

Komponentenbasierte Softwareentwicklung für datenflußorientierte eingebettete Systeme

Dissertation

zur Erlangung des Grades des
Doktors der Naturwissenschaften
an der Fakultät für Mathematik und Informatik
der Universität Passau

Walter Maydl

22. Mai 2005

Ad maiorem Dei gloriam.

*Ich widme diese Arbeit
Frau Margarete Tanner
Meinen Eltern Robert & Berta Maydl
Frau Katharina Maydl († 3. März 2005)*

Vorwort

Incipit.

Allen, die zum Gelingen dieser Arbeit beigetragen haben, danke ich! An herausragender Stelle sind Frau Margarete Tanner, meine Familie und insbesondere meine Eltern, Robert und Bertha Maydl, zu nennen. Mein herzlicher Dank gilt dem Vorstand des Bayerischen Blinden- und Sehbehindertenbundes e.V., der Sehbehindertenschule Nürnberg, dem VdK Bayern und den Arbeitsämtern Deggendorf und Passau, die mir die Promotion erst ermöglicht haben. Stellvertretend für viele seien Herr Schludermann, Herr Seuß, Herr Dr. Kozlik, Frau Willisch-Kozlik, Herr Pientka und Herr Fleischmann genannt.

Ich bedanke mich bei meinem Erstgutachter Prof. Dr.-Ing. Grass für die langjährige gute Betreuung. Außerdem danke ich Prof. Dr.-Ing. Snelting herzlich für seine Bereitschaft, die hier vorliegende Arbeit zu begutachten. Prof. Dr.-Ing. Hahn und Prof. Lengauer, PhD, haben es dankenswerterweise übernommen, mich im Rigorosum zu prüfen.

Es war immer eine Freude, mit meinen Kollegen am Lehrstuhl, Herrn Martin Grajcar, Herrn Markus Ramsauer und Herrn Franz Rautmann, zusammenzuarbeiten. Frau Eva Kapfer beeindruckte durch ihre gleichbleibende beständige Hilfsbereitschaft. Die langjährige gute und fruchtbare Kooperation mit dem Forschungsinstitut Forwiss unter der Leitung von Prof. Dr. Donner verdient spezielle Anerkennung. In besonderem Maße danke ich Herrn Reiner Kickingereeder, der mir den Zugang zu den für meine Fallstudien notwendigen Informationen ermöglicht und meine vielen Fragen fachkundig beantwortet hat. Herrn Klaus Schießl und Herrn Detlef Menzel gilt mein Dank für die großzügige Erlaubnis, die Rechner des CIP-Pools monatelang für meine aufwendigen Untersuchungen nutzen zu dürfen. Stellvertretend für die wissenschaftlichen Mitarbeiter, mit denen ich eine Vielzahl von Konzerten, Theateraufführungen und anderen Festivitäten besucht habe, möchte ich Herrn Andreas Fischer, Herrn Thomas Kast und Herrn Wolfgang Völkl erwähnen.

Zum Gelingen dieser Dissertation haben eine große Zahl von Hiwis, Praktikanten und Diplomanden durch ihre unermüdliche Arbeit und viele fruchtbringende Diskussionen beigetragen: Herr Stephan Absmeier, Herr Thomas Fischl, Herr Markus Heininger, Herr Michael Hekel, Herr Martin Klepper, Herr Josef Köck, Herr Markus Lausser, Herr Quoc Bao Le, Herr Roland Praml, Herr Peter Reitinger, Herr Peter Seidenhofer, Herr Christian Spies und Herr Jörg Stockinger. Für die gute und freundschaftliche Zusammenarbeit sei allen gedankt.

Die Mitarbeit in verschiedenen Gremien wie dem Fachbereichsrat, dem erweiterten Senat

und dem Konvent der wissenschaftlichen und künstlerischen Mitarbeiter hat mir viele interessante Einblicke über die Fakultät und die Universität hinaus ermöglicht. In diesem Zusammenhang möchte ich Herrn Dr. Martin Griehl und Herrn Dr. Christoph Herrmann, die mich durch ihr Engagement begeistert haben, dankend erwähnen.

Bergham, 22. Mai 2005

Walter Maydl

Zusammenfassung

Quid multa?

Diese Dissertation beschäftigt sich mit den Problemen bei der Entwicklung von effizienter und zuverlässiger Software für eingebettete Systeme.

Eingebettete Systeme sind inhärent nebenläufig, was mit einem Grund für ihre hohe Entwurfskomplexität darstellt. Aus dieser Nebenläufigkeit resultiert ein hoher Grad an Kommunikation zwischen den einzelnen Komponenten. Eine wichtige Forderung zur Vereinfachung des Entwurfsprozesses besteht in der getrennten Modellierung von Kommunikationsprotokollen und eigentlichen Verarbeitungsalgorithmen. Daraus resultiert eine höhere Wiederverwendbarkeit bei sich ändernden Kommunikationsstrukturen.

Die Grundlage für die sogenannten Datenflußsprachen bildet eine einfache von Gilles Kahn konzipierte Sprache für Parallelverarbeitung. In dieser Sprache besteht ein System aus einer Menge sequentieller Prozesse (Komponenten), die über Fifokanäle miteinander kommunizieren. Ein Prozess ist rechenbereit, wenn seine Eingangsfifos mit entsprechenden Daten gefüllt sind. Übertragen werden physikalische Signale, die als Ströme bezeichnet werden. Ströme sind Folgen von Werten ohne explizite Zeitangaben. Das Einsatzgebiet von Datenflußsprachen liegt in der Entwicklung von Programmen zur Bild- und Signalverarbeitung, typischen Aufgaben in eingebetteten Systemen. Die Programmierung erfolgt visuell, wobei man Icons als Repräsentanten parametrisierbarer Komponenten aus einer Bibliothek auswählt und mittels Kanten (Fifos) verbindet. Ein im allgemeinen dynamischer Scheduler überwacht die Ausführung des fertiggestellten Anwendungsprogramms.

Diese Arbeit schlägt ein universelleres Modell physikalischer Signale vor. Dabei werden zwei Ziele verfolgt:

1. Effiziente Kommunikation zwischen den Komponenten
2. Entwurfsbegleitende Überprüfung von Programmeigenschaften unter Verwendung komplexerer Komponentenmodelle

Zur Effizienzsteigerung werden nur relevante Werte innerhalb von Strömen übertragen. Dies erhöht zwar den Mehraufwand zur Kennzeichnung des Aufbaus eines Teilstroms, in praktischen Anwendungen ist die hier vorgestellte Methode jedoch effizienter. Die Einführung neuer Signalmerkmale erlaubt unterschiedlichste Überprüfungen der Einhaltung von Typregeln durch

die Eingangs- und Ausgangsströme einer Komponente. Anstelle einfacher Schaltregeln werden aufwendigere Kommunikationsprotokolle für die verschiedenen Arten von Komponenten eingeführt. Fifomaten (Fifo-Automaten) dienen als formale Grundlage. Mittels eines dezidierten Model-Checking-Verfahrens wird das Zusammenspiel der Fifomaten daraufhin untersucht, ob ein zyklischer Schedule existiert. Die Existenz eines solchen zyklischen Schedules schließt Speicherüberlauf und Deadlocks aus und garantiert darüber hinaus, daß das Programm nach endlicher Zeit wieder in die Ausgangssituation zurückfindet. Da im allgemeinen die Datenflußprogramme turingäquivalent sind, kann es allerdings zyklische Schedules geben, die das Verfahren nicht entdeckt.

Mit der hier vorgestellten und implementierten Methode wird die Entwicklungszeit korrekter Datenflußprogramme deutlich reduziert. Das neue Modell physikalischer Signale macht zudem die Ausführung effizienter.

Inhaltsverzeichnis

Vorwort	1
Zusammenfassung	3
Inhaltsverzeichnis	5
1 Motivation	11
1.1 Einführung	11
1.2 Problem der Qualitätssicherung	18
1.3 Relevanz der Problematik	19
1.4 Lösungsvorschlag	20
1.4.1 Zielsetzung	20
1.4.2 Lösungsansatz	21
1.4.3 Vorteile	23
1.5 Veröffentlichungen	23
1.6 Aufbau der Arbeit	24
2 Stand der Technik	25
2.1 Signalmodell	25
2.1.1 Physikalische Signale	25
2.1.2 Repräsentation physikalischer Signale im Rechner	26
2.1.3 Verschiedene Modelle physikalischer Signale	26
2.1.4 Physikalische Signale und Datenflußgraphen	27
2.2 Komponentenmodell	27
2.2.1 Komponentenbegriff	29
2.2.2 Überblick über Datenflußparadigmen	29
2.2.3 Synchroner Datenfluß	30
2.2.4 Boolescher Datenfluß	31
2.2.5 Dynamischer Datenfluß	31
2.2.6 Bezug zu Petrinetzen	32
2.2.7 Bezug zu Funktionaler Programmierung	33
2.2.8 Denotationelle Semantik	34
2.2.9 Scheduling	35

2.2.10	Analyse	36
2.3	Interface-Typsystem	36
2.3.1	Typen in textbasierten Programmiersprachen	36
2.3.2	Typen in der komponentenbasierten Softwareentwicklung	37
2.3.2.1	Polymorphismus	37
2.3.2.2	Typkonvertierung	38
2.3.3	Das Typsystem von Ptolemy II	38
2.3.3.1	Typdomäne	38
2.3.3.2	Typconstraints	39
2.3.3.3	Typbestimmung	40
2.4	Model Checking	40
2.4.1	Explizite Verifikation des Zustandsraumes	40
2.4.2	Modellierung von Kommunikationsprotokollen	41
2.4.2.1	Interface-Automaten	42
2.4.2.2	Communicating Finite State Machines	43
2.4.2.3	Extended Finite State Machines	44
2.4.3	Suchstrategien für den Erreichbarkeitsgraphen	44
2.4.3.1	Suche ohne Nebenwissen	45
2.4.3.2	Informierte Suche	46
2.4.4	Partial Order Reduction	47
2.4.5	Der Model Checker Spin	48
2.5	Kritikpunkte	49
3	Signalmodell	51
3.1	Probleme mit gebräuchlichen Signalmodellen	51
3.2	Anforderungen	52
3.2.1	Unterstützung bei der Fehlererkennung	55
3.2.2	Flexibilität in der Darstellung	56
3.2.3	Verminderung des Transport-Overheads	56
3.3	Neues Modell	58
3.3.1	Schrittweise Einführung	58
3.3.1.1	Analoge und digitale Signale	58
3.3.1.2	Signale als Folgen von Segmenten	59
3.3.1.3	Signale als Folgen von Blöcken	61
3.3.1.4	Signale als Folgen von gefärbten Token	61
3.3.2	Neue Signalmerkmale	61
3.3.3	Formale Beschreibung	62
3.3.4	Probleme und Lösungen	65
3.4	Innovative Aspekte	66

4	Komponentenmodell	69
4.1	Probleme mit gebräuchlichen Komponentenmodellen	69
4.2	Anforderungen	73
4.2.1	Kompatibilität mit dem neuen Signalmodell	73
4.2.2	Unterstützung bei der Fehlererkennung	73
4.2.3	Kontrolle über Komplexität und Modellierungsmächtigkeit	74
4.3	Neues Modell	75
4.3.1	Schrittweise Einführung	75
4.3.1.1	Funktion definiert auf Punkten	75
4.3.1.2	Funktion definiert auf Blöcken	76
4.3.1.3	Funktion definiert auf Token	76
4.3.1.4	Funktion definiert auf Strömen	77
4.3.2	Klassifikation von Datenflußkomponenten	77
4.3.3	Gefärbte Datenflußparadigmen	78
4.3.3.1	Colored SDF	78
4.3.3.2	Colored BDF	80
4.3.3.3	Colored DDF	85
4.3.3.4	Anwendungsbeispiel	86
4.3.4	Denotationelle Semantik	88
4.3.5	Probleme und Lösungen	89
4.4	Innovative Aspekte	91
5	Interface-Typsystem	93
5.1	Anforderungen	93
5.2	Beispiele zu Typconstraints	94
5.3	Neues Modell der Interfacetypen	96
5.3.1	Definitionen	96
5.3.2	Typdomäne	97
5.3.2.1	Typtraits	97
5.3.2.2	Polymorphismus	108
5.3.3	Typconstraints	109
5.4	Neues Typbestimmungsverfahren	109
5.4.1	Überblick über den Typbestimmungsalgorithmus	109
5.4.2	Constraint-Propagationsregeln	110
5.4.3	Beispiel zur Typbestimmung	113
5.4.4	Formale Beschreibung des Typbestimmungsalgorithmus	113
5.4.5	Probleme und Lösungen	118
5.4.6	Typkonvertierung	123
5.5	Innovative Aspekte	124

6	Model Checking	127
6.1	Anforderungen	127
6.2	Beispiel eines Kommunikationsprotokolls	128
6.3	Neues Modell der Kommunikationsprotokolle	129
6.3.1	Definitionen	130
6.3.2	Fifomaten	130
6.3.3	Kommunikationsprotokolle	132
6.4	Neues Model-Checking-Verfahren	134
6.4.1	Aufgabe	134
6.4.2	Überblick über den Model-Checking-Algorithmus	134
6.4.3	Komposition	135
6.4.4	Simulation	139
6.4.5	Formale Beschreibung des Model-Checking-Algorithmus	146
6.4.6	Probleme und Lösungen	172
6.4.7	Komposition vs. Simulation	177
6.5	Innovative Aspekte	178
7	Bild- und Signalverarbeitungswerkzeug Skylla	181
7.1	Anforderungen	181
7.2	Beispiel eines Entwicklungsablaufs	183
7.3	Das Werkzeug Skylla	185
7.3.1	Editor	187
7.3.2	Analyzer	187
7.3.2.1	Interface-Typsystem	188
7.3.2.2	Model Checker	188
7.3.3	Ablaufsteuerung	190
7.3.3.1	Server	191
7.3.3.2	Clients	192
7.3.3.3	Untersuchungen zur Kommunikation	193
7.3.3.4	Alternative Implementierung in ACE	194
7.3.3.5	Alternative Implementierung in CORBA	194
7.3.4	Komponentenbibliothek	195
7.3.4.1	Implementierung des Komponentenmodells in Haskell	195
7.3.4.2	Implementierung des Komponentenmodells in Design/CPN	195
7.3.4.3	C++-Bibliothek	195
7.3.5	Schedulbibliothek	197
7.4	Innovative Aspekte	198
8	Ergebnisse	201
8.1	Anforderungen	201
8.2	Testbedingungen	202
8.3	Interface-Typsystem	203
8.3.1	Systematische Tests	203

8.3.1.1	Einfügereihenfolge	203
8.3.1.2	Constraintkomplexität	205
8.3.1.3	Graphgröße	206
8.3.1.4	Löschen	206
8.3.2	Fallstudie: Qualitätskontrolle von Getränkedosen	207
8.4	Model Checking	210
8.4.1	Systematische Tests	211
8.4.1.1	Graphstruktur Rows	215
8.4.1.2	Graphstruktur Net	222
8.4.1.3	Graphstruktur Rows mit Rückkopplungen	225
8.4.1.4	Graphstruktur Net mit Rückkopplungen	231
8.4.1.5	Graphstruktur Random	235
8.4.1.6	Weitere Untersuchungen	235
8.4.1.7	Abschließende Bewertung	244
8.4.2	Vergleich mit dem Model Checker Spin	244
8.4.2.1	Graphstruktur Rows	248
8.4.2.2	Graphstruktur Net	249
8.4.2.3	Graphstruktur Rows mit Rückkopplungen	254
8.4.2.4	Graphstruktur Net mit Rückkopplungen	254
8.4.2.5	Weitere Untersuchungen	254
8.4.2.6	Abschließende Gegenüberstellung	254
8.5	Ablaufsteuerung	259
8.5.1	Fallstudie: Projektive Rekonstruktion in der Stereobildverarbeitung	259
8.5.2	Fallstudie: Qualitätskontrolle von Beilagscheiben	264
8.5.3	Fallstudie: Ermittlung des gemeinsamen Sichtbarkeitsbereichs zweier Kameras	265
8.6	Zusammenfassung der Ergebnisse	272
9	Zusammenfassung und Ausblick	277
9.1	Wiederaufgreifen der Problemstellung	277
9.2	Problemlösung	277
9.2.1	Modelle	278
9.2.2	Verfahren	279
9.2.3	Grenzen der Modelle und Verfahren	280
9.2.4	Überprüfung der Modelle und Verfahren	281
9.3	Innovative Aspekte und Bedeutung der Arbeit	281
9.4	Zusätzliche Einsatzgebiete der zentralen Ideen	283
	Abbildungsverzeichnis	285
	Tabellenverzeichnis	291
	Literaturverzeichnis	293

Kapitel 1

Motivation

Tolle lege! [Augustinus]

Das Thema dieser Arbeit lautet „Komponentenbasierte Softwareentwicklung für datenflußorientierte eingebettete Systeme“. Auf eine kurze Einführung in die Thematik folgt die Konkretisierung der Aufgabenstellung der vorliegenden Arbeit. Es wird die eigene Lösung kurz skizziert, wobei deren Vorteile und innovativen Aspekte beleuchtet werden. Ein Abriß über den Aufbau der Ausarbeitung schließt dieses Kapitel ab.

1.1 Einführung

Im folgenden werden zentrale Begriffe wie eingebettetes System, Datenflußgraph, Datenflußkomponente und Datenflußparadigma erläutert. Anhand eines einfachen Beispiels ist die Abarbeitung eines Datenflußgraphen erklärt. Danach fokussiert sich dieser Abschnitt auf zyklische Schedules, Speicherbedarf, Deadlocks und Typfehler in Datenflußgraphen. Abschließend wird noch auf komponentenbasierte Softwareentwicklung eingegangen.

Eingebettete Systeme: Abbildung 1.1 zeigt den schematischen Aufbau eines eingebetteten Systems. Ein eingebettetes System ist wie folgt charakterisiert (vergleiche [Lee02, BP03, Rus04]):

DEFINITION 1.1.1: Ein EINGEBETTETES SYSTEM ist eine Software-/Hardware-Einheit, welche in eine Maschine oder ein Gerät integriert ist. Das so entstehende Gesamtsystem ist nach außen nicht als Rechner, sondern nur als Träger intelligenter Systemfunktionen erkennbar. Es ist über Sensoren und Aktoren mit einem technischen Prozeß verbunden und übernimmt darin Überwachungs-, Steuerungs- beziehungsweise Regelungsaufgaben.

Eingebettete Systeme finden sich sowohl in alltäglichen Gebrauchsgegenständen wie Fernsehgeräten, Mobiltelefonen, Spielwaren als auch in komplexeren Systemen wie Autos, Fließbandsteuerungen und Flugzeugen.

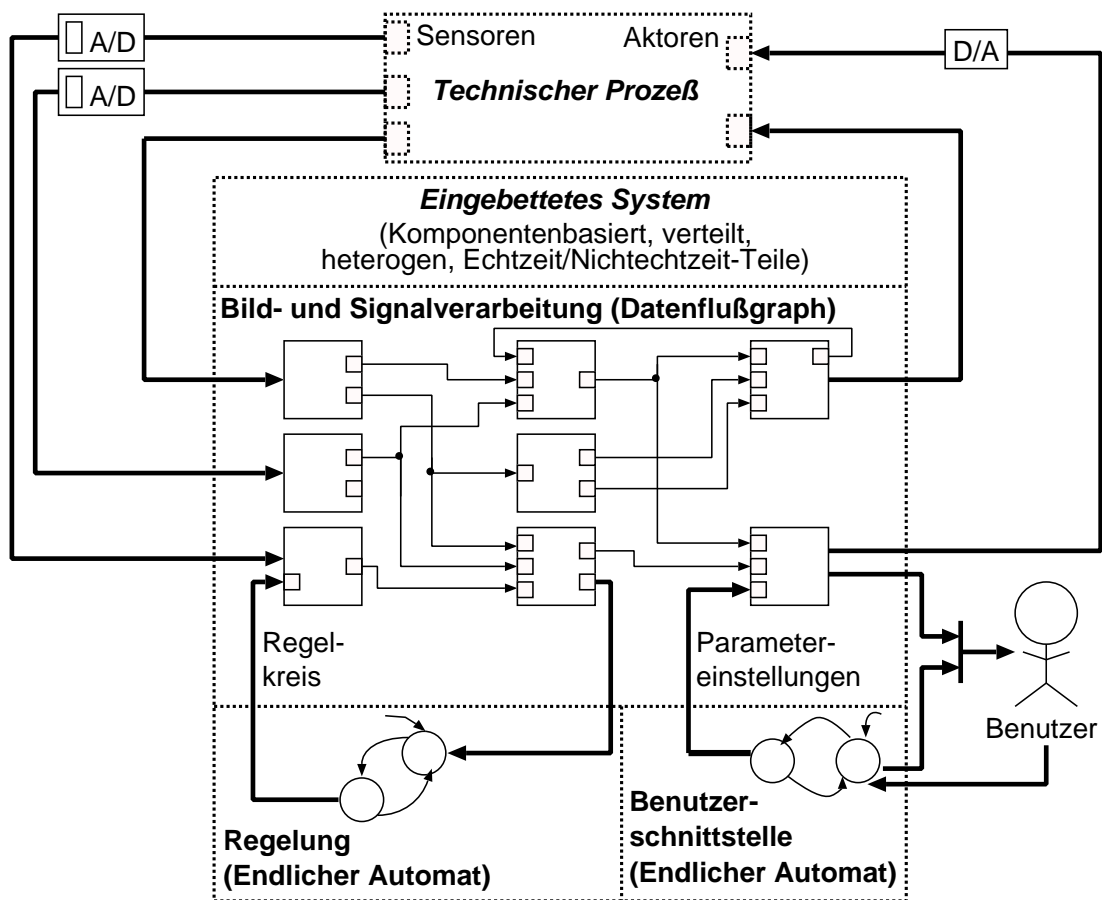


Abbildung 1.1: Eingebettetes System (Schematische Darstellung)

Für den Entwurf eines eingebetteten Systems werden gewöhnlich verschiedene Spezifikationssprachen verwendet. Eine grobe Einteilung unterscheidet zwischen KONTROLL- und DATENFLUSSDOMINANT, wobei in einem Programm beide Paradigmen eingesetzt werden können [BML⁺97, ELLSV97, Lee99, SLSV00, Bör00]. Dabei werden datenflußdominante Programmteile, welche im folgenden als DATENFLUSSGRAPHEN dargestellt und bezeichnet werden, vor allem zur Bild- und Signalverarbeitung eingesetzt. Kontrollflußdominante Programmteile (endliche Automaten und deren Erweiterungen [Har87, Ber98, HP99]) dienen beispielsweise zur Feinjustierung des Datenflußgraphen, indem sie Parameter zur Laufzeit anpassen beziehungsweise auf Benutzereingaben reagieren. Ein eingebettetes System [BML⁺97] setzt sich in der Regel aus Echtzeit- und Nichtechtzeiteilen zusammen. Echtzeiteile sind zum Beispiel für die Ansteuerung eines Regelkreises zuständig, wohingegen das Sammeln von Statistikdaten keine Echtzeitfähigkeit erfordert. Tasks, die diesen Teilen zugeordnet werden, können nun periodisch, aperiodisch beziehungsweise sporadisch sein. Die Ausführung eines eingebetteten Systems erfolgt im allgemeinen asynchron und verteilt [Bro03a].

Datenflußgraphen und Datenflußkomponenten: Diese Arbeit ist fokussiert auf datenflußorientierte eingebettete Systeme [Lee02]. Betrachten wir daher einen DATENFLUSSGRAPHEN genauer (siehe Abbildung 1.1). Die Knoten des Datenflußgraphen repräsentieren Programmbausteine (KOMponentEN)¹. Um zu betonen, daß eine Komponente in Zusammenhang mit Datenflußgraphen verwendet wird, spricht man auch von Datenflußkomponente. Beispiele für Datenflußkomponenten aus dem Bereich der Bild- und Signalverarbeitung sind digitale Filter, Fast-Fourier-Transformation, Kantenextraktion oder PID-Regler. Diese Datenflußkomponenten können in einer beliebigen Programmiersprache implementiert sein und sind gekennzeichnet durch klare Schnittstellen und explizite Kontextabhängigkeiten. Datenflußkomponenten werden in der Regel in binärer Form als Shared Libraries weitergegeben. Diese Bausteine sind mittels Fifos, deren Kapazitäten gegebenenfalls beschränkt sind, zu komplexeren Anwendungen verbunden. Das heißt, die Datenflußkomponenten kommunizieren asynchron. Datenflußparadigmen stellen somit DOMÄNENSPEZIFISCHE PROGRAMMIERSPRACHEN dar, die unter anderem auf der von Gilles Kahn definierten FIXPUNKTSEMANTIK [Kah74, KM77, Lee97] aufbauen. Bei der Entwicklung von Datenflußgraphen werden in der Regel VISUELLE PROGRAMMIERTECHNIKEN eingesetzt.

Für das Verständnis von Datenflußgraphen ist es wesentlich, sich die einzelnen Schritte bei ihrer Abarbeitung vor Augen zu führen. In Abbildung 1.2 (a) sind drei Datenflußkomponenten aus dem Bereich der Bildverarbeitung dargestellt. Die Bildquelle liest Bilder zum Beispiel von einer Kamera ein. Die interne Komponente glättet diese Bilder. Die Displaykomponente stellt dann sowohl das unveränderte als auch das geglättete Bild auf einem Bildschirm dar. Die von den Datenflußkomponenten erzeugten beziehungsweise konsumierten Daten sind in Datencontainern, sogenannten TOKEN, gekapselt, welche in der Abbildung als große schwarze Punkte wiedergegeben sind.

1. Als erstes wird die Bildquelle rechenbereit, sobald die Kamerahardware das Vorhandensein eines digitalen Bildes signalisiert (Abbildung 1.2 (b)).
2. Die Bildquelle wird nun durch einen Scheduler rechnerisch gesetzt, liest das Bild ein und schreibt es in seine Ausgabefifos. Ist eine Datenflußkomponente rechnerisch, wird dies hier durch Schwarzfärbung des zugehörigen Knotens visualisiert (Abbildung 1.2 (c)).
3. Die Glättungskomponente wird rechenbereit (Abbildung 1.2 (d)).
4. Sobald die Glättungskomponente vom Scheduler rechnerisch gesetzt wird, übernimmt sie das an ihrem Eingang anliegende Bild, glättet dieses und gibt es an ihre Ausgabefifo weiter (Abbildung 1.2 (e)).
5. Jetzt ist auch die Displaykomponente rechenbereit, da an ihren beiden Eingängen Daten anliegen. Sobald die Datenflußkomponente ausgeführt wird, übernimmt sie die Bilder und stellt diese auf dem zugeordneten Monitor dar (Abbildung 1.2 (f)).
6. Der Datenflußgraph befindet sich wieder im Ausgangszustand und die Abarbeitung von Eingabedaten beginnt von vorne (Abbildung 1.2 (g)).

¹Der hier zugrundeliegende Komponentenbegriff basiert auf [Szy02] (vergleiche Abschnitt 2.2.1).

