kurven Variablenwiederverwendung und Variablenwiederverwendung → Kopierpropagation, sowie die Kurven Kopierpropagation und Kopierpropagation → Variablenwiederverwendung überschneiden sich.)

Abbildung 6.2 zeigt den Ausschnitt bis 50 Knoten der Abbildung 6.1 und verdeutlicht den Einfluss der Variablenwiederverwendung auf die Kopierpropagation.

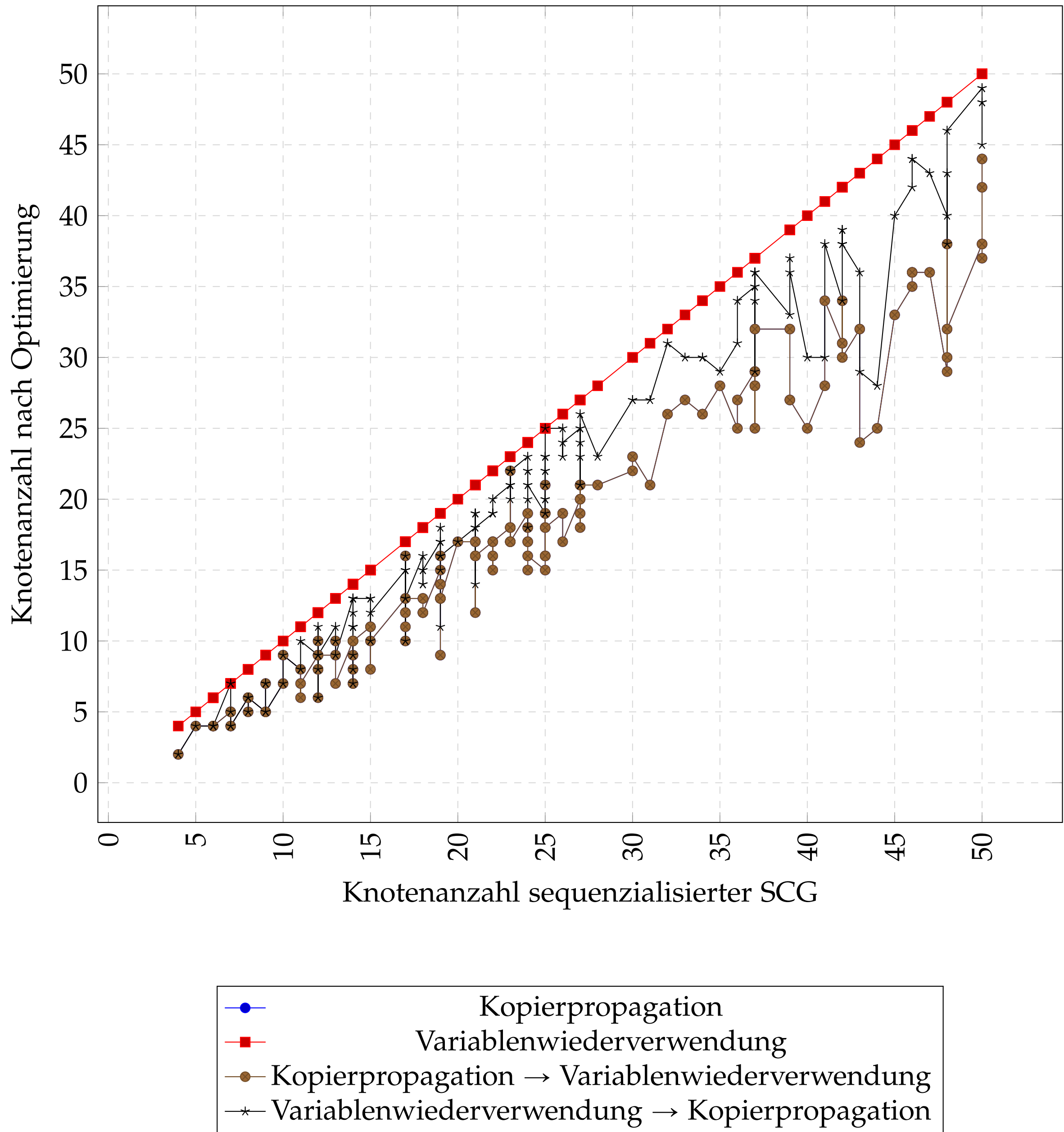


Abbildung 6.2. Vergleich Knotenanzahl (bis 50) nach Optimierung - (Die Kurven Kopierpropagation und Kopierpropagation → Variablenwiederverwendung überschneiden sich.)

Die Überlappung der Kurven der Kopierpropagation und der Kopierpropagation mit nachfolgender Variablenwiederverwendung zeigt, dass nach einer durchgeführten Kopierpropagation nur noch wenig Optimierungspotential durch die Variablenwie-

derverwendung besteht. Wird die Kombination anders herum ausgeführt, so fällt die Gesamtoptimierung geringer aus. Ausreißerwerte sind sogar gar nicht weiter optimierbar durch die Kopierpropagation. Ein Beispiel hierfür ist der Punkt (25,25) des Diagramms. Die Effekte der Variablenwiederverwendung auf die Kopierpropagation machen es notwendig auch die Knotenanzahl nach der Optimierungen durch die Variablenwiederverwendung mit in die Diagramme aufzunehmen. Des weiteren zeigt der Ausschnitt, dass auch im kleinen Maßstab ähnliche Ergebnisse erzielt werden, wie sie in der Gesamtansicht zu sehen waren. Lediglich der Zwischenbereich weist eine größere Streuung auf.

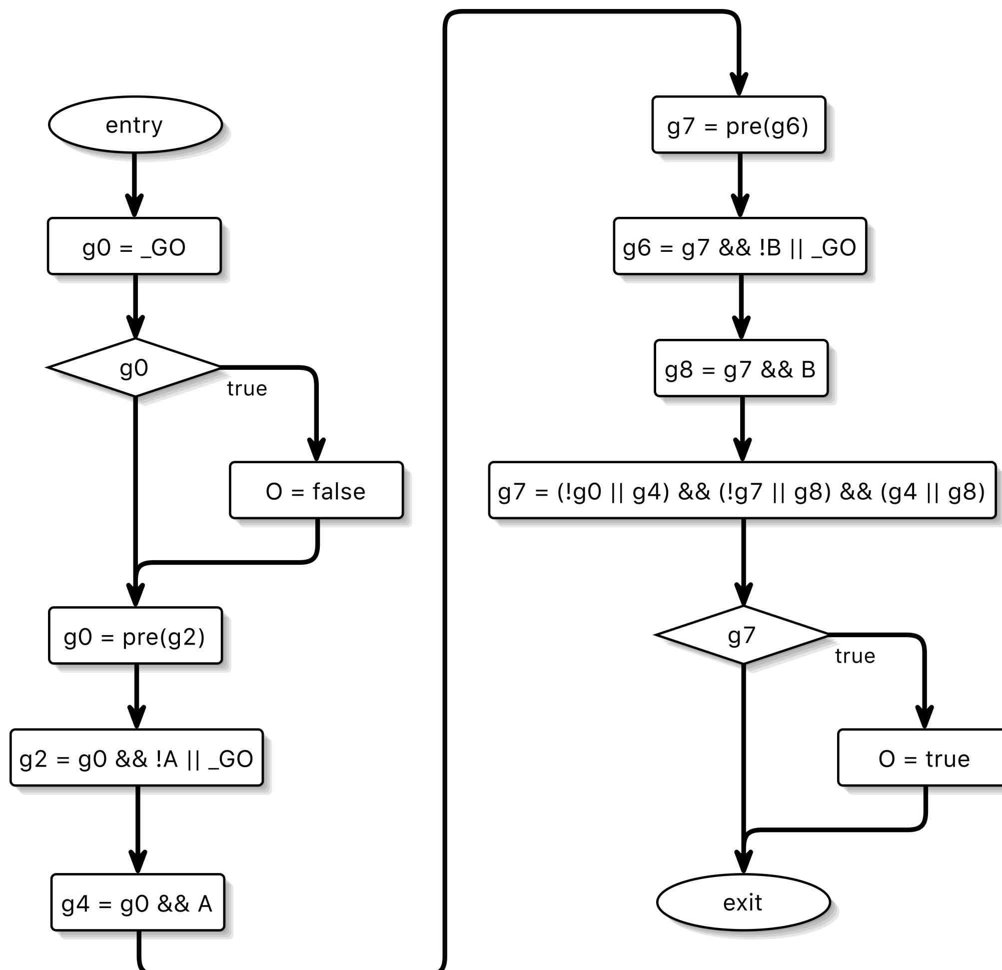


Abbildung 6.3. ABO Modell Optimierung durch Variablenwiederverwendung → Kopierpropagation

Um den Effekt auf die Kopierpropagation an einem Beispiel zu erklären, wird ein weiteres Mal das ABO-Modell herangezogen. In Abbildung 4.6 ist die Optimierung

der ABO-Modells mittels der Variablenwiederverwendung zu verzeichnen. Es fällt auf, dass anders als bei dem unoptimierten ABO-Modells aus Abbildung 4.1 einige Variablen mehrere Zuweisungen aufweisen. Zu diesen Variablen gehören genau die zwei Variablen, die durch die Variablenwiederverwendung optimiert wurden. Genauer sind dies `g0` und `g7`. Da die Kopierpropagation als Bedingung hat, dass nur Variablen propagiert werden die lediglich eine initiale Zuweisung aufzeigen, werden die Variablen `g0` und `g7` nicht mit optimiert. Abbildung 6.3 das Ergebnis aus der Variablenwiederverwendung mit nachfolgender Kopierpropagation.

Optimierung in %	Minimum	Maximum	Durchschnitt	Median
Kopierpropagation (KP)	0,61	52,63	25,73	25,58
Variablenwiederverw. (VW)	0,00	0,00	0,00	0,00
KP → VW	0,61	52,63	25,73	25,58
VW → KP	0,00	50,00	13,32	10,00

Abbildung 6.4. Minimum, Maximum, Durchschnitt und Median der prozentualen Optimierung der Knotenanzahl aus Abbildung 6.1

Bei einer weitere Betrachtung des Diagramms 6.1 fällt die untere der beiden überlappenden Kurven auf. Diese verläuft annähernd linear und zeigt an, dass die Kopierpropagation und die Kopierpropagation mit nachgeschalteter Variablenwiederverwendung Knoten einspart. Zusätzlich zu dem Diagramm sind in der Tabelle 6.4 die prozentualen Reduktion der einzelnen Optimierungen zu dem ursprünglichen SCG zu erkennen. Die Werte der Tabelle werden aus den selben Werten generiert wie das Diagramm. Die durchschnittliche Einsparung über die Menge der analysierten Modelle liegt bei der Kopierpropagation mit und ohne nachgeschalteter Variablenwiederverwendung bei ca. 25%. Der Median liegt ebenfalls bei ca. 25%. Die Abweichung sowohl zur minimalen als auch maximalen Ersparnis ist dennoch sehr groß. Die geringste Optimierung liegt bei unter 1% Knotenreduktion, wohingegen die stärkste Optimierung bei fast 53% Reduktion angesiedelt ist. Im Allgemeinen ist festzustellen, dass die Optimierbarkeit mit steigender Knotenanzahl sinkt. Mit steigender Knotenanzahl steigt auch die Komplexität der Modelle. Dieser Komplexität ist es geschuldet, dass auch die Terme innerhalb des seq. SCGs größer werden und somit nicht weiter in das Schema der verschiedenen Optimierungen passen. Zudem ist festzuhalten, dass es sich hierbei um relative Werte handelt. Bei einem Modell mit beispielsweise 1000 Knoten im seq. SCG ist eine Optimierung von 5%, in diesem Fall 50 Knoten, erfreulich. Es ist festzuhalten, dass die aktuell in KIELER verwendete Kompilerkette für große Modelle bereits optimierte SCG-Strukturen generiert.

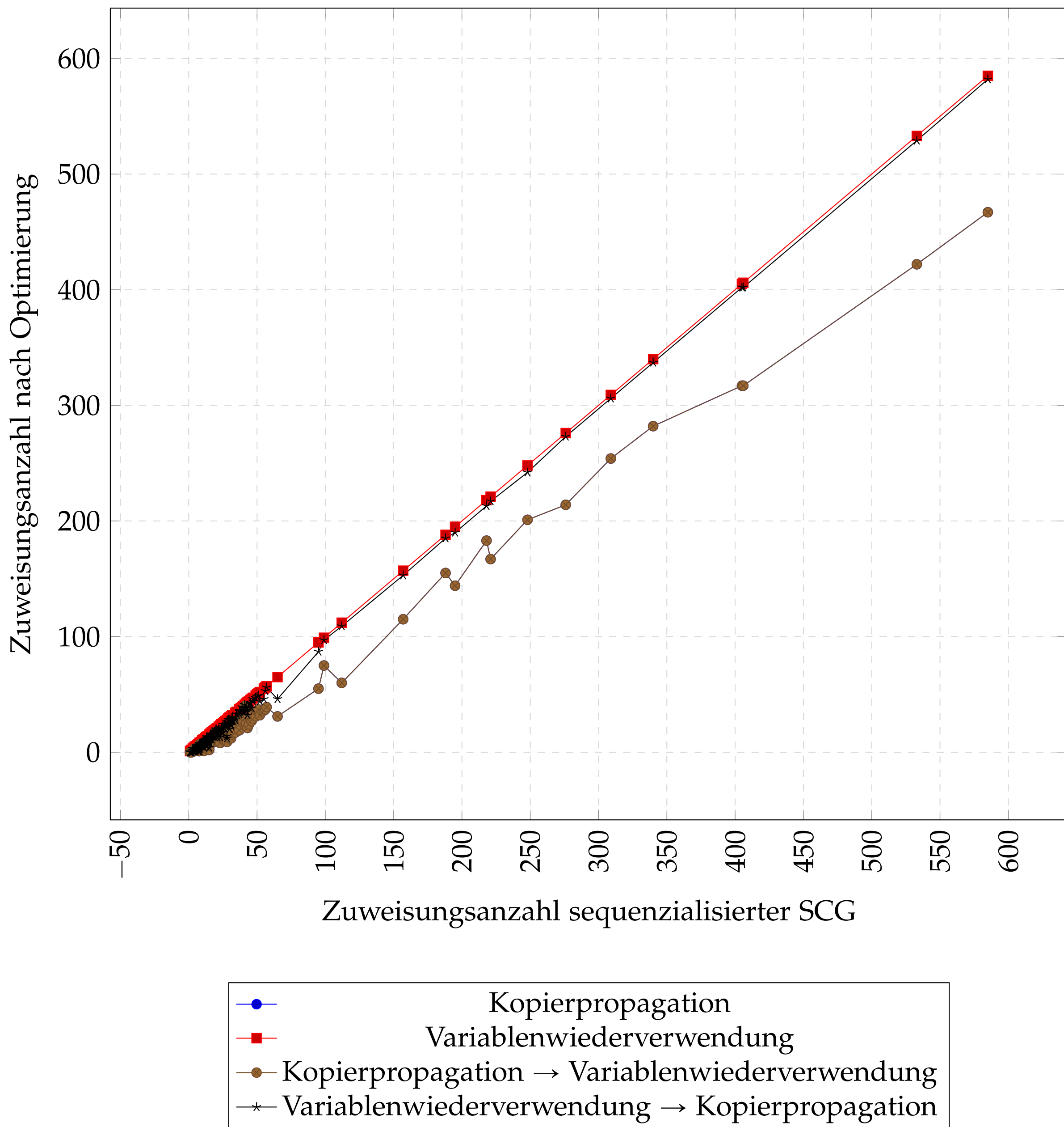


Abbildung 6.5. Vergleich Zuweisungsanzahl nach Optimierung - (Die Kurven Variablenwiederverwendung und Variablenwiederverwendung → Kopierpropagation, sowie die Kurven Kopierpropagation und Kopierpropagation → Variablenwiederverwendung überschneiden sich.)