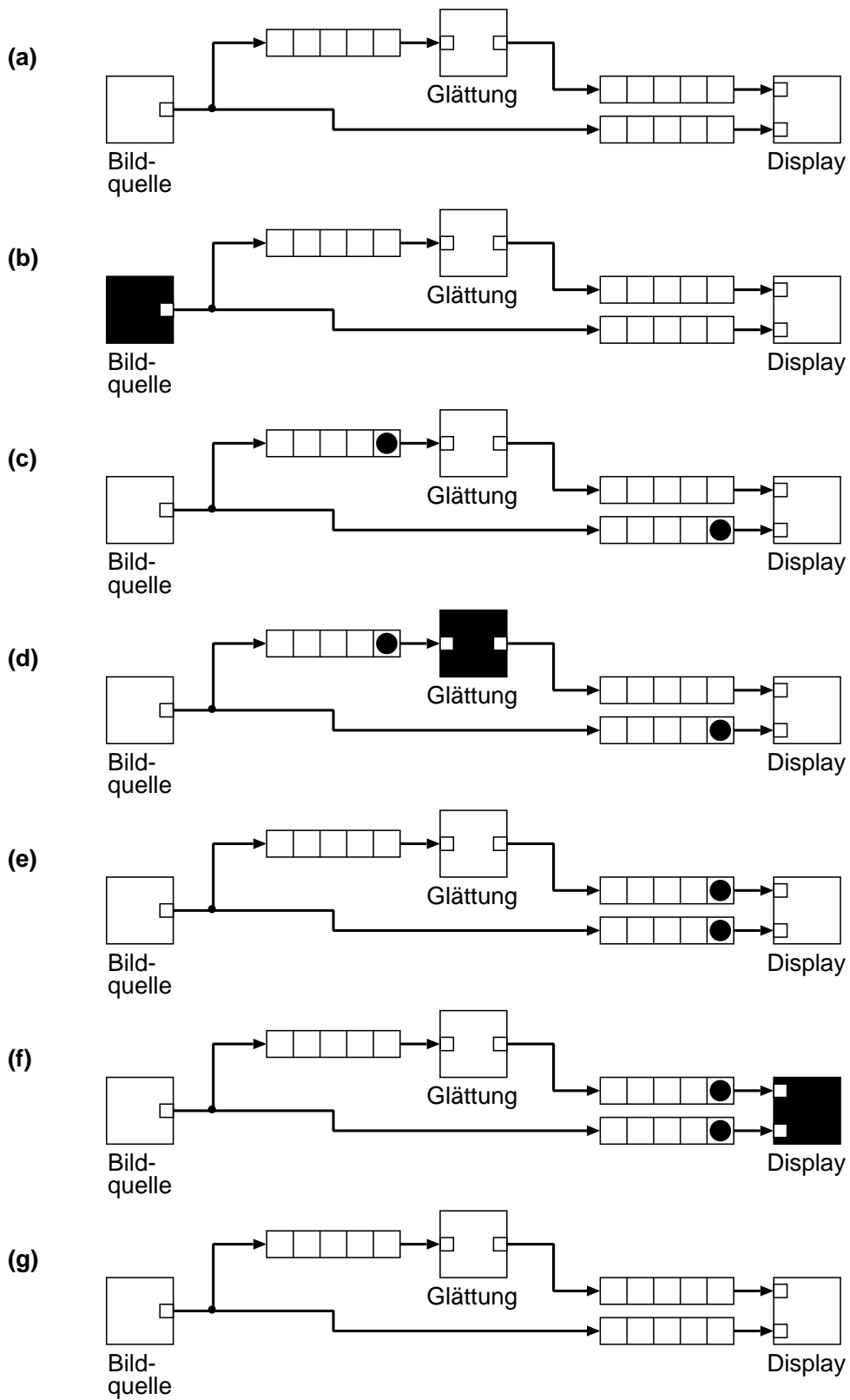


Abbildung 1.2: Beispielhafte Abarbeitung eines Datenflußgraphen

Die Daten fließen zwischen den Datenflußkomponenten, woraus sich der Begriff „DATENFLUSS“ ableitet.

Datenflußparadigmen: Man unterscheidet Datenflußkomponenten anhand der Regeln zur Bestimmung ihrer Rechenbereitschaft. Datenflußkomponenten, die dem Datenflußparadigma – dem Analogon einer klassischen Programmiersprache – SYNCHRONER DATENFLUSS (SDF) zugeordnet werden, sind rechenbereit, wenn entsprechend den jeweiligen Gewichten ausreichend viele Token an ihren Eingängen vorhanden sind. Gemäß den zugrundeliegenden Schaltregeln konsumiert eine rechenbereite Datenflußkomponente dieser Kategorie die durch die Gewichtung vorgegebene Anzahl von Token von ihren Eingängen. Nach der internen Verarbeitung der Daten werden von der Datenflußkomponente soviele Token in die Ausgabefifos geschrieben, wie die jeweiligen Kantengewichte vorschreiben. Datenflußkomponenten dieses Datenflußparadigmas speichern intern keine Daten. Datenflußgraphen, die nur solche Datenflußkomponenten enthalten, sind analysierbar, da synchroner Datenfluß nicht turingäquivalent ist. Es ist möglich, einen zyklischen Schedule zu berechnen und zu entscheiden, ob ein Datenflußgraph einen Deadlock enthält beziehungsweise wie groß die Fifokapazitäten für eine erfolgreiche Ausführung des Datenflußgraphen sein müssen (siehe unten).

Durch die Ergänzung von SDF um die Datenflußkomponenten **Switch** und **Select** erhält man das Paradigma BOOLESCHER DATENFLUSS (BDF). Derartige Datenflußprogramme sind turingäquivalent. Fragen nach zyklischen Schedules, Deadlockfreiheit und Speicherbedarf sind nicht mehr entscheidbar. Fügt man zu BDF noch die nichtdeterministische Datenflußkomponente **Merge** hinzu, so entsteht das Datenflußparadigma DYNAMISCHER DATENFLUSS (DDF). Datenflußprogramme verlieren dadurch zusätzlich noch die Eigenschaft, für gleiche Eingaben immer gleiche Ausgaben zu liefern. Damit liegt der bekannte GEGENSATZ ZWISCHEN ANALYSIERBARKEIT UND MODELLIERUNGSMÄCHTIGKEIT vor. Auf diese Sachverhalte wird in Kapitel 2 genauer eingegangen.

Da also die Kanten zwischen den Datenflußkomponenten nur eine partielle Ausführungsreihenfolge festlegen, erfordert die vorhandene explizite Parallelität einen Scheduler, der entscheidet, wann und auf welchem Rechner (bei verteilter Ausführung) rechenbereite Datenflußkomponenten ausgeführt werden. In der Regel – außer bei SDF – ist DYNAMISCHES SCHEDULING notwendig, da die Bestimmung der Rechenbereitschaft im allgemeinen Fall erst zur Laufzeit erfolgen kann.

Zyklischer Schedule: Eine Aktivierungsreihenfolge von Datenflußkomponenten, die wieder in der Ausgangssituation, welche durch eine spezifische Belegung der Fifos und durch spezifische interne Zustände der Datenflußkomponenten gegeben ist, endet, nennt man ZYKLISCHER SCHEDULE. Zyklische Schedules sind deshalb besonders interessant, da sie sich analysieren lassen – man muß nur eine endliche Folge von Komponentenaktivierungen untersuchen – und sich für eine unendliche Ausführung eignen.

Speicherbedarf: Anhand eines zyklischen Schedules läßt sich der SPEICHERBEDARF des Datenflußgraphen, der durch die Fifokapazitäten gekennzeichnet ist, bestimmen. In obigem Beispiel

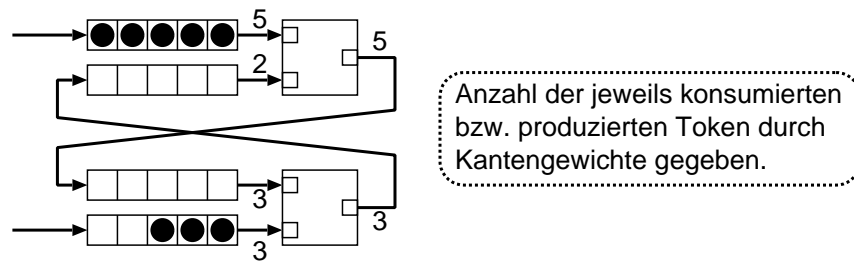


Abbildung 1.3: Deadlock (blockiertes Lesen)

reicht eine Fifokapazität von 1 für alle Fifos aus. Wächst der Speicherbedarf eines Datenflußgraphen während seiner Ausführung ständig an, so spricht man von **SPEICHERÜBERLAUF**.

Deadlocks: Häufige Fehlerquellen in einem Datenflußgraphen stellen **DEADLOCKS** dar. In einem Datenflußgraphen können Deadlocks aus **BLOCKIERTEM LESEN** beziehungsweise **BLOCKIERTEM SCHREIBEN** entstehen. Abbildung 1.3 zeigt eine typische Leseblockade. Beide Datenflußkomponenten erwarten jeweils eine fixe durch Gewichte vorgegebene Anzahl von Token an ihren Eingängen, um rechenbereit zu werden. Füllt man die Fifos vor Beginn der Abarbeitung mit **INITIALISIERUNGSTOKEN** auf, so daß die beiden Datenflußkomponenten rechenbereit werden, so ist die Leseblockade aufgehoben. Allerdings kommt es nach einigen Ausführungsschritten in Abbildung 1.4 (e) zu einer Schreibblockade, da die Fifokapazität, welche durch die Anzahl der Plätze symbolisiert ist, überschritten würde. Blockiertes Schreiben resultiert also aus festen Fifokapazitäten, die nicht überschritten werden dürfen. Ist eine Fifo voll, dann kann die produzierende Datenflußkomponente keine weiteren Daten mehr erzeugen.

Aus diesen Beispielen wird ersichtlich, daß die Rechenbereitschaft von Datenflußkomponenten sowohl vom Vorhandensein von Token an den jeweiligen Eingängen als auch vom Vorhandensein ausreichenden Fifospeicherplatzes an den jeweiligen Ausgängen abhängt.

Inkompatible Daten: Neben den geschilderten Fehlern wie Deadlocks wegen blockiertem Lesen oder Schreiben, die auf der Abstraktionsebene der Token auftreten, gibt es auch eine Reihe von möglichen Fehlern auf niedrigeren Abstraktionsebenen von digitalisierten physikalischen Signalen. Betrachtet man eine beliebige Datenflußkomponente und die an ihren Schnittstellen anliegenden digitalen physikalischen Signale (vergleiche Abbildung 1.5), so können unterschiedlichste Probleme wie beispielsweise nichtzusammenpassende Abstraten oder inkompatible Wertemengen wie **int** und **double** auftreten. Neben dem Definitionsbereich und dem Wertebereich spielen auch Aspekte des Datentransports bei der Ermittlung der Kompatibilität zweier Komponenteninterfaces eine wesentliche Rolle (vergleiche Kapitel 3 beziehungsweise Kapitel 5).

Komponentenbasierte Softwareentwicklung: Werden komplexe Programme wie zum Beispiel Datenflußgraphen aus Komponenten zusammengesetzt, spricht man von **KOMPONENTEN-**

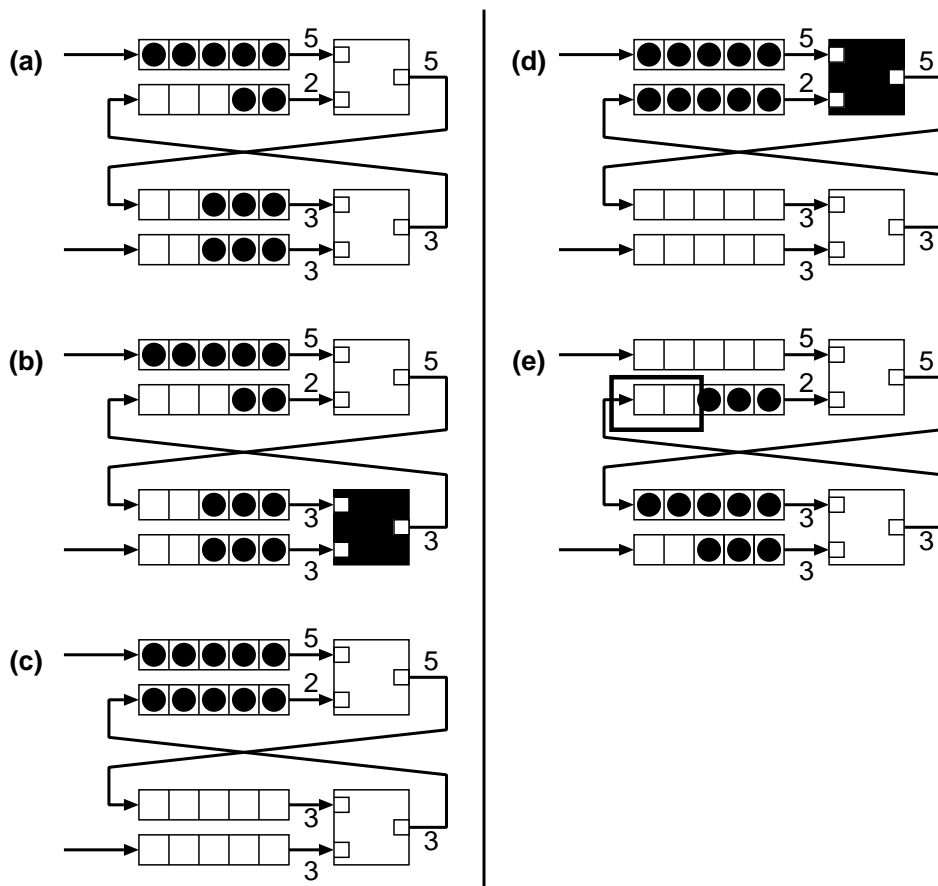


Abbildung 1.4: Deadlock (blockiertes Schreiben)

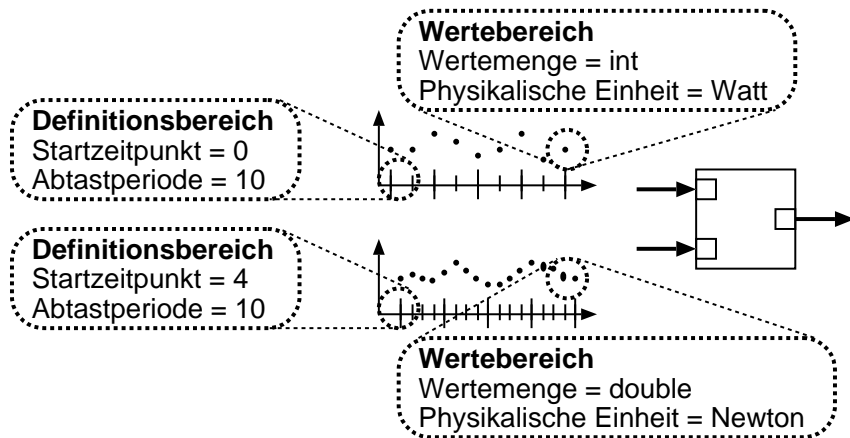


Abbildung 1.5: Datenflußkomponente mit digitalen Signalen

BASIERTER SOFTWAREENTWICKLUNG. Bei der Kombination von Datenflußparadigmen mit komponentenbasierter Softwareentwicklung kommen alle wesentlichen Vorteile des letztgenannten Verfahrens zum Tragen [Szy02]:

- WIEDERVERWENDUNG findet in großem Ausmaß statt, da die in den Komponenten gekapselten Bild- und Signalverarbeitungsalgorithmen wie zum Beispiel Fast-Fourier-Transformation beziehungsweise Filteralgorithmen Bestandteile vieler Anwendungen im Bereich der Bild- und Signalverarbeitung sind.
- Komponenten können LEICHT an sich ändernde Anforderungen ANGEPASST werden, da nur explizite Schnittstellen nach außen existieren.
- Dies zieht eine VERBESSERTE WARTBARKEIT von Datenflußgraphen nach sich, da bei gleichbleibenden Schnittstellen nur alte Komponenten durch neue ausgetauscht werden müssen.
- Ein wesentlicher Punkt ist der GESTEIGERTE SCHUTZ GEISTIGEN EIGENTUMS, der daraus resultiert, daß die Komponenten in binärer Form weitergegeben werden.

1.2 Problem der Qualitätssicherung

Abbildung 1.6 veranschaulicht das zentrale Problem beim Entwurf von Datenflußgraphen: In einem großen unübersichtlichen Datenflußprogramm tritt ein Laufzeitfehler auf. Ein solcher Fehler zieht in der Regel schwerwiegende Folgen nach sich, da von eingebetteten Systemen, zum Beispiel der Kontrolleinheit eines Fließbandes, häufig gefordert wird, daß diese endlos und ohne Unterbrechung ausgeführt werden. Jeder Deadlock und daraus gegebenenfalls resultierende Systemabsturz kostet zumindest Zeit und Geld, in manchen Fällen sogar Menschenleben. Aufgrund der Größe des Datenflußgraphen muß man mit einer langwierigen Debugphase rechnen.

Verallgemeinert man diese Beobachtung, so liegt das zentrale Problem in folgendem begründet: Je später ein Fehler erkannt wird, desto größer sind die aus diesem Fehler resultierenden Kosten. Abbildung 1.7 zeigt den rapiden Anstieg der Fehlerkosten in späten Phasen des Softwareentwurfsprozesses wie beispielsweise Test und Inbetriebnahme. Aktuell wird der größte Aufwand zur Qualitätssicherung in den späten Softwareentwurfsphasen eines eingebetteten Systems getrieben (siehe Abbildung 1.8) [Ame02, Rus04]. Andererseits entstehen die meisten Fehler in den frühen Phasen wie Analyse und Design (vergleiche Abbildung 1.7).

Diese Problematik der späten und kostenintensiven Fehlererkennung wird weiter verschärft, da eine zunehmende Komplexität eingebetteter Software kombiniert mit immer kürzeren TIME-TO-MARKET-ZYKLEN und einem steigenden Zwang zur Kostensenkung gleichbleibend hohen Qualitätsanforderungen gegenübersteht [RB02]. Da sich dadurch der Zeitaufwand für Test und Inbetriebnahme in Zukunft noch vergrößern wird [Egg02], spricht man in diesem Zusammenhang auch von einer QUALITÄTSFALLE [Ame02, Rus04]. In [Mey99] ist das Problem klar beschrieben:

„We build software that’s not very good and, through brute force, debug it into correctness.“

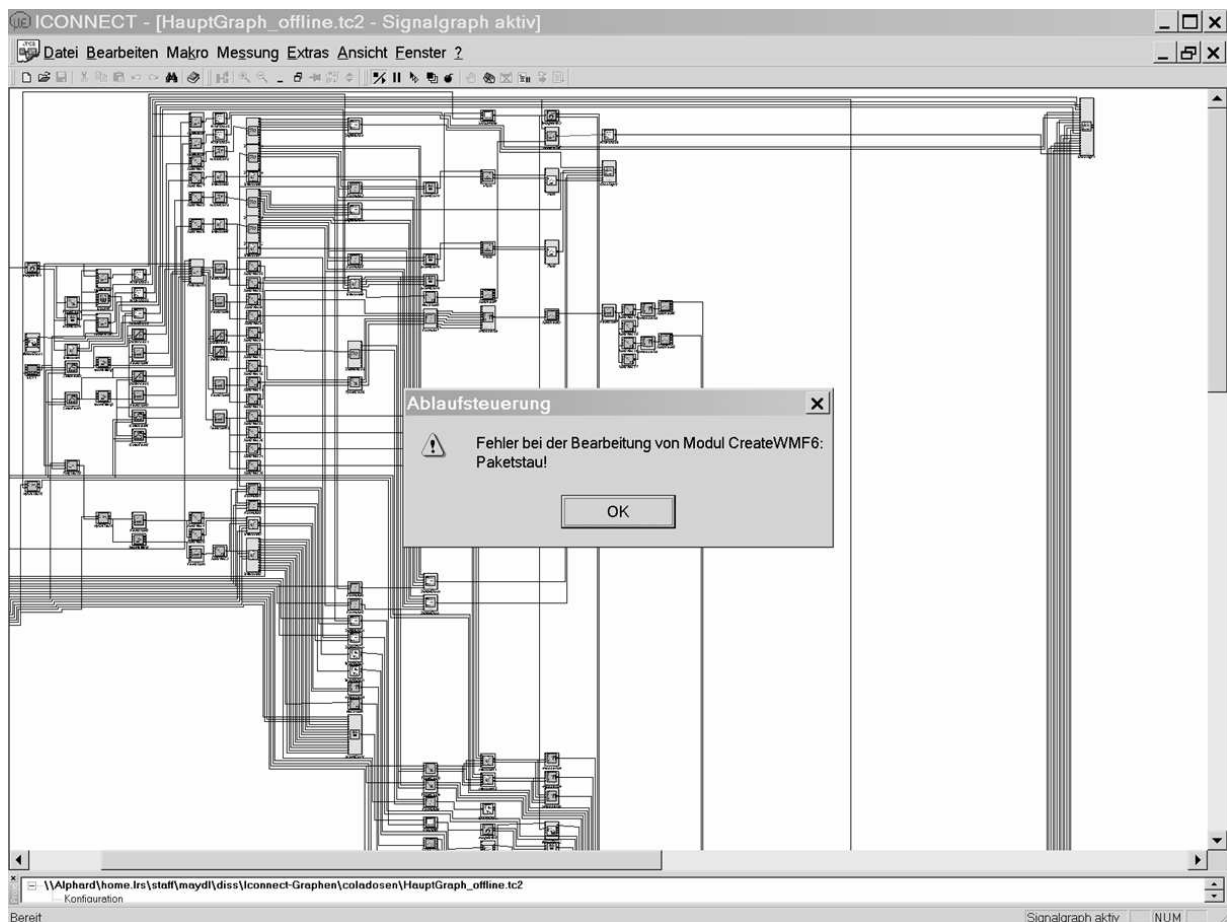


Abbildung 1.6: Unübersichtlicher Datenflußgraph mit Laufzeitfehler (kommerziell eingesetztes Tool)

1.3 Relevanz der Problematik

Das im vorigen Abschnitt diskutierte Problem stellt einen lohnenden und aktuellen Forschungsschwerpunkt dar. Dies zeigen die vielfältigen Veröffentlichungen zu diesem Thema. Dabei sehen viele einen geeigneten Lösungsansatz in einer Kombination aus KOMPONENTENBASIERTER SOFTWAREENTWICKLUNG mit ENTWURFSBEGLEITENDER FRÜHER QUALITÄTSSICHERUNG. Es seien dazu nur der Wegweiser für Forschung und Lehre im Software-Engineering eingebetteter Systeme von Manfred Broy und Wolfgang Pree [BP03], das Verbundprojekt Embedded Quality (Equal) [KCF⁺02, Rus04], die Studie des Bundesamtes für Sicherheit in der Informationstechnik bezüglich neuer Trends und Entwicklungen in der Informationstechnik [GHS⁺03] beziehungsweise die diversen Arbeiten im Rahmen des Projektes Ptolemy [Lee99, Lee02] genannt. Der EINSATZ FORMALER METHODEN wird in diesem Zusammenhang als ein wichtiger Schritt zur entwurfsbegleitenden frühen Qualitätssicherung angesehen [Mey99].

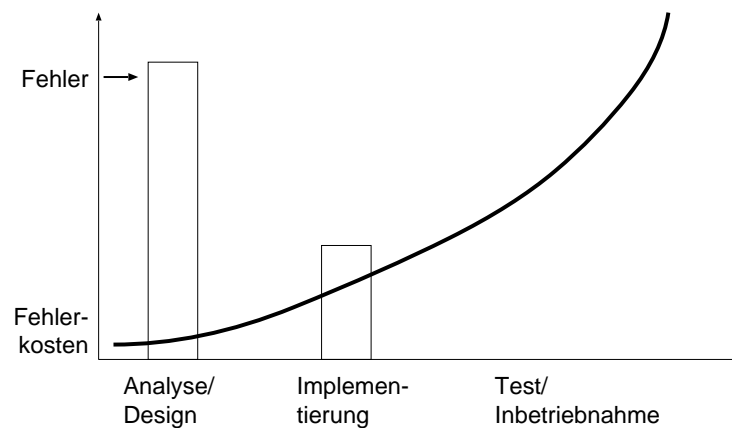


Abbildung 1.7: Fehlerverteilung und Fehlerkosten in unterschiedlichen Phasen eines Softwareprojekts

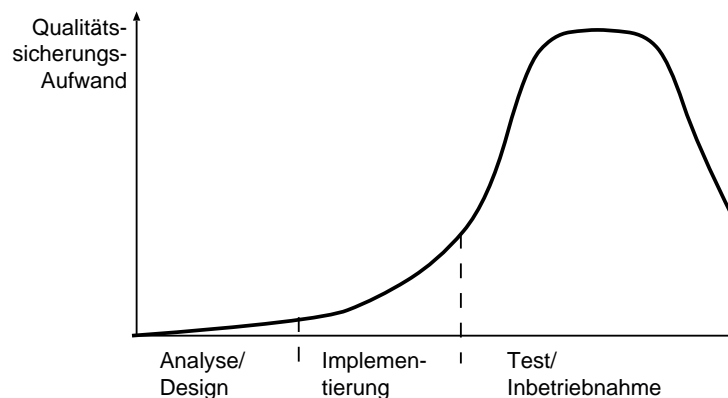


Abbildung 1.8: Qualitätssicherungsaufwand in unterschiedlichen Phasen eines Softwareprojekts

1.4 Lösungsvorschlag

Im folgenden werden die Ziele dieser Arbeit, der Lösungsansatz mit den darin enthaltenen innovativen Aspekten und dessen Vorteile erläutert.

1.4.1 Zielsetzung

In der hier vorliegenden Arbeit werden zwei Ziele verfolgt:

1. EFFIZIENTE KOMMUNIKATION zwischen den Komponenten
2. ENTWURFSBEGLEITENDE ÜBERPRÜFUNG VON PROGRAMMEIGENSCHAFTEN wie zyklischen Schedules, Deadlocks und Speicherbedarf unter Verwendung komplexerer Komponentenmodelle

1.4.2 Lösungsansatz

Als Lösung schlage ich folgende Maßnahmen vor (siehe Abbildung 1.9), wobei jeweils kurz beschrieben ist, wie der Stand der Technik in Entwurf und Analyse von datenflußorientierten komponentenbasierten eingebetteten Systemen dadurch erweitert wird:

- **SIGNALMODELL:** Grundlage der entwurfsbegleitenden Fehlererkennung bildet ein neues Signalmodell, welches die effiziente Beschreibung von unendlichen physikalischen Signalen erlaubt, die aus
 - äquidistant abgetasteten Signalwerten,
 - Signalsegmenten bestehend aus äquidistant abgetasteten relevanten Signalwerten und dazwischenliegenden Pausen, sogenannten Totzeiten, wie sie beispielsweise bei an einem Fließband gemessenen Signalen, akustischen Signalen wie Sprache beziehungsweise neuronalen Signalen auftreten, oder
 - nicht äquidistant abgetasteten Signalwerten (Events) wie zum Beispiel Tastatureingaben,

zusammengesetzt sind. Somit ist ein relevanter Aspekt dieses Signalmodells auch der effiziente Datentransport, welcher wichtig für die verteilte Ausführung von Datenflußgraphen ist. Insgesamt führt dies zu einer Erweiterung aktueller Signalmodelle um zusätzliche Signalmerkmale bezüglich des Zeitbereichs (Startzeiten und Abtastperioden), des Wertebereichs (Wertemengen und physikalische Einheiten) und bezüglich Attributen des Signal- datentransportes.

- **KOMPONENTENMODELL:** Dieses Signalmodell erfordert ein neues Komponentenmodell, was in der Definition neuer Datenflußparadigmen resultiert. Die klassischen Datenflußparadigmen SDF, BDF und DDF (siehe Abschnitt 2.2) werden erweitert zu gefärbten Datenflußparadigmen. Farben erlauben nicht nur eine Markierung Boolescher Werte, wie bei BDF, sondern können auch zur Darstellung von Signalmerkmalen verwendet werden. Dieses neue Komponentenmodell paßt gut in Szyperkis Definition einer Komponente (siehe Abschnitt 2.2.1). Außerdem erlaubt dieses Komponentenmodell die Definition von Typconstraints, die zum Beispiel die Gleichheit von Abtastperioden fordern, und von Kommunikationsprotokollen. Damit eine Komponente und der zugehörige Datenflußgraph **WOHLDEFINIERT** sind, müssen die genannten Constraints und Protokolle erfüllt sein.
- **INTERFACE-TYPSYSTEM:** Die durch Interfacetypen erfaßte Information wird durch die zusätzlichen Signalmerkmale signifikant erweitert. Die Typconstraints, welche in diesem **NEUEN MODELL DER INTERFACETYPEN** spezifiziert werden können, sind komplexer als bei anderen Typsystemen in diesem Anwendungsfeld. So ist es zum Beispiel möglich, mehrwertige Polynome zu definieren. Damit kann das Verhalten von Datenflußkomponenten präziser erfaßt werden. Um die Einhaltung von Typconstraints zu garantieren, wird ein **NEUARTIGES VERFAHREN ZUR TYPBESTIMMUNG** angewandt. Durch den Aufruf des Typbestimmungsalgorithmus während jedes Entwurfsschrittes erhält der

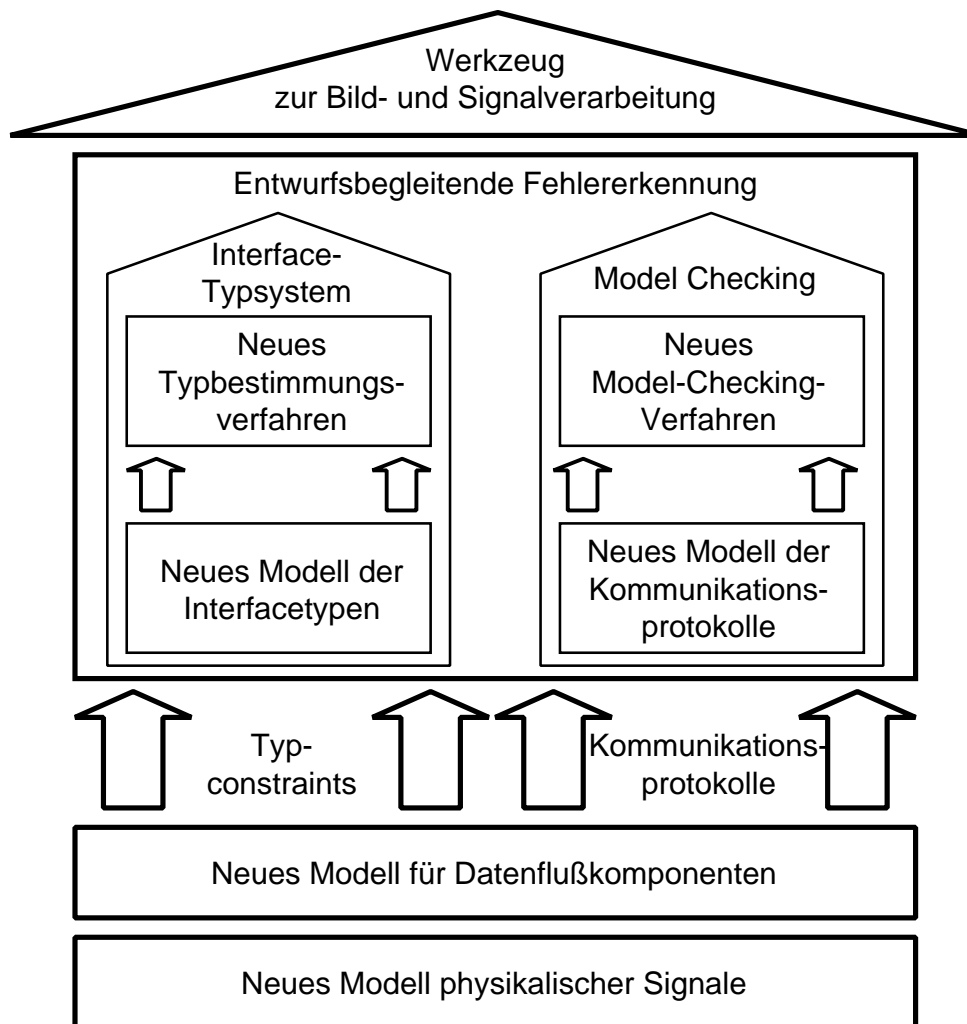


Abbildung 1.9: Eigener Lösungsansatz

Benutzer eine frühzeitige Rückmeldung. Außerdem nutzt der Typbestimmungsalgorithmus INKREMENTELL soweit wie möglich die Ergebnisse vorheriger Berechnungsschritte. Probleme, welche durch das Interface-Typsystem aufgespürt werden, umfassen SPEICHERÜBERLAUF, VORZEITIGE PROGRAMMTERMINIERUNG oder FEHLERHAFTE VERRECHNUNG VON nicht demselben Meßobjekt zugeordneten SIGNALDATEN. Diese Probleme sind gegebenenfalls gefährlich für die Prozeßumgebung. Zudem ist das Interface-Typsystem sehr flexibel, indem es POLYMORPHISMUS, OVERLOADING und AUTOMATISCHE TYPKONVERTIERUNG erlaubt. Das Interface-Typsystem ist zudem leicht anpaßbar, indem man Typchecks hinzugefügt oder bestehende verändert.

- MODEL CHECKER: FIFOMATEN (Fifo-Automaten) stellen eine Erweiterung von Communicating Finite State Machines [Hol91] dar und bilden ein NEUES MODELL DER KOM-

MUNIKATIONSPROTOKOLLE von Datenflußkomponenten. Mit Hilfe dieses Kommunikationsmodells wird durch ein NEUES MODEL-CHECKING-VERFAHREN die Protokollkompatibilität der einzelnen Datenflußkomponenten untersucht. Dabei spielen Aspekte wie die BESTIMMUNG EINES ZYKLISCHEN SCHEDULES, von DEADLOCKS und des SPEICHERBEDARFS – in Abhängigkeit von der Berechnungskomplexität – eine wichtige Rolle. Ist ein betrachtetes Datenflußparadigma turingäquivalent, so können nur noch mittels Simulation Gegenbeispiele gefunden werden.

Diese neuen Modelle und Verfahren bilden die Grundlage für ein Werkzeug zur Bild- und Signalverarbeitung namens **Skylla**. Damit entspricht dieser Lösungsansatz der in [Bro03a] geforderten Vorgehensweise:

„Constructing, analyzing, and arguing in terms of models
is at the heart of science.“

1.4.3 Vorteile

Die Hauptvorteile dieses Lösungsansatzes liegen

- in einer kompakten Darstellung physikalischer Signale, die einen EFFIZIENTEN DATENTRANSPORT zur Folge hat,
- im FRÜHZEITIGEN ENTWURFSBEGLEITENDEN AUFDECKEN VON FEHLERN wie Speicherüberlauf und Deadlocks,
- im Ausnutzen der Vorteile der KOMPONENTENBASIERTEN SOFTWAREENTWICKLUNG (vergleiche Abschnitt 1.1) und
- darin, daß KEINE VERTIEFTEN KENNTNISSE der angewandten formalen Verfahren notwendig sind.

Der letztgenannte Punkt ist für Anwendungsentwickler, die Experten in ihrer Anwendungsdomäne wie beispielsweise der Bild- und Signalverarbeitung sind, aber keine vertieften Kenntnisse über Programmanalysemethodiken besitzen, besonders interessant. Diese Entwickler können sich ganz auf die zu lösende Aufgabe konzentrieren.

1.5 Veröffentlichungen

Einen Überblick über das Signalmodell, das Komponentenmodell und das darauf aufbauende Interface-Typsystem und den Model Checker liefert [May04a]. [MSG02] präsentiert diverse Komponentenmodelle für Signalverarbeitungs-, Adapter- und Kontrollflußkomponenten. [MG03] stellt das Interface-Typsystem im Detail vor. Fifomaten als Modellierungsmethodik für das Kommunikationsverhalten von Datenflußkomponenten und die Regeln zu deren Komposition finden sich in [May04c]. Das Model-Checking-Verfahren ist in [May04b] dargelegt. Die

Ablaufsteuerung für die verteilte Ausführung von Datenflußgraphen ist in [MSG04] dargestellt. Visuelle Programmieretechniken für Datenflußgraphen wurden in [MRSS01] vorgestellt. [MG05] gibt zusammenfassend den aktuellen Stand der Technik für Behavioral Types, welche Modelle und Verfahren zur Überprüfung der Kompatibilität von Kommunikationsprotokollen darstellen, wieder. Außerdem wurde im Jahr 2003 ein Überblick über diese Dissertation auf dem Ph.D.-Forum der Konferenz „Design Automation and Test in Europe“ (DATE) präsentiert.

1.6 Aufbau der Arbeit

In Kapitel 2 wird der Stand der Technik beleuchtet. Kapitel 3 beinhaltet das neue Signalmodell. Darauf aufbauend stellt Kapitel 4 das neue Modell für Datenflußkomponenten vor. Kapitel 5 ist auf das neuartige Interface-Typsystem fokussiert, wohingegen Kapitel 6 den auf Datenflußgraphen zugeschnittenen Model Checker beschreibt. Das Signalverarbeitungswerkzeug *Skylla* ist in Kapitel 7 beschrieben. Die wesentlichen Ergebnisse der implementierten Verfahren werden in Kapitel 8 vorgestellt. Kapitel 9 liefert eine Zusammenfassung und einen kurzen Ausblick.

Kapitel 2

Stand der Technik

Multi multum dicunt.

Dieses Kapitel präsentiert den Stand der Technik zu den vier Schwerpunkten dieser Arbeit: Signalmodell, Komponentenmodell, Interface-Typsystem und Model Checking. Die abschließenden Bemerkungen bewerten diesen Stand kritisch.

2.1 Signalmodell

Nach der Definition des Begriffs physikalisches Signal wird auf die Darstellung physikalischer Signale im Rechner, auf unterschiedlich genaue Signalmodelle und auf das in Datenflußparadigmen üblicherweise verwendete Signalmodell eingegangen.

2.1.1 Physikalische Signale

DEFINITION 2.1.1: *Ein PHYSIKALISCHES SIGNAL ist das Erscheinungsbild einer physikalischen Information [Hes93, Czi91]. Dabei ist ein physikalisches Signal beschreibbar durch*

- *eine mathematische Funktion in geschlossener Form,*
- *ein Verteilungsgesetz (zum Beispiel für ein stochastisches Signal) oder*
- *eine Meßreihe.*

Beispiele für physikalische Signale sind [Czi91]:

- akustische Signale wie Sprache oder Musik
- Signale aus dem Bereich der Medizin, wie sie zum Beispiel beim Elektrokardiogramm oder Elektroencephalogramm entstehen

- Signale aus dem Bereich der Geologie wie zum Beispiel seismische Wellen
- Ortungssignale wie Radar oder Sonar.

Ein analoges physikalisches Signal f_a ist eine Abbildung, welche sich durch einen kontinuierlichen Zeitbereich \mathbb{T} und einen kontinuierlichen Wertebereich \mathbb{X} auszeichnet:

$$f_a : \mathbb{T} \rightarrow \mathbb{X} . \quad (2.1)$$

2.1.2 Repräsentation physikalischer Signale im Rechner

Um physikalische Signale im Rechner weiterverarbeiten zu können, müssen diese zuvor digitalisiert werden. Bei der Digitalisierung durch einen Analog/Digital-Wandler wird nun ein Analogsignal durch Abtastung zeitdiskret und durch Quantisierung wertediskret. Dabei kann man zum Beispiel bei der Zeitdiskretisierung zwischen gleichmäßiger und ungleichmäßiger Abtastung unterscheiden [Czi91].

Ist f_a ein analoges physikalisches Signal, so ist durch f_d ein dazugehöriges gleichmäßig abgetastetes zeitdiskretes und quantisiertes Signal gegeben:

$$\begin{aligned} f_d & : \mathbb{T}_a \rightarrow \mathbb{X}_q \\ f_d(n) & \approx f_a(n \cdot t_p) . \end{aligned} \quad (2.2)$$

Dabei bezeichnet \mathbb{T}_a den diskreten Zeitbereich, \mathbb{X}_q den diskreten Wertebereich, t_p die Abtastperiode und $f_p = 1/t_p$ die Abtastrate.

Ein ZENTRALES PROBLEM bei der Erfassung physikalischer Signale stellt sich dabei wie folgt dar: Bei Einsatz mehrerer Analog/Digital-Wandler werden unterschiedliche Uhren für die Abtastung eingesetzt. Damit kann aufgrund der INHÄRENTEN UNGENAUIGKEITEN, wie beispielsweise einer Phasenverschiebung der Taktsignale beziehungsweise kleinen Abweichungen in der Abtastperiode, keine absolut zeitsynchrone Erfassung zweier physikalischer Signale durch zwei verschiedene Analog/Digital-Wandler durchgeführt werden. Dies hat zur Folge, daß zum Beispiel im Falle einer Additionskomponente Signalwerten verrechnet werden, die aufgrund der unterschiedlichen Zeitpunkte, an denen diese digitalisiert wurden, nicht zusammenpassen.

2.1.3 Verschiedene Modelle physikalischer Signale

Nach dem Grundsatz, daß ein Modell so genau wie nötig und so abstrakt wie möglich sein soll, gibt es auch unterschiedliche Ansätze, physikalische Signale im Rechner darzustellen. Dabei spielt vor allem die UNTERSCHIEDLICH GENAUE MODELLIERUNG DER ZEIT eine zentrale Rolle [Bro97, BBD⁺00, BS01]. Auf der einen Seite wird das Argument aufgeführt, daß die Modellierung physikalischer Zeit zu Überspezifizierung führt, was letztendlich in ineffizienteren Entwürfen mündet [LSV96]. Demgegenüber kann man anführen, daß eine unzulängliche Berücksichtigung zeitlicher Aspekte eine Ignorierung von Problemen und deren Verschleppung zur Folge hat.

Im TAGGED-SIGNAL-MODELL [LSV96, LSV98, BBD⁺00] wird ein Event als ein Wert-Tag-Paar $(v, t) \in V \times \bar{T}$ definiert. Tags werden unter anderem dazu verwendet, Zeit zu modellieren. Ein DETERMINISTISCHES SIGNAL wird als Folge von Events und somit als eine – möglicherweise partielle – Funktion $f : \bar{T} \rightarrow V$ modelliert.

Abbildung 2.1 zeigt verschiedene Möglichkeiten, physikalische Signale zu modellieren [LSV96]. Zum einen kann man eine totale Ordnung auf den Tags definieren, in welchem Fall diese die physikalische Zeit modellieren (vergleiche Abbildung 2.1 (a.i)). Es können noch weitere Rahmenbedingungen spezifiziert werden, wie beispielsweise daß alle Abtastperioden ein ganzzahliges Vielfaches einer Grundperiode sein sollen (vergleiche Abbildung 2.1 (a.ii)). Ein spezielles Signalmodell wird für synchrone Systeme [Hal93, Hou02] verwendet, in denen eine globale Uhr existiert. In diesem Fall wird auch die Abwesenheit eines Ereignisses durch das spezielle Event \perp markiert (vergleiche Abbildung 2.1 (a.iii)). In [DL04] wird ein Modell für stückweise konstante Signale vorgestellt. In diesem Modell wird zu jedem Signalwert die Zeitdauer relativ zum vorherigen Signalwert angegeben, für die dieser Wert Gültigkeit besitzt (siehe Abbildung 2.1 (a.iv)). Wird nur eine partielle Ordnung auf den Tags definiert, dann kann man nicht mehr von einer eindeutigen Darstellung physikalischer Zeit sprechen (vergleiche Abbildung 2.1 (b.i)). Eine Vernachlässigung jeglicher Ordnungsbeziehung zwischen zwei Eingabeströmen ist in Abbildung 2.1 (b ii) dargestellt.

2.1.4 Physikalische Signale und Datenflußgraphen

Digitalsignale sind in Datenflußgraphen in Form von UNENDLICHEN STRÖMEN von Token repräsentiert. Token dienen dabei als Datencontainer für Signalwerte. Der Begriff Token wird im folgenden anstelle des in Abschnitt 2.1.3 verwendeten Begriffs Event verwendet. In der Regel wird implizit eine partielle Ordnung zwischen den Token durch deren Reihenfolge in den Fifo-Kanälen angenommen (vergleiche Abbildung 2.1 (b.ii)). Dabei können aber keine Ordnungsbeziehungen zwischen unterschiedlichen Strömen angegeben werden.

2.2 Komponentenmodell

Auf die Definition einer Softwarekomponente folgt ein Überblick über diverse Datenflußparadigmen. Die im Rahmen dieser Arbeit relevanten Datenflußparadigmen werden detaillierter beschrieben. In diesem Zusammenhang wird auch der Begriff einer Komponente und ihre Rechenbereitschaft erläutert. Es wird der Bezug zwischen den vorgestellten Datenflußparadigmen und Petrinetzen beziehungsweise funktionaler Programmierung hergestellt. Eine Skizzierung der diesen Datenflußparadigmen zugrundeliegenden Semantik und der gängigen Analysemethodik runden diesen Teilabschnitt ab.

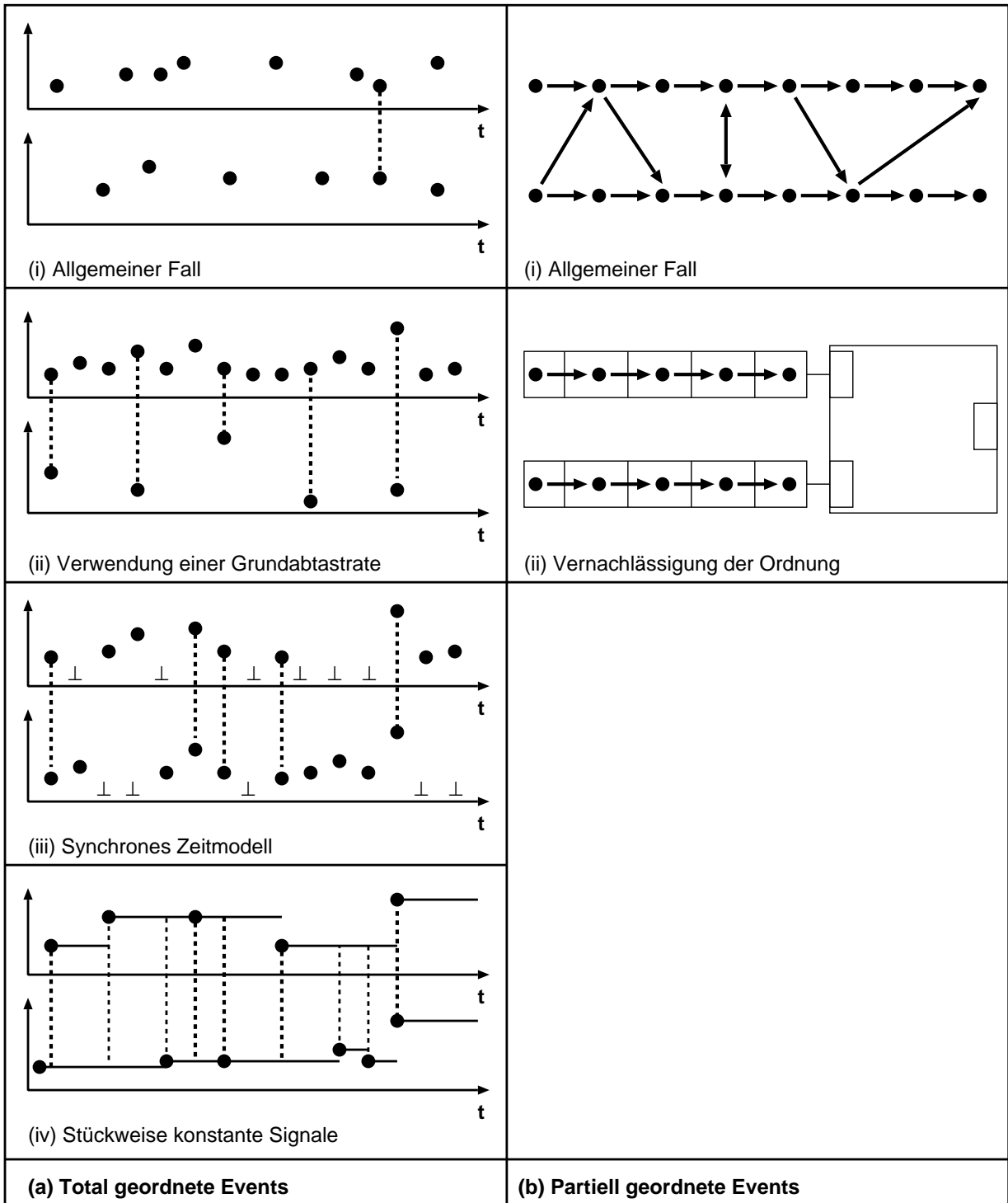


Abbildung 2.1: Signalmodelle

2.2.1 Komponentenbegriff

Der in dieser Arbeit verwendete Komponentenbegriff basiert auf der Definition von Szyperski [Szy02]:

DEFINITION 2.2.1: A SOFTWARE COMPONENT *is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.*

In einem Nachsatz fügt Szyperski hinzu: „software components are 'binary' units that are composed without modification“. Im Vergleich zu anderen Definitionen [Sam97, BW98, BRS⁺00, DGOS02, Szy02] ist diese sehr eindeutig. Die herausragenden Eigenschaften eines komponentenbasierten Ansatzes sind (vergleiche Abschnitt 1.1) [Szy02]:

- verbesserte Qualität der Anwendungen aufgrund WIEDERVERWENDUNG von Komponenten
- Unterstützung von Rapid Prototyping und Rapid Development (kürzere TIME-TO-MARKET-ZYKLEN)
- klare SYSTEMSTRUKTUR
- Vereinfachung von VERIFIKATION
- leichte Anpassung an sich ändernde Anforderungen und eine verbesserte KONFIGURIERBARKEIT. Dies ist besonders wichtig für eingebettete Systeme mit sich ändernden Umgebungen.
- erhöhte WARTBARKEIT (zum Beispiel durch Softwareupdates auf Komponentenebene)
- klare TRENNUNG DER AUFGABEN eines Anwendungsprogrammierers, nämlich der Komposition von Komponenten, von denen eines Komponentenprogrammierers.
- Entwicklung von Komponentenbibliotheken, die GEISTIGES EIGENTUM in Form neuartiger Algorithmen durch ihre binäre Form schützen.

Die hier vorgestellte Definition einer Komponente läßt sich vortrefflich auf Datenflußparadigmen anwenden. In diesem Zusammenhang spricht man auch von Datenflußkomponenten.

2.2.2 Überblick über Datenflußparadigmen

[Ste97, NLG99] geben einen Überblick über die historische Entwicklung von Datenflußparadigmen (vergleiche Abschnitt 1.1) seit den sechziger Jahren des vorigen Jahrhunderts. Interessante Stationen waren dabei die Berechnungsgraphen von Karp und Miller [KM66] beziehungsweise die Data Flow Procedure Language von Dennis [Den74]. Einen sehr weitreichenden Einfluß hatte die Arbeit von Gilles Kahn [Kah74], der eine einfache denotationelle Semantik entwarf, welche die Grundlage vieler Datenflußparadigmen bildet. Einen Überblick über die unterschiedlichen

Themenbereiche, die mit Datenfluß in Beziehung stehen, bietet [GBG95]. Die hier vorliegende Dissertation baut auf einer Hierarchie von Datenflußparadigmen, die an der Universität von Kalifornien in Berkeley konzipiert wurde, auf [WELP96, Tei97]:

- Synchroner Datenfluß
- Boolescher Datenfluß
- Dynamischer Datenfluß

Der Grund für die Verwendung dieser Variante ist darin zu suchen, daß bei ihrer Konzeption der Schwerpunkt auf die auch dieser Arbeit zugrundeliegende Anwendung in der Bild- und Signalverarbeitung gelegt wurde [BML96]. Außerdem existiert eine Vielzahl von Veröffentlichungen zu unterschiedlichsten Aspekten dieser Datenflußparadigmen, angefangen von diversen Schedulingstrategien bis hin zu Semantikdefinitionen. Weitere im Laufe der Zeit vorgestellte Datenflußparadigmen [WELP96, PCH99, BB00, ML02] wie beispielsweise

- Zyklostatistischer Datenfluß
- Zyklodynamischer Datenfluß
- Mehrdimensionaler synchroner Datenfluß
- Parametrisierter synchroner Datenfluß
- Erweiterter synchroner Datenfluß

spielen keine Rolle und werden im folgenden auch nicht betrachtet.

2.2.3 Synchroner Datenfluß

Synchroner Datenfluß (SDF) repräsentiert das grundlegende Datenflußparadigma. Die enthaltenen atomaren Datenflußkomponenten zeichnen sich durch eine einfache Regel zur Bestimmung ihrer Rechenbereitschaft aus. Betrachtet man Abbildung 2.2 (a), so ist jeder Komponentenschnittstelle eine natürlichzahlige Markierung zugeordnet. Man bezeichnet diese Markierung auch als Gewicht. Jede Datenflußkomponente, die nun mindestens so viele Token an ihren Eingängen anliegen hat, wie diese Markierungen fordern, ist rechenbereit. Sobald diese rechenbereite Datenflußkomponente durch einen Scheduler aktiviert (also rechnend gesetzt) wird, konsumiert sie so viele Token von ihren Eingängen, wie diese Markierungen verlangen und produziert eine entsprechende Anzahl von Token an ihren Ausgängen. Die Kanten zwischen den Datenflußkomponenten repräsentieren Fifokanäle. Abbildung 2.2 (b) zeigt eine Datenflußkomponente mit eigenem Speicher. Dieser Speicher wird durch eine Rückkopplungskante der Datenflußkomponente auf sich selber, welche mit einem Initialisierungstoken belegt ist, modelliert.

Die Berechnungsmächtigkeit von SDF ist geringer als die einer Turingmaschine [BML96, Buc93]. Daher sind interessante Fragen betreffs zyklischer Schedules, Deadlocks oder bezüglich des Speicherbedarfs entscheidbar (vergleiche Abschnitt 1.1).

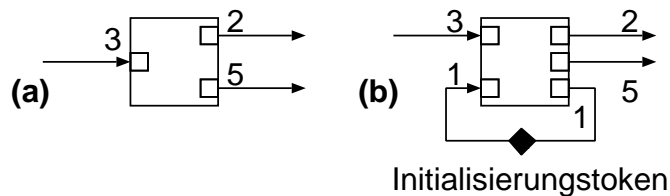


Abbildung 2.2: SDF-Komponenten (a) ohne und (b) mit eigenem Speicher

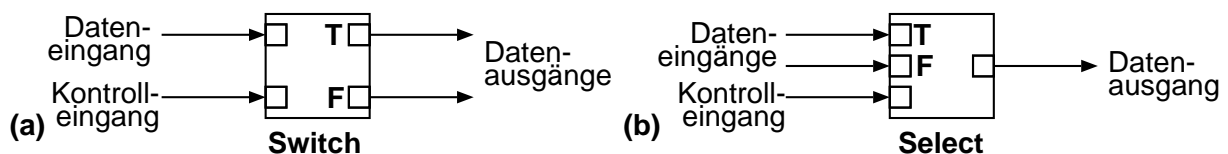


Abbildung 2.3: BDF-Komponenten Switch und Select

2.2.4 Boolescher Datenfluß

Fügt man zu dem SDF-Paradigma die Datenflußkomponenten **Switch** und **Select** (siehe Abbildung 2.3) hinzu, so erhält man das Paradigma Boolescher Datenfluß (BDF). **Switch** und **Select** unterscheiden sich von SDF-Komponenten durch ihre Steuereingänge. Im Falle von **Select** wird je nach dem am Steuereingang anliegenden Booleschen Wert das Token am mit **true** (T) beziehungsweise **false** (F) markierten Eingang an den Ausgang weitergeleitet. **Switch** ist analog definiert.

Das Paradigma BDF hat die gleiche Berechnungsmächtigkeit wie eine Turingmaschine [Buc93], da Turingmaschinen mittels BDF-Graphen und umgekehrt BDF-Graphen mittels Turingmaschinen simuliert werden können. Dies hat zur Folge, daß die in Abschnitt 2.2.3 genannten Fragen nicht mehr entscheidbar sind. Allerdings ist BDF immer noch deterministisch.

DEFINITION 2.2.2: Ein Datenflußgraph wird als DETERMINISTISCH bezeichnet, wenn – für alle zulässigen Ausführungsreihenfolgen seiner Datenflußkomponenten – identische Eingaben identische Ausgaben zur Folge haben. Können in einem Datenflußparadigma nur deterministische Datenflußgraphen konstruiert werden, so bezeichnet man dieses Paradigma ebenfalls als DETERMINISTISCH.

2.2.5 Dynamischer Datenfluß

Wird BDF um die Datenflußkomponente **Merge** (vergleiche Abbildung 2.4) erweitert, so erhält man das Paradigma Dynamischer Datenfluß (DDF). Trifft ein Token an einem der Eingänge der Datenflußkomponente **Merge** ein, so wird dieses – sobald die Datenflußkomponente aktiviert wird – an den Ausgang weitergeleitet. Falls gleichzeitig Token an beiden Eingängen anliegen,

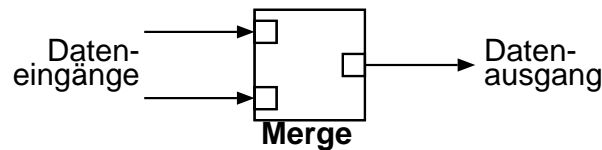


Abbildung 2.4: DDF-Komponente Merge

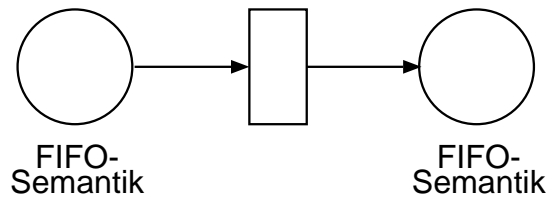


Abbildung 2.5: Petrinetzmodell einer SDF-Komponente

werden diese in zufälliger Reihenfolge weitergereicht. Die Datenflußkomponente **Merge** führt **NICHTDETERMINISMUS** in die Modellierung ein [BA81]. Das heißt, es können Datenflußgraphen konstruiert werden, die auf eine identische Eingabe unterschiedliche Ausgaben produzieren (vergleiche Definition 2.2.2).

2.2.6 Bezug zu Petrinetzen

Eine interessante Eigenschaft der vorgestellten Datenflußparadigmen ist, daß **Zeit** nicht modelliert wird, **Kausalität** jedoch schon. Somit können SDF-Graphen mittels Stellen/Transitionsnetzen [Rei91, Rei98], die bekanntermaßen nicht turingäquivalent sind [Mur89], nachgebildet werden. Abbildung 2.5 zeigt das Stellen/Transitionsnetz-Äquivalent einer einfachen SDF-Komponente. Dabei ist aber festzuhalten, daß die Stellen des Petrinetzmodells eine **Fifo-Semantik** besitzen.