Als Nächstes soll das Verhalten der Anzahl der Zuweisungen untersucht werden. Hierzu bietet die Abbildung 6.5 eine Übersicht. Ebenfalls wird in der Abbildung

die Anzahl der Zuweisungen des sequenzialisierten SCG gegen die Anzahl nach den Optimierungen aufgetragen. In der Tabelle 6.6 werden die prozentualen Einsparungen der Optimierungen im Vergleich zum seq. SCG aufgeführt. Ähnlich zu der Knotenanzahl überlagern sich die Kurven der Variablenwiederverwendung und der Variablenwiederverwendung mit nachfolgender Kopierpropagation sowie der Kopierpropagation und der Kopierpropagation mit nachgeschalteter Variablenwiederverwendung. Ebenfalls ist hier die Variablenwiederverwendung der ausschlaggebende Faktor. Es werden durch die Wiederverwendung auch Zuweisungen verändert, allerdings nicht überflüssig gemacht. Es ändert sich lediglich der Name der Zuweisung. Die Kopierpropagation fällt bei den Zuweisungen als einzige Optimierung ins Gewicht. Die obere Kurve (auf der Ursprungsgerade) beinhaltet die Optimierungen der Variablenwiederverwendung, welche keine Auswirkung auf die Anzahl der Zuweisungen hat. Knapp unterhalb der Kurve sind die Werte der Kopierpropagation mit vorgeschalteter Variablenwiederverwendung zu finden. Es ist zu beobachten, dass das Variablenrecycling einen negativen Einfluss auf die Optimierung durch die Kopierpropagation hat. Im Durchschnitt werden in dieser Optimierung lediglich 24% Zuweisungen eingespart. Der Median liegt bei ca. 17%. Vergleicht man vorherige Optimierung mit der der Kopierpropagation nachgeschalteten Variablenwiederverwendung, so ist bereits in der Abbildung 6.5 zu erkennen, dass eine größere Anzahl an Zuweisungen eingespart wird. Im Durchschnitt als auch Median werden ca. 45% der Zuweisungen aus den SCGs entfernt. Bei sehr kleinen und simplen Modellen können sogar 100% der Zuweisungen eingespart werden. Die Auswertung der Terme erfolgt dann direkt in den Bedingungen der Fallunterscheidungen. Im Minimum erfolgt eine Ersparnis von ca. 2%.

Optimierung in %	Minimum	Maximum	Durchschnitt	Median
Kopierpropagation (KP)	2,43	100,00	46,44	45,45
Variablenwiederverw. (VW)	0,00	0,00	0,00	0,00
KP → VW	2,43	100,00	46,44	45,45
VW → KP	0	100,00	24,48	16,66

Abbildung 6.6. Minimum, Maximum, Durchschnitt und Median der prozentualen Optimierung der Zuweisungsanzahl aus Abbildung 6.5

Wird nun Abbildung 6.7 betrachtet, so ist ein Ausschnitt der Abbildung 6.5 zu erkennen. Dieser Ausschnitt beschränkt sich auf die sequenzialsierten SCG-Modelle mit einer maximalen Anzahl von 50 Zuweisungen im unoptimierten Zustand. Wie auch in dem Übersichtsdiagramm ist zu verzeichnen, dass die Variablenwiederverwendung alleine keine Optimierung der Zuweisungszahl durchführt.

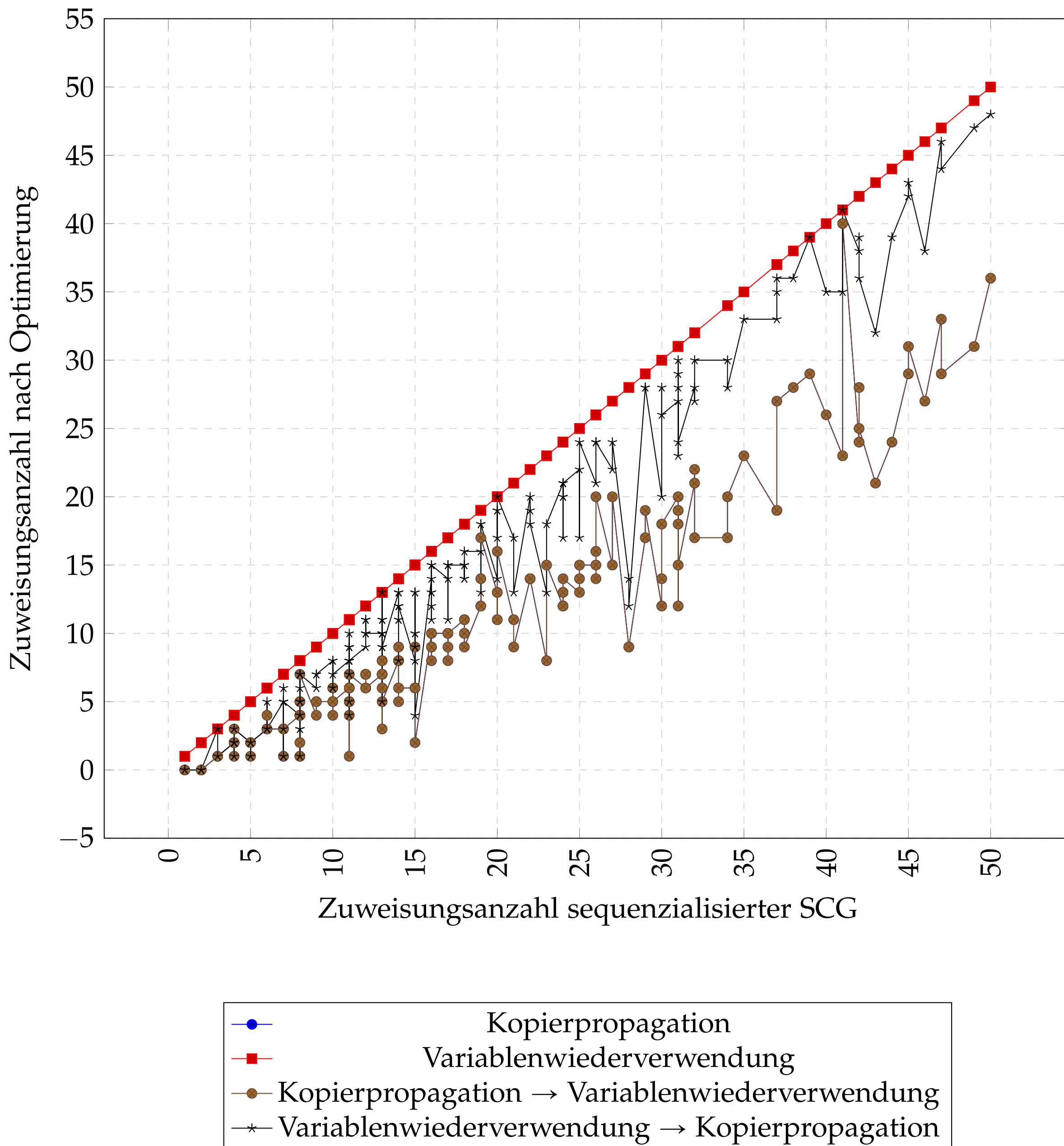


Abbildung 6.7. Vergleich Zuweisungsanzahl (bis 50) nach Optimierung - (Überschneidung der Kurven Kopierpropagation und Kopierpropagation → Variablenwiederverwendung)

Wird allerdings die Kopierpropagation hinzugezogen ändert sich das Bild. Wird diese auf die vorher mit der Variablenwiederverwendung behandelten Modelle angewendet, so können bei fast allen Modellen Optimierungen der Zuweisungsanzahl

vorgenommen werden. Wie bei der Evaluation der Knotenzahl ist die Optimierung durch die Variablenwiederverwendung mit nachgeschalteter Kopierpropagation nicht so stark wie die reine Kopierpropagation. Hierbei kann ebenfalls mit der Argumentation zur Abbildung 6.3 gearbeitet werden.

Angeschlossen an die Auswertung der Zuweisungsanzahl folgt nun die Evaluation der Deklarationsanzahl. Ähnlich zu den vorhergehenden Diagrammen wird die Anzahl der Deklarationen des seq. SCG der der Optimierungen in Abbildung 6.9 gegenübergestellt. Zu erkennen sind erneut vier Kurven von denen sich in diesem Fall zwei überschatten. Dieses Mal überlappen sich die Variablenwiederverwendung und die der Kopierpropagation vorgeschaltete Variablenwiederverwendung. In Tabelle 6.8 werden zudem die prozentualen Reduktionen der einzelnen Optimierungen zu dem seq. SCG dargestellt.

Optimierung in %	Minimum	Maximum	Durchschnitt	Median
Kopierpropagation (KP)	1,02	66,66	28,22	29,26
Variablenwiederverw. (VW)	0,00	28,85	13,94	14,63
KP → VW	12,24	66,66	32,12	33,33
VW → KP	10,20	66,66	27,86	28,57

Abbildung 6.8. Minimum, Maximum, Durchschnitt und Median der prozentualen Optimierung der Deklarationsanzahl aus Abbildung 6.9

Angefangen mit der obersten Kurve ist festzustellen, dass die Kopierpropagation durchweg unterhalb der Ursprungsgerade verläuft. Im Median wird eine Reduktion von 29% erreicht. Der für Ausreißer anfällige Durchschnitt beläuft sich ebenfalls auf 29% Deklarationsersparnis. Die nächsttiefere Kurve (Variablenwiederverwendung und Variablenwiederverwendung → Kopierpropagation) lässt vermuten, dass eine bessere Optimierung erzielt wird. Hier trügt der erste Eindruck. Lediglich bei großen Modellen wird durch das Variablenrecycling eine höhere Anzahl an Deklarationen entfernt als durch die Kopierpropagation. Wird in diesem Kontext die Abbildung 6.10 betrachtet, so erhärtet sich die Aussage. Die Variablenwiederverwendung ist bei Modellen bis 50 Deklarationen die ineffizienteste der vier vorgestellten Optimierungen. Lediglich einige wenige Ausreißer der Kopierpropagation optimieren das betrachtete Modell weniger gut. Im Durchschnitt bewegt sich die Optimierung durch die Variablenwiederverwendung bei ungefähr 14%. Diesen Wert nimmt auch der Median an, somit eine deutlich geringere Optimierung als durch die Kopierpropagation. Arbeiten nun beide Optimierungen in einer Reihenfolge zusammen, in der keine der beiden Optimierungen negativ beeinflusst wird, so fallen die Stärken beider zusammen.

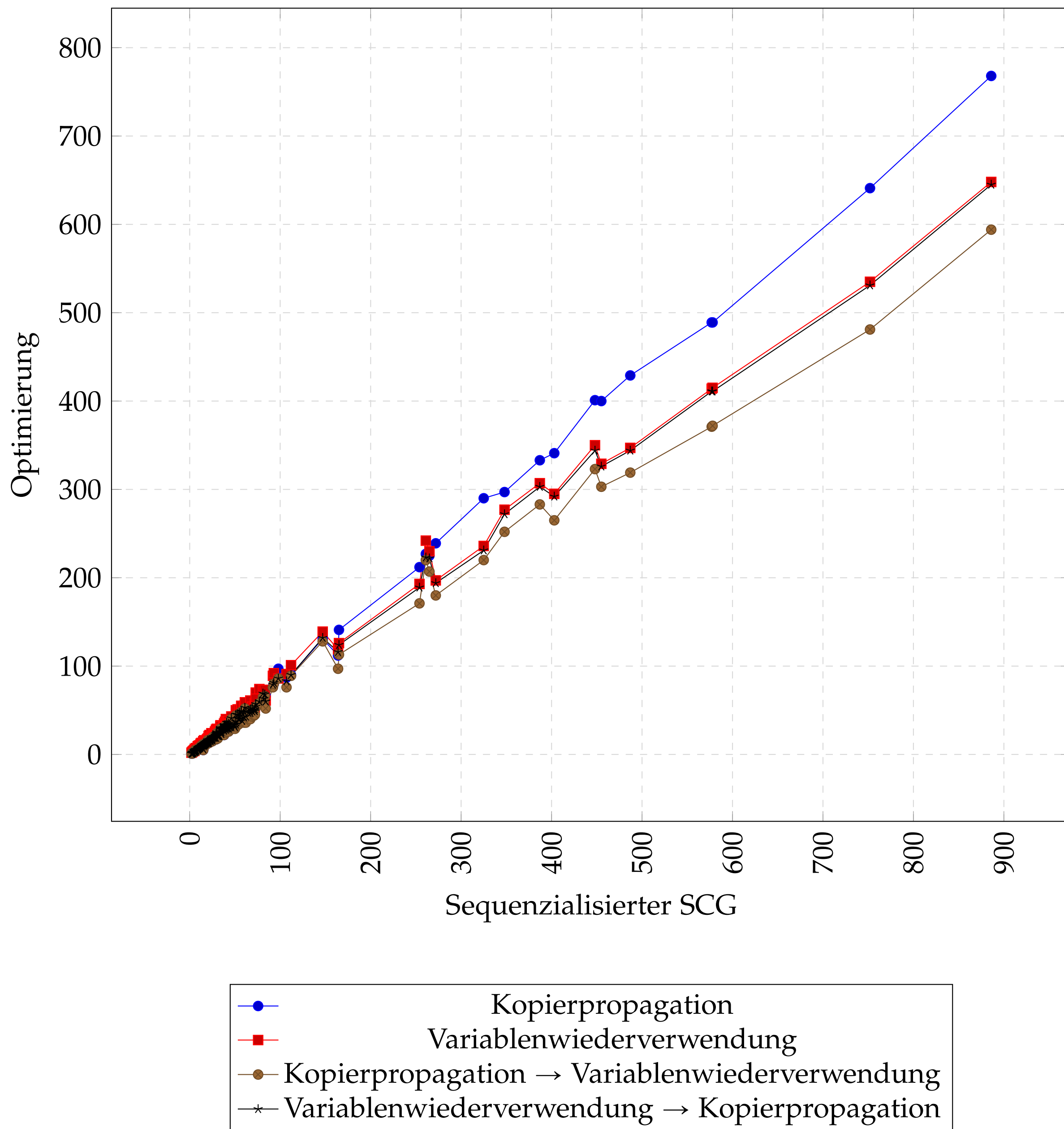


Abbildung 6.9. Vergleich Deklarationsanzahl nach Optimierung - (Die Kurven Variablenwiederverwendung und Variablenwiederverwendung → Kopierpropagation überschneiden sich.)

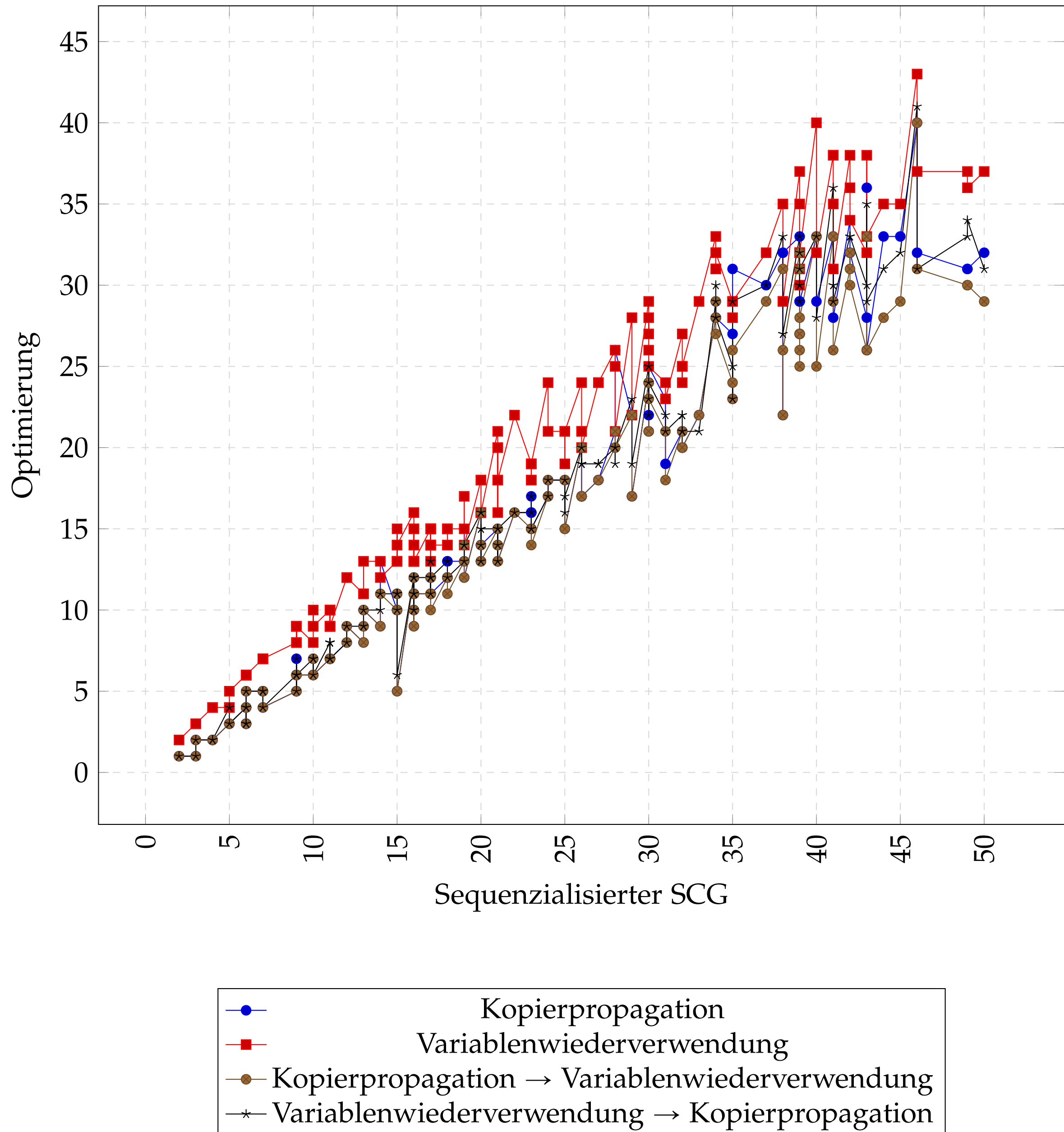


Abbildung 6.10. Vergleich Deklarationsanzahl (bis 50) nach Optimierung

Dies ist bei der Ausführung der Kopierpropagation mit anschließender Variablenwiederverwendung der Fall. Es ist eine Optimierung von durchschnittlich 33% zu verzeichnen. Bei der Deklarationsanzahl ist ebenfalls zu erkennen, dass die Optimierung für große Modelle prozentual gesehen weniger gut arbeitet. Für sehr kleine

Modelle ist allerdings eine Reduktion bis zu 66% möglich. Zusammenfassend ist zu der Größenreduktion zu sagen, dass die Kopierpropagation mit anschließendem Variablenrecycling die besten Werte liefert. Ebenfalls sind kleine Modelle relativ betrachtet besser zu optimieren als ihre großen Vertreter. Ein Grund dafür ist, dass die aktuelle Kompilerkette bei großen Modellen mehr Möglichkeiten der Zusammenfassung von Termen hat. Dies fällt bereits bei dem Vergleich des ABO-Modells aus Abbildung 1.5 mit dem noch folgenden ThreeRegionsNormalTermination-Modell aus Abbildung 6.19 auf. Die sequenzialisierten SCGs der beiden Modelle, dargestellt in Abbildung 4.1 (ABO-Modell) und Abbildung 6.19 (ThreeRegionsNormalTermination), zeigen, dass bei vielen simplen Übergängen, wie es bei ThreeRegionsNormalTermination der Fall ist, sehr viel simpler Code der Form `g3 = pre(_GO)` oder `g3_e1 = !g3` generiert wird. Werden, wie beim ABO-Modell, Eingangssignale hinzugenommen, so werden die generierten Terme komplexer. Ein Beispiel hierfür ist `g2 = _GO || g3 && !A`. Kleine Modelle können zwar ebenfalls komplexe Strukturen enthalten. Die Zuweisung der Variablen `g8` aus ThreeRegionsNormalTermination ist ein Beispiel hierfür. Größere Modelle weisen somit typischerweise mehr Komplexität auf die sich nicht in simplen booleschen Termen ausdrücken lässt.

Die Verringerung der Größe der SCGs spiegelt sich auch in der Größe des kompilierten Java-Bytecodes wieder. Für die vorher optimierten Modelle zeigt Abbildung 6.11 die Größe der generierten class-Dateien der optimierten Modelle aufgetragen gegen die Bytecodegröße der unoptimierten Modelle. Die Abbildungen 6.11 und 6.12 zeigen, dass keine Optimierung einen größeren Bytecode erzeugen als die Basislinie. Ebenfalls ist zu sehen, dass die Kopierpropagation mit anschließender Variablenwiederverwendung die besten Ergebnisse liefert. Die reine Variablenwiederverwendung liefert durchweg die niedrigste Optimierung der Dateigröße. Vermutlich optimiert an dieser Stelle der Java-Kompiler (javac) die verwendeten Variablen bereits, sodass die in dieser Arbeit implementierte Optimierung der Variablenwiederverwendung lediglich Optimierungsarbeit vorweg nimmt.

Die Kurven der Optimierungen Kopierpropagation und Variablenwiederverwendung → Kopierpropagation bewegen sich zwischen den Kurven der Variablenwiederverwendung und der Kopierpropagation mit nachfolgender Variablenwiederverwendung. Es fällt auf, dass die Variablenwiederverwendung, wenn sie der Kopierpropagation vorgeschaltet ist, auch die Effektivität der Kopierpropagation auf die Bytecodegröße negativ beeinflusst.

Tabelle 6.13 zeigt die minimale, maximale, durchnittliche und mediane prozentuale Verkleinerung durch die einzelnen Optimierungen. Wie bereits aus dem Diagramm zu entnehmen ist, wirkt sich keine Optimierung negativ auf die Bytecodegröße aus, dies ist der Spalte Minimum zu entnehmen. In der Spalte Maximum ist zu erfassen, dass bis auf die Kopierpropagation mit nachfolgender Variblenwiederverwendung

alle Optimierungen bis zu 43% an Bytecodegröße einsparen. Die Kopierpropagation → Variablenwiederverwendung spart sogar bis zu 49%. Ein Blick auf den Median und den Durchschnitt zeigt, dass die Maximalwerte starke Ausreißer sind. Im Median kann keine Optimierung die Größe um mehr als zehn Prozent verringern.

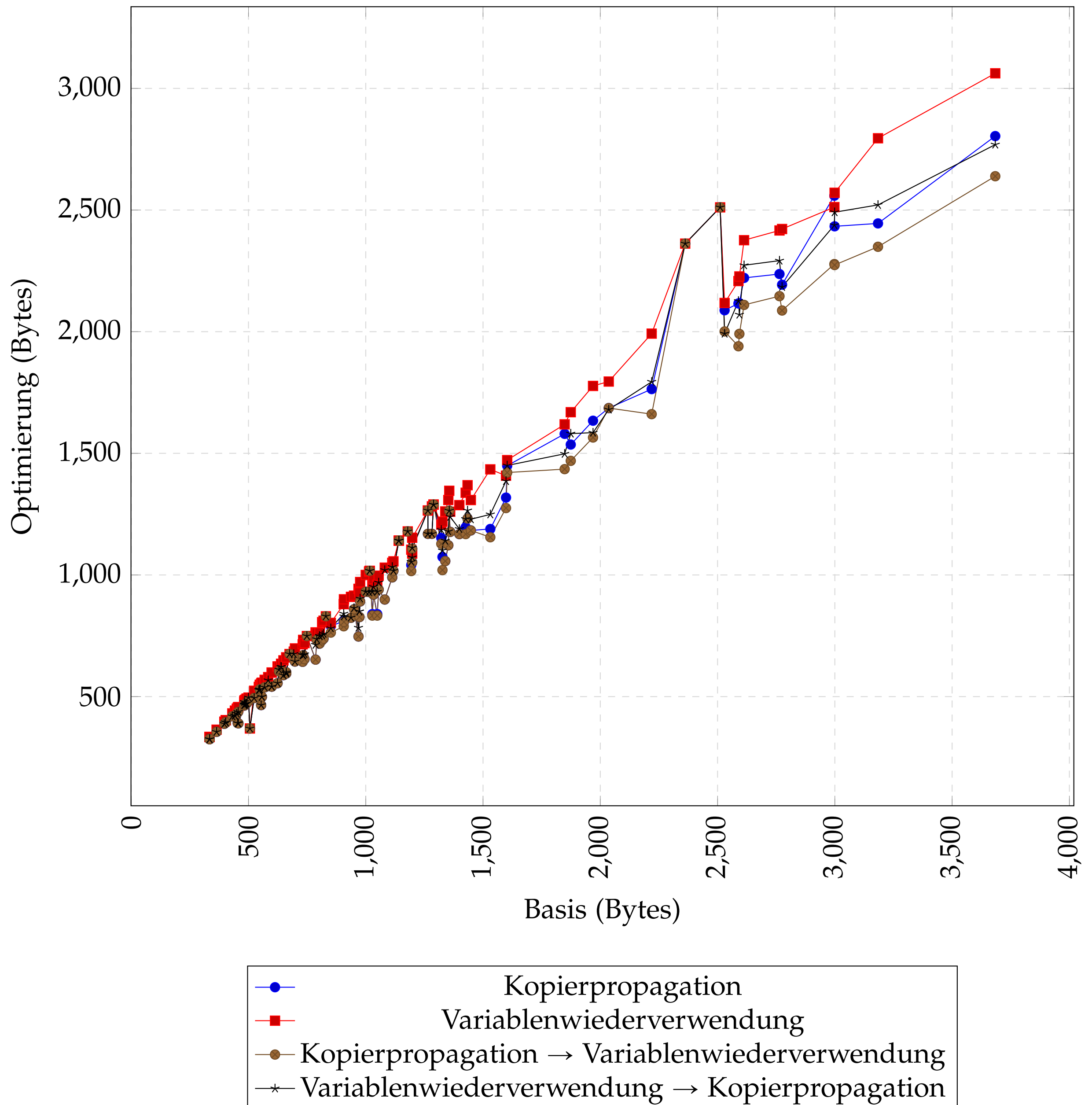


Abbildung 6.11. Vergleich Java-Bytecodegröße nach Optimierung

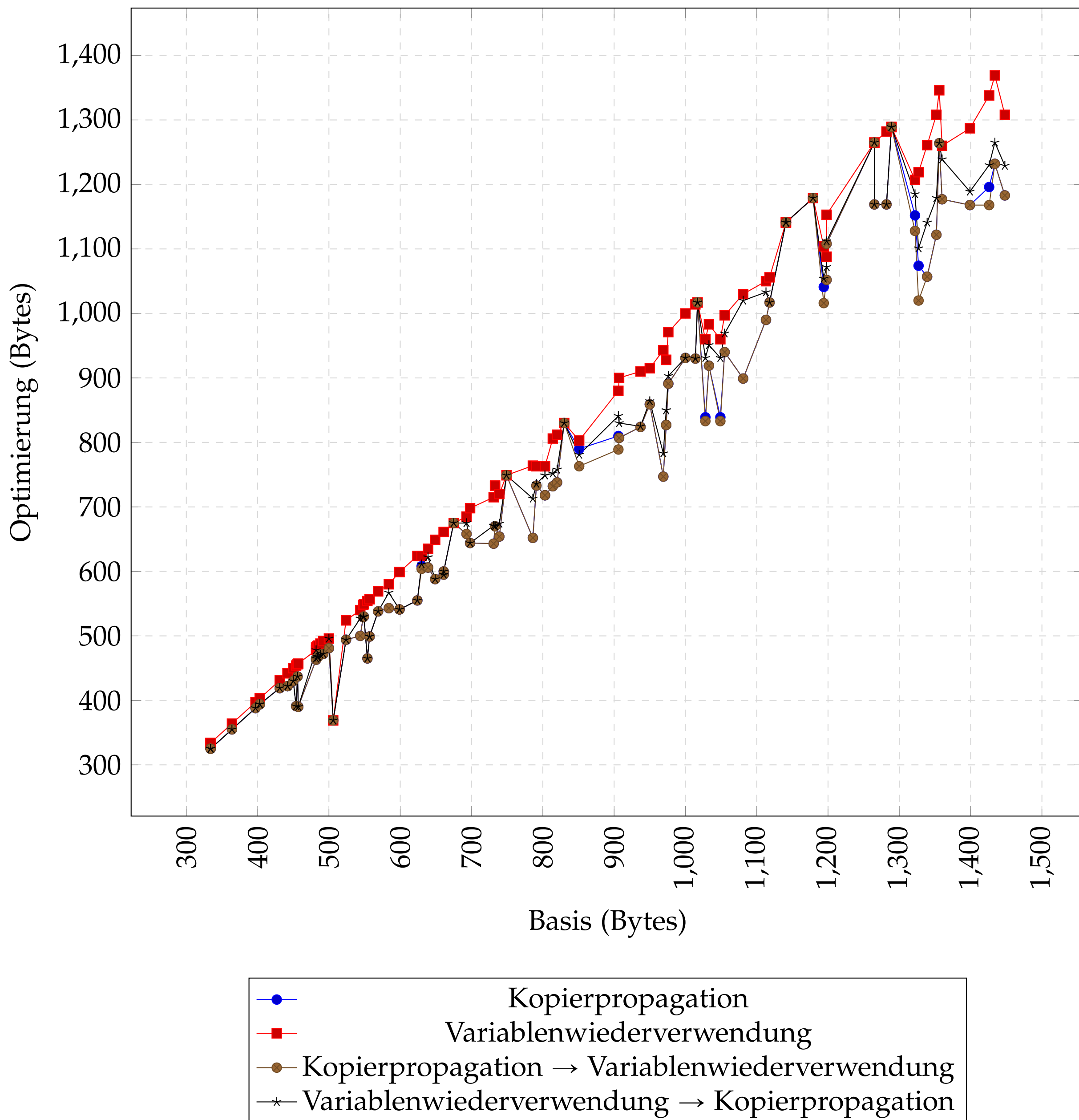


Abbildung 6.12. Vergleich Java-Bytecodegröße nach Optimierung bis 1500 Byte Ausgangsgröße

Wie bereits aus den Diagrammen 6.11 und 6.12 zu entnehmen ist, hat die reine Variablenwiederverwendung die durchschnittliche geringste Optimierung. Die Verringerung des Bytecodes beläuft sich im Schnitt auf ungefähr vier Prozent.

Optimierung in %	Minimum	Maximum	Durchschnitt	Median
Kopierpropagation (KP)	0,00	42,69	9,73	9,22
Variablenwiederverw. (VW)	0,00	42,69	4,07	0,68
KP → VW	0,00	48,79	10,65	9,39
VW → KP	0,00	42,69	8,73	7,73

Abbildung 6.13. Minimum, Maximum, Durchschnitt und Median der prozentualen Optimierung der Bytecodegröße aus Abbildung 6.11

Im Median fällt das Ergebnis noch geringer aus. Hier sind es gerade einmal 0,68% Reduktion. Im Median als auch im Durchschnitt ist die Kopierpropagation mit nachfolgender Variablenwiederverwendung die Optimierung mit den besten Ergebnissen. Im Durchschnitt erfolgt eine Optimierung des Bytecodes von 10,65%. Eine Reduktion von 9,39% ist im Median zu verzeichnen.

Zusammengefasst bieten alle Optimierungen eine Optimierung der Bytecodegröße. Wie auch bei den anderen gewählten Messwerten bietet die kombinierte Optimierung KP → VW die besten Ergebnisse. Ebenfalls verzeichnet diese die größten positiven Ausreißer.

Zum Ende der quantitativen Evaluation realer Modelle wird die Ausführungszeit verglichen. Typischerweise sind Ausführungszeitdaten systemabhängig und weisen, je nach Auslastung des Systems durch andere Prozesse, große Streuungen auf. Um diese Streuung herauszufiltern werden mehrere Durchläufe der Modelle simuliert. Für jedes Modell werden 50 Durchläufe mit Spur(Trace)-Dateien von 100 Ticks absolviert. Pro Durchlauf werden die schnellsten zehn Prozent der Zeiten zur Durchschnittsbildung herangezogen. Weiterhin wird über alle Durchläufe eines Modells der Durchschnitt über die schnellsten zehn Prozent gebildet. Die Daten der Ausführungszeiten werden mit dem in Kapitel 3.1.5 vorgestellten KIEM ermittelt. Es handelt sich dabei um die Ausführungszeit als C-Programm. Anschließend werden die Werte der Optimierungen ins Verhältnis zu den Ausführungszeitwerten des seq. SCGs gesetzt. Grafisch dargestellt werden die Werte in Abbildung 6.14 in einem Boxplot, um Ausreißer besser erkennen zu können und somit diese differenziert von den nah beieinander liegenden Werten zu betrachten. Negative prozentuale Werte bedeuten eine langsamere Ausführung, positive Werte einen Speedup.

Zu erkennen ist, dass alle Optimierungen sowohl im Durchschnitt als auch im Median die Ausführung beschleunigen. Ebenfalls ist eine große Streuung zu verzeichnen.

Weiterhin ist zu erkennen, dass die Variablenwiederverwendung (VW in der Abbildung 6.14) die beste Beschleunigung erreicht. Die Hälfte der Ausführungszeiten ist zwischen 2% und 7% schneller. Bis auf die Kopierpropagation (KP) bewegen sich alle

Optimierungen mit ihren Quartilen im positiven Bereich, also einer Beschleunigung.

Die Kopierpropagation hingegen deckt einen Bereich von -3% bis 5% ab. Im Median verlangsamt die Kopierpropagation somit die Ausführungszeit. Die kombinierten Ausführungen der einzelnen Optimierungen bewegen sich, relativ zu dem Verhältnis der Kopierpropagation und der Variablenwiederverwedung, eng beieinander.

Zusammenfassend ist festzuhalten, dass die Kopierpropagation, obwohl die Codegröße stark verringert wird, nicht zwingend eine beschleunigte Ausführung erzielt. Dies liegt vermutlich an der angeführten Mehrfachauswertung der Negationen, welche durch die Kopierpropagation erzeugt werden.