Um BDF-Graphen mittels Stellen/Transitionsnetzen nachbilden zu können, müssen diese Stellen/Transitionsnetze um **Verbotskanten** erweitert werden. Damit werden Stellen/Transitionsnetze turingäquivalent [Mur89]. Abbildung 2.6 stellt ein Petrinetz dar, das eine Datenflußkomponente **Switch** nachbildet. Dabei wird **true** durch zwei Token und **false** durch Abwesenheit von Token modelliert. Liegen zwei Token am Steuereingang, so werden die Token des Dateneingangs an den **true**-Ausgang (T) geschaltet. Liegt kein Token am Steuereingang an, dann werden die Token des Dateneingangs an den **false**-Ausgang (F) weitergeleitet. Das Petrinetzmodell der Datenflußkomponente **Select** ist analog definiert (vergleiche Abbildung 2.7). Nichtdeterminismus läßt sich mittels Petrinetzen sehr einfach erzeugen, wie das Modell der Datenflußkomponente **Merge** in Abbildung 2.8 zeigt.

Der wesentliche Vorteil, den die vorgestellten Datenflußparadigmen im Vergleich zu Stellen/Transitionsnetzen ohne beziehungsweise mit Verbotskanten bieten, liegt in der Einschränkung der Verwendung von Nichtdeterminismus beziehungsweise Verbotskanten. Diese Model-

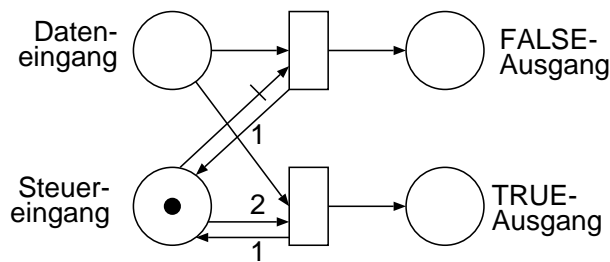


Abbildung 2.6: Petrinetzmodell einer Switch-Komponente

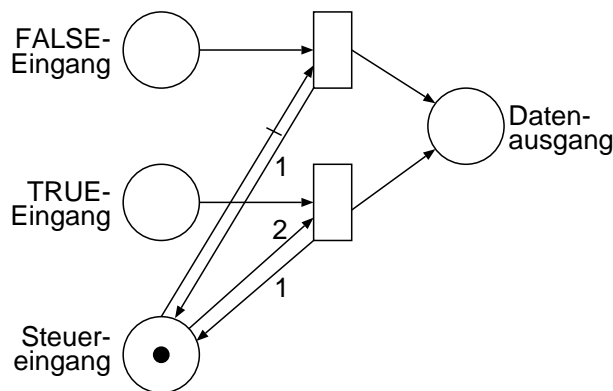


Abbildung 2.7: Petrinetzmodell einer Select-Komponente

lierungsmöglichkeiten werden in den Datenflußkomponenten **Switch**, **Select** beziehungsweise **Merge** gekapselt. Diese Kapselung entspricht in etwa dem Schritt von Assembler- zu Goto-Programmierung. Es ist also, insgesamt betrachtet, durchaus gerechtfertigt, SDF, BDF und DDF als spezielle Petrinetzdialekte zu bezeichnen.

2.2.7 Bezug zu Funktionaler Programmierung

Man kann deterministische Datenflußkomponenten auch als Funktionen auf unendlichen Strömen von Daten modellieren [Den95, KRB96, Bro03b]. In [Ree95] wird dies für grundlegende SDF-Komponenten mit der funktionalen Programmiersprache **Haskell** durchgeführt. Dabei wird argumentiert, daß jeder SDF-Graph mit den in Abbildung 2.9 vorgestellten Funktionen auf Datenströmen nachgebildet werden kann.

- Der sogenannte cons-Operator ($:-$) fügt vorne in einen Strom ein Element ein. Dies entspricht dem Einfügen eines INITIALISIERUNGSTOKENS.
- **groupS** zerteilt einen Strom in einen Strom von Vektoren der Länge k . Auf diese Weise kann die Abtastrate eines Stromes verringert werden.

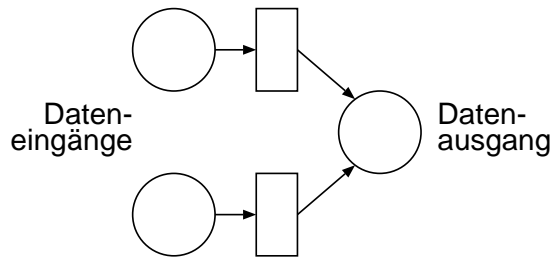


Abbildung 2.8: Petrinetzmodell einer Merge-Komponente

$(:-) :: \alpha \rightarrow \text{Stream}^n \alpha \rightarrow \text{Stream}^n \alpha$ groupS $:: \text{Int} \rightarrow \text{Stream}^{nk} \alpha \rightarrow \text{Stream}^n (\text{Vector}^k \alpha)$ concatS $:: \text{Stream}^n (\text{Vector}^k \alpha) \rightarrow \text{Stream}^{nk} \alpha$ zipS $:: \text{Stream}^n \alpha \rightarrow \text{Stream}^n \beta \rightarrow \text{Stream}^n (\alpha, \beta)$ unzipS $:: \text{Stream}^n (\alpha, \beta) \rightarrow (\text{Stream}^n \alpha, \text{Stream}^n \beta)$ mapS $:: (\alpha \rightarrow \beta) \rightarrow \text{Stream}^n \alpha \rightarrow \text{Stream}^n \beta$

Abbildung 2.9: Haskell-Modelle grundlegender SDF-Komponenten

- **concatS** stellt das Gegenstück zu **groupS** dar. Hier wird ein Strom von Vektoren in einen Strom von Elementen gewandelt.
- **zipS** kombiniert zwei Ströme punktweise zu einem einzigen Strom.
- **unzipS** erzeugt aus einem Strom von Paaren ein Paar von Strömen.
- **mapS** wendet eine Funktion auf jedes Element eines Stromes an.

In ähnlicher Weise können **switchS** und **selectS** definiert werden [Ree95]. Da das nichtdeterministische **mergeS** eine Relation und keine Funktion darstellt, kann dieses in einer funktionalen Programmiersprache wie Haskell nicht modelliert werden.

2.2.8 Denotationelle Semantik

Aufbauend auf den vorausgegangenen Abschnitt 2.2.7 liegt es nahe, eine denotationelle Semantik für Datenflußparadigmen zu definieren. Der bekannteste Ansatz einer denotationellen Semantik für Datenflußgraphen basiert auf der Arbeit von Gilles Kahn [Kah74, Par95, BJJ00]. Datensequenzen, die entlang von Kanten zwischen Datenflußkomponenten übertragen werden, werden als Complete Partial Orders (CPOs) modelliert. Jede Datenflußkomponente berechnet eine stetige Funktion. Ein Kahn-Prozeß-Netzwerk ist gegeben als eine Komposition solcher Funktionen. Das CPO-Fixpunkt-Theorem I [DP02] dient als Grundlage, um eine denotationelle Semantik für diese Komposition von Funktionen zu bestimmen: der kleinste Fixpunkt eines stetigen Funktionals, welches auf diese Komposition von Funktionen angewandt wird. Diese Semantik wird

verwendet, um Programmeigenschaften wie Terminierung, Nichtterminierung und Eigenschaften der Ausgabe zu ermitteln [Kah74, Ste97].

In [LP95, Lee97] wird diese denotationelle Semantik um das Konzept des Feuerns einer Datenflußkomponente (dort Aktor genannt) erweitert. Ein stetiger Kahn-Prozeß wird durch Sequenzen von Komponentenaktivierungen definiert. Schaltregeln werden als Vorbedingung von Komponentenaktivierungen angewendet. Diese Schaltregeln stellen den Determinismus des Prozeßnetzes sicher. Diese Vorgehensweise funktioniert für SDF- und BDF-Komponenten. Da DDF-Komponenten wie zum Beispiel das nichtdeterministische **Merge** sich nicht in Form von Funktionen darstellen lassen (vergleiche Abschnitt 2.2.7), kann man dafür auch keine denotationelle Semantik angeben.

In [Ste97] wird ein Überblick über verschiedene datenstromverarbeitende und datenflußbasierte Berechnungsmodelle geliefert. Darunter gibt es mehrere Erweiterungen von Kahn-Prozeßnetzen um Nichtdeterminismus [BA81, Kok86]. Jedoch sind diese Erweiterungen in der Regel nicht kompositionell. Broy verwendet Orakel, welche Komponenten mit eigenem Zustand darstellen, um das Ergebnis einer nichtdeterministischen Auswahl zu bestimmen [BS01].

2.2.9 Scheduling

In [Par95] werden zwei Anforderungen an einen Scheduler, der für die Ausführung eines Datenflußgraphen in einer Einprozessor- beziehungsweise Mehrprozessor- oder Mehrrechnerumgebung zuständig ist, gestellt:

1. **VOLLSTÄNDIGE AUSFÜHRUNG:** Der Scheduler soll eine vollständige Ausführung des Datenflußgraphen realisieren. Ist das Programm nicht-terminierend, soll dieses für immer ausgeführt werden.
2. **AUSFÜHRUNG IN BESCHRÄNKTEM SPEICHER:** Der Scheduler soll das Datenflußprogramm so ausführen, daß sich während des gesamten Programmablaufs nur eine beschränkte Anzahl von Token auf einem Kommunikationskanal ansammelt.

Dabei kann man Schedulingstrategien als DATA-DRIVEN, DEMAND-DRIVEN oder als eine KOMBINATION aus beiden kategorisieren [Par95]:

- **DATA-DRIVEN:** Eine Datenflußkomponente wird aktiviert, sobald genügend Daten an ihren Eingängen anliegen.
- **DEMAND-DRIVEN:** Eine Datenflußkomponente wird aktiviert, sobald ihre Ausgabe von einer anderen Datenflußkomponente benötigt wird.
- **KOMBINATION AUS DATA-DRIVEN UND DEMAND-DRIVEN:** Da sowohl eine als data-driven als auch eine als demand-driven charakterisierte Schedulingstrategie gegen die Anforderung einer Ausführung in beschränktem Speicher verstoßen kann [Par95], werden auch Kombinationen aus beiden Ansätzen untersucht [Par95, BH01].

2.2.10 Analyse

Ziel der Analyse von Datenflußgraphen ist die Herleitung eines zyklischen Schedules, anhand dessen Fragen wie Deadlockfreiheit beziehungsweise Speicherbedarf geklärt werden.

Betrachtet man zwei SDF-Komponenten **A** und **B**, die durch eine Kante miteinander verbunden sind, so kann man für diese beiden Datenflußkomponenten eine Balancegleichung aufstellen [Buc93, BML96, NLG99]:

$$\nu_A \cdot N = \nu_B \cdot M . \quad (2.3)$$

Dabei bezeichnen die Variablen ν_A und ν_B die Anzahl der Aktivierungen der beiden Datenflußkomponenten und N beziehungsweise M die Anzahl der produzierten beziehungsweise konsumierten Token. Verallgemeinert man dieses Verfahren zur Herleitung von Gleichungen auf größere Datenflußgraphen, so bekommt man pro Kante eine solche Gleichung. Das Ziel ist, eine positive ganzzahlige Lösung für die Zahl der Aktivierungen ν_i der Datenflußkomponenten i zu bestimmen. Existiert eine ganzzahlige positive Lösung für die Komponentenaktivierungen ν_i , wird beispielsweise die Ausführung des Datenflußgraphen simuliert, um festzustellen, ob Deadlocks aufgrund einer Leseblockade existieren. Durch einen geeigneten Algorithmus können solche Deadlocks aufgelöst werden, indem INITIALISIERUNGSTOKEN auf die jeweiligen Kanten plaziert werden [BML96].

Bei BDF-Graphen wird statt einer numerischen eine symbolische Berechnung durchgeführt [Buc93, NLG99]. Dies ist notwendig, da zur Entwurfszeit nicht bekannt ist, wie sich zur Laufzeit die Anzahl der Token auf den jeweiligen Kanten entwickelt. Dieses symbolische Gleichungssystem kann nun, muß aber nicht lösbar sein. Dies ergibt sich aus der Turingäquivalenz des Paradigmas BDF. Nachteilig wirkt sich hier unter anderem aus, daß kleine Änderungen im Datenflußgraphen das zugehörige Gleichungssystem unlösbar machen können [NLG99].

Aufbauend auf diese grundlegende Vorgehensweise werden in [BML96] beziehungsweise [Buc93] weitere Techniken zur Bestimmung eines zyklischen Schedules wie zum Beispiel Clustering-Verfahren vorgestellt.

2.3 Interface-Typsystem

Am Anfang dieses Abschnittes steht eine kurze Einführung in die Begriffswelt von Typsystemen, die aus der textbasierten Programmierung stammt (vergleiche aber auch [BRS⁺00]). Im Anschluß findet sich eine Beleuchtung der besonderen Herausforderungen, mit denen Typsysteme im Bereich der komponentenbasierten Programmierung konfrontiert sind. Abschließend wird das Typsystem von Ptolemy II [HLL⁺03, BLL⁺04] näher beleuchtet.

2.3.1 Typen in textbasierten Programmiersprachen

Typsysteme wurden in erster Linie für textbasierte Programmiersprachen entworfen und sind in diesem Bereich auch sehr gründlich erforscht worden. Die in diesem Abschnitt aufgeführten

Definitionen stammen im wesentlichen aus [CW85, Car97]. Im Zentrum steht dabei natürlich der Begriff Typ:

DEFINITION 2.3.1: *Ein TYP beschreibt eine Abschätzung der Werte, die ein Programmfragment während seiner Ausführung annehmen kann.*

Um die Bestimmung und Einhaltung dieser Typen kümmert sich in modernen Programmiersprachen ein Typsystem:

DEFINITION 2.3.2: *Ein TYPSYSTEM ist der Bestandteil einer typisierten Programmiersprache, der sich um die Typen von allen Ausdrücken in einem Programm wie zum Beispiel Variablen kümmert.*

Weitere grundlegende Begriffe sind Typprüfung und Typchecker.

DEFINITION 2.3.3: *Den Vorgang, ein Programm vor seiner Ausführung auf Konformität zu einem gegebenen Typsystem zu überprüfen mit dem Ziel, Typfehler zu vermeiden, bezeichnet man als TYPPRÜFUNG.*

DEFINITION 2.3.4: *Unter einem TYPCHECKER versteht man den Teil eines Compilers oder Interpreters, der die Typprüfung durchführt.*

Beachtenswerte Typsysteme beinhalten beispielsweise der typisierte Lambda-Calculus und funktionale Sprachen wie Haskell beziehungsweise ML. Für eine detaillierte Beschreibung sei auf [Mil78, Sch94, Car97, Bir98] verwiesen.

2.3.2 Typen in der komponentenbasierten Softwareentwicklung

Ein wichtiger Unterschied zwischen textbasierter und komponentenbasierter Softwareentwicklung ist, daß bei letzterer die atomaren Bausteine in der Regel ausschließlich in binärer Form als Shared Libraries vorliegen. Dadurch hat die Typprüfung keinen Zugang zu den internen Variablen und Funktionsdeklarationen. Gegenstand der Typprüfung sind die Typen der Schnittstellen der einzelnen Komponenten. Mittels einer Schnittstellenbeschreibungssprache können beispielsweise diese Typen und Typconstraints zwischen diesen Typen für jede einzelne Komponente definiert werden. Neben der Kompatibilität der verbundenen Schnittstellentypen sind Polymorphismus und automatische Typkonvertierung von zentraler Bedeutung.

2.3.2.1 Polymorphismus

Polymorphismus [Car97] spielt in der komponentenbasierten Softwareentwicklung eine besonders zentrale Rolle, da dadurch die Wiederverwendungsmöglichkeit [Kar95, BHH00] einzelner Komponenten wesentlich erhöht wird.

DEFINITION 2.3.5: *Unter POLYMORPHISMUS versteht man die Fähigkeit eines Programmfragmentes beziehungsweise einer Komponente mehrere Typen annehmen zu können.*

Cardelli und Wegner [CW85] unterscheiden zwei Hauptarten von Polymorphismus: universellen Polymorphismus und Ad-Hoc-Polymorphismus. Eine universell polymorphe Funktion führt

denselben Code für unterschiedliche Typen von Eingabedaten aus, wohingegen eine ad-hoc-polymorphe Funktion unterschiedlichen Code ausführt. Dabei kann man den UNIVERSELLEN POLYMORPHISMUS weiter unterteilen in parametrischen Polymorphismus und Inklusionspolymorphismus.

- PARAMETRISCHER POLYMORPHISMUS bedeutet, daß eine Funktion in gleicher Weise mit einer Reihe von Typen arbeiten kann.
- INKLUSIONSPOLYMORPHISMUS tritt bei objektorientierter Programmierung auf, wenn eine Unterklasse anstelle einer Oberklasse verwendet werden kann.

AD-HOC-POLYMORPHISMUS wird unterteilt in Overloading und Coercion:

- OVERLOADING bezeichnet die Verwendung desselben Operator- oder Funktionsnamens für unterschiedliche Berechnungen. Mit Hilfe des Kontexts wird entschieden, welche Code ausgeführt werden soll.
- COERCION konvertiert Argumente einer Funktion in den Typ, den die Funktion erwartet.

2.3.2.2 Typkonvertierung

Bei der Verwendung vorgegebener Komponenten kommt es häufig vor, daß Werte von einem Typ in einen anderen konvertiert werden müssen [Xio02]. Diese Konvertierung kann nun verlustfrei erfolgen, wie zum Beispiel bei einer Konvertierung von int nach double. Die Konvertierung kann aber auch verlustbehaftet sein, beispielsweise bei einer Konvertierung von double nach int. Die Konvertierung kann implizit, das heißt automatisch, stattfinden, indem während der Konstruktion eine geeignete Konvertierungskomponente eingefügt wird. Dies ist vor allem bei verlustfreien Konvertierungen sinnvoll. Bei verlustbehafteten Konvertierungen sollte der Benutzer befragt werden.

2.3.3 Das Typsystem von Ptolemy II

Dieses Typsystem ist in [XL00, Xio02, LX04] beschrieben und stellt den aktuellen Stand der Technik für komponentenbasierte datenflußorientierte eingebettete Systeme dar.

2.3.3.1 Typdomäne

In Ptolemy II wird die Wertemenge eines physikalischen Signals typisiert, indem man den einzelnen Werten Basistypen aus Abbildung 2.10 beziehungsweise Aggregationstypen wie Records oder Arrays zuordnet. Die Basistypen sind in einem Verband organisiert, wobei eine Verbindungslinie ausdrückt, daß der unten stehende Typ in den oben stehenden Typ verlustfrei konvertiert werden kann. Dies ist zum Beispiel bei int und double der Fall.

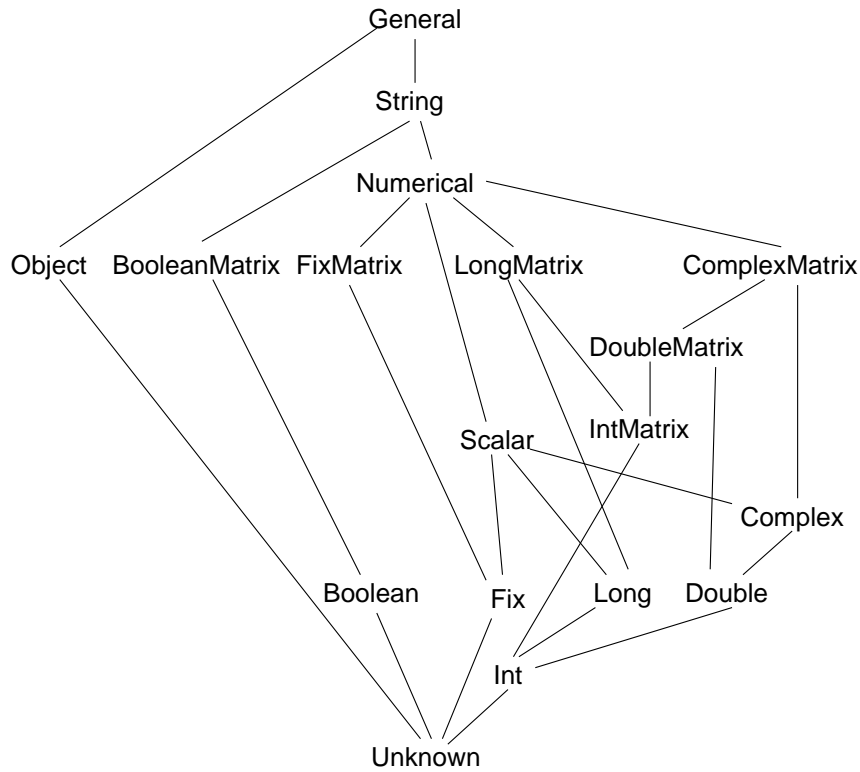


Abbildung 2.10: Verband zur Typbestimmung in Ptolemy II

2.3.3.2 Typconstraints

Von Kanten implizierte Typconstraints werden in Form von Ungleichungen

$$\text{sendType} \leq \text{receiveType} \quad (2.4)$$

angegeben. Innerhalb einer Komponente können zusätzliche Typconstraints zwischen Typen von Schnittstellen und Parametern definiert werden. Diese Typconstraints haben alle die Form von Ungleichung 2.4. Dabei werden deklarierte Typen als Konstanten und nichtdeklarierte Typen als Variablen repräsentiert. Die Domäne einer Typvariable ist durch die Elemente der Verbände der Basis- und Aggregationstypen gegeben [Xio02].

Abbildung 2.11 zeigt einen kleinen Datenflußgraphen. Die durch die Kanten bedingten Typconstraints lauten

$$\begin{aligned} \text{int} &\leq \alpha \\ \text{double} &\leq \beta \\ \gamma &\leq \text{double} \end{aligned} \quad (2.5)$$

Zusätzlich werden durch die Datenflußkomponente K2, welche einen überladenen Addierer re-

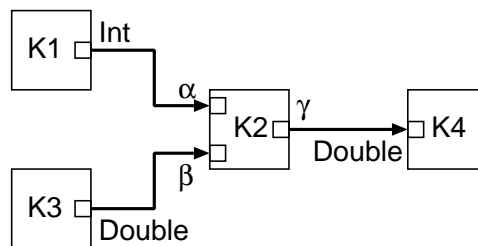


Abbildung 2.11: Beispiel zur Typbestimmung in Ptolemy II

präsentiert, die folgenden Typconstraints eingeführt:

$$\begin{aligned}
 \alpha &\leq \gamma \\
 \beta &\leq \gamma \\
 \gamma &\leq \text{complex}
 \end{aligned}
 \tag{2.6}$$

Die ersten beiden Typconstraints besagen, daß die Genauigkeit des Ergebnisses nicht geringer sein soll als die der Eingaben. Der letzte Typconstraint bewirkt, daß die Daten an den Schnittstellen verlustfrei in den Typ `complex` konvertiert werden können.

2.3.3.3 Typbestimmung

Typbestimmung entspricht in Ptolemy II der Lösung einer Menge von einfachen Ungleichungen (siehe Abschnitt 2.3.3.2), welche über einem endlichen Verband definiert sind. Dazu wird der Algorithmus aus [RM96] verwendet. Zu Anfang sind alle Typvariablen auf `unknown` gesetzt. Dann werden die einzelnen Variablen wiederholt aktualisiert bis alle Typconstraints erfüllt sind. Dieses Vorgehen entspricht der wiederholten Auswertung einer monotonen Funktion. Der Algorithmus liefert somit als Ergebnis den kleinsten Fixpunkt dieser Funktion zurück.

2.4 Model Checking

Dieser Abschnitt stellt zuerst Model Checking als eine explizite Methodik zur Verifikation eines Zustandsraumes eines Systems vor. Dem schließt sich die Beschreibung verschiedener Erweiterungen endlicher Automaten [HU88] an, die zur Modellierung von Kommunikationsprotokollen verwendet werden können. Im Zusammenhang mit der Analyse großer Erreichbarkeitsgraphen spielen effiziente Suchstrategien und Techniken zur Suchoptimierung wie Partial Order Reduction eine zentrale Rolle. Den Abschluß bildet die Vorstellung des Model Checkers Spin.

2.4.1 Explizite Verifikation des Zustandsraumes

Der ZUSTANDSRAUM (State Space) eines Systems umfaßt alle Zustände dieses Systems. Dabei können sich diese Systemzustände beispielsweise durch unterschiedliche Variablenbelegun-

gen unterscheiden. Model Checking befaßt sich mit der Verifikation solcher Zustandsräume. In [CW96] ist Model Checking wie folgt definiert (vergleiche auch [CGP99, Sch02, Laf03]):

DEFINITION 2.4.1: *MODEL CHECKING is a technique that relies on building a finite model of a system and checking that a desired property holds in that model.*

Der Zustandsraum eines beispielsweise über Fifos miteinander kommunizierenden Systems endlicher Automaten wird durch ein Model-Checking-Verfahren in Form eines Erreichbarkeitsgraphen dargestellt. Das heißt, diese Analysemethodik basiert auf einer vollständigen Durchsuchung des Erreichbarkeitsgraphen, welche aufgrund der Endlichkeit des Modells terminiert.

DEFINITION 2.4.2: *Ein ERREICHBARKEITSGRAPH ist ein gerichteter Graph, dessen Knoten die möglichen Zustände des untersuchten Systems beschreiben. Die Kanten des Erreichbarkeitsgraphen repräsentieren die Übergänge zwischen den einzelnen Zuständen.*

Werden die Zustände eines Systems explizit dargestellt, so spricht man von EXPLIZITER Verifikation des Zustandsraums. Demgegenüber steht symbolisches Model Checking, in welchem eine IMPLIZITE Darstellung des Zustandsraumes basierend auf Ordered Binary Decision Diagrams verwendet wird [McM93, Sch02].

Die Vorgehensweise beim Model Checking läßt sich laut [CGP99] in drei Schritte unterteilen:

1. **MODELLIERUNG:** Als erstes muß ein Entwurf in einen Formalismus übertragen werden, der automatisch durch ein Model-Checking-Werkzeug analysiert werden kann.
2. **SPEZIFIKATION:** Als nächstes müssen die Eigenschaften, welchen der Entwurf genügen soll, spezifiziert werden. Dies geschieht in der Regel mit Hilfe einer temporalen Logik wie zum Beispiel LTL (Linear Temporal Logic) oder CTL (Computation Tree Logic). Da in dieser Arbeit die zu untersuchenden Eigenschaften immer dieselben sind, wird hier keine temporale Logik verwendet. Aus diesem Grund entfällt auch deren Beschreibung in diesem Abschnitt. Für eine detaillierte Einführung sei zum Beispiel auf [CGP99, BJJ00, Hol04] verwiesen.
3. **VERIFIKATION:** In diesem Schritt werden die spezifizierten Eigenschaften mittels einer erschöpfenden Betrachtung der Knoten des Erreichbarkeitsgraphen verifiziert.

2.4.2 Modellierung von Kommunikationsprotokollen

Der erste wesentliche Schritt beim Model Checking ist die Spezifikation eines dem zu verifizierenden Sachverhaltes entsprechenden Modells [BD02]. Als Grundlage der Modellierung werden Erweiterungen von endlichen Automaten verwendet. Diese Erweiterungen können sich sowohl auf die Kommunikation zwischen einzelnen Automaten als auch auf die Darstellung interner Zustände beziehen: Bei der Kommunikation unterscheidet man synchrone [Ber98, AH01] und asynchrone Kommunikation. Die asynchrone Kommunikation erfolgt in der Regel mittels Fifos, wobei man unterscheidet:

- verlustfreie [BZ83, SU96, FM97, FIS00, FS01] und verlustbehaftete Datenübertragung [EN98, AABJ04]

- begrenzte [Hol04] oder unbegrenzte Fifokapazität [AABJ04]
- Sortierung der Fifoinhalte nach Einfügereihenfolge beziehungsweise numerische Sortierung [Hol04]
- Auslesen der Fifoinhalte nach Einfügereihenfolge beziehungsweise nach Pattern-Matching („zufällige“ Auslesereihenfolge) [Hol04]

Synchrone Kommunikation läßt sich mittels asynchroner Kommunikation nachbilden, indem man die Fifokapazitäten auf Null setzt. Die internen Zustände eines erweiterten Automaten können in unterschiedlichster Form gespeichert werden [Hol91, Hol04]:

- eigentliche Zustände des Automaten
- Fifos für die interne Kommunikation
- Zählvariablen

Diese Reihe kann beliebig erweitert werden. Die drei Hauptkriterien für die Bewertung der Eignung einer Modellierungsmethodik sind [Hol91]:

- Modellierungsmächtigkeit
- Analysierbarkeit
- Klarheit der Beschreibung

Im folgenden werden Interface-Automaten, Communicating Finite State Machines und Extended Finite State Machines näher betrachtet. Neben diesen hier vorgestellten Modellierungsmethodiken gibt es eine ganze Reihe anderer Erweiterungen von endlichen Automaten wie zum Beispiel Regular State Machines [TTS00] beziehungsweise Algebraic State Machines [BW00], auf die hier nicht weiter eingegangen werden kann.

2.4.2.1 Interface-Automaten

Abbildung 2.12 zeigt ein Beispiel zweier kommunizierender Interface-Automaten [AH01, LX01, Xio02]. Das Hauptmerkmal von Interface-Automaten ist die Verwendung synchroner Kommunikation. Interface-Automaten besitzen Zustände und Transitionen, wobei letztere in Eingabe- (markiert mit „?“), Ausgabe- (gekennzeichnet durch „!“) und interne Transitionen (symbolisiert mittels „;“) unterteilt werden. Eingabetransitionen entsprechen dabei Methodenaufrufen der modellierten Komponente, Ausgabetransitionen repräsentieren Methodenaufrufe anderer Komponenten durch die modellierte Komponente und interne Transitionen stellen Berechnungen der modellierten Komponente dar.

Das in Abbildung 2.12 dargestellte Beispiel ist aus [AH01] entnommen und modelliert einen Nachrichtenübertragungsdienst. Die Komponente **Comp** besitzt eine Methode **msg**, die aufgerufen wird, sobald eine Nachricht übertragen werden soll. **Comp** selber ruft die Methode **send** auf. Bei erfolgreichem Senden wird eine Bestätigung **ack** zurückgeschickt. Im Falle eines Mißerfolgs

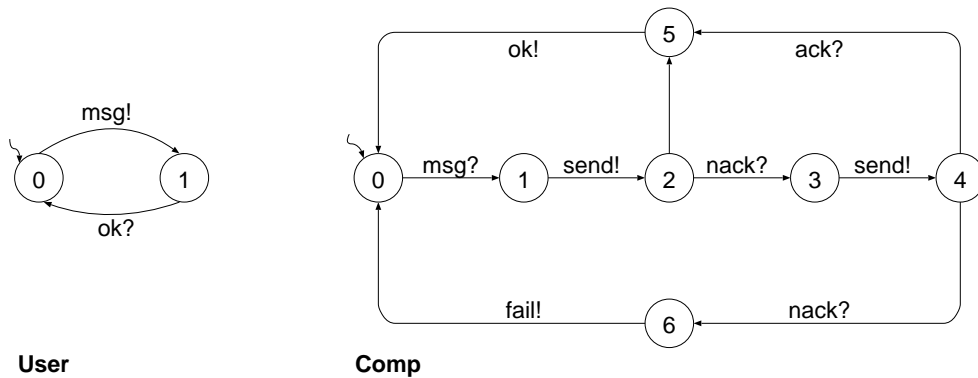


Abbildung 2.12: Beispiel zweier kommunizierender Interface-Automaten

wird `nack` zurückgesendet. Erhält `Comp` eine positive Rückmeldung `ack`, sendet es seinerseits ein `ok` an den Aufrufer. Im anderen Fall wird `fail` zurückgesendet.

Der große Vorteil von Interface-Automaten ist ihre Analysierbarkeit. Da keine Fifos beziehungsweise internen Variablen verwendet werden, ist die Anzahl der Zustände, die ein Interface-Automat annehmen kann, vergleichsweise klein. Dieser Vorteil wird mit der eingeschränkten Modellierungsmächtigkeit bezahlt.

2.4.2.2 Communicating Finite State Machines

Communicating Finite State Machines (CFSM) [Hol91] können als Erweiterung von Interface-Automaten um begrenzte fehlerfreie Fifos angesehen werden. Damit ist neben synchroner Kommunikation auch asynchrone Kommunikation möglich.

Dadurch bleibt die Anzahl von Zuständen einer CFSM zwar endlich. Verknüpft man aber einige solcher Modelle zu einem Gesamtsystem, dann kann die so entstehende Anzahl von Zuständen ohne weiteres die Speicherkapazität eines modernen Rechners überschreiten.

Betrachten wir zur Erläuterung folgendes kleine Beispiel (vergleiche [Hol04]): Sei c die Anzahl der Fifokanäle, n_c die Kapazität der Fifos, m_c die Mächtigkeit des Nachrichtenalphabets. Jede Fifo kann nun $m \in [0; c]$ Nachrichten speichern und jede Nachricht wird aus m_c Möglichkeiten ausgewählt. Die Anzahl der möglichen unterschiedlichen Zustände dieser Menge von Fifos ergibt sich als:

$$f(n_c, m_c, c) = \left(\sum_{i=0}^{n_c} m_c^i \right)^c \quad (2.7)$$

$$f(5, 5, 5) = 909203700718879776 \quad (2.8)$$

In Gleichung 2.8 ist das Ergebnis für fünf Fifos mit einer Kapazität von fünf und fünf möglichen Nachrichtentypen ausgerechnet. Dies soll einen ersten Eindruck der inhärenten Problematik der ZUSTANDSRAUMEXPLOSION vermitteln.

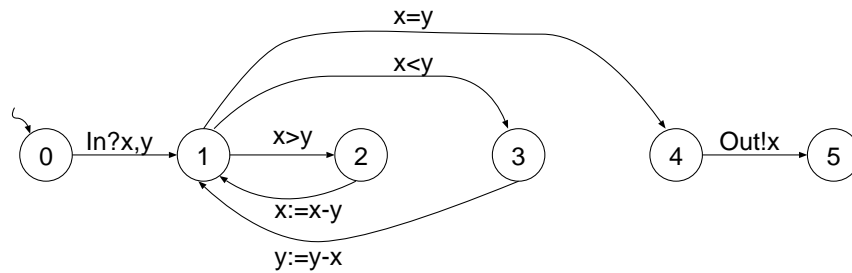


Abbildung 2.13: Beispiel einer Extended Finite State Machine

2.4.2.3 Extended Finite State Machines

Extended Finite State Machines [Hol91] erweitern CFSMs durch die Hinzunahmen von Integer-Variablen. Als zweites werden in den Fifos nur noch Integer-Daten anstelle von abstrakten Objekten übertragen. Als dritte Änderung werden eine Reihe arithmetischer und logischer Operatoren eingeführt, welche den Inhalt von Variablen manipulieren können. Abbildung 2.13 zeigt als Beispiel eine Extended Finite State Machine, die aus zwei Eingabewerten den größten gemeinsamen Teiler berechnet.

Die Hinzunahme von Variablen mit begrenztem Wertebereich erhöht die Berechnungsmächtigkeit im Vergleich zu CFSMs mit begrenzten Fifokanälen nicht. Denn eine solche Variable kann durch einen endlichen Automaten simuliert werden [Hol91]. Nachteilig dabei ist, daß diese implizit ausgedrückten endlichen Automaten in der Regel eine relativ große Anzahl von Zuständen besitzen. Dies hat zur Folge, daß die für die Analyse benötigte Zeit erheblich ansteigen kann.

2.4.3 Suchstrategien für den Erreichbarkeitsgraphen

Die einem Model-Checking-Verfahren zugrundeliegende Strategie, den Zustandsraum zu durchsuchen, ist von entscheidender Bedeutung für dessen Erfolg. Der Zustandsraum liegt dabei in Form eines Erreichbarkeitsgraphen vor (vergleiche Definition 2.4.2). Diese Suchstrategien folgen dem prinzipiellen Aufbau von Algorithmus 2.4.1 [Laf03].

Generell kann man die Menge der Knoten des Erreichbarkeitsgraphen in bereits besuchte und noch nicht besuchte Knoten unterteilen. Bei den bereits besuchten Knoten kann man zwischen der Menge V_E der expandierten Knoten, die als fertig bearbeitet markiert und deren Söhne – soweit vom Algorithmus angestrebt – bereits alle erzeugt wurden, und der Menge V_P der noch nicht (vollständig) expandierten Knoten unterscheiden. Initialisiert wird die Menge V_P zum Zeitpunkt 0 mit dem Wurzelknoten des Erreichbarkeitsgraphen. Während eines Bearbeitungsschrittes wird jeweils der höchstpriorisierte Knoten mit `select()` ausgewählt. Dabei bestimmt diese Priorisierung die Suchstrategie (vergleiche Abschnitt 2.4.3.1). Es werden mittels `expand()` die Nachfolger des Knotens ermittelt. Ist das Suchziel erreicht, was durch `goal()` überprüft wird, bricht das Verfahren ab.

ALGORITHMUS 2.4.1:

```
1  Algorithm generalSearch() {
2       $V_E^0 := \emptyset$ 
3       $V_P^0 := \{v_{\text{root}}\}$ 
4       $i := 0$ 
5      while ( $V_P^i \neq \emptyset$ ) {
6           $v_{\text{best}} := \text{select}(V_P^i)$ ;
7           $V_P^i := V_P^i \setminus \{v_{\text{best}}\}$ 
8           $V_E^{i+1} := V_E^i \cup \{v_{\text{best}}\}$ 
9           $V_{\text{sons}} := \text{expand}(v_{\text{best}})$ 
10          $V_P^{i+1} := V_P^i \cup (V_{\text{sons}} \setminus V_E^{i+1})$ 
11         if ( $\text{goal}(V_{\text{sons}})$ ) {
12             return solution
13         }
14          $i := i + 1$ 
15     }
16 }
```

Bei der Beurteilung von Suchstrategien spielen folgende Kriterien eine Rolle:

- **TERMINIERUNG:** Wenn der Suchalgorithmus eine Lösung liefert, terminiert er.
- **VOLLSTÄNDIGKEIT:** Ein Suchalgorithmus terminiert genau dann, wenn er eine Lösung liefert.
- **ZULÄSSIGKEIT:** Ein Suchalgorithmus ist zulässig, wenn er immer eine optimale Lösung liefert. Die optimale Lösung in einem ungewichteten Graphen ist der kürzeste Pfad zu einem Zielknoten. In einem gewichteten Graphen ist dies der Pfad zu dem Zielknoten mit minimalen Kosten.

Generell unterscheidet man zwischen Suche ohne Verwendung von Nebenwissen und informierter Suche (Guided Search) [Laf03]. Suchstrategien, die kein Nebenwissen verwenden, sind dadurch gekennzeichnet, daß sie allein auf der Struktur des zu durchsuchenden Graphen und auf dem Fortschritt der Suche basieren. Vom dem zugrundeliegenden Problem wird vollkommen abstrahiert.

2.4.3.1 Suche ohne Nebenwissen

Die hier behandelten Strategien zur Suche ohne Verwendung von Nebenwissen sind Tiefensuche, Breitensuche und Dijkstras Single-Source-Shortest-Path-Algorithmus (vergleiche [Sch02, Laf03, Hol04]).

Tiefensuche: Bei der Tiefensuche wird der aktuell am tiefsten liegende Knoten in dem bis zu diesem Zeitpunkt untersuchten Teil des Erreichbarkeitsgraphen zur Expansion ausgewählt. Dies erreicht man, indem man die Prioritäten so wählt, daß die Auswahl einem Stack entspricht. Dieses Verfahren terminiert für endliche Graphen. Bei unendlichen Graphen muß eine Beschränkung der Suchtiefe eingeführt werden. Das Verfahren ist vollständig, da jeder vom Startknoten aus erreichbare Knoten untersucht wird. Allerdings ist der Algorithmus nicht zulässig, da er keine optimale Lösung garantiert.

Breitensuche: Hier werden die am wenigsten tief liegenden Knoten in dem bis zu diesem Zeitpunkt untersuchten Teil des Erreichbarkeitsgraphen bevorzugt. Dies erreicht man, indem die Prioritätsvergabe eine Auswahl nach der Fifo-Strategie bedingt. Das Verfahren terminiert, wenn das Suchziel gefunden werden kann. Ist dies nicht möglich und der Graph unendlich, so terminiert das Verfahren nicht. Das Verfahren ist vollständig, da jeder vom Startknoten aus erreichbare Knoten besucht wird. Das Verfahren ist zulässig für nicht gewichtete Graphen, da jeder Knoten in kürzestmöglichem Abstand gefunden wird.

Single-Source-Shortest-Path: Dabei wird der Pfad mit den minimalen Kosten ermittelt, indem der Algorithmus in jedem Schritt den Knoten zu expandieren sucht, der durch den Pfad mit den augenblicklich günstigsten Kosten erreicht wurde. Damit wird die Auswahlfunktion `select()` so definiert, daß sie eine Prioritätswarteschlange realisiert. Die Priorität jedes Knotens ergibt sich aus den Kosten des augenblicklich günstigsten Pfades zu diesem Knoten. Kann das Suchziel gefunden werden, so terminiert das Verfahren. Ist der Graph unendlich und kann das Suchziel nicht gefunden werden, so terminiert der Algorithmus nicht. Das Verfahren ist vollständig, da jeder vom Startknoten aus erreichbare Knoten besucht wird. Der Algorithmus ist zulässig für sowohl nicht gewichtete als auch nicht negativ gewichtete Graphen.

Bewertung: Da keinerlei Informationen über das Ziel der Suche verwendet werden, dauert diese bei großen Zustandsräumen in der Regel sehr lange. Es gibt Ansätze, die Suchzeit einzuschränken, indem beispielsweise bei der Tiefensuche eine maximale Suchtiefe festgelegt wird. Dies hat aber den Nachteil, daß der Zustandsraum nicht mehr vollständig untersucht wird.

2.4.3.2 Informierte Suche

Diesen Suchstrategien ohne Nebenwissen werden sogenannte informierte Suchstrategien gegenübergestellt, die Nebenwissen über das zugrundeliegende Problem ausnutzen und damit in den meisten Fällen schneller ein Ergebnis liefern können. Diese Strategien verwenden in der Regel Funktionen, die den zu untersuchenden Knoten des Erreichbarkeitsgraphen eine geeignete Priorität zuweisen.

Der Suchalgorithmus A*: Der Suchalgorithmus A* ist mit Tiefensuche und Dijkstras Single-Source-Shortest-Path-Algorithmus verwandt. Im Gegensatz zu diesen werden die Knoten des Erreichbarkeitsgraphen entsprechend einem zugeordneten Prioritätswert expandiert, der sich aus

zwei Bestandteilen zusammensetzt. Zum einen werden die Kosten des augenblicklich optimalen Pfades zu dem jeweiligen Knoten berücksichtigt. Zum anderen wird eine Abschätzungsfunktion ausgewertet, die problemabhängig spezifiziert werden muß. In [Laf03] wird zum Beispiel für die Suche in einem zweidimensionalen Gitter als Abschätzung der Euklidische Abstand verwendet. Der A*-Algorithmus terminiert für endliche Graphen, wobei – falls notwendig – der gesamte Suchraum abgedeckt wird. Die Terminierung und Vollständigkeit ist auch für unendliche Graphen gewährleistet, falls die Kosten jedes unendlichen Pfades unbegrenzt sind.

Der Suchalgorithmus IDA*: Der Algorithmus Iterative Deepening A* (IDA*) führt mehrere A*-Suchläufe beginnend bei dem gleichen Startknoten mit jeweils erhöhten Kostenbeschränkungen durch. Knoten, die diese Beschränkung überschreiten, werden nicht weiter berücksichtigt. Initialisiert wird die Kostenbeschränkung mit einer heuristischen Abschätzung des Startknotens. Von allen Kostenwerten, die in einem Suchlauf bestimmt werden, wird der kleinste als Kostenbeschränkung für den nächsten Suchlauf verwendet. Der Algorithmus terminiert, falls der Graph endlich ist oder ein Knoten mit den gesuchten Eigenschaften existiert. Der Algorithmus ist vollständig und, abhängig von der verwendeten Abschätzung, gegebenenfalls auch zulässig.

Weitere Verfahren: Es existieren eine Reihe weiterer Verfahren zur informierten Suche, die Abwandlungen der vorgestellten Methoden darstellen. Beispiele dafür sind Hill Climbing, WA*, Best-First, Beam Search und Frontier Search [Laf03].

2.4.4 Partial Order Reduction

In der Regel ist die Kommunikation in zu verifizierender Software asynchron, was bedeutet, daß viele Komponentenaktivierungen unabhängig voneinander ausgeführt werden können. Dies führt mit zu dem Problem der ZUSTANDSRAUMEXPLOSION. So kann beispielsweise ein vergleichsweise kleines System über Fifos kommunizierender endlicher Automaten einen Erreichbarkeitsgraphen mit unerwartet vielen Knoten bedingen. Eine Möglichkeit, die Zustandsraumexplosion einzudämmen, besteht in der PARTIAL ORDER REDUCTION. Diese Techniken basieren auf der Unabhängigkeit nebenläufig ausführbarer Ereignisse.

DEFINITION 2.4.3: Zwei EREIGNISSE sind voneinander UNABHÄNGIG, wenn man sie in beliebiger Reihenfolge ausführen kann und trotzdem dasselbe Endergebnis (denselben Zustand) erhält.

Der Name Partial Order Reduction leitet sich vom PARTIAL ORDER MODEL OF PROGRAM EXECUTION her [CGP99]. In diesem Modell werden parallel ausführbare Events nicht geordnet. Jede partiell geordnete Ausführung kann somit einer Vielzahl von Sequentialisierungen dieser Events entsprechen. Werden nun bei der Untersuchung asynchron kommunizierender Software alle möglichen Reihenfolgen von Ereignissen berücksichtigt, resultiert dies in einem sehr großen Zustandsraum. Wenn es für eine Untersuchung irrelevant ist, in welcher genauen Abfolge die auftretenden Events abgearbeitet werden, reicht es aus, eine mögliche Abfolge zu untersuchen.

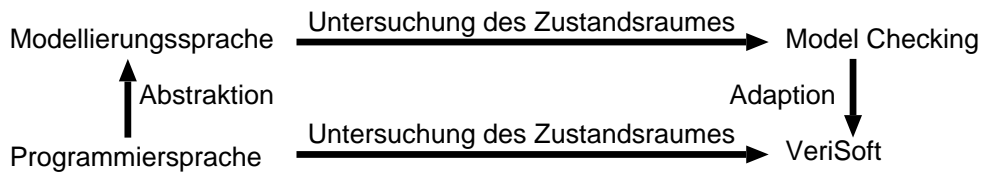


Abbildung 2.14: Model-Checking-Strategie von VeriSoft

DEFINITION 2.4.4: *Unter PARTIAL ORDER REDUCTION versteht man die Einschränkung eines Model-Checking-Verfahrens auf einen Repräsentanten einer Menge von Sequentialisierungen derselben unabhängigen Ereignisse.*

Mittlerweile gibt es eine Reihe von Ansätzen für eine solche Partial Order Reduction wie beispielsweise Stubborn Sets, Persistent Sets und Ample Sets [Sch02, Laf03].

2.4.5 Der Model Checker Spin

Der Model Checker Spin [Hol91, Hol04] verwendet als Modellierungssprache im wesentlichen Extended Finite State Machines (siehe Abschnitt 2.4.2.3). Dabei werden aber zum Beispiel auch andere Schreib- und Lesestrategien für Fifos ermöglicht (vergleiche Abschnitt 2.4.2). Als Suchstrategien werden Tiefensuche, Tiefensuche mit Beschränkung der Suchtiefe, verschachtelte Tiefensuche beziehungsweise Breitensuche eingesetzt.

Als Suchoptimierungen sind Methoden zur Partial Order Reduction im Sinne von Abschnitt 2.4.4 beziehungsweise STATEMENT MERGING als Spezialfall implementiert. Unter Statement Merging versteht man den Versuch der Zusammenlegung von Transitionssequenzen eines Automaten in eine Transition [Hol04]. All diese Verfahren haben zum Ziel, die Anzahl der Zustände, welche gespeichert werden müssen, zu minimieren.

Orthogonal dazu sind Speicherkomprimierungsverfahren zu sehen, die entweder verlustfrei oder verlustbehaftet arbeiten. So kann man zum einen den Speicherbedarf für einzelne Zustände des Erreichbarkeitsgraphen minimieren. Dazu werden Teile der Zustände separat abgespeichert und im eigentlichen Zustand nur eine Indexnummer abgelegt. Eine weitere Maßnahme zur Verminderung des Speicherbedarfs besteht in der Verwendung eines minimalen deterministischen endlichen Automaten, der Zustände anhand ihrer Indexmenge erkennt und anstelle einer konventionellen Lookup-Tabelle verwendet wird. Die Laufzeiteinbusen können allerdings erheblich sein [Hol04]. Gibt man die Bedingung auf, daß der Zustandsraum unter Garantie vollständig analysiert wird, dann können auch verlustbehaftete Komprimierungsverfahren beispielsweise basierend auf Bitstate Hashing verwendet werden.

Weitere bekannte Model Checker sind VeriSoft [God97, GHJ98, God03] und SMV [McM93, McM99, McM02]. VeriSoft wird dabei im Gegensatz zu Spin nicht auf Modelle in einer Modellierungssprache, die auf endlichen Automaten basiert, angewendet, sondern direkt auf Programme in einer Programmiersprache. Dabei entfallen die Schritte der Abstraktion und Adaption (vergleiche Abbildung 2.14). SMV ist ein symbolischer Model Checker, der intern auf Binary Decision Diagrams aufbaut.

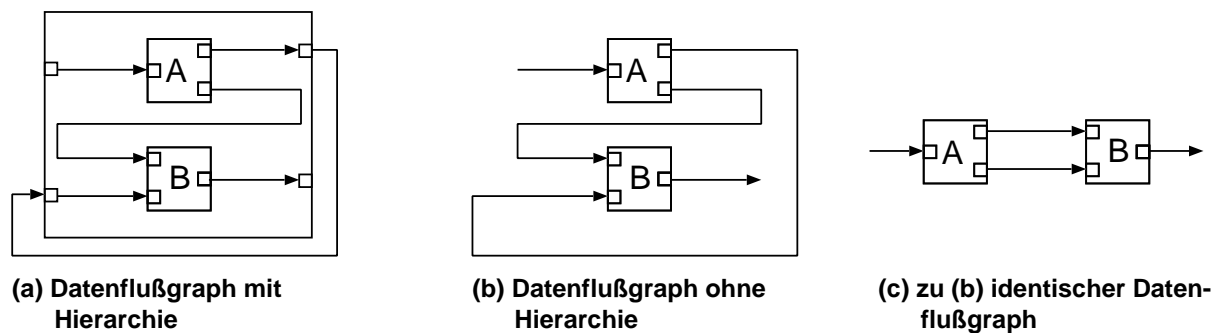


Abbildung 2.15: Beispiel eines künstlichen Deadlocks

Da Spin von der Art der Modellierung und der Analyse dieser Modelle am ehesten dem in dieser Arbeit vorgestellten Ansatz (siehe Kapitel 6) entspricht, wird in Abschnitt 8.4.2 ein Vergleich durchgeführt.

2.5 Kritikpunkte

Betrachtet man die vorgestellten Methodiken, so fallen einige Kritikpunkte ins Auge:

- **Signalmodell:** Aufgrund der BEABSICHTIGTEN UNGENAUIGKEIT der existierenden Signalmodelle für Datenflußgraphen (vergleiche Abschnitt 2.1.4) werden einige Probleme einfach wegabstrahiert. Darunter fallen beispielsweise Speicherüberläufe aufgrund nicht absolut synchron laufender Analog/Digital-Wandler. Auch Probleme beispielsweise wegen unterschiedlich eingestellter Abtastraten sind aus den in diesen Signalmodellen erfaßten Signalmerkmalen nicht ersichtlich.
- **Komponentenmodell:** Die vorgestellten Datenflußparadigmen sind NICHT KOMPOSITIO-NELL. So ist zum Beispiel in Abbildung 2.15 (a) ein KÜNSTLICHER DEADLOCK aufgrund der eingeführten Hierarchie entstanden [PPL95]. Der hierarchische Knoten kann, wenn man die einfachen SDF-Aktivierungsregeln zugrundelegt, nicht ohne ein Initialisierungstoken rechenbereit werden. Der Deadlock heißt künstlich, da er im entsprechenden Datenflußgraphen ohne Hierarchie nicht auftritt (vergleiche Abbildungen 2.15 (b) und c)).

Auch die Analyse von Datenflußgraphen mittels Gleichungssystemen stößt rasch an ihre Grenzen. So wirkt bereits der Schritt von SDF zu BDF eher erzwungen durch die Einführung von stochastischen Variablen [Buc93]. Jede zusätzliche Erweiterung, zum Beispiel um Farben analog zu gefärbten Petrinetzen [Jen97], ist mit linearen Gleichungssystemen so nicht machbar.

- **Interface-Typsystem:** In dem vorgestellten Typsystem von Ptolemy II werden nur die Wertebereiche von physikalischen Signalen typisiert. Außerdem sind die Typconstraints auf sehr einfache Ungleichungen beschränkt. Was auch auffällt, ist die unterschiedliche

Behandlung von Arrays und Matrizen, was relativ häufig zu unnötigen Konvertierungsoperationen führt.

- **Model Checking:** Die im Model Checker **Spin** verwendete Modellierungsmethodik ist sehr allgemein gehalten, um in einer beliebigen Programmiersprache definierte Algorithmen nachbilden zu können. Unter diese für Datenflußparadigmen überflüssigen Merkmale fallen zum Beispiel unterschiedlichste Zugriffsstrategien auf Fifos. Auch ist im Falle von Datenflußgraphen eine Spezifizierung der zu analysierenden Merkmale mittels temporaler Logik überflüssig, da die zu untersuchenden Merkmale wie beispielsweise das Vorhandensein zyklischer Schedules immer dieselben sind. Die Fokussierung auf Datenflußkomponenten und deren spezifische Eigenschaften ermöglicht die Verwendung einer INTELLIGENTEREN SUCHSTRATEGIE als beispielsweise Tiefensuche.

Die in diesem Kapitel wiedergegebenen Modellierungsmethoden und Verfahren stellen den aktuellen Stand der Technik dar. Im nachfolgenden Teil dieser Arbeit werden neue Modelle und Verfahren eingeführt, die eine Verbesserung dieses Stands der Technik für komponentenbasierte datenflußorientierte eingebettete Systeme hinsichtlich der in Abschnitt 1.4.1 genannten Ziele darstellen.

Kapitel 3

Signalmodell

Daß Keiner eintrete, der nicht Mathematik beherrscht. [Platon]

In diesem Kapitel wird das dieser Arbeit zugrundeliegende neue Modell physikalischer Signale vorgestellt. Nach einem kurzen Abriß über Probleme gebräuchlicher Signalmodelle werden die Anforderungen an ein neues Modell formuliert. Daran schließt sich die schrittweise Einführung des neuen Signalmodells an. Am Ende des Kapitels werden die innovativen Aspekte dieses Ansatzes beleuchtet. Teile dieses neuen Signalmodells wurden in [MG03, May04a] vorgestellt.

3.1 Probleme mit gebräuchlichen Signalmodellen

In Abbildung 3.1 (a) ist eine Additionskomponente dargestellt, die aus zwei Strömen von Eingabewerten einen Strom von Ausgabewerten berechnet. Dabei werden jeweils zwei Signalwerte miteinander verknüpft. Ist wie in Abschnitt 2.1.3 Zeit implizit durch die Position der Signalwerte in den Fifos ausgedrückt, können sehr leicht Probleme auftreten. Wird beispielsweise das am oberen Eingang anliegende physikalische Signal schneller abgetastet, so ist die Additionsoperation NICHT WOHLDEFINIERT, da nun Signalwerte, die zu unterschiedlichen Zeitpunkten erfaßt wurden, miteinander verrechnet werden (vergleiche Abbildung 3.1 (b)). Die so berechneten Ergebnisse sind sinnlos. Als zweites Problem tritt SPEICHERÜBERLAUF auf, da das am oberen Eingang anliegende physikalische Signal im gleichen Zeitraum öfter abgetastet wird als das am unteren Eingang anliegende physikalische Signal. Aus den Fifos werden aber oben genausoviele Signalwerte wie unten entfernt. Aufgrund dieses Speicherüberlaufs kommt es in absehbarer Zeit zu einer VORZEITIGEN PROGRAMMTERMINIERUNG. Als Lösung werden im folgenden neue Signalmerkmale eingeführt mit dem Ziel, solche und ähnliche Fehler mit Hilfe eines Typsystems zu erkennen und zu vermeiden.