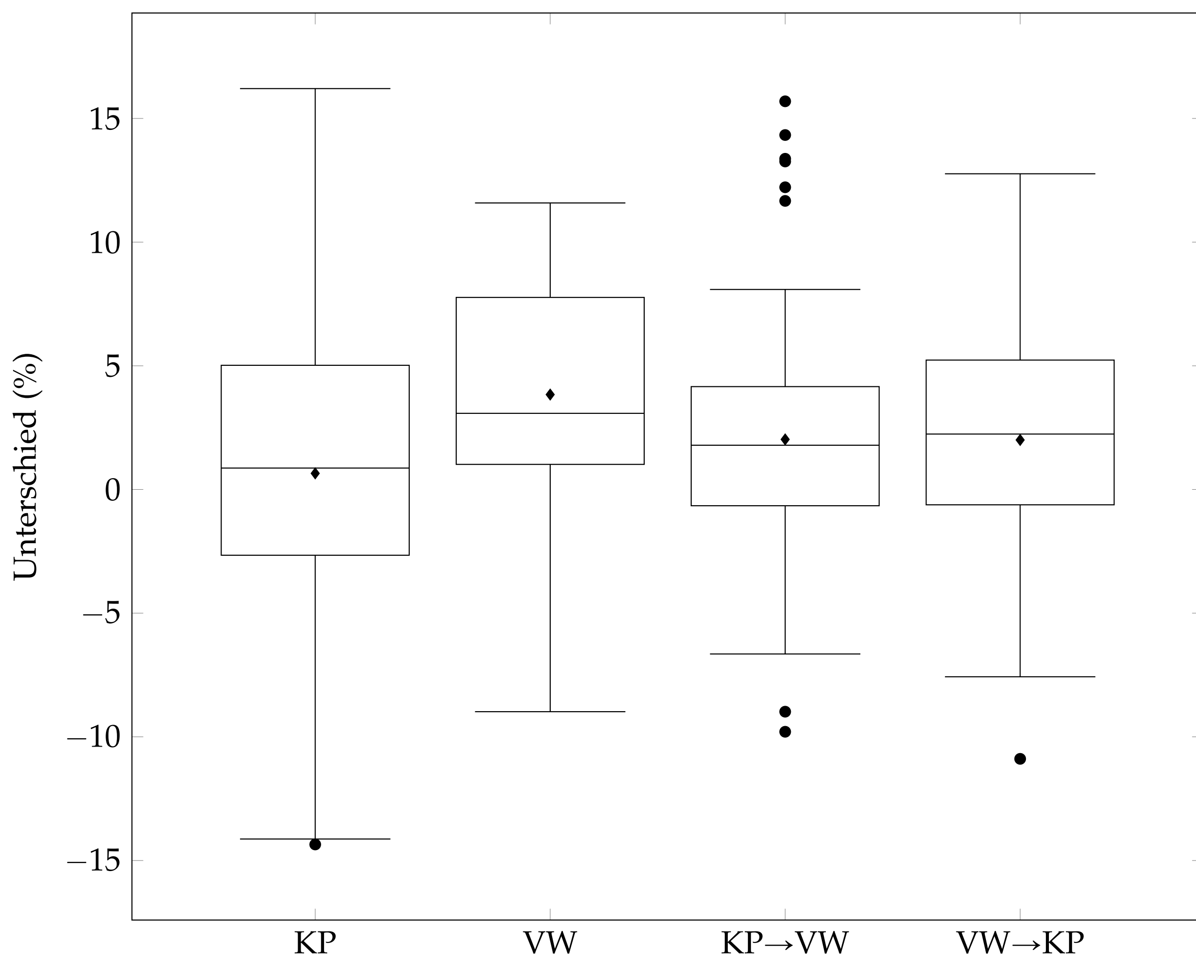


Abbildung 6.14. Vergleich Ausführungszeiten realer Modelle - (Prozentualer Unterschied der Ausführungszeit des optimierten SCGs zu dem unoptimierten SCG - KP: Kopierpropagation, VW: Variablenwiederverwendung)

Das Variablenrecycling hingegen sorgt in den meisten Fällen für eine Beschleunigung. Dies liegt an der zeitlichen Nähe der Zugriffe auf Variablen gleichen Namens. Im besten Fall befindet sich die Variable, auf die gerade zugegriffen werden soll, bereits im Cache. Dieser Zugriff kann schneller durchgeführt werden, als eine Kopie neuer Variablen aus einer höheren Speicherebene in den Cache oder die Register.

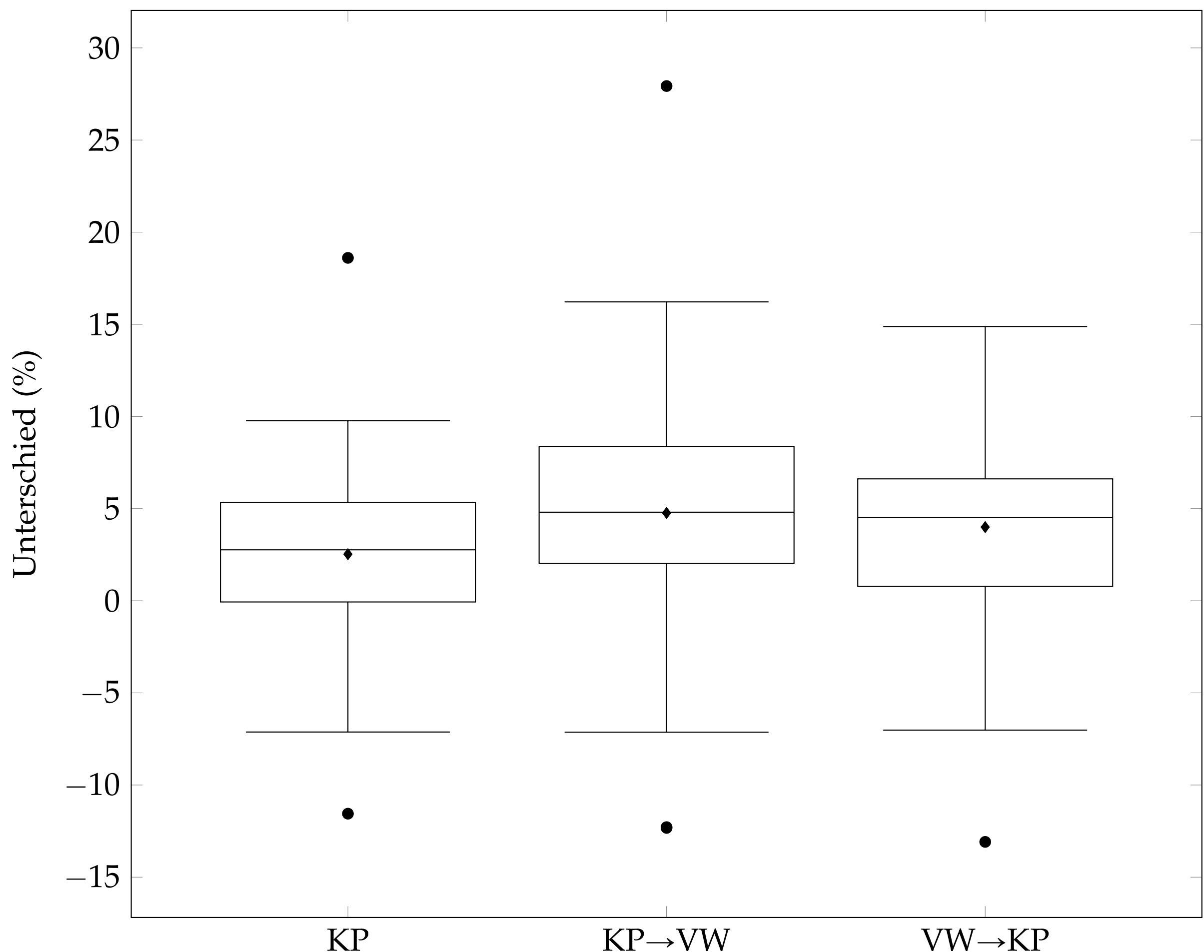


Abbildung 6.15. Vergleich Ausführungszeiten realer Modelle (Kopierpropagation ohne negierte Terme) - (Prozentualer Unterschied der Ausführungszeit des optimierten SCGs zu dem unoptimierten SCG - KP: Kopierpropagation, VW: Variablenwiederverwendung)

Der Überprüfung der These, dass die Mehrfachauswertung der Negationen, welche durch die Kopierpropagation generiert werden, die Ausführungszeit negativ beeinflusst, dient die Abbildung 6.15. Die Daten wurden mit einer angepassten Version der vorgestellten Kopierpropagation erstellt. Die Kopierpropagation ignoriert

negierte Terme bei der Wahl der Substitutionskandidaten.

Der Aufbau der Abbildung 6.15 ähnelt der aus Abbildung 6.14. Lediglich die Variablenwiederverwendung als eigenständige Optimierung ist nicht wiederzufinden, da diese durch die Änderungen an der Kopierpropagation nicht beeinflusst wird. Es ist zu erkennen, dass die Variablenwiederverwendung sich weiterhin positiv auf die Ausführungszeit auswirkt. Ebenfalls ist zu verzeichnen, dass alle Optimierungen mit der Hälfte ihrer Messwerte über der Null-Prozent-Linie liegen und somit die Ausführung beschleunigen. Dies ist in Abbildung 6.14 lediglich bei der reinen Variablenwiederverwendung zu sehen. Weiterhin fällt auf, dass die negativen Ausreißer, also die verlangsamte Ausführung, ähnlich zu den negativen Ausreißern in Abbildung 6.14. Die positiven Ausreißer hingegen verzeichnen deutlich bessere Ergebnisse. Generell sind die Ausführungszeitunterschiede näher gepackt als bei der Ausführung mit Propagation negierter Terme. Es ist festzuhalten, dass die Kopierpropagation ohne die Propagation von negierten Termen im Vergleich zu der Kopierpropagation mit der Propagation von negierten Termen eine bessere Ausführungszeitoptimierung ermöglicht.

6.3 Ergebnisse ausgewählter Beispiele

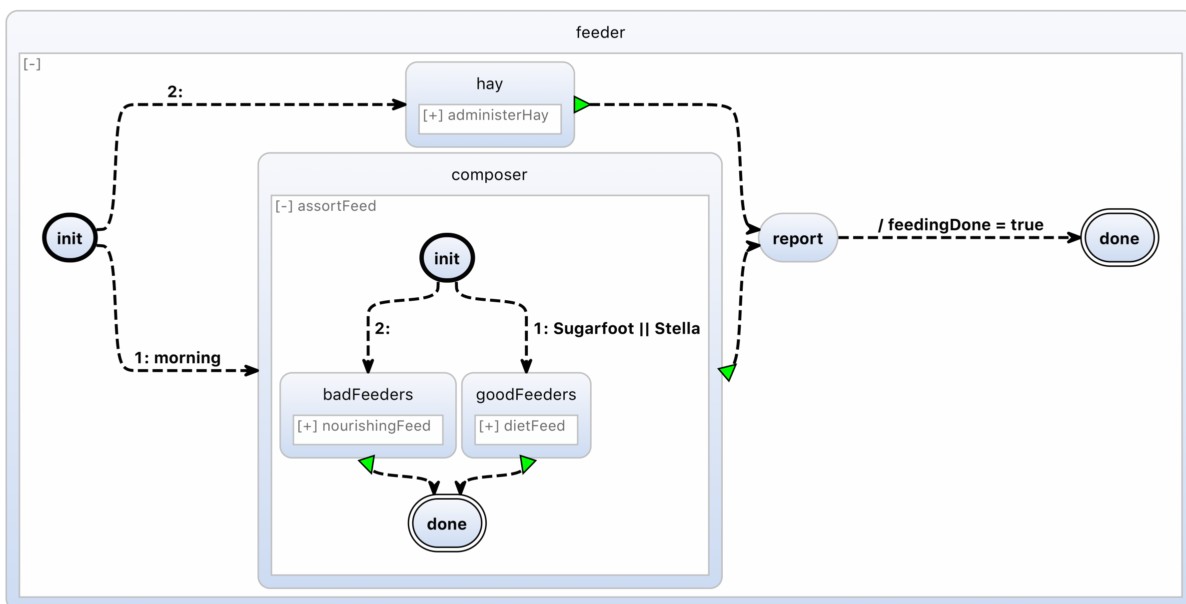


Abbildung 6.16. Beispiel 1 - SCChart

Die in diesem Kapitel vorgestellten Modelle sind Ausreißer der quantitativen Auswertung realer Modelle. Dies heißt, dass besonders gut aber auch besonders schlecht optimierbare Modelle diskutiert werden. Während der Auswertung sind vier Modelle (zwei gut und zwei schlecht optimierbar) auffällig geworden. In welchem Kontext die betrachteten Modelle auffällig sind, wird während der Beschreibung der Modelle mit aufgenommen.

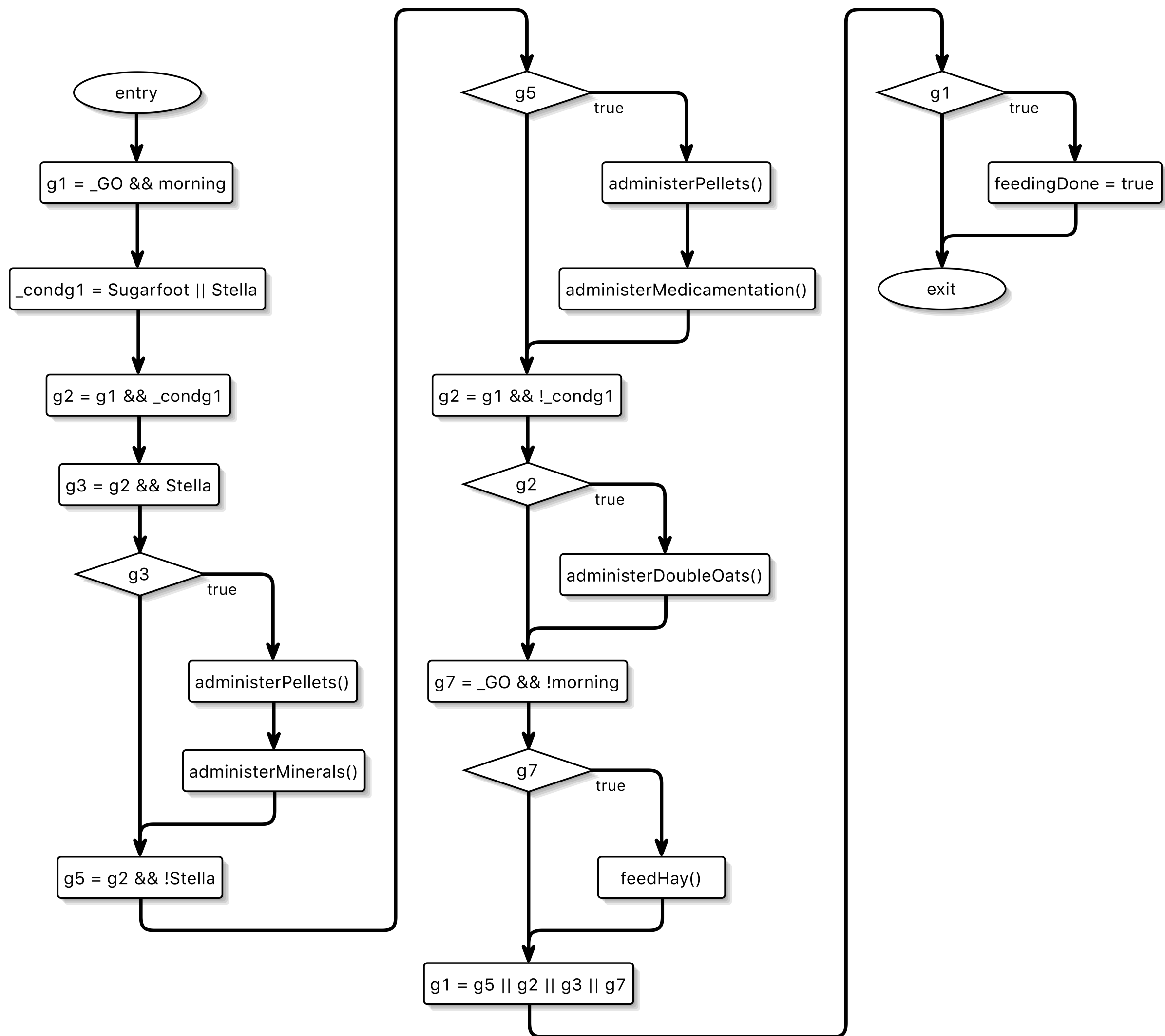


Abbildung 6.17. Beispiel 1 - Kopierpropagation → Variablenwiederverwendung

Angefangen mit den schlecht optimierbaren Modellen werden zwei Modelle mit teils vielen Hostcode-Aufrufen vorgestellt. Die schlechte Optimierbarkeit hängt direkt mit der Anzahl der eingebetteten Hostcode-Aufrufe zusammen. Diese werden nicht

optimiert und bleiben somit als Knoten im SCG enthalten. Würden Hostcodeaufrufe aus der Größenbestimmung herausgefiltert werden so wäre der prozentuale Gewinn durch die Optimierung besser.