In Abbildung 2.1 (a.iv) ist das Modell stückweiser konstanter physikalischer Signale, in dem man nur Wertänderungen und zugeordnete Zeitpunkte überträgt, veranschaulicht [DL04]. Dieses Modell erscheint auf den ersten Blick sehr gut dafür geeignet, physikalische Signale kompakt

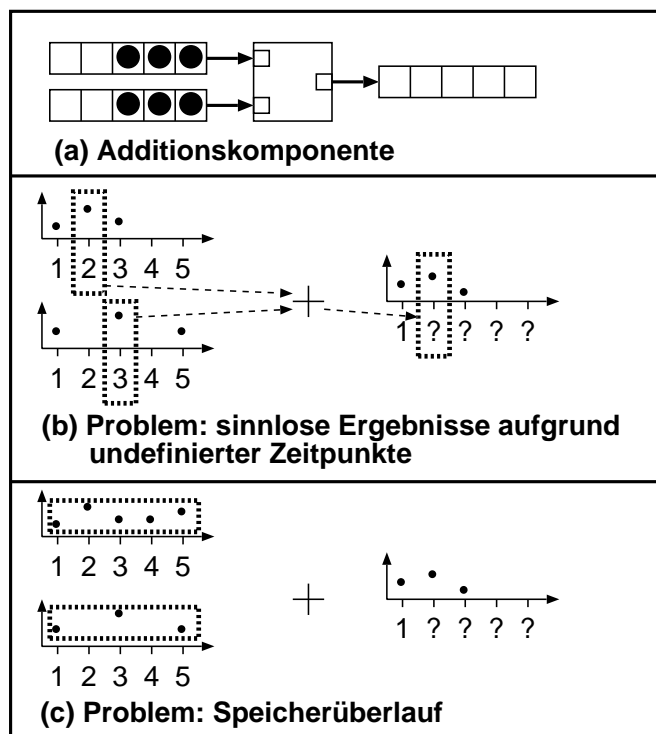


Abbildung 3.1: Probleme bei Vernachlässigung des Zeitbereichs physikalischer Signale

darzustellen. Daß dies im Rahmen von Datenflußgraphen problematisch ist, veranschaulicht Abbildung 3.2. Wenn zwei physikalische Signale, die von zwei Quellkomponenten eingelesen werden, an einer beliebigen Stelle im Datenflußgraphen miteinander verrechnet werden, dann tritt in der Regel folgende Situation auf: Ändert sich das physikalische Signal von Quellkomponente A, so kann das Gültigkeitsintervall des VORAUSGEGANGENEN Signalwertes erst jetzt bestimmt und übertragen werden. Das heißt, je nach Größe des Datenflußgraphen und dem Abstand zwischen Signaländerungen kann es zu erheblichen Verzögerungen in der Reaktionszeit kommen. Man kann natürlich auch das Gültigkeitsintervall unterteilen, indem man zwischendurch immer wieder die Signalwerte überträgt, obwohl eigentlich keine Änderung stattgefunden hat. Denkt man diesen Ansatz konsequent zu Ende, kommt man letztendlich zur gleichmäßigen Abtastung der beiden physikalischen Signale. Wie man ein effizientes Signalmodell für Datenflußgraphen erhält, wird im folgenden dargestellt.

3.2 Anforderungen

Aus den in Abschnitt 3.1 dargestellten Problemen ergeben sich diverse Anforderungen an ein neues Modell physikalischer Signale. Weitere Anforderungen werden im folgenden mittels einer Reihe von Beispielen für physikalische Signale (vergleiche Abbildung 3.3) motiviert.

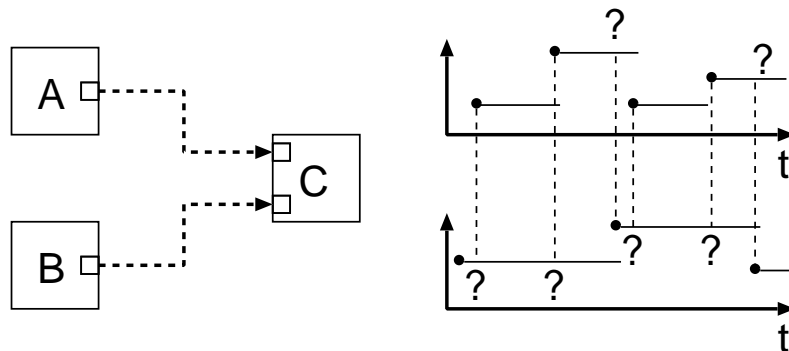


Abbildung 3.2: Problem bei stückweise konstanten physikalischen Signalen

- **SONAR BEZIEHUNGSWEISE ECHOLOT:** Die Echolokation oder auch Sonarortung (Sonar: Sound Navigation and Ranging) [Uri83, AES98] funktioniert gleich dem zur Orientierung verwendeten Echolot. Beim Echolot wird eine Serie von Tönen gleicher Frequenz im Ultraschallbereich ausgesendet, welche zum Beispiel vom Meeresboden reflektiert werden. Diese reflektierten Töne werden im Schiff ausgewertet und die Meerestiefe bestimmt. Die Sonarortung verwendet mehrere Frequenzen und erlaubt neben der Abstandsmessung auch eine Bestimmung von Form, Dichte, Oberflächenstruktur und – bis zu einem gewissen Maß – auch Aufbau des anvisierten Objekts. Abbildung 3.3 (a) zeigt das Meßprinzip und typische Verläufe von gesendeten und empfangenen Signalen. Charakteristisch sind die unterschiedlich langen Antwortzeiten. Diese stellen einen **WESENTLICHEN INFORMATIONSTRAGENDE ANTEIL** der Signale dar und sind daher in das Signalmodell miteinzubeziehen.
- **RADAR:** Radarsignale (Radio Detection and Ranging) [RF99, Pal02, SMH02] werden unter anderem zur Abstandsbestimmung zwischen Fahrzeugen, zur Füllstandsmessung von Behältern (vergleiche Abbildung 3.3 (b)) beziehungsweise zur Ortung von Flugzeugen (siehe Abbildung 3.3 (c)) eingesetzt. Bei der Füllstandsmessung werden kurze Impulse ausgesandt. Die von der Umgebung und insbesondere vom Füllgut reflektierten Impulse werden als Radarechos empfangen. Die **LAUFZEIT** eines Impulses vom Aussenden bis zum Empfangen ist dabei dem Abstand zum Füllgut proportional. Die Positionsbestimmung von Flugzeugen ist dagegen aufwendiger. Nach dem Aussenden des Ortungsimpulses wird nach einer kurzen Erholzeit auf Empfang umgeschaltet. Um falsche Abstandbestimmungen aufgrund Reflexion zeitlich weiter zurückliegender Ortungsimpulse durch weit entfernte Objekte zu vermeiden, wird nach einer gewissen Zeit der Empfang deaktiviert. Anschließend wird der nächste Ortungsimpuls ausgesandt. Damit wird das Empfangssignal auf **ZEITBEREICHE BESTEHEND AUS RELEVANTEN DATEN** eingeschränkt.
- **GLOBAL POSITIONING SYSTEM (GPS):** Ziel von GPS [HWLC97, NJN98] ist die Ortung von Objekten auf und in der Nähe der Erde mittels Satelliten. Neben der Position

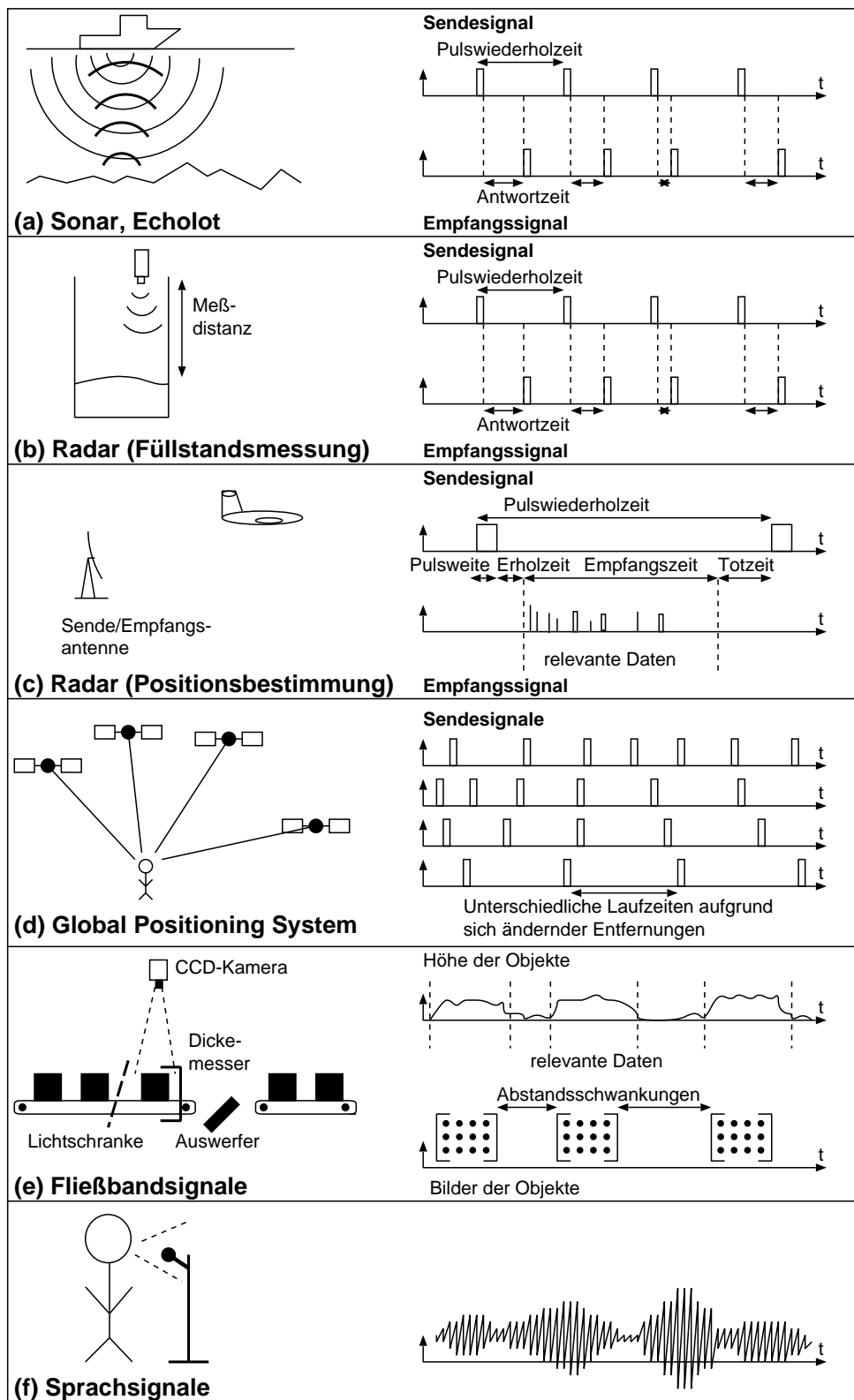


Abbildung 3.3: Beispiele für physikalische Signale

der Objekte ist auch deren Bewegungsrichtung und Geschwindigkeit feststellbar. Zu jedem Zeitpunkt befinden sich mindestens vier Satelliten in brauchbarer Höhe über dem Horizont. Dabei sendet jeder Satellit laufend ein Datenpaket aus, das unter anderem die SENDEZEIT und die augenblickliche Position des Satelliten enthält. Aus der LAUFZEIT ergibt sich dann die Entfernung zum Satelliten. Mit drei solchen Abständen zu verschiedenen Satelliten kann man nun die Position des Empfängers im Raum bestimmen, indem man einfach die drei Kugeln um die Satelliten mit ihrem Radius gleich dem jeweils ermittelten Abstand schneidet. Ganz so einfach ist die Berechnung aber aufgrund relativistischer Effekte und der ungenauen Uhr des Empfängers nicht (vergleiche dazu [HWLC97]). Abbildung 3.3 (d) zeigt den prinzipiellen Aufbau und typische Signalverläufe. Man erkennt, daß der ZEITINFORMATION auch hier eine ZENTRALE ROLLE zukommt.

- **FLIESSBANDSIGNALLE:** Betrachtet man ein typisches Fließband [Cro98], wie in Abbildung 3.3 (e) dargestellt, so liegen in der Regel die transportierten Objekte darauf mit ungleichem Abstand voneinander. Zudem kann man zwischen RELEVANTEN SIGNALDATEN, die einem Objekt zugeordnet werden können, und irrelevanten Signalwerten, die keinem Objekt zugeordnet sind, unterscheiden. Gegebenenfalls wird diese Trennung durch ein TRIGGERSIGNAL, das zum Beispiel mittels einer Lichtschranke ausgelöst wird, unterstützt. Werden von verschiedenen Sensoren physikalische Signale erfaßt, so müssen diese geeignet einander zugeordnet werden.
- **SPRACHSIGNALLE:** Sprachsignale (siehe Abbildung 3.3 (f)) zeichnen sich dadurch aus, daß sich Sätze zwar aus einzelnen Wörtern aufbauen, deren Abgrenzung aber nicht ohne Vorarbeit möglich ist. Dies liegt daran, daß in der Regel keine Sprechpausen eingelegt werden. Für die Analyse von Sprachsignalen werden neben dem Zeitbereich auch ANDERE DEFINITIONSBEREICHE betrachtet. So wird beispielsweise das Signal auch in den Frequenzbereich transferiert, um das Amplitudenspektrum zu analysieren.

Aus den in Abschnitt 3.1 und den gerade genannten Beispielen lassen sich die Anforderungen an ein neues Modell physikalischer Signale für Datenflußgraphen ableiten.

3.2.1 Unterstützung bei der Fehlererkennung

Die zentrale Anforderung an das neue Signalmodell ist die Unterstützung von ENTWURFSBEGLEITENDER FEHLERERKENNUNG:

- Zum einen soll das Signalmodell die Definition von Typconstraints unterstützen. Mit diesen werden Fehler erkannt, die auf nichtzusammenpassende Definitions- und Wertebereiche von physikalischen Signalen basieren. Auch Aspekte des Datentransports (siehe Abschnitt 3.3.2) spielen eine wichtige Rolle. Daraus ergibt sich die Forderung, zentrale Signalcharakteristika herauszuarbeiten, welche DEFINITIONSBEREICH, WERTEBEREICH und ASPEKTE DES DATENTRANSPORTS ausreichend genau beschreiben, so daß auf dieser Basis Typconstraints definiert werden können. Diese Typconstraints werden dann mit Hilfe eines Interface-Typsysteams (siehe Kapitel 5) überprüft.

- Zum anderen sollen auch DYNAMISCHE ABLÄUFE, die in Form von KOMMUNIKATIONS-PROTOKOLLEN festgehalten sind, überprüft werden. Dazu ist es erforderlich, Signalmerkmale zu definieren, welche eine einfache Erfassung solcher dynamischer Vorgänge in Form von Kommunikationsprotokollen ermöglichen. Die Kompatibilität dieser Kommunikationsprotokolle wird dann mittels eines Model Checkers geprüft (vergleiche Kapitel 6).

3.2.2 Flexibilität in der Darstellung

Ein zweiter wichtiger Punkt, der für die Relevanz eines Signalmodells entscheidend ist, ist die FLEXIBILITÄT, mit welcher verschiedene Arten von physikalischen Signalen dargestellt werden können.

- Zum einen möchte man effizient die folgenden Arten von ZEITBEREICHSSIGNALEN modellieren (siehe Abbildung 3.4).
 1. Physikalische Signale bestehend aus unendlich vielen äquidistant abgetasteten Signalwerten: dies ist die gängige Darstellungsart für Zeitsignale.
 2. Physikalische Signale bestehend aus unendlich vielen Signalsegmenten, die äquidistant abgetastete relevante Signalwerte beinhalten: solche Signale treten zum Beispiel bei Fließbandsteuerungen (siehe Abbildung 3.3 (e)) oder bei Radarsignalen (siehe Abbildung 3.3 (c)) auf. Weitere Beispiele sind Sprachsignale, Signale biologischer Neuronen beziehungsweise Herzrhythmus-signale [AMG02].
 3. Physikalische Signale bestehend aus unendlich vielen nichtäquidistant abgetasteten Signalwerten: Ein Beispiel dafür sind Tastatureingaben (siehe Abbildung 3.4).
- Zum anderen sind in vielen Signalverarbeitungsanwendungen Transformationen von Signalen aus dem Zeitbereich in ANDERE DEFINITIONSBEREICHE wie beispielsweise den Frequenzbereich erforderlich. Auch diese physikalischen Signale sollen modelliert werden können.

3.2.3 Verminderung des Transport-Overheads

Der dritte wichtige Gesichtspunkt beim Entwurf eines Signalmodells ist die Unterstützung einer EFFIZIENTEN DATENÜBERTRAGUNG. Dazu werden folgende Anforderungen gestellt:

- Es soll eine TRENNUNG VON RELEVANTEN UND IRRELEVANTEN SIGNALDATEN möglich sein, so daß man letztere verwerfen kann.
- Die Signaldaten sollen in geeignet großen Datenpaketen übertragen werden, um sowohl TRANSPORT-OVERHEAD als auch lange und stark schwankende WARTEZEITEN zu vermeiden. Transport-Overhead entsteht, wenn die jeweils übertragene Datenmenge zu klein ist und somit Verwaltungsaufwand zum Beispiel resultierend aus Prozeßumschaltungen überproportional zu Buche schlägt. Lange und stark schwankende Wartezeiten ergeben sich, wenn die Größe der zu übertragenden Datenmenge zur Laufzeit stark variiert.

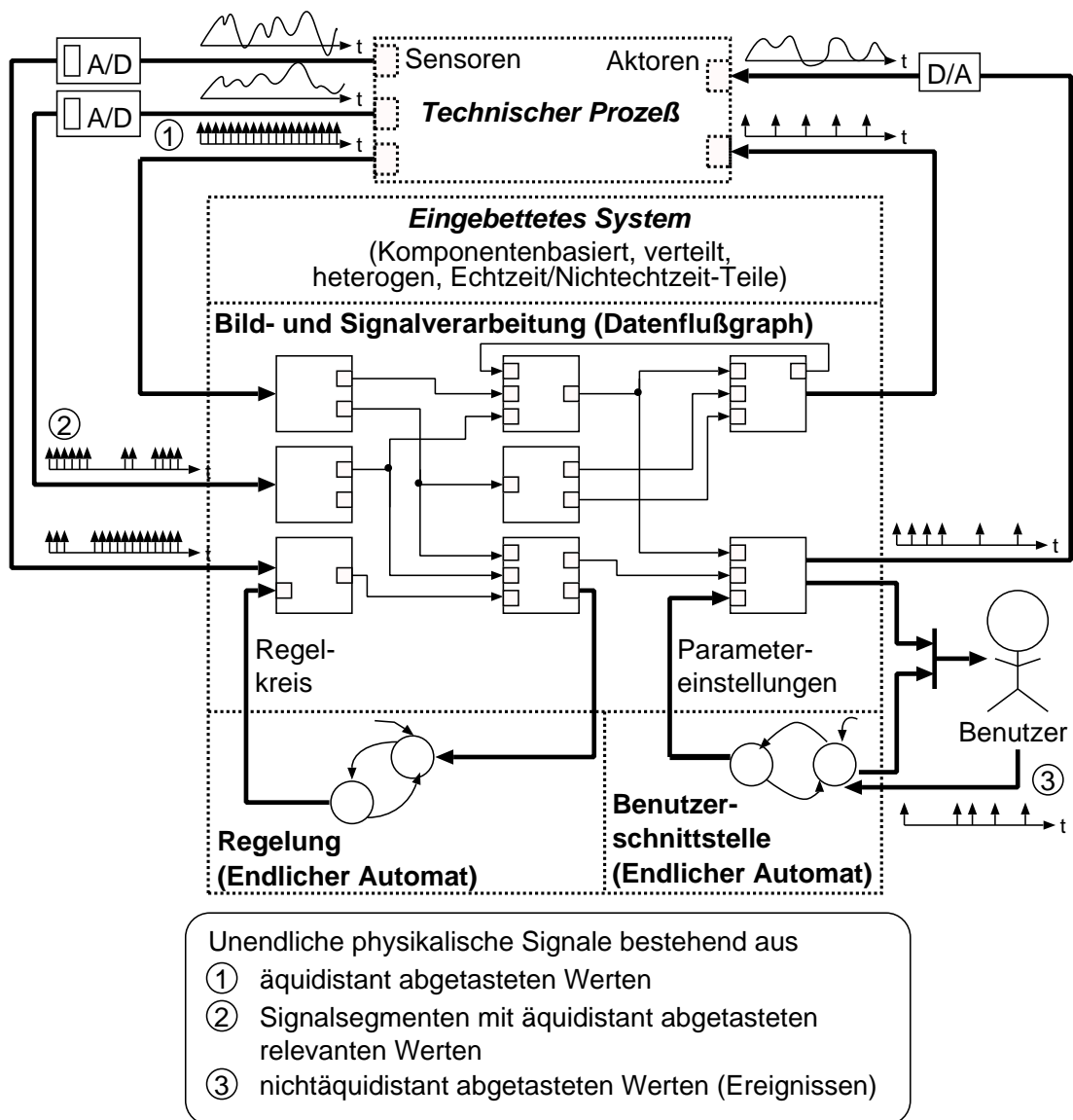


Abbildung 3.4: Physikalische Signale in einem eingebetteten System

- Die einzelnen ZEITPUNKTE der Paare bestehend aus Zeit und zugehörigem Signalwert sollen nicht mitübertragen sondern REKONSTRUIERT werden. Dies bietet sich bei äquidistant abgetasteten physikalischen (Teil-)Signalen an und führt zu einer weiteren Verringerung der zu übertragenden Datenmenge.

3.3 Neues Modell

Ausgehend von Analogsignalen wird das neue Modell physikalischer Signale schrittweise eingeführt. Daran schließt sich eine Zusammenfassung der neuen Signalmerkmale an. Eine formale Beschreibung des Signalmodells rundet diesen Abschnitt ab.

3.3.1 Schrittweise Einführung

Ausgehend von einem analogen physikalischen Signal ergibt sich die Beschreibung eines digitalen physikalischen Signals. Darauf bauen die Darstellungen physikalischer Signale als Folgen von Signalsegmenten, Signalblöcken beziehungsweise gefärbten Token auf.

3.3.1.1 Analoge und digitale Signale

Im allgemeinen sind die physikalischen Signale, die an einem technischen Prozess gemessen werden, analog (vergleiche Abbildung 3.5 (a)) und müssen vor Übernahme in einen Rechner abgetastet und quantisiert werden. Daher kann man physikalische Signale als Funktionen

$$f_d : \mathbb{T}_a \rightarrow \mathbb{X}_q \quad (3.1)$$

von einer diskreten Zeitmenge \mathbb{T}_a in eine diskrete Wertemenge \mathbb{X}_q darstellen. Der Zeitbereich \mathbb{T}_a ist eine unendliche Menge äquidistanter Zeitpunkte¹:

$$\mathbb{T}_a = \{t_i = t_s + i \cdot t_p \mid t_s, i \in \mathbb{N}_0, t_p \in \mathbb{N}\} \quad (3.2)$$

t_s : Startzeitpunkt

t_p : Abtastperiode

(3.3)

Der Wertebereich \mathbb{X}_q ist gegeben durch eine Wertemenge, welche den klassischen Datentypen entspricht, und einer physikalische Einheit pu wie beispielsweise Volt, Ampère beziehungsweise Newton. Die klassischen Datentypen setzen sich dabei aus Basisdatentypen **BT** und Aggregationstypen **AT** zusammen. Eine genauere Beschreibung des Wertebereichs wird in Abschnitt 5.3.2.1 gegeben.

$$\mathbb{X}_q \in (\mathbf{BT} \cup \mathbf{AT}) \times \mathbf{PU} \quad (3.4)$$

BT = {number, int, long, float, double, complex, bool, string}

AT : Aggregationstypen

PU : physikalische Einheiten

¹In dieser Arbeit ist die Menge der natürlichen Zahlen ohne die 0 durch \mathbb{N} , mit der 0 durch \mathbb{N}_0 gekennzeichnet.

Diskrete Signale werden gewöhnlich als unendliche Folge² von Paaren aus Zeiten und Werten (vergleiche Abbildung 3.5 (b)) dargestellt:

$$[(t_s + i \cdot t_p, x_i)]_{i=0}^{\infty} \quad (3.5)$$

t_s : Startzeitpunkt
 t_p : Abtastperiode
 $x_i \in \mathbb{X}_q$.

3.3.1.2 Signale als Folgen von Segmenten

In vielen Anwendungen kann man zwischen relevanten und irrelevanten Signaldaten unterscheiden (vergleiche Abbildung 3.3 (c) beziehungsweise (e)). So werden zum Beispiel nur solche Signaldaten als relevant eingestuft, die oberhalb eines Schwellwertes x_{th} liegen. Die in den Totzeiten auftretenden Signaldaten werden entweder überhaupt nicht durch die Hardware erfaßt beziehungsweise von der Software verworfen und somit bei der weiteren Verarbeitung nicht berücksichtigt. Dadurch wird das ursprüngliche physikalische Signal (zum Beispiel durch eine Einsteckkarte eines Sensors) in mehrere Signalsegmente, die ausschließlich relevante Signalwerte $x_i > x_{th}$ enthalten, zerteilt (vergleiche Abbildung 3.5 (c)):

$$\underbrace{[(t_{s_j} + i \cdot t_p, x_i)]_{i=0}^{n_j-1}}_{\text{Segment } j} \bullet \underbrace{[(t_{s_{j+1}} + i \cdot t_p, x_i)]_{i=0}^{n_{j+1}-1}}_{\text{Segment } j+1} \bullet \dots \quad (3.6)$$

n_j : Anzahl der Elemente von Segment j .

Die Konkatenation zweier Listen wird mittels des Operators \bullet dargestellt³. Dabei gilt natürlich, daß sich aufeinanderfolgende Signalsegmente, die zu demselben physikalischen Signal gehören, zeitlich nicht überlappen:

$$t_{s_j} + n_j \cdot t_p < t_{s_{j+1}} . \quad (3.7)$$

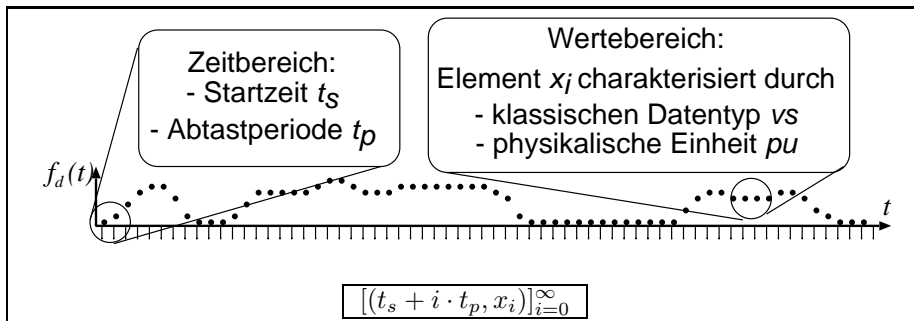
Dieses neue Signalmodell ist in der Lage, alle drei Arten unendlicher physikalischer Signale, die in Abbildung 3.4 dargestellt sind, zu repräsentieren. Physikalische Signale der dritten dargestellten Art entsprechen einer unendlichen Folge von endlichen Signalsegmenten, die jeweils nur einen Signalwert beinhalten.

²Folgen werden in dieser Arbeit mit Hilfe von eckigen Klammern [...] dargestellt.

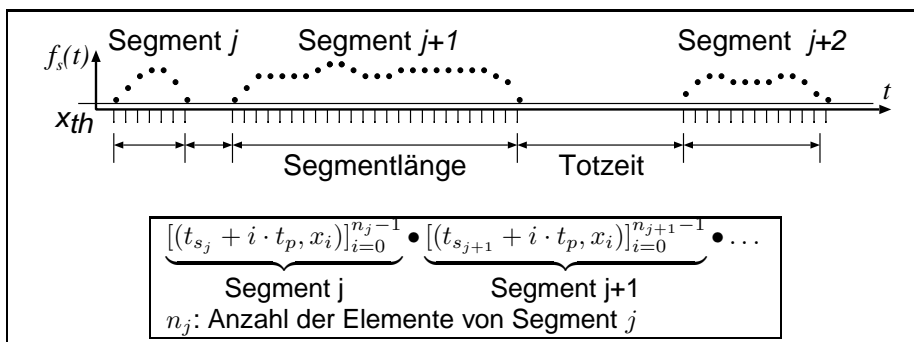
³In dieser Arbeit ist der Operator \bullet dergestalt überladen, daß er sowohl für die Konkatenation von Listen als auch für das Anhängen von Elementen an eine Liste verwendet wird. Die jeweilige Verwendungsart erschließt sich aus dem Kontext.