Das Beispiel in Abbildung 6.16 stellt das erste Negativbeispiel dar. Beispiel 1 zeigt mehrere Makrozustände. Diese sind in der Abbildung minimiert, sodass die Zustandsnamen noch zu erkennen sind. Die Makrozustände enthalten Subzustände und Transitionen mit Hostcodeaufrufen. Das Modell wurde zuerst mit der Kopierpropagation und dann mit der Variablenwiederverwendung optimiert. Das Ergebnis ist in Abbildung 6.17 dargestellt. Lediglich eine Optimierung von ca. 8% Modellgröße, relativ zum sequenzialisieren SCG, konnte erreicht werden. Das Modell enthält viele Hostcodeaufrufe, die nicht optimiert werden können. Jeder Hostcodeaufruf wird im SCG als eigener Knoten abgebildet.

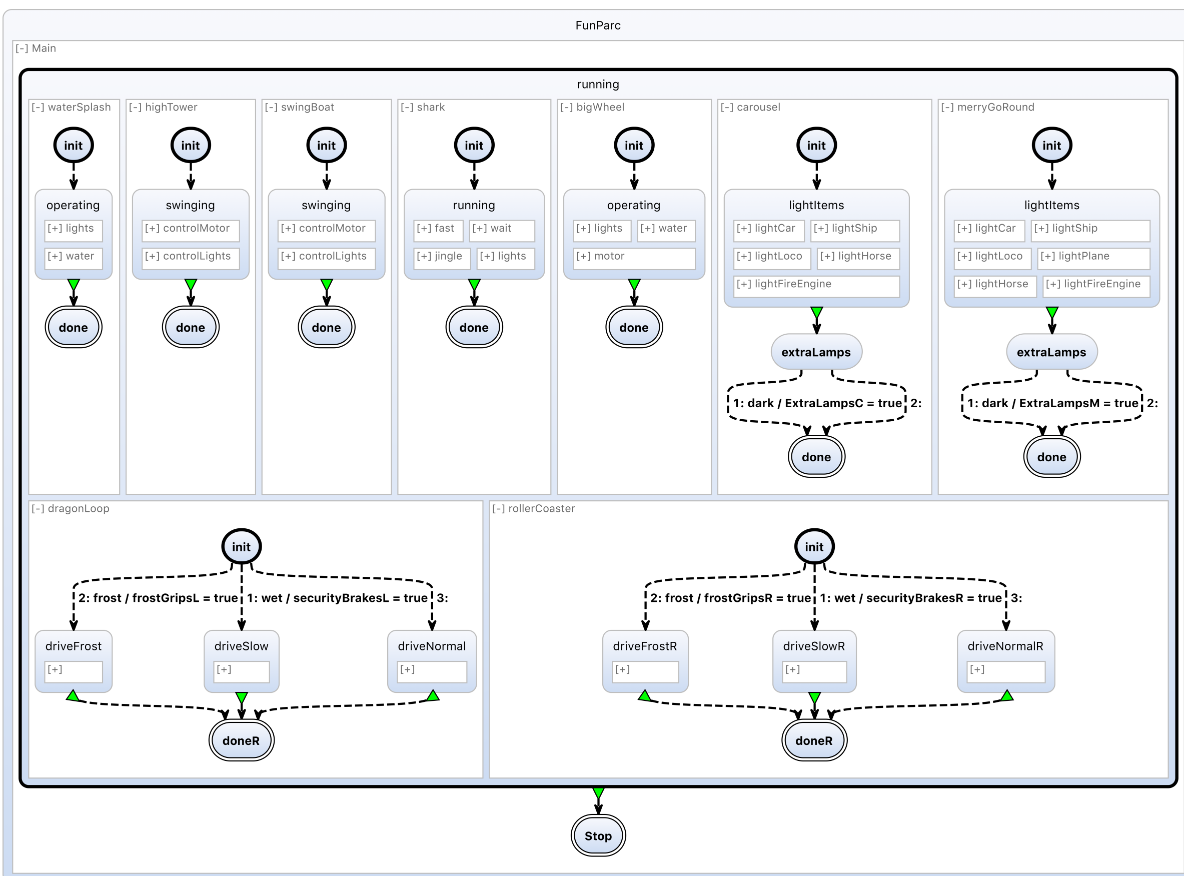


Abbildung 6.18. Beispiel 2 - SCChart

Knoten dieser Art können durch die getroffenen Optimierungen nicht eingespart werden und verschlechtern somit die relative Optimierbarkeit der betrachteten Mo-

dells. Dies heißt nicht, dass Hostcodeaufrufe die Modelloptimierung verhindern. Es wird lediglich viel nicht optimierbarer Code erzeugt.

Das zweite Negativbeispiel, zu sehen in Abbildung 6.18, enthält viele benutzerdefinierte Variablen mit mehrfachen Zuweisungen auf diesen. Dadurch, dass die Variablen mehrfach zugewiesen werden, kann die Kopierpropagation sehr wenig optimieren. Die Optimierbarkeit liegt bei einem Prozent. Auf die Darstellung des SCGs wird an dieser Stelle verzichtet, da die Abbildung sehr viele Knoten enthält und nicht sinnvoll auf dieser Seite dargestellt werden kann.

Nachfolgend werden nun die Positivbeispiele diskutiert. Es stellt sich die Frage, weshalb eine gute Optimierung möglich war. Ein möglicher Grund sind viele Transformationen mit lediglich einem Trigger. Da jeder Trigger eine eigene Variable erzeugt, welche anschließend in einer Fallunterscheidung ausgewertet wird, ist ein großes Optimierungspotenzial gegeben. Hierzu soll das Modell in Abbildung 6.19 betrachtet werden.

M ist der Startzustand des Automaten und ist in drei Regionen mit jeweils zwei Zuständen, je ein Start- und ein Endzustand, unterteilt.

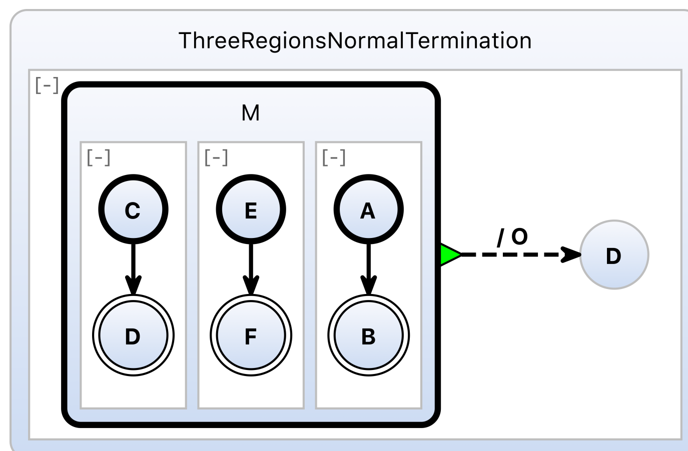
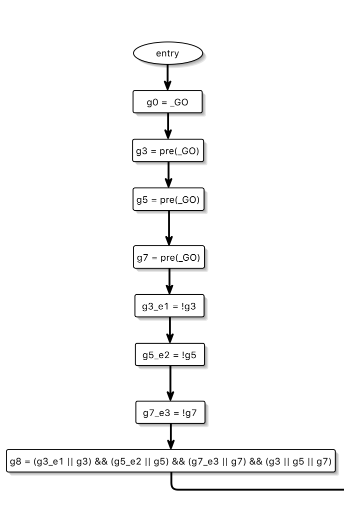


Abbildung 6.19. Beispiel 3 - SCT-Modell

Sobald alle Regionen ihren Endzustand erreicht haben wechselt der Automat in den Zustand D und setzt das Ausgangssignal O auf wahr. Wird nun der seq. SCG (Abbildung 6.20) betrachtet, fällt auf, weshalb das Modell gut optimiert werden kann. Nicht nur findet g0 keine Verwendung, auch diverse Zuweisungen von Variablen mit dem Wert pre(_GO) finden statt. Wird auf das Modell die, für diesen Fall gedachte, Kopierpropagation angewendet, komprimiert sich der SCG zu folgender Abbildung 6.21. Zu sehen ist, dass alle pre-Anweisungen an dem Punkt der Verwendung eingesetzt werden. Dadurch expandiert die Zuweisung der Variable g8.



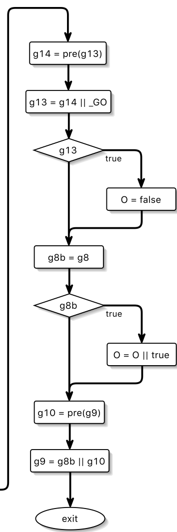


Abbildung 6.20. Beispiel 3 - seq. SCG

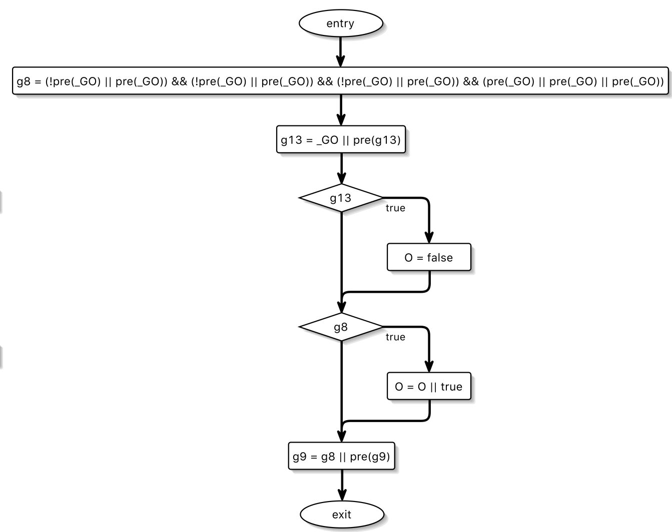


Abbildung 6.21. Beispiel 3 - Kopierpropagation

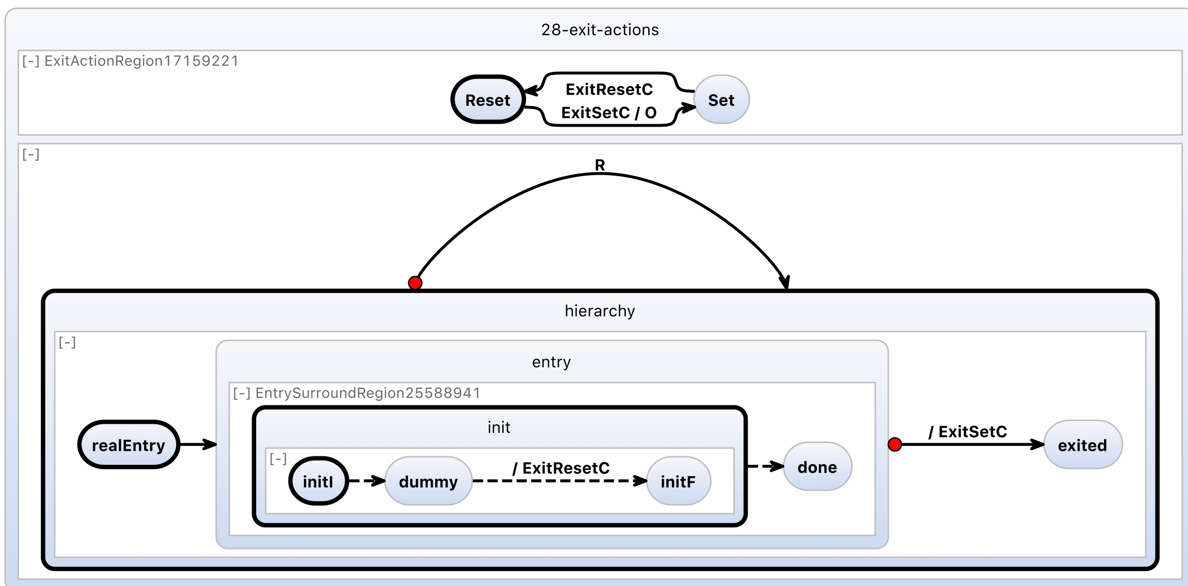


Abbildung 6.22. Beispiel 4 - SCT-Modell

Der Term der Zuweisung lässt sich zu pre(_GO) vereinfachen. Die Optimierung dieser Arbeit decken eine Optimierung der booleschen Terme nicht ab. Eine weiterführende, aufbauende Optimierung könnte dies umsetzen. Im Ausblick in Kapitel 7.1 wird dies genauer diskutiert.

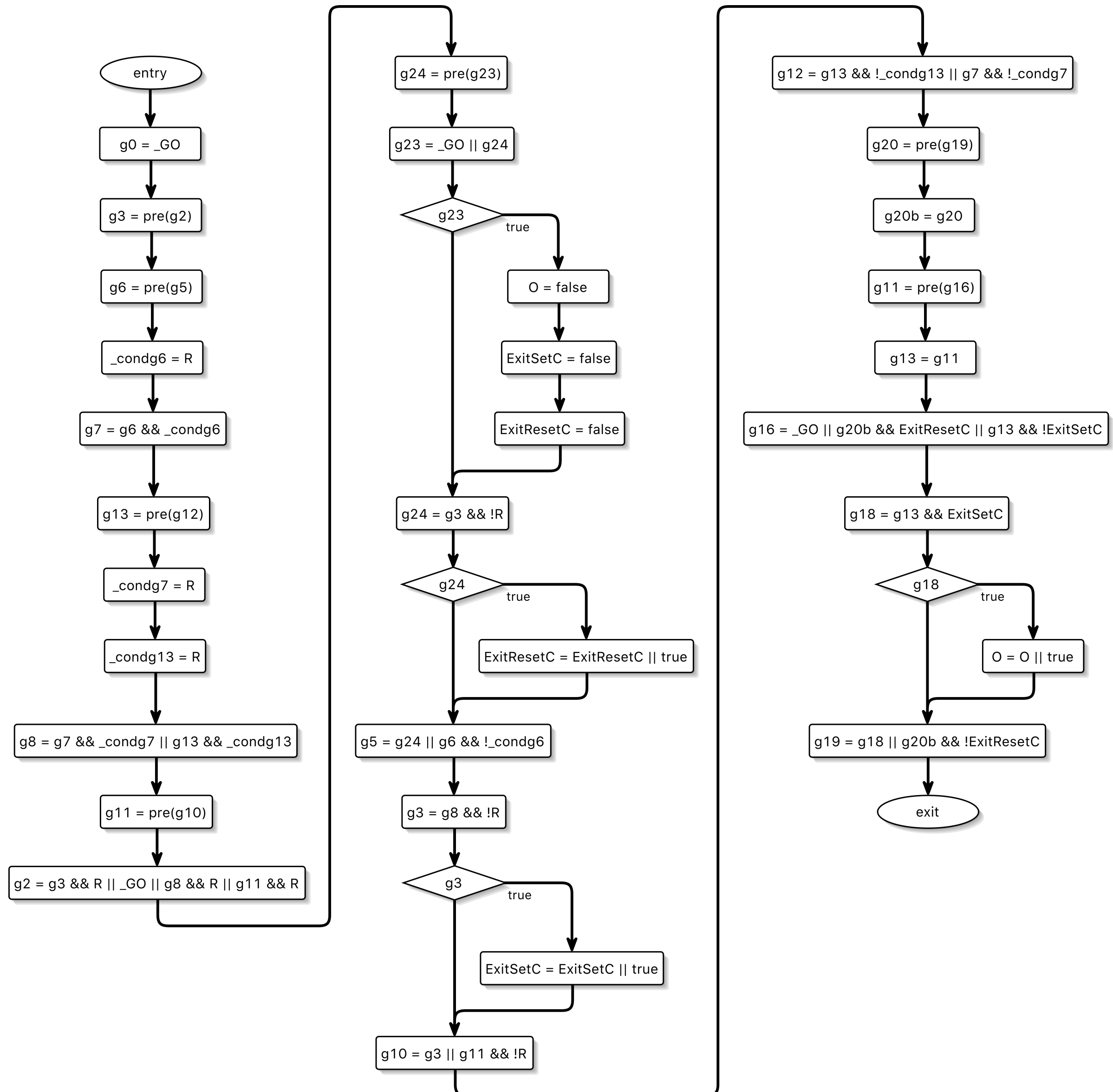


Abbildung 6.23. Beispiel 4 - Variablenwiederverwendung

Durch die Kopierpropagation können bei diesem Modell ca. 50% der Knoten eingespart werden. Zudem ist nur noch ein Viertel der Variablen vorhanden.

Das zweite Positivbeispiel, dargestellt in Abbildung 6.22, fällt mit einer Einsparung von ca. 10% nach der definierten Metrik bei der Variablenwiederverwendung auf. Das Modell zeigt zwei nebenläufige Regionen.

Die erste Region ist im Vergleich zur zweiten Region einfach. Es sind lediglich zwei Zustände mit mit zwei dazwischen liegenden Transitionen. Die zweite nebenläu-

fige Region enthält neben zwei normalen Zuständen noch einen Makrozustand und besitzt eine Strong-Abort-Selbsttransition mit Trigger R. Der Makrozustand enthält einen weiteren Makrozustand sowie einen normalen Zustand. Der innerste Makrozustand enthält 3 Zustände. Wird das Modell mittels der Variablenwiederverwendung optimiert, so ist Abbildung 6.23 das Ergebnis. Die Optimierung ersetzt insgesamt vier Variablennamen. Als Ersetzungen tauchen `g4` → `g24`, `g9` → `g3`, `g17` → `g11` und `g17b` → `g13` auf. Zusammenfassend ist festzuhalten, dass Hostcodeaufrufe nicht optimiert werden können und somit Modelle mit vielen dieser Aufrufe weniger gut optimierbar sind.

Besser optimierbar sind Modelle mit Transitionen ohne komplexe Triggerterme, da für diese Transitionen simple Fallunterscheidungen generiert werden.

Diese wiederum sind prädestiniert für die Optimierung durch die Kopierpropagation. Weiterhin ist festzustellen, dass Modelle mit Variablen, die kurz vor ihrer letzten Verwendung eingeführt werden, besser durch die Variablenwiederverwendung optimiert werden können, da nach ihrer letzten Verwendung, die direkt im nächsten Knoten ist, die Variable als nicht verwendet markiert werden kann und dementsprechend eine Weiterverwendung in einem anderen Kontext möglich ist.

6.4 Quantitative Auswertung automatisch generierter Modelle

Die quantitative Auswertung automatisch generierter Modelle basiert auf Modellen aus dem Modellgenerator von Smyth. Die Einstellungen der Generators werden in Abbildung 6.24 dargestellt. Es werden 10000 Modelle generiert und mit dem TestRunner aus dem vorhergehenden Kapitel zur Größenermittlung vermessen. Die Modelle haben eine Größe zwischen 1 und 500 Zuständen mit einer gleichmäßigen Verteilung dieser über die Gesamtmenge der Modelle.

Die Auftrittswahrscheinlichkeit eines Makrozustand liegt bei 10%. Maximal können 50 Makrozustände auftreten. Nebenläufigkeit hat eine Wahrscheinlichkeit von ebenfalls 10%. Die maximale Anzahl an nebenläufigen Regionen pro Modell beläuft sich auf 20. Die Chance für eine neue Transition nach einem Zustand liegt bei standardmäßigen 20%. Die generierten Modelle haben eine Eingangs- und Ausgangsanzahl von jeweils maximal 50. Die Terme der Bedingungen von Fallunterscheidungen können maximal drei Ausdrücke enthalten.

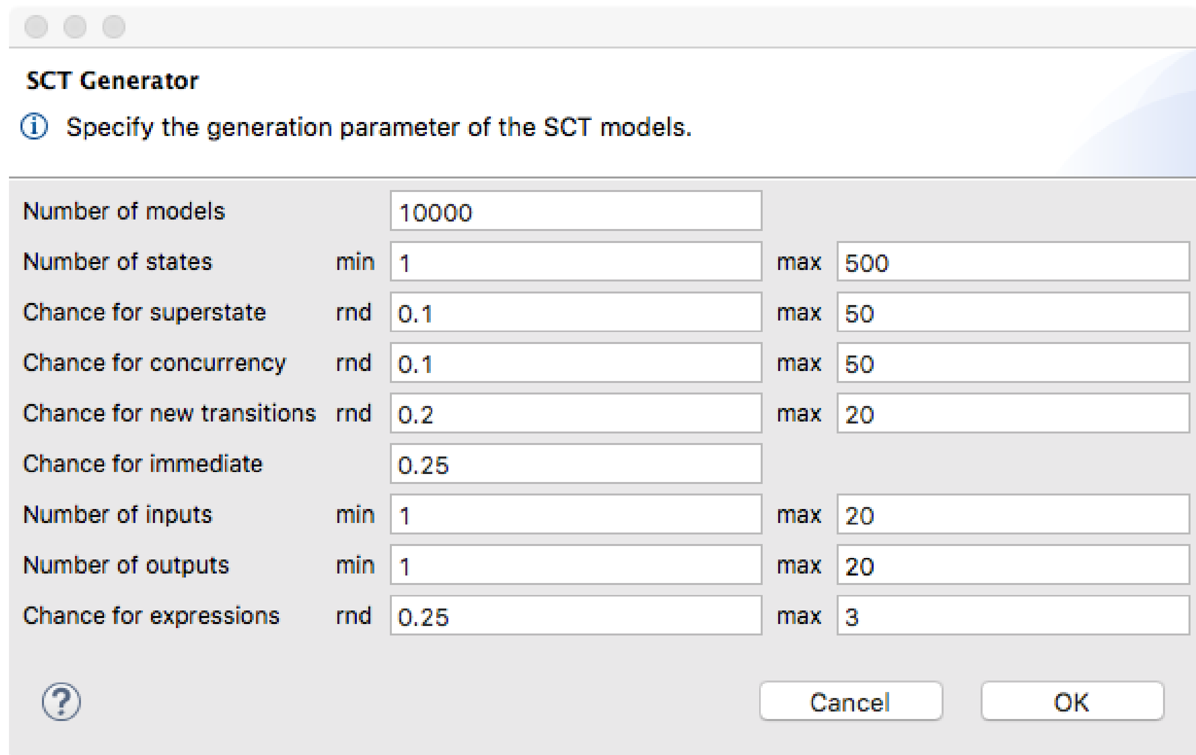


Abbildung 6.24. SCTGenerator - Einstellungen

Dargestellt werden die Werte als prozentuale Optimierung zum sequenzialisierten SCG. Es wird über die Knoten-, Zuweisungs- und Deklarationsanzahl mit einer selbst definierten Metrik eine Summe über die Werte definiert. Die Werte dieser Metrik werden für den Vergleich verwendet. Die Metrik stellt durch die Kumulation der drei oben genannten Werte eine Vergleichbarkeit der Größe her. Für die Darstellung der Werte wird in der folgenden Abbildung 6.25 das Boxplotverfahren verwendet. Der prozentuale Unterschied sagt aus, wie viel Größe nach der Metrik prozentual eingespart wird. Somit gilt, je größer der Wert desto mehr Einsparung.

Der Boxplot zeigt an, dass bei den Optimierungen Kopierpropagation, Kopierpropagation → Variablenwiederverwendung und Variablenwiederverwendung → Kopierpropagation viele Ausreißer (runde schwarze Punkte) vorhanden sind. Dies ist ähnlich zu der Auswertung der realen Modelle. Die Ausreißer im unteren Prozentbereich sind große und somit prozentual weniger gut optimierbare Modelle. Im oberen Prozentbereich zeigen die Ausreißer kleine Modelle die überdurchschnittlich gut durch die Kopierpropagation optimiert werden. Eine Diskussion dieser Eigenschaft wurde in Kapitel 6.2 bereits durchgeführt. Man erkennt, dass die Variablenwiederverwendung (VW) mit allen Werten zwischen 0% und 10% Reduktion liegt.

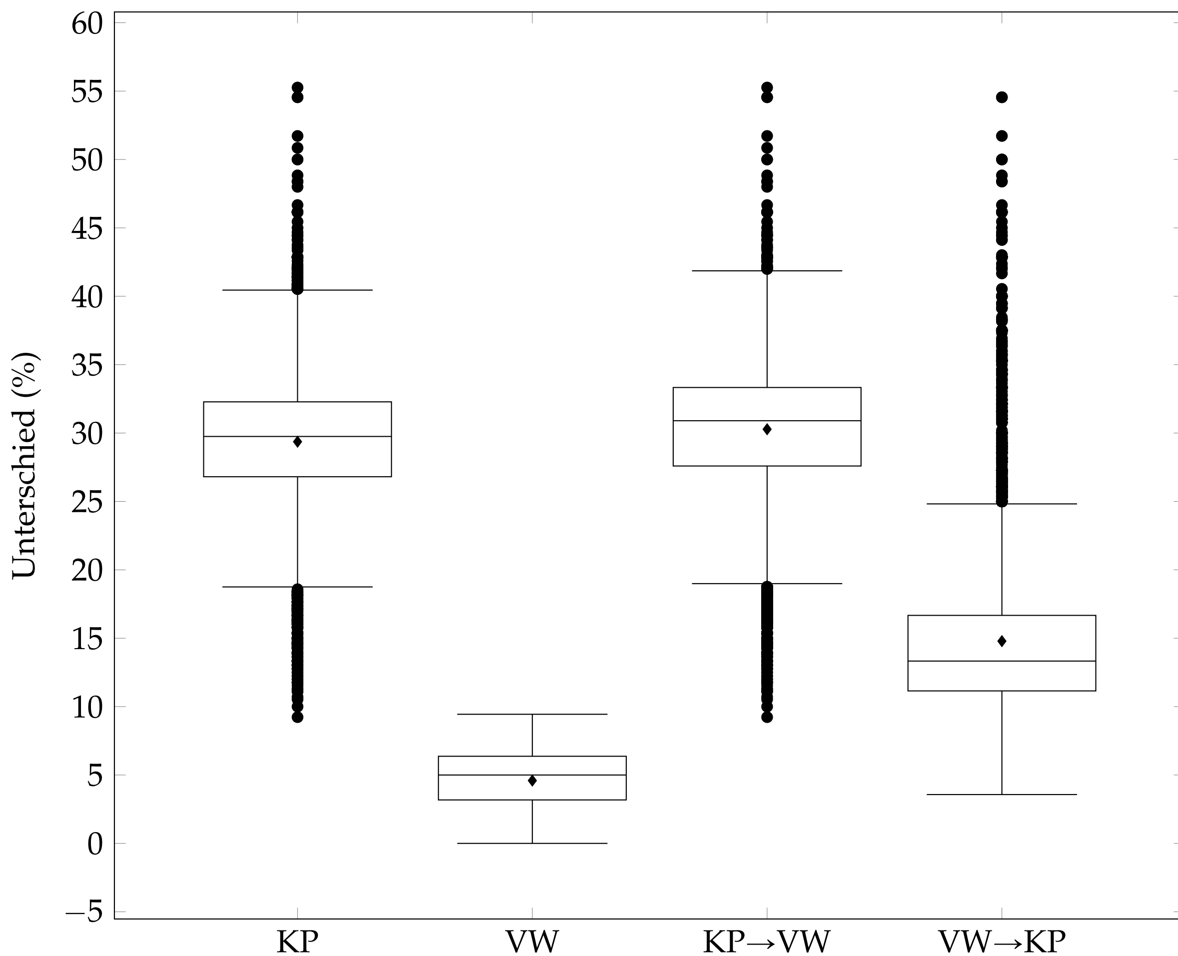


Abbildung 6.25. Vergleich prozentuale Optimierung automatisch generierter Daten - (Prozentualer Unterschied der Modellgröße des optimierten SCGs zu dem unoptimierten SCG - KP: Kopierpropagation, VW: Variablenwiederverwendung)

Dies ist im Vergleich zu den anderen Optimierungen die kompakteste Menge, aber auch die mit der geringsten Auswirkung auf die Größe. Die im Durchschnitt und Median beste Optimierung ist, wie bei der Auswertung der realen Modelle, die Kopierpropagation mit nachgeschalteter Variablenwiederverwendung (KP→VW). Es ist allerdings auch zu beobachten, dass die Streuung der Ergebnisse sehr groß ist. Der meisten Ergebnisse befinden sich allerdings um die 30% Linie im Median. Ähnliches gilt für die reine Kopierpropagation (KP). Lediglich die Variablenwiederverwendung mit anschließender Kopierpropagation (VW→KP) streut nur positiv. Dies hängt mit der Positionierung der direkt zugewiesenen Negationen zusammen. Die Variablen werden direkt nach ihrer Zuweisung verwendet. Hierzu soll ein weite-

res Mal der seq. SCG des ABO-Modells (Abbildung 4.1) betrachetet werden, genauer die Variablen `g4_e1`, `g8_e2` und `g9`. Die ersten beiden genannten Variablen werden direkt einen bzw. zwei Knoten nach ihrer Zuweisung in `g9` verwendet. Davor, und auch danach, sind diese Variablennamen für die Variablenwiederverwendung für die Substitution verwendbar. Es werden somit viele der Konstrukte der Form `X = !Y` durch die Variablenwiederverwendung ersetzt und der Variablenname erhält somit mehrere Zuweisungen. Diese Zuweisungen können im Anschluss nicht mehr durch die Kopierpropagation optimiert werden.

Es ist festzuhalten, dass die Variablenwiederverwendung eine stabile Optimierung ermöglicht aber keine Ersparnis über 10% bietet. Eine größere Ersparnis mit mehr Unsicherheit ermöglicht die Kopierpropagation mit oder ohne nachgeschaltetem Variablenrecycling. Im Durchschnitt und im Median ist eine Einsparung von 30% nach Metrik möglich. Da die negativen Ausreißer der Kopierpropagation immer noch eine bessere Optimierung bieten als die besten Werte der Variablenwiederverwendung ist die stabile Optimierung durch die Variablenwiederverwendung nebensächlich.

Die Werte decken sich mit denen aus dem vorhergehenden Kapitel. Die Optimierungen ermöglichen somit sowohl auf realen als auch generierten Modellen eine solide Verbesserung der Codegröße.

6.5 Analyse der High-Level-Transformationen

Für die Programmierung eingebetteter Systeme mit KIELER und SCCharts ist es interessant zu wissen, welche Modellkonstrukte höchster Ebene (Extended SCCharts) durch die Expansion welche Größenauswirkung im generierten Quellcode vorweisen. Um dieser Frage nachzugehen werden minimale Modelle zu den einzelnen High-Level-Konstrukten erstellt und mittels Testrunnern vermessen.

Um die Vorgehensweise zu verdeutlichen ist in Abbildung 6.26 als Beispiel das Extended-Feature 'Simple Transition with Trigger' zu sehen. Die Abbildung 6.27 zeigt die sequenzialisierte SCG-Form der Modells aus der Abbildung 6.26.

Das Modell aus Abbildung 6.26 zeigt einen Zustand mit einer Selbsttransition. Die Transition enthält einen Trigger mit dem Term `IN`. Mittels Transitionen wird der seq. SCG generiert. Dieser bildet die Funktionalität semantisch äquivalent zu der SCCharts-Darstellung ab. Die Expansion bewirkt, dass das seq. SCG fünf Knoten, drei Zuweisungen und fünf Deklarationen (`g0`, `g1`, `g2`, `_GO` und `IN`) enthält. Der Wert der Metrik beläuft sich auf 13.

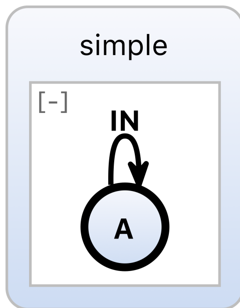


Abbildung 6.26. Transition with Trigger - SCChart

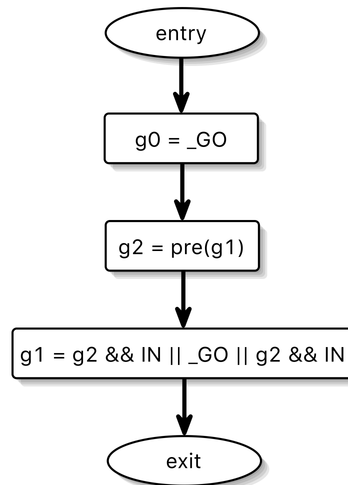


Abbildung 6.27. Transition with Trigger - seq. SCG

Eine Übersicht der High-Level-Befehle bietet die Übersicht der Core und Extended SCCharts aus Abbildung 1.1. Ebenfalls sind Modelle mit mehreren Verwendungen der komplexen Konstrukte vorhanden. Dies dient der Feststellung, ob nur die erste Verwendung oder auch nachfolgende Verwendungen der einzelnen Konstrukte eine starke Expansion verursachen. Das Ergebnis sind absolute Werte über Knoten-, Zuweisungens- und Deklarationsanzahl. Diese Werte werden im Folgenden sortiert und verglichen. Zu Beginn wird es eine Übersicht über Konstrukte mit geringer Auswirkung auf die Codegröße geben. Im Verlauf werden dann Konstrukte mittlerer und großer Auswirkung besprochen.