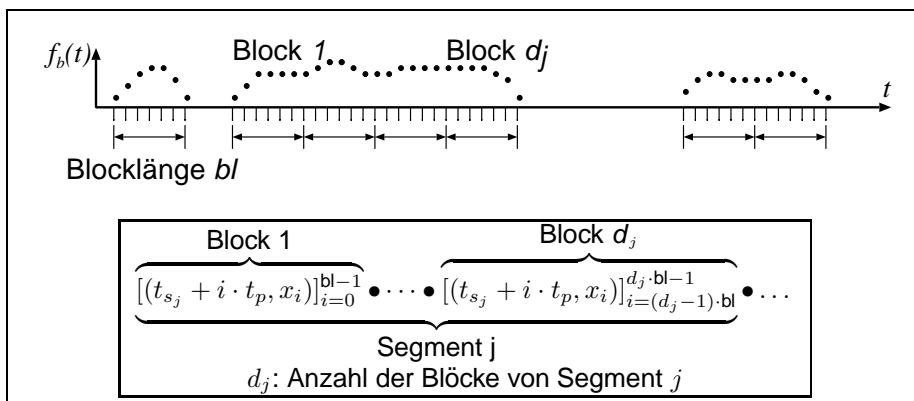
(a) Analoges Signal



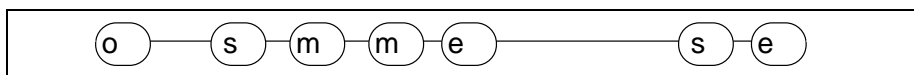
(b) Digitales (abgetastetes und quantisiertes) Signal



(c) Signal als Folge von Segmenten



(d) Signal als Folge von Blöcken



(e) Signal als Strom gefärbter Token

Abbildung 3.5: Schrittweise Einführung des neuen Signalmodells

3.3.1.3 Signale als Folgen von Blöcken

Signalsegmente werden weiter in gleich große Signalblöcke unterteilt (vergleiche Abbildung 3.5 (d)):

$$\underbrace{\overbrace{[(t_{s_j} + i \cdot t_p, x_i)]_{i=0}^{bl-1}}^{\text{Block 1}} \bullet \dots \bullet \overbrace{[(t_{s_j} + i \cdot t_p, x_i)]_{i=(d_j-1) \cdot bl}^{d_j \cdot bl-1}}^{\text{Block } d_j}}_{\text{Segment } j}} \bullet \dots \bullet \quad (3.8)$$

d_j : Anzahl der Blöcke von Segment j .

Die Blocklänge bl ist dabei für alle Signalsegmente eines physikalischen Signals in der Regel identisch, kann aber zwischen verschiedenen physikalischen Signalen unterschiedlich sein.

Für diese Unterteilung gibt es mehrere Gründe:

- Die Blocklänge bl kann – eventuell während einer Kalibrierungsphase – so eingestellt werden, daß der TRANSPORT-OVERHEAD in einem verteilten eingebetteten Echtzeitsystem möglichst minimiert wird. Der Transfer einzelner Signalwerte führt zu einem erheblichen Verwaltungsaufwand. Demgegenüber resultiert die Übertragung kompletter Signalsegmente in zu langen Reaktionszeiten des eingebetteten Systems. Außerdem ist im letztgenannten Fall die Länge der Signalsegmente oft nicht vorhersagbar.
- Mehrere Datenflußkomponenten, wie beispielsweise eine FFT-Komponente, benötigen einen Block von Signalwerten bestimmter Länge für ihre Ausführung.
- Außerdem produziert die Meßhardware gegebenenfalls sowieso Signalblöcke.

3.3.1.4 Signale als Folgen von gefärbten Token

Um die Segmentstruktur eines Signals rekonstruieren zu können, wird jedem Signalblock eine Blockmarkierung, die nachfolgend auch als Farbe bezeichnet wird, zugeordnet. Die verschiedenen Werte dieser Blockmarkierung bm sind: s für Startblock, m für interne (mittlere) Signalblöcke und e für den Endblock eines Signalsegments. Besteht ein Signalsegment nur aus einem Signalblock, der dann gleichzeitig Start- und Endblock ist, wird dies durch die Blockmarke o gekennzeichnet. Die Kombination aus Signalblock und Blockmarkierung wird im folgenden als gefärbtes Token bezeichnet. In diesem Zusammenhang spricht man von Starttoken, internen (mittleren) Token und Endtoken. Beispiele finden sich in Abbildung 3.5 (e).

3.3.2 Neue Signalmerkmale

Zusammenfassend wird ein physikalisches Signal durch folgende sechs Signalmerkmale charakterisiert:

- DEFINITIONSBEREICH: Der Definitionsbereich eines physikalischen Signals ist ein wesentlicher Bestandteil und wird durch zwei Signalcharakteristika erfaßt.

1. Für jedes Signalsegment wird die Startzeit t_s festgehalten.
2. Außerdem benötigt man noch die Abtastperiode t_p , um die einzelnen Zeitpunkte t_i rekonstruieren zu können:

$$t_i = t_s + i \cdot t_p \quad (3.9)$$

Dabei stellt i den Index des betrachteten Signalwertes in der Signalwertfolge dar (vergleiche Abbildung 3.5 (c)).

- WERTEBEREICH: Der Wertebereich eines physikalischen Signals wird durch folgende zwei Signalmerkmale charakterisiert.
 3. Die WERTEMENGE beschreibt die zulässigen Signalwerte. Dazu werden klassische Datentypen wie `int` oder `double` verwendet. Um komplexere Werte wie beispielsweise Bilder oder von einer Sensorphalanx erfaßte Datentupel beschreiben zu können, werden Aggregationstypen wie Arrays oder Records eingesetzt.
 4. Neben der Wertemenge ist bei physikalischen Signalen die zugehörige physikalische Einheit von großer Bedeutung. Diese wird mittels SI-Einheiten angegeben.
- ASPEKTE DES DATENTRANSPORTS: Da in eingebetteten Systemen auch der Transport der Daten und ihre Verarbeitung durch die Datenflußkomponenten eine wesentliche Rolle spielen, werden die folgenden zwei Merkmale berücksichtigt.
 5. Die BLOCKLÄNGE beschreibt die Anzahl der Signalwerte in einem Datenblock.
 6. Die FARBIGE BLOCKMARKIERUNG dient zur Rekonstruktion von Signalsegmenten.

3.3.3 Formale Beschreibung

Als Basis für das in Kapitel 4 vorgestellte Komponentenmodell ist eine formale Beschreibung physikalischer Signale erforderlich. Die dabei verwendeten mathematischen Grundlagen finden sich in [DP02].

Das formale Modell eines PHYSIKALISCHEN SIGNALS basiert auf den folgenden Mengenbeschreibungen:

DEFINITION 3.3.1: Die MENGE DER PUNKTE P ist wie folgt definiert:

$$P = \mathbb{T}_a \times \mathbb{X}_q \quad (3.10)$$

Dabei stellt \mathbb{T}_a die diskrete Zeitmenge und \mathbb{X}_q die diskrete Wertemenge dar.

Zur Veranschaulichung dient folgendes Zahlenbeispiel, in welchem ein Signalpunkt jeweils durch einen Zeitwert und ein Element der Wertemenge gegeben ist. Das Element der Wertemenge setzt sich in diesem Beispiel je aus einem Zahlenwert und der physikalischen Einheit Ampère zusammen.

$$\{(1, (9, A)), (3, (7, A)), (2, (10, A)), (4, (5, A)), \dots\} \quad (3.11)$$

Mit Hilfe von Signalpunkten lassen sich Signalblöcke definieren.

DEFINITION 3.3.2: *Die Menge aller endlichen Folgen von Punkten heißt BLOCKMENGE B (t_s : Startzeit, t_p : Abtastperiode, bl : Blocklänge).*

$$B = \{[(t_s + i \cdot t_p, x_i)]_{i=0}^{bl-1} | t_s \in \mathbb{N}_0, t_p \in \mathbb{N}, t_s + i \cdot t_p \in \mathbb{T}_a, x_i \in \mathbb{X}_q, bl \in \mathbb{N}\} \cup \{\varepsilon_B\} \quad (3.12)$$

Der LEERE BLOCK wird mit ε_B bezeichnet.

In Fortsetzung des obigen Zahlenbeispiels erhält man als Beispiel für eine Blockmenge:

$$\begin{aligned} & \{[(0, (7, A)), (1, (9, A)), (2, (10, A)), (3, (7, A)), (4, (5, A))], \\ & [(6, (2, A)), (7, (4, A)), (8, (9, A)), (9, (11, A)), (10, (20, A))], \\ & [(13, (5, A)), (14, (5, A)), (15, (27, A)), (16, (5, A)), (17, (4, A))], \\ & [(20, (84, A)), (21, (15, A)), (22, (8, A)), (23, (47, A)), (24, (52, A))], \\ & [(33, (1, A)), (34, (3, A)), (35, (3, A)), (36, (3, A)), (37, (1, A))], \\ & [(40, (3, A)), (41, (5, A)), (42, (34, A)), (43, (42, A)), (44, (7, A))]\} \end{aligned} \quad (3.13)$$

Mit Hilfe von Blöcken kann man eine Menge von Token definieren:

DEFINITION 3.3.3: *Ein TOKEN ist ein Paar bestehend aus einer Blockmarkierung (Farbe) $c \in C$ und einem Signalblock $b \in B$. Dabei ist die Menge der Farben gegeben als $C = \{\mathbf{s}, \mathbf{m}, \mathbf{e}, \mathbf{o}\}$ (vergleiche Abschnitt 3.3.1.4). Die MENGE ALLER TOKEN ist gegeben durch:*

$$T = C \times B \quad (3.14)$$

In Fortsetzung obigen Zahlenbeispiels erhält man als Beispiel für eine Tokenmenge:

$$\begin{aligned} & \{(\mathbf{s}, [(0, (7, A)), (1, (9, A)), (2, (10, A)), (3, (7, A)), (4, (5, A))]), \\ & (\mathbf{m}, [(6, (2, A)), (7, (4, A)), (8, (9, A)), (9, (11, A)), (10, (20, A))]), \\ & (\mathbf{m}, [(13, (5, A)), (14, (5, A)), (15, (27, A)), (16, (5, A)), (17, (4, A))]), \\ & (\mathbf{m}, [(20, (84, A)), (21, (15, A)), (22, (8, A)), (23, (47, A)), (24, (52, A))]), \\ & (\mathbf{e}, [(33, (1, A)), (34, (3, A)), (35, (3, A)), (36, (3, A)), (37, (1, A))]), \\ & (\mathbf{o}, [(40, (3, A)), (41, (5, A)), (42, (34, A)), (43, (42, A)), (44, (7, A))])\} \end{aligned} \quad (3.15)$$

Zur Vereinfachung des Formalismus wird gelegentlich von der Blockbeschreibung abstrahiert. Damit wird ein Token allein durch seine Färbung beschrieben. Diese Darstellung wird insbesondere bei der Modellierung des Kommunikationsverhaltens in Kapitel 6 verwendet. Daraus ergibt sich die Definition einer VEREINFACHTEN TOKENMENGE:

DEFINITION 3.3.4: *Die vereinfachte Tokenmenge ist gegeben durch:*

$$T_s = C \quad (3.16)$$

Aufbauend auf die Tokenmenge wird ein Signalsegment definiert:

DEFINITION 3.3.5: Ein SIGNALSEGMENT $\hat{S}_{t_{s_j}, t_p, \mathbb{X}_q, d_j, bl}$ ist eine Folge von Token aus $T = C \times B$:

$$\hat{S}_{t_{s_j}, t_p, \mathbb{X}_q, d_j, bl} = \begin{cases} (\mathbf{s}, [(t_{s_j} + i \cdot t_p, x_i \in \mathbb{X}_q)]_{i=0}^{bl-1}) \\ \bullet \left[(\mathbf{m}, [(t_{s_j} + i \cdot t_p, x_i)]_{i=(k-1) \cdot bl}^{k \cdot bl-1}) \right]_{k=1}^{d_j-2} \\ \bullet \left(\mathbf{e}, [(t_{s_j} + i \cdot t_p, x_i)]_{i=(d_j-2) \cdot bl}^{(d_j-1) \cdot bl-1} \right) \text{ falls } d_j > 1 \\ (\mathbf{o}, [(t_{s_j} + i \cdot t_p, x_i)]_{i=0}^{bl-1}) \text{ falls } d_j = 1 \end{cases} \quad (3.17)$$

d_j repräsentiert die Anzahl der Blöcke des Signalsegmentes.

Das gegebene Zahlenbeispiel wird erweitert zu einem Beispiel eines Signalsegmentes:

$$\begin{aligned} & [(\mathbf{s}, [(0, (7, A)), (1, (9, A)), (2, (10, A)), (3, (7, A)), (4, (5, A))]), \\ & (\mathbf{m}, [(6, (2, A)), (7, (4, A)), (8, (9, A)), (9, (11, A)), (10, (20, A))]), \\ & (\mathbf{m}, [(13, (5, A)), (14, (5, A)), (15, (27, A)), (16, (5, A)), (17, (4, A))]), \\ & (\mathbf{m}, [(20, (84, A)), (21, (15, A)), (22, (8, A)), (23, (47, A)), (24, (52, A))]), \\ & (\mathbf{e}, [(33, (1, A)), (34, (3, A)), (35, (3, A)), (36, (3, A)), (37, (1, A))]) \end{aligned} \quad (3.18)$$

Physikalische Signale in Form von Tokenströmen bestehen aus Signalsegmenten, wie aus nachfolgender Definition ersichtlich wird.

DEFINITION 3.3.6: Die MENGE DER ENDLICHEN STRÖME S^f beschreibt alle endlichen Ströme von Token und ist gegeben durch:

$$S^f = \left\{ \left[\hat{S}_{t_{s_j}, t_p, \mathbb{X}_q, d_j, bl} \right]_{j=0}^{n-1} \left| d_j = \left\lceil \frac{n_j}{bl} \right\rceil, t_{s_j} \in \mathbb{N}_0, n, n_j, t_p \in \mathbb{N}, t_{s_j} + n_j \cdot t_p \leq t_{s_{j+1}} \right\} \quad (3.19)$$

mit n_j als Anzahl der Punkte des j -ten Signalsegmentes.

Die MENGE DER UNENDLICHEN STRÖME S^∞ beschreibt alle unendlichen Ströme von Token und ist definiert als:

$$S^\infty = \left\{ \left[\hat{S}_{t_{s_j}, t_p, \mathbb{X}_q, d_j, bl} \right]_{j=0}^\infty \left| d_j = \left\lceil \frac{n_j}{bl} \right\rceil, t_{s_j} \in \mathbb{N}_0, n_j, t_p \in \mathbb{N}, t_{s_j} + n_j \cdot t_p \leq t_{s_{j+1}} \right\}. \quad (3.20)$$

Die MENGE DER STRÖME S beschreibt alle endlichen und unendlichen Ströme von Token:

$$S = S^f \cup S^\infty. \quad (3.21)$$

Der leere Strom wird dabei durch ε_S symbolisiert.

Damit ist ein physikalisches Signal als ein Strom $s \in S$ modelliert. Ein Beispiel für einen solchen Strom von Token, der ein physikalisches Signal repräsentiert, ist gegeben durch

$$\begin{aligned} & [(\mathbf{s}, [(0, (7, A)), (1, (9, A)), (2, (10, A)), (3, (7, A)), (4, (5, A))]), \\ & (\mathbf{m}, [(6, (2, A)), (7, (4, A)), (8, (9, A)), (9, (11, A)), (10, (20, A))]), \\ & (\mathbf{m}, [(13, (5, A)), (14, (5, A)), (15, (27, A)), (16, (5, A)), (17, (4, A))]), \\ & (\mathbf{m}, [(20, (84, A)), (21, (15, A)), (22, (8, A)), (23, (47, A)), (24, (52, A))]), \\ & (\mathbf{e}, [(33, (1, A)), (34, (3, A)), (35, (3, A)), (36, (3, A)), (37, (1, A))]), \\ & (\mathbf{o}, [(40, (3, A)), (41, (5, A)), (42, (34, A)), (43, (42, A)), (44, (7, A))]) \end{aligned} \quad (3.22)$$

Neben der Strommenge S wird auch die vereinfachte Strommenge S_s definiert, die auf der vereinfachten Tokenmenge T_s aufbaut.

DEFINITION 3.3.7: *Aufbauend auf die vereinfachte Tokenmenge T_s ist die VEREINFACHTE MENGE DER STRÖME durch folgenden regulären Ausdruck gegeben:*

$$S_s = (sm^*e|o)^* \text{ mit } s, m, e, o \in T_s \quad (3.23)$$

Das obige Beispiel in Bezug auf die vereinfachte Strommenge S_s lautet:

$$\text{smmmeo} \quad (3.24)$$

Die vorgestellte Menge von Strömen S erfüllt nun mehrere wichtige Eigenschaften (vergleiche dazu auch [Lee97, NLG99]), die als Grundlage der Definition eines dazu passenden Komponentenmodells (siehe Kapitel 4) dienen:

1. Auf der Strommenge S ist eine Präfixordnung definiert (\bullet repräsentiert Konkatination)

$$\forall x, y \in S : x \sqsubseteq y \Leftrightarrow \exists z \in S : x \bullet z = y . \quad (3.25)$$

Die Strommenge S bildet zusammen mit der Präfixordnung \sqsubseteq eine PARTIELL GEORDNETE MENGE (POSET) (S, \sqsubseteq) .

2. Als geordnete Menge besitzt die Strommenge S ein BOTTOM-ELEMENT \perp , falls $\perp \sqsubseteq s$ für alle möglichen Ströme $s \in S$. Der leere Strom ε_S erfüllt diese Eigenschaft.
3. Eine Teilmenge \widehat{S} der Strommenge S ist eine KETTE (chain), falls für je zwei ihrer Elemente gilt:

$$\forall x, y \in \widehat{S} : x \sqsubseteq y \vee y \sqsubseteq x . \quad (3.26)$$

4. Eine OBERE SCHRANKE einer Teilmenge \widehat{S} der Strommenge S ist ein Element u von S , wobei jedes Element von \widehat{S} ein Präfix von u ist.
5. Ein SUPREMUM ist eine obere Schranke, welche selber Präfix jeder anderen oberen Schranke ist.
6. Der Poset (S, \sqsubseteq) bildet eine VOLLSTÄNDIGE PARTIELLE ORDNUNG (CPO), da jede Kette in S ein Supremum in S besitzt. Damit ist auch das Kreuzprodukt von n Strömen S^n eine CPO [DP02].

3.3.4 Probleme und Lösungen

Aufgrund der Unterteilung von physikalischen Signalen in Signalsegmente und Signalblöcke treten zwei Probleme in Erscheinung:

- **TEILWEISE FÜLLUNG DES LETZTEN SIGNALBLOCKES:** Da sich die Länge eines Signalsegmentes erst zur Laufzeit ermitteln läßt, kann nicht immer garantiert werden, daß der letzte Signalblock eines Signalsegmentes vollständig mit Signalwerten gefüllt ist. Zur Lösung dieses Problems wird dem Benutzer vorgeschlagen, daß die entsprechende Quellkomponente entweder den letzten Signalblock mit Dummywerten auffüllt beziehungsweise diesen Signalblock verwirft.
- **BESTIMMUNG DER SIGNALSEGMENTE:** Auf welche Weise die Unterscheidung zwischen relevanten und irrelevanten Signalen, wodurch die Signalsegmente festgelegt werden, erfolgt, ist anwendungsabhängig. Diese Unterscheidung kann sowohl mittels Hardware als auch durch die Software erfolgen. Beispiele für Hardwarelösungen sind:
 - Verwendung von Lichtschranken, die beispielsweise die Sensorerfassung von Objekten auf einem Fließband triggern.
 - Zeitliche Limitierung der Datenerfassung durch die Sensoren wie beispielsweise durch das Umschalten von Senden auf Empfang bei Radaranlagen.

Softwarelösungen sind in den Quellkomponenten eines Datenflußgraphen implementiert. Eine Trennung von relevanten und irrelevanten Daten kann auch hier in unterschiedlichster Weise erfolgen:

- Bei Unter- beziehungsweise Überschreiten einfacher Schwellwerte werden Signalen weitergeleitet beziehungsweise verworfen.
- Um den Einfluß von Störungen der physikalischen Signale zu verringern, werden auch kompliziertere Verfahren eingesetzt. Diese können beispielsweise die Schalthysterese eines Schmitt-Triggers nutzen.
- In verschiedenen Anwendungen sind auch zeitliche Schranken denkbar. So könnte beispielsweise die oben geschilderte Übernahme von Radarsignalen auch mittels Software gesteuert werden.

Die geeignete Lösung zur Bestimmung der Signalsegmente stellt dabei einen zentralen Aspekt bei der Kopplung des datenflußorientierten eingebetteten Systems an den jeweiligen technischen Prozeß dar.

3.4 Innovative Aspekte

Die innovativen Aspekte des vorgestellten Modells physikalischer Signale lassen sich wie folgt zusammenfassen:

- **Unterstützung bei der Fehlererkennung:** Zur Unterstützung der entwurfsbegleitenden Fehlererkennung wurden sechs Signalmerkmale eingeführt, die den Definitionsbereich, den Wertebereich und Aspekte des Datentransports charakterisieren. Dies ermöglicht:

1. **DEFINITION VON TYPCONSTRAINTS:** Es lassen sich nun Typconstraints, welche zum Beispiel die Wohldefiniiertheit einer Komponentenoperation sicherstellen, definieren und mittels eines geeigneten Typsystems überprüfen (siehe Kapitel 5). Datenflußkomponenten können Algorithmen beinhalten, die komplexe Typconstraints für die an ihren Ein- und Ausgängen anliegenden physikalischen Signale zur Folge haben, wie die Fast-Fourier-Transformation zeigt [Goo97].
 2. **DEFINITION VON KOMMUNIKATIONS PROTOKOLLEN:** Neben der Überprüfung statischer Constraints lassen sich auch dynamische Constraints, welche das Kommunikationsverhalten einer Datenflußkomponente in Form eines Kommunikationsprotokolls beschreiben, definieren. Mittels Model Checking kann dann die Kompatibilität dieser Kommunikationsprotokolle geprüft werden (siehe Kapitel 6).
- **Flexibilität in der Darstellung:** Die gewünschte Flexibilität in der Darstellung von physikalischen Signalen zeigt sich in den zwei folgenden Punkten:
 1. **ZEITSIGNALE MIT UNTERSCHIEDLICHEN ABTASTUNGEN:** Dies erreicht man durch die Unterteilung der Signale in Signalsegmente. Somit kann man sowohl unendliche Signale mit äquidistanter Abtastung als auch unendliche Signale bestehend aus Signalsegmenten mit äquidistanter Abtastung und auch Signale mit nichtäquidistanter Abtastung darstellen.
 2. **TRANSFORMATIONEN IN ANDERE DEFINITIONSBEREICHE:** Die Ergebnisse aufeinanderfolgender Transformationen eines Signals aus dem Zeitbereich in einen anderen Definitionsbereich können auch als aufeinanderfolgende Signalwerte eines Zeitbereichssignals aufgefaßt werden. So kann man beispielsweise Signalwertfolgen fester Länge aus dem Zeitbereich in den Frequenzbereich transferieren. Das Ergebnis ist eine Folge komplexwertiger Vektoren, die den Startzeiten der ursprünglichen Signalwertfolgen zugeordnet werden (siehe Abschnitt 5.2).
 - **Verminderung des Transport-Overheads:** Für die Reduzierung des Transport-Overheads wurden folgende Maßnahmen vorgeschlagen:
 1. **UNTERSCHIEDUNG RELEVANTER UND IRRELEVANTER SIGNALDATEN:** Als erstes werden nur relevante Signaldaten übertragen. Dies führt neben einer Verminderung des Transport- und Rechenaufwandes auch zu einer Vermeidung von fehlerhaften Ergebnissen (vergleiche Abbildung 3.3 (c)).
 2. **ÜBERTRAGUNG VON DATENMENGEN MIT GERINGEM TRANSPORT-OVERHEAD:** Um sowohl lange Wartezeiten bei der Übertragung ganzer Signalsegmente als auch den Transport-Overhead bei der Übertragung von Einzelwerten zu vermeiden, werden Blöcke aus Signaldaten in Token (Datencontainer) gekapselt und übertragen.
 3. **REKONSTRUKTION DES ZEITBEREICHS VON SIGNALSEGMENTEN AUS STARTZEITPUNKT UND ABTASTPERIODE:** Eine weitere Vermeidung unnötigen Datenaufkommens bewirkt die Rekonstruktion des Zeitbereichs eines Signalsegmentes aus

Startzeitpunkt und Abtastperiode. Dazu werden pro Signalsegment nur einmal Startzeitpunkt und Abtastperiode übertragen.

In den folgenden Kapiteln wird die Überprüfung der Kompatibilität von Datenflußkomponenten mit Hilfe eines neuartigen Interface-Typs (siehe Kapitel 5) beziehungsweise mittels eines auf diese spezielle Problemstellung zugeschnittenen neuen Model-Checking-Verfahrens (vergleiche Kapitel 6) vorgestellt.

Kapitel 4

Komponentenmodell

Wer hohe Türme bauen will, muß lange beim Fundament verweilen.

Im ersten Abschnitt dieses Kapitels sind die grundlegenden Probleme gebräuchlicher Komponentenmodelle dargestellt. Aus diesen Problemen werden die Anforderungen an das neu zu erstellende Modell von Datenflußkomponenten abgeleitet. Daran schließt sich die Vorstellung des neuen Komponentenmodells an. Nach einer schrittweisen Einführung dieses neuen Komponentenmodells werden neue gefärbte Datenflußparadigmen definiert. Aufbauend auf die formale Beschreibung dieses Modells rundet eine abschließende Zusammenfassung der innovativen Aspekte dieses Kapitel ab. Beispiele für diese neuen Datenflußkomponenten finden sich in [MSG02, MSG04].

4.1 Probleme mit gebräuchlichen Komponentenmodellen

In Abbildung 4.1 wird die Abarbeitung eines aus drei Datenflußkomponenten bestehenden Datenflußgraphen veranschaulicht. Eine **Quelle** liest ein Signal von einem Sensor ein und liefert dieses sowohl an eine **Duplikator-Komponente** als auch eine **Senke**, die zur Visualisierung verwendet wird, weiter. Physikalische Signale werden als Ströme gefärbter Token repräsentiert (vergleiche Kapitel 3.3.1.4). Die an einem Interface zulässigen Farben sind als Mengen in der Abbildung angegeben. Der Ablauf sieht im einzelnen wie folgt aus:

1. Als erstes wird die **Quelle** rechenbereit (vergleiche Abbildung 4.1 (b)).
2. Sobald die **Quelle** rechnend wird, liest sie das erste Token eines Signalsegments ein und leitet dieses an ihre Nachfolgerkomponenten weiter (vergleiche Abbildung 4.1 (c)).
3. Jetzt ist die **Duplikator-Komponente** rechenbereit (vergleiche Abbildung 4.1 (d)).
4. Die **Duplikator-Komponente** verdoppelt die am Eingang anliegenden Token. Damit die Struktur eines Signalsegments erhalten bleibt, erhält das zweite Token die Farbe m (vergleiche Abbildung 4.1 (e)).

5. Die Datenflußkomponente **Senke** wird rechenbereit (vergleiche Abbildung 4.1 (f)).
6. Die Datenflußkomponente **Senke** konsumiert je ein Token von ihren beiden Eingangsfifos (vergleiche Abbildung 4.1 (g)).
7. Die **Quelle** wird wieder rechenbereit (vergleiche Abbildung 4.1 (h)).
8. Es wird ein Endtoken **e** eingelesen. Damit tritt bei der **Senke** im nachfolgenden Schritt (der nicht mehr dargestellt ist) ein Problem auf, da jetzt **TOKEN UNTERSCHIEDLICHER FÄRBUNG MITEINANDER VERRECHNET** würden (vergleiche Abbildung 4.1 (i)).

In traditionellen Komponentenmodellen würde dieses Problem, das letztendlich ein Verrechnen nichtzusammenpassender Signalsegmente nach sich ziehen würde, unbemerkt bleiben. Eine Anforderung an das neue Modell ist die **FRÜHZEITIGE ENTWURFSBEGLEITENDE ERKENNUNG** solcher Probleme. Eine weitere Forderung ist, daß dem Benutzer verschiedene Möglichkeiten, das Problem zu beheben, angeboten werden. Dies kann beispielsweise mittels **ADAPTERKOMponenten** geschehen. Adapterkomponenten passen zum Beispiel die Länge von Signalsegmenten durch Einfügen von Dummywerten beziehungsweise durch Weglassen überzähliger Signalwerte an.