Die nachfolgenden Tabellen sind gleich aufgebaut. Die Spalte 'Konstrukt' beschreibt den verwendeten High-Level-Befehl sowie die Quantität der Verwendung und, wenn vorhanden, Optionen wie Trigger und Effekt einer Transition oder Verknüpfungen per Kon- oder Disjunktion von Befehlen. Die verwendete Metrik (Kumulation der Werte Knoten, Zuweisungen und Deklarationen) wird zum Zweck der Sortierung und somit auch der Vergleichbarkeit genutzt. Die eigentliche Länge des Quellcodes ergibt sich aus der Anzahl der Knoten (Logik des Programmes) und der Anzahl der Deklarationen (Eingeführte Variablen). Alle Daten werden auf Basis der sequenzialisierten SCG-Form des betrachteten Modells erhoben.

Die Tabelle 6.28 zum ersten Teil der Analyse zeigt größtenteils das häufig genutzte simple Befehle eine geringe Auswirkung auf die Codegröße, also eine kleine Expansion, aufweisen. Für Programmierer ist die Verwendung der Operationen somit unproblematisch. Beispielweise das ABO-Modell 1.5 besteht aus den aufgeführten Befehlen.

Konstrukt	Kno.	Zuw.	Dekl.	Metrik
Immediate Strong Abort	3	1	2	6
Immediate Termination	3	1	2	6
Immediate	3	1	2	6
Superstate Final	3	1	2	6
Termination	3	1	2	6
Immediate Strong Abort Effect	5	1	3	9
Immediate Termination Effect	5	1	3	9
Immediate Effect	5	1	3	9
Exit	4	2	4	10
Complex Final	5	3	4	12
Simple	5	3	4	12
Strong Abort	5	3	4	12
Superstate	5	3	4	12
During	5	3	5	13
Entry	5	3	5	13
Simple Trigger	5	3	5	13
Strong Abort Trigger	5	3	5	13

Abbildung 6.28. High-Level-Transformationen Auswertung - Ressourcenschonende Konstrukte

Die nachfolgende Tabelle deckt den Übergangsbereich zwischen simplen Operationen mit wenig Auswirkung auf die Codegröße und komplexen Operationen mit sehr vielen generierten Quellcodezeilen ab. Ebenfalls tauchen in der zweiten Tabelle die ersten Doppelverwendungen von Operationen auf.

Zum Großteil sind Operationen aus der zweiten Tabelle 6.29 Operationen aus der ersten Tabelle mit einem Trigger oder/sowie einem Effekt.

Die einfache Verwendung eines Eingabesignals weist gegenüber der Doppelverwendung eine um 3 Punkte erhöhte Metrik auf. Somit wird bei Signalen für die Erstverwendung eine große Anzahl an Codezeilen mehr generiert. Jede weitere Verwendung verursacht eine minimale Expansion, da die Strukturen zum Zurücksetzen der Signale bereits mit der ersten Verwendung generiert wurden und somit weitere Signale nur hinzugefügt werden müssen. Ähnlich verhält es sich mit der Verwendung von Signalen als Ausgabe. Für die Betrachtung ist ein Blick auf die Tabelle 6.30 notwendig, da die Doppelverwendung bereits in den Bereich der Konstrukte mit einer vergleichsweise großen Auswirkung auf die Quellcodelänge fällt. Tabelle 6.30 umfasst Konstrukte großer Expansion.

Konstrukt	Kno.	Zuw.	Dekl.	Metrik
Simple Effect	7	3	5	15
Strong Abort Effect	7	3	5	15
Immediate Strong Abort Trigger	6	4	6	16
Immediate Trigger	6	4	6	16
Superstate Concurrenct Regions	7	5	6	18
Immediate Strong Abort Effect Trigger	8	4	7	19
Signal Input	7	5	7	19
Immediate Trigger Effect	8	4	7	19
Simple Trigger Effect	8	4	7	19
Strong Abort Trigger Effect	8	4	7	19
Count Delay	5	3	12	20
Immediate During	8	6	8	22
2 Signals Input Conj	8	5	9	22
2 Signals Input Disj	8	5	9	22
Suspend	11	5	8	24
Superstate 2 Strong Aborts	8	4	13	25
Signal Output	12	6	8	26

Abbildung 6.29. High-Level-Transformationen Auswertung - Übergangsbereich zwischen Ressourcenschonenden und stark expandierenden Konstrukten

Als Programmierer sollte bei der Programmierung für ressourcenbegrenzte Systeme somit auf die Vermeidung dieser Konstrukte geachtet werden. Da allerdings viele der aufgelisteten Befehle sehr viel Logik enthalten und teils schwierig durch simplere Konstrukte zu ersetzen sind, kann nicht immer auf diese verzichtet werden. Als Entwickler sollte allerdings immer daran gedacht werden, dass beispielsweise eine Verwendung einer tiefen Historie für einen Superstate mit zwei Subzuständen über 100 Zeilen Quellcode generiert werden können.

Während der Beschreibung der Tabelle 6.29 wird auf die Ausgabesignale bereits eingegangen. Die Verwendung zweier Ausgabesignale ist in der Tabelle 6.30 ebenfalls zu sehen. Anders als bei der Verwendung der Eingabesignale erhöht sich die Metrik bei einer weiteren Verwendung eines Ausgabesignals um 7. Genauer werden 5 weitere Knoten, wovon ein Knoten eine weitere Zuweisung ist, generiert. Ebenfalls werden zwei neue Variablen eingeführt.

Zusammengefasst ist sind die Tabellen 6.28 und 6.29 für die Programmierung eingebetter Systeme mit limitierten Ressourcen sehr gut geeignet. Die aufgelisteten Operationen haben jeweils eine geringe Auswirkung auf die Codegröße. Sobald

komplexere Strukturen abgebildet werden müssen, bleibt es abzuwägen, ob eine Implementierung durch kleine aber simple Operationen mit jeweils wenigen Codezeilen oder aber durch komplexe Operationen mit einer großen Auswirkung auf die Codelänge abzubilden ist. Bei solchen Anwendungsfällen ist allerdings der Anwendungsentwickler gefragt.

Eine Untersuchung, ob die stark expandierenden Konstrukte auf eine ineffiziente Implementierung zurückgehen wurde an dieser Stelle nicht betrachtet. Die Idee und eine Diskussion hierzu ist in Kapitel 7.1 aufzufinden.

Konstrukt	Kno.	Zuw.	Dekl.	Metrik
Deferred	11	3	14	28
Pre	12	6	10	28
Deferred Effect	11	3	15	29
Superstate Strong Abort	8	4	17	29
Deferred Trigger Effect	12	4	17	33
Deep History	15	7	11	33
Shallow History	15	7	11	33
2 Signals Output	16	7	10	33
Deferred Trigger	12	4	18	34
Superstate Weak Abort	16	10	12	38
Immediate Termination Trigger	19	8	14	41
2 Pres Conj	19	8	16	43
2 Pres Disj	19	8	16	43
Immediate Termination Trigger Effect	21	8	15	44
Superstate 2 Weak Aborts	20	11	16	47
Weak Suspend	22	10	39	71
Shallow History 2 Inner Stats	46	22	31	99
Deep History 2 Inner States	65	29	46	140

Abbildung 6.30. High-Level-Transformationen Auswertung - Stark expandierende Konstrukte

6.6 Analyse der Optimierungen im Kontext der Hardwaresynthese

Ebenfalls interessant ist eine Untersuchung der Auswirkungen der einzelnen Optimierungen auf die Hardwaresynthese von Rybicki [Ryb16]. Wird das bereits in Abbildung 1.5 vorgestellte unoptimierte ABO-Modell zu Hardware synthetisiert, so

erhält man die Abbildung 6.31. Es sind die Variablen aus Abbildung 4.1 zu erkennen. Die Kon- und Disjunktionen werden in UND- und ODER-Gatter übersetzt. Insgesamt sind 21 Elemente in der Hardwaredarstellung vorhanden.

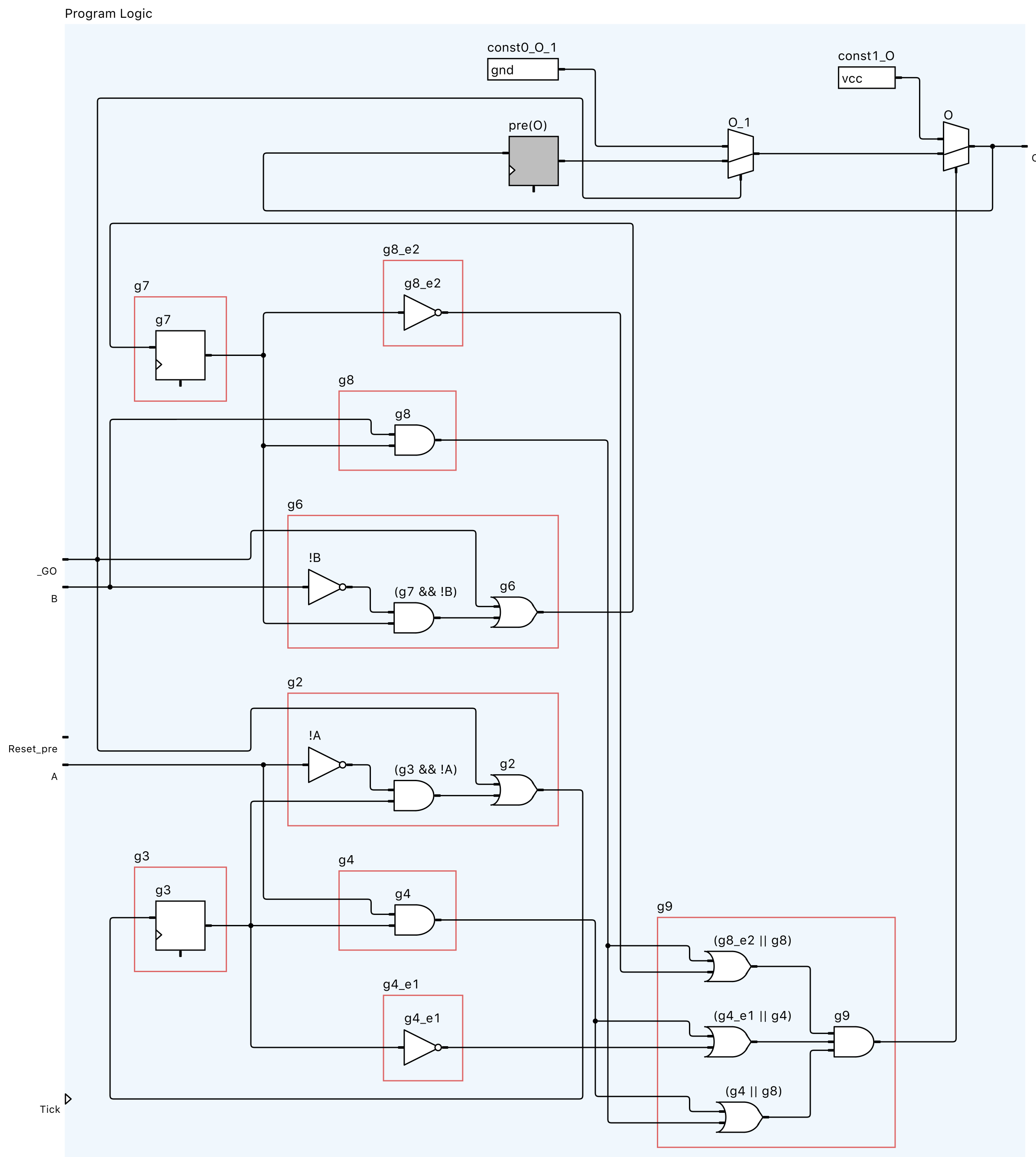


Abbildung 6.31. ABO-Modell - Hardwaresynthese

Wird nun das Modell mit der Kopierpropagation und anschließend mit der Variablenwiederverwendung optimiert wird das Modell in Abbildung 6.33 mittels Hardwaresynthese generiert. Die Abbildung zeigt die Variablen aus Abbildung 4.7. Obwohl die generierte Schaltung kleiner wirkt als Abbildung 6.31 hat Abbildung 6.33 ebenfalls 21 Elemente. Lediglich die Anzahl der rot eingezeichneten Regionen hat sich verringert. Somit erbringt die Optimierung für dieses Modell keine Reduktion in der Hardwaresynthese. Problematisch ist, dass die SSA-Form, welche für diese Art von Schaltkreissynthese erforderlich ist (vgl. [Ryb16]), die Variablenwiederverwendung quasi aufhebt und die Kopierpropagation die Logik lediglich in weniger Variablen verschiebt und nicht verschlankt.

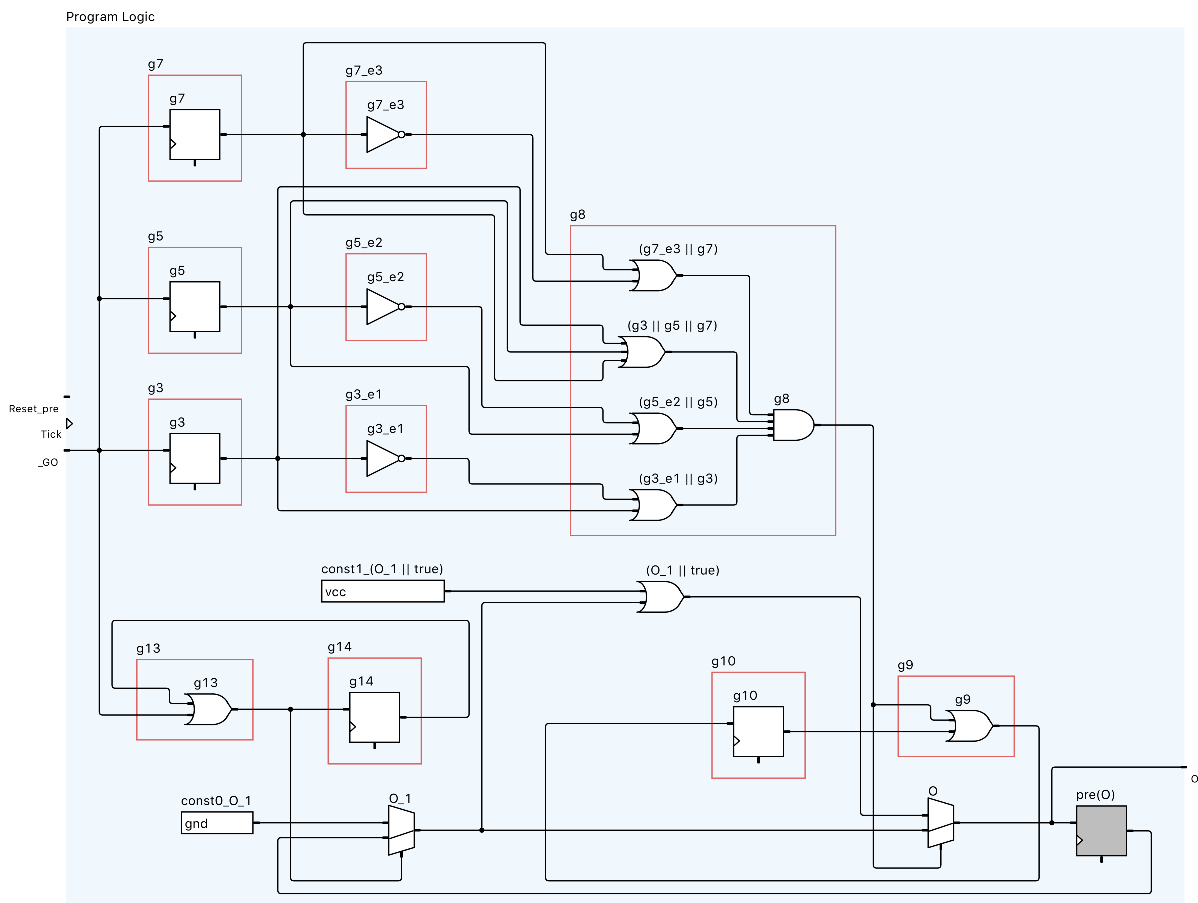


Abbildung 6.32. ThreeRegionsNormalTermination-Modell - Hardwaresynthese

Betrachtet man im gleichen Kontext das Modell aus Abbildung 6.19 unoptimiert und optimiert als Hardwaredarstellung ist eine deutliche Reduktion der Hardwarekomponenten zu verzeichnen.

Die Abbildung 6.32 zeigt die Variablen des Modells aus Abbildung 6.19. Ähnlich wie bei der Reduktion der Quellcodelänge aus Abbildung 6.20 verhält es sich mit der Hardwaresynthese. Die nachfolgende Abbildung 6.34 zeigt die Reduktion der Hardware.

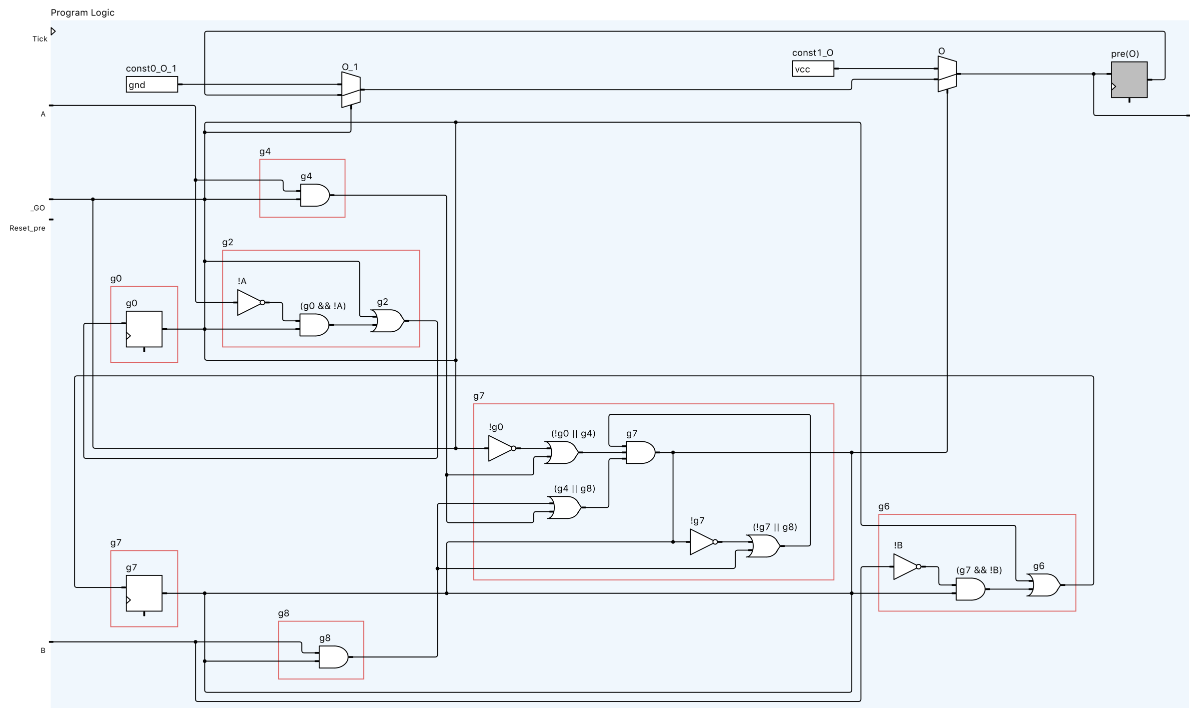


Abbildung 6.33. ABO-Modell optimiert - Hardwaresynthese

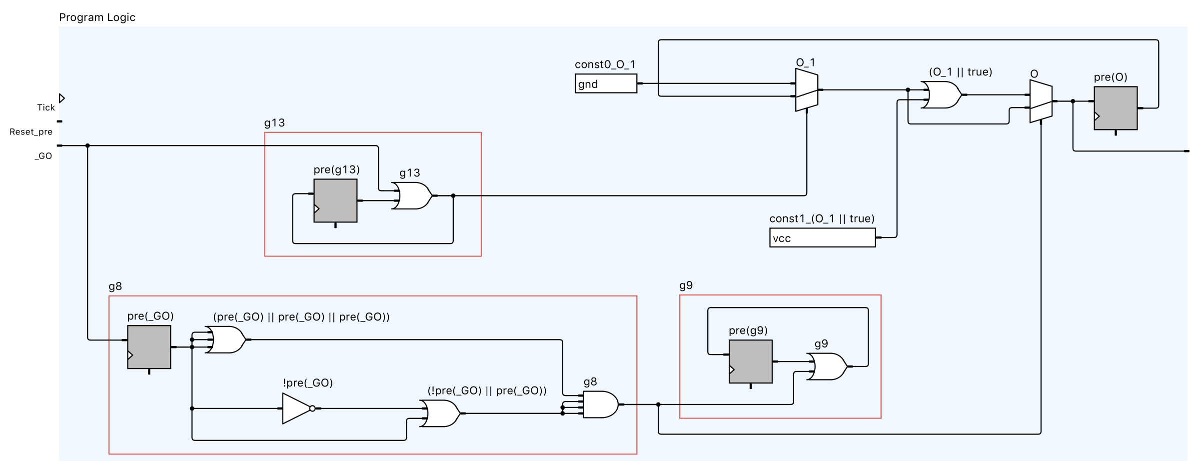


Abbildung 6.34. ThreeRegionsNormalTermination-Modell optimiert - Hardwaresynthese

Insgesamt werden durch die Kopierpropagation sechs Hardwarekomponenten eingespart. In relativen Zahlen macht dies eine Ersparnis von 28,5%. Es ist somit stark von dem zu optimierenden Modell abhängig, ob und wie stark das Modell auch für die Hardwaresynthese optimiert werden kann.

Ähnlich wie in Abbildung 6.21 bei der Betrachtung ausgewählter Beispiele der Evaluation der Quellcodeoptimierung ist in Abbildung 6.34 zu erkennen, dass `g8` noch weiter optimierbar ist. Durch eine Optimierung boolescher Terme können vier weitere Elemente eingespart werden. Zudem erfüllt die als `g9` gekennzeichnete Region keinen Nutzen. Eine Betrachtung des betreffenden Terms erfolgt im Ausblick dieser Arbeit im Anschluss an die Evaluation.

Kapitel 7

Zusammenfassung

Zusammengefasst stellt die Arbeit die Optimierung der Quellcodegröße und Ausführungszeit von aus SCCharts-Modellen generierten Programmen dar. Ebenfalls gibt sie einen Einblick in die Expansion verschiedener Extended SCCharts-Befehle und somit auch auf die Verwendbarkeit für ressourcenbeschränkte Systeme. Genauer wird die Größenreduktion mittels der Kompileroptimierung Kopierpropagation und der Variablenwiederverwendung, welche der Registerwiederverwendung gleicht, erreicht. Ein Nebeneffekt der Variablenwiederverwendung ist zudem die beschleunigte Ausführung der optimierten Programme. Es konnte auch bei der Kopierpropagation eine Verbesserung der Ausführungsgeschwindigkeit verzeichnet werden, allerdings war hierbei die Streuung der Werte sehr groß. Auch Kombinationen der beiden Optimierungen werden durchgeführt und vermessen. Hierbei wird festgestellt, dass eine Ausführung der Kopierpropagation mit nachfolgender Variablenwiederverwendung die maximale Größenreduktion mit sich bringt. Die Analyse der Expansion der High-Level-Konstrukte auf Extended SCCharts-Ebene hat zudem ergeben, dass die meisten Operationen wenige Codezeilen bei der Quellcodegenerierung erzeugen. Lediglich ein paar, in ihrer Funktion sehr komplexe, Konstrukte erzeugen mehr als 20 Codezeilen nach ihrer Expansion. Für ressourcen-beschränkte Systeme wird daher empfohlen auf die Features ’Immediate Termination with Trigger and Effekt’, ’Weak Suspend’, ’Weak Abort’, ’Shallow History’, und ’Deep History’ zu verzichten. Gerade die beiden ’History’-Konstrukte lassen den Code stark expandieren.

7.1 Ausblick

Die Evaluation hat gezeigt, dass die Optimierungen Quellcode in vielen Fällen stark verkleinern können. Nun mag eine nächste Fragestellung sein, ob durch Repositionierung der Knoten, primär der Zuweisungen, beispielsweise die Variablenwiederverwendung bessere Ergebnisse liefert und somit mehr Deklarationen eingespart werden können. Wird zu diesem Zweck das ABO-Modell aus Abbildung 1.5 bzw. dessen sequenzialisierte SCG-Form aus Abbildung 4.1 betrachtet und mit der lediglich durch die Variablenwiederverwendung optimierten Form (Abbildung 4.6) verglichen so

fällt auf, dass es keine Optimierungsmöglichkeit durch Repositionierung gibt. Keines der weiteren betrachteten Modell ist in diesem Kontext positiv aufgefallen. Ohne umfassende Analysen in Hinsicht auf die weitere Optimierbarkeit durch Repositionierung von Knoten ist allerdings schwer abzuschätzen ob eine Betrachtung des Themas sinnvoll ist.

Ein weiteres Optimierungsthema wurde bereits im Kapitel 6.3 angesprochen. Nachdem die Kopierpropagation durchgeführt wurde, kann es zu stark expandierten Variablenzuweisungen mit simplen boolschen Termen kommen. Diese können im Ausführungszeit- als auch Hardwaresynthesekontext durch eine Vereinfachung boolescher Terme optimiert werden.

Bei der Betrachtung der `g9`-Region in der Hardwaredarstellung in Abbildung 6.34 der Term `g9 = g8 || pre(g9)` auf. Diese Zeile hat keine Auswirkung auf den Programmverlauf, da `g9` ausschließlich in diesem Term gelesen und geschrieben wird. Es handelt sich aber nicht um nicht verwendeten oder nicht erreichbaren Code. Es wird eine Art ineffektiver oder unnützer Code dargestellt. Eine Entfernung dieser Codestellen ist interessant.

Am Ende des Kapitel 6.5 wurde angesprochen, dass nicht überprüft wurde, ob die Expansion der Tabelle 6.30 auf eine ineffektive Implementierung zurückzuführen ist. Eine Betrachtung der Extended Features in diesem Kontext ist interessant und sinnvoll.

7.2 Fazit

Die Arbeit leistet eine nachweisliche Reduktion der Codegröße und Ausführungszeit von aus SCCharts-Modellen generierten Quellcodes. Ebenfalls werden Verwendungsvorschläge für High-Level-Konstrukte der Extended SCCharts gegeben. Sowohl auf realen als auch auf automatisch generierten Modellen leisten die Optimierungen ähnlich gute Ergebnisse. Lediglich bei speziellen Modellen und der Hardwaresynthese kann es vorkommen, dass gar keine Optimierung vorgenommen werden kann. Das Ausgangsproblem wird durch die getroffenen Implementierungen gelöst.

Literatur

[And96] Charles André. *SyncCharts: A visual representation of reactive behaviors*. Techn. Ber. RR 95–52, rev. RR 96–56. Sophia-Antipolis, France: I3S, Rev. April 1996.

[AP13] G. Ann Campbell und P. P. Papapetrou. *Sonarqube in action*. Manning Publications, 2013, S. 3–25, 82–95. ISBN: 9781617290954.

[ASU70] Alfred V. Aho, Ravi Sethi und J. D. Ullman. „A formal approach to code optimization“. In: *SIGPLAN Not.* 5.7 (Juli 1970), S. 86–100. ISSN: 0362-1340. DOI: 10.1145/390013.808486. URL: http://doi.acm.org/10.1145/390013.808486.

[ASU86] Alfred V. Aho, Ravi Sethi und Jeffrey D. Ullman. *Compilers - principles, techniques, and tools*. Reading, Massachusetts: Addison-Wesley, 1986, S. 590–591, 632–637. ISBN: 0-201-10088-6.

[BC84] Gérard Berry und Laurent Cosserat. „The ESTEREL Synchronous Programming Language and its Mathematical Semantics“. In: *Seminar on Concurrency, Carnegie-Mellon University*. Bd. 197. LNCS. Springer-Verlag, 1984, S. 389–448. ISBN: 3-540-15670-4.

[Bun08] J. Bungo. „The use of compiler optimizations for embedded systems software“. In: *Crossroads - Volume 15* (2008), S. 8–15.

[DEM+00] Saumya K. Debray, William Evans, Robert Muth und Bjorn De Sutter. „Compiler techniques for code compaction“. In: *ACM Trans. Program. Lang. Syst.* 22.2 (März 2000), S. 378–415. ISSN: 0164-0925. DOI: 10.1145/349214.349233. URL: http://doi.acm.org/10.1145/349214.349233.

[GH88] J. R. Goodman und W.-C. Hsu. „Code scheduling and register allocation in large basic blocks“. In: *ICS '88 Proceedings of the 2nd international conference on Supercomputing* (1988), S. 442–452.

[Gue99] Martin Gueting Ralf Hartmut und Erwig. *Übersetzerbau (Techniken, Werkzeuge, Anwendungen)*. Springer, 1999, S. 255–288. ISBN: 3-540-65389-9.

[Har87] David Harel. „Statecharts: A visual formalism for complex systems“. In: *Science of Computer Programming* 8.3 (Juni 1987), S. 231–274.

[HDM+14] Reinhard von Hanxleden, Björn Duderstadt, Christian Motika, Steven Smyth, Michael Mendler, Joaquín Aguado, Stephen Mercer und Owen O'Brien. „SCCharts: Sequentially Constructive Statecharts for safety-critical applications". In: *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'14)*. Edinburgh, UK: ACM, Juni 2014, S. 372–383.

[HMA+14] Reinhard von Hanxleden, Michael Mendler, Joaquín Aguado, Björn Duderstadt, Insa Fuhrmann, Christian Motika, Stephen Mercer, Owen O'Brien und Partha Roop. „Sequentially Constructive Concurrency—A conservative extension of the synchronous model of computation". In: *ACM Transactions on Embedded Computing Systems, Special Issue on Applications of Concurrency to System Design* 13.4s (Juli 2014), 144:1–144:26.

[Ker88] Brian W. Kernighan. *The C programming language - second edition*. Hrsg. von Dennis M. Ritchie. 2nd. Prentice Hall Professional Technical Reference, 1988, S. 118–121. ISBN: 0131103709.

[Kik89] S. Kikuchi. *Compile method using copy propagation of a variable*. US Patent 4,843,545. 1989.

[Kil73] Gary A. Kildall. „A unified approach to global program optimization". In: *POPL '73 Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages* (1973), S. 194–206.

[LWL05] Benjamin Livshits, John Whaley und Monica S. Lam. *Reflection analysis for java*. APLAS'05. Tsukuba, Japan: Springer-Verlag, 2005, S. 139–160. ISBN: 3-540-29735-9, 978-3-540-29735-2. DOI: 10.1007/11575467_11. URL: http://dx.doi.org/10.1007/11575467_11.

[MP90] Samuel P. Midkiff und David A. Padua. „Issues in the optimization of parallel programs". In: *Proceedings of the 1990 International Conference on Parallel Processing, Urbana-Champaign, IL, USA, August 1990. Volume 2: Software.* 1990, S. 105–113.

[MSH+13] Christian Motika, Steven Smyth, Reinhard von Hanxleden und Michael Mendler. *Sequentially Constructive Charts (SCCharts)*. Poster presented at 10th Biennial Ptolemy Miniconference (PTCONF'13), Berkeley, CA, USA. Juli 2013.

[MSH14] Christian Motika, Steven Smyth und Reinhard von Hanxleden. „Compiling SCCharts—A case-study on interactive model-based compilation". In: *Proceedings of the 6th International Symposium on Leveraging Applications*

of Formal Methods, Verification and Validation (ISoLA 2014). Bd. 8802. LNCS. Corfu, Greece, Okt. 2014, S. 443–462. DOI: 10.1007/978-3-662-45234-9.

[Muc97] Steven S. Muchnick. *Advanced compiler design and implementation*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997, S. 356–362. ISBN: 1-55860-320-4.

[Ryb16] Francesca Rybicki. „Interactive incremental hardware synthesis for SCCharts“. http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/fry-bt.pdf. Bachelor Thesis. Kiel University, Department of Computer Science, März 2016.

[Sch06] G. Schlossnagle. *Professionelle PHP 5-Programmierung - Entwicklerleitfaden für große Webprojekte mit PHP 5 (Studentenausgabe)*. Open source library. Addison Wesley in Pearson Education Deutschland, 2006. ISBN: 9783827325136.

[Sch14] Alexander Schulz-Rosengarten. „Framework zum Tracing von EMF-Modelltransformationen“. http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/als-bt.pdf. Bachelor thesis. Kiel University, Department of Computer Science, März 2014.

[Sch73] Paul B. Schneck. „A survey of compiler optimization techniques“. In: ACM '73 (1973), S. 106–113. DOI: 10.1145/800192.805690. URL: http://doi.acm.org/10.1145/800192.805690.

[Smy13] Steven Smyth. „Code generation for sequential constructiveness“. http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/ssm-dt.pdf. Diploma thesis. Kiel University, Department of Computer Science, Juli 2013.

[SS07] Y. N. Srikant und Priti Shankar. *The compiler design handbook: optimizations and machine code generation, second edition*. 2nd. Boca Raton, FL, USA: CRC Press, Inc., 2007, S. 68–74. ISBN: 142004382X, 9781420043822.

[WZ91] Mark N. Wegman und F. Kenneth Zadeck. „Constant propagation with conditional branches“. In: *ACM Trans. Program. Lang. Syst.* 13.2 (Apr. 1991), S. 181–210. ISSN: 0164-0925. DOI: 10.1145/103135.103136. URL: http://doi.acm.org/10.1145/103135.103136.