Ein zusätzliches Problem resultiert aus **IMPLIZIT EINGEFÜHRTEM NICHTDETERMINISMUS**. In Abbildung 4.2 (a) ist das Modell einer Komponentenschnittstelle dargestellt. Dieses Modell basiert auf den klassischen Datenflußparadigmen, die in Abschnitt 2.2.2 vorgestellt wurden. Diese Schnittstelle wiederholt früher empfangene Token solange, bis ein neues Token ankommt. Dies resultiert zum einen aus dem Wunsch, **DATEN NUR BEI ÄNDERUNGEN ZU ÜBERTRAGEN**, zum Beispiel wenn sich die Temperatur eines Meßobjekts nur langsam ändert. Denn obwohl die Daten nur bei Änderungen übertragen werden, sorgt die Schnittstelle dafür, daß die als letztes gesendeten Daten wiederholt werden, so daß die nachfolgende SDF-Komponente rechenbereit werden kann (vergleiche Abschnitt 2.2.3). Ein anderer Verwendungszweck liegt in der **AUFLÖSUNG VON DEADLOCKS BASIEREND AUF ZYKLEN**. Da auf der Rückkopplungskante keine Daten vorhanden sind, die Regeln zur Bestimmung der Rechenbereitschaft einer SDF-Komponente aber das Vorhandensein von Daten fordern, ermöglicht es die vorgestellte Schnittstelle, einen Defaultwert zu verwenden und somit die SDF-Komponente rechenbereit werden zu lassen. Das letztgenannte Problem ließe sich aber auch durch Initialisierungstoken ohne Verwendung nichtdeterministischer Datenflußkomponenten (vergleiche Abschnitt 2.2.3) vermeiden. Betrachtet man das dargestellte Modell von Abbildung 4.2 (a), so stellt sich dessen Verhalten wie folgt dar:

- Kommt ein Datentoken von außerhalb an, so wird dieses von der klassischen nichtdeterministischen **Merge**-Komponente an den SDF-Speicher **B** weitergeleitet.
- **B** speichert aufgrund der Rückkopplungskante das Datentoken und sendet es zudem an die Datenflußkomponente **Switch**. Über die untere Kante sendet **B** gleichzeitig ein Steuertoken an die Datenflußkomponente **Switch**. Dieses Steuertoken bewirkt, daß das Datentoken von der Datensenke **C** konsumiert wird.
- Ausschließlich die Quellkomponente **A** veranlaßt durch Senden eines passenden Steuertokens die Weitergabe der in **B** gespeicherten Daten.

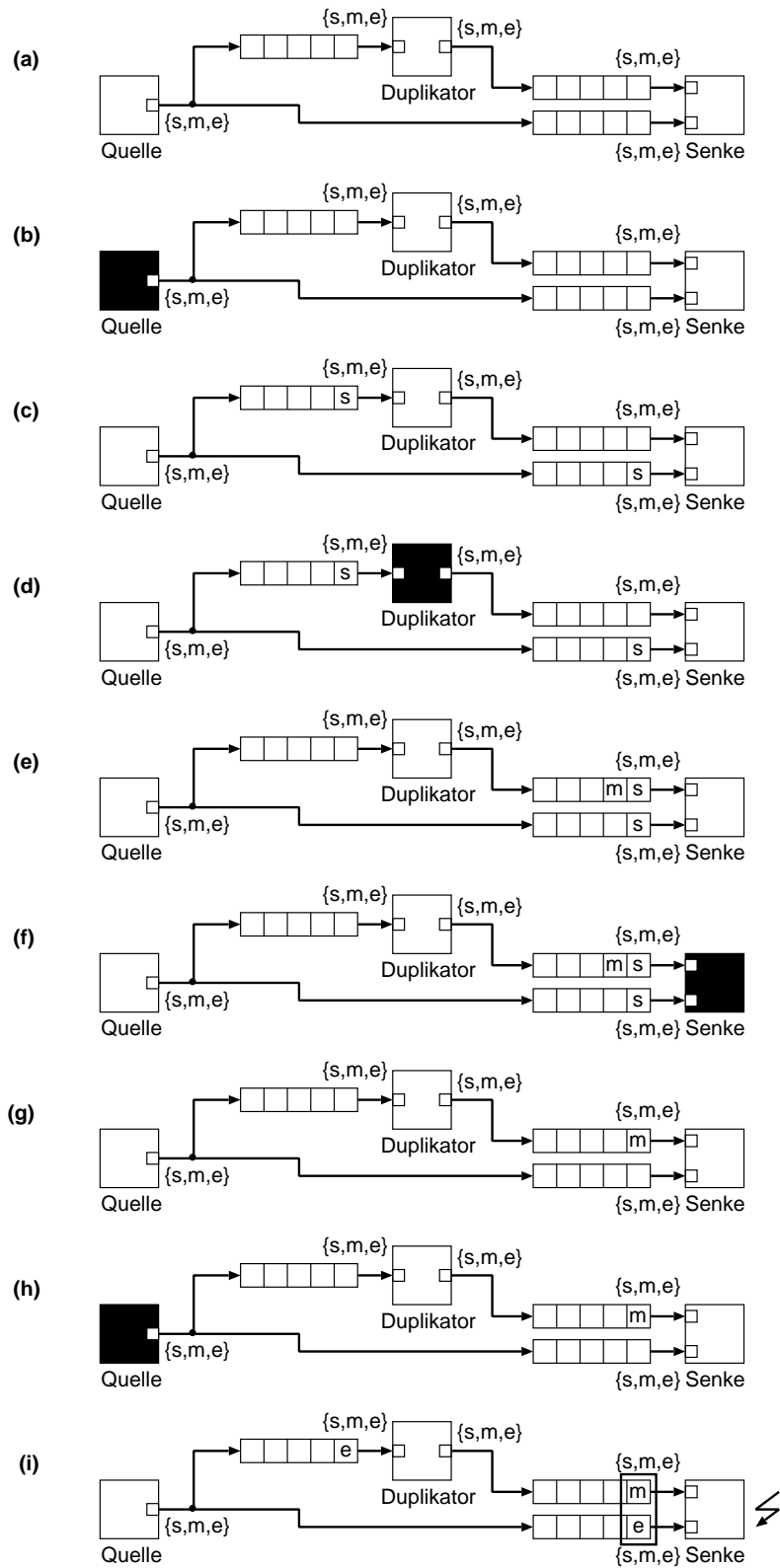


Abbildung 4.1: Probleme bei der Kombination von traditionellem Komponentenmodell mit neuem Signalmodell

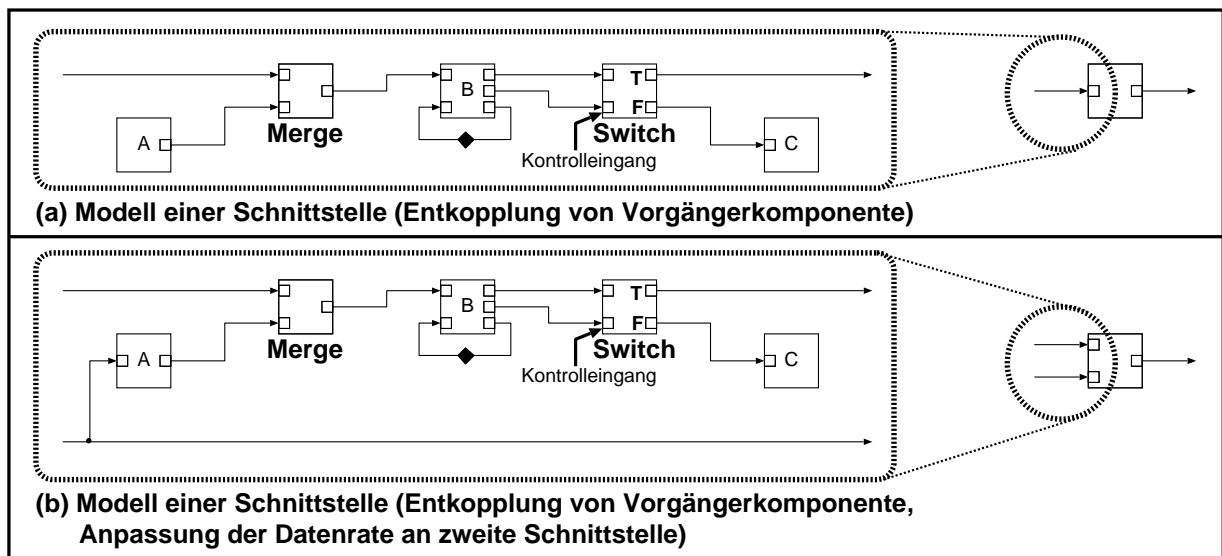


Abbildung 4.2: Nichtdeterminismus in Datenflußgraphen (klassisches Komponentenmodell)

Betrachtet man zur weiteren Verdeutlichung Abbildung 4.3, so erkennt man, daß Datenflußkomponente D aufgrund des vorhandenen Nichtdeterminismus nicht notwendigerweise vor Datenflußkomponente E rechnend werden muß, damit E rechenbereit ist. Die Rechenbereitschaft der Datenflußkomponente E ist völlig unabhängig von D. Zudem wählt die nichtdeterministische Merge-Komponente, wenn an ihren beiden Dateneingängen Token anliegen, zufällig aus, welches davon als erstes an den Ausgang weitergeleitet wird. Damit hat man ein sehr SCHWERWIEGENDES PROBLEM in den Datenflußgraphen eingeführt:

1. Die Kante zur Vorgängerkomponente, die durch das Modell in Abbildung 4.2 beschrieben ist, verhält sich wie eine (VERSTECKTE) QUELLE, was vom Scheduler entsprechend berücksichtigt werden muß.
2. Der IMPLIZITE NICHTDETERMINISMUS, der durch die Merge-Komponente (siehe Abschnitt 2.2.5) bedingt ist, hat zur Folge, daß bei gleichzeitiger Ankunft eines neuen Tokens von Datenflußkomponente D und der Aktivierung der Quellkomponente A nicht klar ist, welche Daten weitergeleitet werden.
3. Damit erhält man bei unterschiedlicher Reihenfolge der Ausführung der Datenflußkomponenten nicht mehr für dieselben Eingabedaten dieselben Ausgabedaten. Es kann sogar zu SPEICHERÜBERLAUF kommen, da sich auf einer Kante Daten aufsammeln, während an einer nachfolgenden Stelle im Datenflußgraphen veraltete Signalwerte verwendet werden. Will man diesen Nichtdeterminismus und die daraus resultierenden Probleme unterdrücken, muß dies zum Beispiel durch PRIORITÄTEN geregelt werden. Dies ist in einem großen unübersichtlichen Datenflußgraphen eine häufige und schwer handhabbare Fehlerquelle (vergleiche Abbildung 1.6).

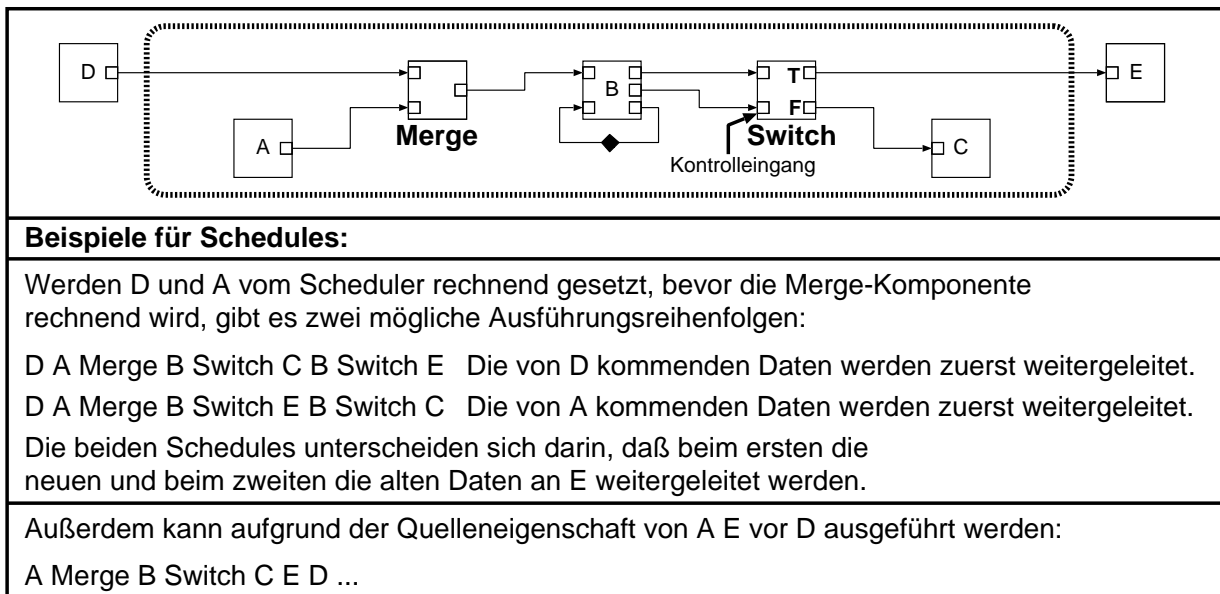


Abbildung 4.3: Nichtdeterminismus in Datenflußgraphen (Beispiele für Schedules)

Abbildung 4.2 (b) zeigt eine weitere Variante des gerade vorgestellten Problems. Hier wird die Weitergabe des gespeicherten Tokens durch die Ankunft eines Tokens in einem Kanal, welcher der gewohnten Fifo-Semantik gehorcht, getriggert. Damit wird zwar die ungewollte Quelleneigenschaft vermieden, die restlichen negativen Folgen bleiben jedoch bestehen.

4.2 Anforderungen

Aus den in Abschnitt 4.1 geschilderten Problemstellungen lassen sich folgende Anforderungen an das Komponentenmodell herleiten.

4.2.1 Kompatibilität mit dem neuen Signalmodell

Als erste Anforderung ist die Definition eines Komponentenmodells zu nennen, welches mit dem neuartigen Signalmodell in Kapitel 3 kompatibel ist. Dabei sollte insbesondere die Handhabung von Signalsegmenten, Signalblöcken und Signalwerten eine zentrale Rolle spielen.

4.2.2 Unterstützung bei der Fehlererkennung

Eine präzise Beschreibung des Komponentenverhaltens soll es ermöglichen, Regeln für die Wohldefiniertheit einer Datenflußkomponente abzuleiten.

- **TYPCONSTRAINTS:** Aus der formalen Beschreibung von Datenflußkomponenten sollen Typconstraints hergeleitet werden, die Regeln bezüglich der sechs vorgestellten Signal-

		Horizontale Aufteilung			
Vertikale Aufteilung	Tokenmaschine	Nicht turing-äquivalent	Turing-äquivalent	Turing-äquivalent	+ Nicht-deterministisch
	Gekapselter Algorithmus	In der Regel Turingäquivalent			

Abbildung 4.4: Horizontale und vertikale Berechnungskomplexität

merkmale (vergleiche Abschnitt 3.3.2) zwischen den einzelnen an den verschiedenen Komponentenschnittstellen anliegenden physikalischen Signalen definieren. Damit wird die WOHLDEFINIERTHEIT der von den Datenflußkomponenten gekapselten Operationen gewährleistet.

- **KOMMUNIKATIONSPROTOKOLLE:** Außerdem können Querbezüge in den erlaubten Tokenabfolgen an den diversen Komponentenschnittstellen aus der formalen Komponentenbeschreibung abgeleitet werden. Damit wird – abhängig von der Berechnungskomplexität – die **KOMPATIBILITÄT** der Datenflußkomponenten sichergestellt.

Diese Regeln werden dann von einem Interface-Typsystem (siehe Kapitel 5) und einem Model Checker (vergleiche Kapitel 6) auf ihre Einhaltung hin überprüft.

4.2.3 Kontrolle über Komplexität und Modellierungsmächtigkeit

Ein weiterer zentraler Punkt ist die Aufteilung von Datenflußkomponenten anhand ihrer Aufgabe. Dabei spielen die folgenden Aspekte eine wichtige Rolle (vergleiche Abbildung 4.4):

- **UNTERSCHIEDUNG ZWISCHEN TOKENMASCHINE UND GEKAPSELTEM ALGORITHMUS:** Alle Aspekte, die den Transport von Token (Datencontainern) durch den Datenflußgraphen betreffen, sind auf der Ebene der Tokenmaschine anzusiedeln. Davon klar zu unterscheiden ist der gekapselte Algorithmus, der auf den in einem Token enthaltenen Daten arbeitet. Gekapselte Algorithmen stammen in der Regel aus dem Bereich der Bild- und Signalverarbeitung wie zum Beispiel Kantenglättung, Filter, PID-Regler oder Regler basierend auf Neuronalen Netzen [May99, MS00]. Diese Unterscheidung wird auch **VERTIKALE AUFTEILUNG** genannt.
- **UNTERSCHIEDUNG ZWISCHEN TOKENMASCHINEN UNTERSCHIEDLICHER BERECHNUNGSKOMPLEXITÄT:** Die Tokenmaschinen lassen sich anhand ihrer Berechnungskomplexität in nichtturingäquivalente, turingäquivalente und turingäquivalente und gleichzeitig nichtdeterministische Tokenmaschinen unterteilen. Dies wird auch als **HORIZONTALE AUFTEILUNG** bezeichnet. Die Komplexität einer Tokenmaschine ergibt sich dabei aus den Regeln, anhand derer ihre Rechenbereitschaft ermittelt wird. Daraus resultieren zwei Teilanforderungen für das neue Komponentenmodell:

- **KAPSELUNG VON SIGNALVERARBEITUNGsalgorithmen in NICHTTURING-ÄQUIVALENTEN TOKENMASCHINEN:** Der Entwickler soll die Möglichkeit haben, alle Signalverarbeitungsalgorithmen zu nutzen, ohne daß die zugehörigen Tokenmaschinen turingäquivalent sind. Zusätzlich soll eine handhabbare Menge atomarer Datenflußkomponenten zur Verfügung gestellt werden, deren Tokenmaschinen eine höhere Berechnungskomplexität haben. Eine Aufgabe dieser komplexeren Datenflußkomponenten ist dann beispielsweise die Steuerung von Tokenströmen im Datenflußgraphen.
- **WEITGEHENDE VERMEIDUNG VON NICHTDETERMINISMUS:** Insbesondere ist die strenge Trennung von deterministischen und nichtdeterministischen Datenflußkomponenten einzufordern. Nichtdeterminismus (vergleiche Definition 2.2.2) ist zwar für die Modellbildung in einem schrittweisen Verfeinerungsprozeß [Bro99] interessant, im auszuführenden Datenflußgraphen aufgrund der gegebenenfalls nicht vorhandenen Reproduzierbarkeit von Ergebnissen aber fehl am Platze. Dem Entwickler wird allerdings mit einem klaren Hinweis auf die damit entstehenden Nachteile die Möglichkeit eingeräumt, diese Datenflußkomponenten zu verwenden. Aufgrund der gewollten Vermeidung von Nichtdeterminismus tritt der Fall nichtturingäquivalent und gleichzeitig nichtdeterministisch in der horizontalen Aufteilung der Datenflußkomponenten nach ihrer Berechnungskomplexität nicht auf.

4.3 Neues Modell

Dieser Abschnitt stellt ein neues Modell für Datenflußkomponenten vor. Auf eine schrittweise Einführung anhand eines einfachen Beispiels folgt die Erweiterung der klassischen Datenflußparadigmen SDF, BDF und DDF (vergleiche Abschnitt 2.2.2) zu gefärbten Datenflußparadigmen. Abgerundet wird dieser Abschnitt durch eine formale Beschreibung von Datenflußkomponenten.

4.3.1 Schrittweise Einführung

Im folgenden wird anhand des einfachen Beispiels der Addition digitalisierter physikalischer Signale das neue Komponentenmodell, das auf einer funktionalen Beschreibung basiert, eingeführt.

4.3.1.1 Funktion definiert auf Punkten

Auf der untersten Ebene ist die Addition eine Funktion, die jeweils zwei Punkte (siehe Gleichung 3.10) verknüpft. Dabei muß gelten, daß die Zeiten der verknüpften Signalwerte identisch sind. Ein weiteres Constraint fordert die Gleichheit der physikalischen Einheiten.

$$\begin{aligned} \text{add}_{\text{point}} &: P^2 \rightarrow P \cup \{\perp\} \\ \text{add}_{\text{point}}((t_1, (v_1, \mathbf{pu}_1)), (t_2, (v_2, \mathbf{pu}_2))) &= \begin{cases} (t_1, (v_1 + v_2, \mathbf{pu}_1)) & \text{falls } t_1 = t_2, \mathbf{pu}_1 = \mathbf{pu}_2 \\ \perp & \text{sonst} \end{cases} \quad (4.1) \end{aligned}$$

\perp markiert dabei den Fehlerfall. Im folgenden Zahlenbeispiel werden die beiden Punkte $(3, (7, A))$ und $(3, (11, A))$ addiert, wobei beide Signalwerte zum Zeitpunkt 3 gültig sind, die physikalische Einheit gleich Ampère ist und die Signalwerte durch 7 und 11 gegeben sind:

$$\text{add}_{\text{point}}((3, (7, A)), (3, (11, A))) = (3, (18, A)) \quad (4.2)$$

4.3.1.2 Funktion definiert auf Blöcken

Auf der Ebene von Signalblöcken (siehe Gleichung 3.12), die endlichen Folgen von Punkten entsprechen, ist die Addition nur dann wohldefiniert, wenn die beiden Eingabeblocke gleich lang sind und der sukzessive Aufruf von $\text{add}_{\text{point}}$ niemals den undefinierten Wert \perp zurückliefert.

$$\begin{aligned} \text{add}_{\text{block}} &: B^2 \rightarrow B \cup \{\perp\} \\ \text{add}_{\text{block}}(\varepsilon_B, \varepsilon_B) &= \varepsilon_B \\ \text{add}_{\text{block}}(p_1 \bullet b'_1, p_2 \bullet b'_2) &= \text{add}_{\text{point}}(p_1, p_2) \bullet \text{add}_{\text{block}}(b'_1, b'_2) \\ \text{andernfalls: } \text{add}_{\text{block}}(b_1, b_2) &= \perp \end{aligned} \quad (4.3)$$

Werden $\text{add}_{\text{block}}$ zwei leere Blöcke übergeben, so wird ein leerer Block ε_B zurückgeliefert. Wird $\text{add}_{\text{block}}$ mit zwei Signalblöcken aufgerufen, so werden sukzessive jeweils zwei Punkte mittels $\text{add}_{\text{point}}$ verknüpft. In allen anderen Fällen tritt ein Fehler auf, was durch \perp gekennzeichnet ist. Folgendes Zahlenbeispiel soll das Vorgehen weiter veranschaulichen:

$$\begin{aligned} \text{add}_{\text{block}} & \left([(0, (7, A)), (1, (9, A)), (2, (10, A)), (3, (7, A)), (4, (5, A))], \right. \\ & \left. [(0, (2, A)), (1, (4, A)), (2, (9, A)), (3, (11, A)), (4, (20, A))] \right) \\ &= \text{add}_{\text{point}}((0, (7, A)), (0, (2, A))) \bullet \dots \bullet \text{add}_{\text{point}}((4, (5, A)), (4, (20, A))) \bullet \varepsilon_B \\ &= [(0, (9, A)), (1, (13, A)), (2, (19, A)), (3, (18, A)), (4, (25, A))] \end{aligned} \quad (4.4)$$

4.3.1.3 Funktion definiert auf Token

Fügt man zu einem Signalblock eine farbige Markierung hinzu, so erhält man ein gefärbtes Token (siehe Gleichung 3.14). Die Funktion $\text{add}_{\text{token}}$ ist im Fall der Addition wohldefiniert, wenn die Farben der beiden Token übereinstimmen und die Funktion $\text{add}_{\text{block}}$ nicht den undefinierten Wert \perp zurückliefert.

$$\begin{aligned} \text{add}_{\text{token}} &: T^2 \rightarrow T \cup \{\perp\} \\ \text{add}_{\text{token}}(\varepsilon_T, \varepsilon_T) &= \varepsilon_T \\ \text{add}_{\text{token}}((c_1, b_1), (c_2, b_2)) &= \begin{cases} (c_1, \text{add}_{\text{block}}(b_1, b_2)) & \text{falls } c_1 = c_2 \\ \perp & \text{sonst} \end{cases} \\ \text{andernfalls: } \text{add}_{\text{token}}(x_1, x_2) &= \perp \end{aligned} \quad (4.5)$$

In Fortsetzung des obigen Zahlenbeispiels erhält man:

$$\begin{aligned} \text{add}_{\text{token}} & \left((m, [(0, (7, A)), (1, (9, A)), (2, (10, A)), (3, (7, A)), (4, (5, A))]) \right. \\ & \left. (m, [(0, (2, A)), (1, (4, A)), (2, (9, A)), (3, (11, A)), (4, (20, A))]) \right) \\ &= (m, \text{add}_{\text{block}}([(0, (7, A)), (1, (9, A)), (2, (10, A)), (3, (7, A)), (4, (5, A))], \\ & \quad [(0, (2, A)), (1, (4, A)), (2, (9, A)), (3, (11, A)), (4, (20, A))])) \\ &= (m, [(0, (9, A)), (1, (13, A)), (2, (19, A)), (3, (18, A)), (4, (25, A))]) \end{aligned} \quad (4.6)$$

4.3.1.4 Funktion definiert auf Strömen

Ein physikalisches Signal besteht aus einer Folge von Signalsegmenten. Diese Signalsegmente werden mit Hilfe von farbigen Blockmarkierungen gekennzeichnet. Dies führt letztendlich zur Darstellung eines physikalischen Signals als Strom gefärbter Token (siehe Gleichung 3.21). Die Funktion $\text{add}_{\text{stream}}$ bildet demzufolge im Fall der Addition zwei unendliche Ströme gefärbter Token auf einen unendlichen Strom gefärbter Token ab:

$$\begin{aligned} \text{add}_{\text{stream}} : S^2 &\rightarrow S \cup \{\perp\} \\ \text{add}_{\text{stream}}(\varepsilon_S, \varepsilon_S) &= \varepsilon_S \\ \text{add}_{\text{stream}}(x_1 \bullet r'_1, x_2 \bullet r'_2) &= \text{add}_{\text{token}}(x_1, x_2) \bullet \text{add}_{\text{stream}}(r'_1, r'_2) \\ \text{andernfalls: } \text{add}_{\text{stream}}(r_1, r_2) &= \perp \end{aligned} \quad (4.7)$$

Setzt man das oben gegebene Zahlenbeispiel fort, so ergibt sich:

$$\begin{aligned} \text{add}_{\text{stream}} & \left((\mathbf{m}, [(0, (7, A)), (1, (9, A)), (2, (10, A)), (3, (7, A)), (4, (5, A))]) \bullet r'_1, \right. \\ & \left. (\mathbf{m}, [(0, (2, A)), (1, (4, A)), (2, (9, A)), (3, (11, A)), (4, (20, A))]) \bullet r'_2 \right) \\ &= \text{add}_{\text{token}} \left((\mathbf{m}, [(0, (7, A)), (1, (9, A)), (2, (10, A)), (3, (7, A)), (4, (5, A))]), \right. \\ & \quad \left. (\mathbf{m}, [(0, (2, A)), (1, (4, A)), (2, (9, A)), (3, (11, A)), (4, (20, A))]) \right) \quad (4.8) \\ & \bullet \text{add}_{\text{stream}}(r'_1, r'_2) \\ &= (\mathbf{m}, [(0, (9, A)), (1, (13, A)), (2, (19, A)), (3, (18, A)), (4, (25, A))]) \\ & \bullet \text{add}_{\text{stream}}(r'_1, r'_2) \end{aligned}$$

4.3.2 Klassifikation von Datenflußkomponenten

Neben der Klassifikation der Datenflußkomponenten anhand der Berechnungskomplexität ihrer Tokenmaschinen kann man Datenflußkomponenten auch anhand ihrer Aufgaben im Datenflußgraphen gruppieren: SIGNALVERARBEITUNGSKOMPONENTEN, ADAPTERKOMPONENTEN und KONTROLLFLUSSKOMPONENTEN.

- SIGNALVERARBEITUNGSKOMPONENTEN kapseln Algorithmen aus dem Bereich der Bild- und Signalverarbeitung. Dabei kann man wiederum unterscheiden zwischen
 - QUELLEN: Gerätetreiber für Einsteckkarten von Sensoren wie Soundkarten, Framegrabber und Analog/Digital-Wandler.
 - TRANSFORMERN: Beispielkomponenten beinhalten das Extrahieren von Kanten aus Bildern, das Glätten von Bildern, Filteralgorithmen, Regelalgorithmen und Fast-Fourier-Transformation.
 - SENKEN: Gerätetreiber für Aktoren wie Schrittmotorleistungskarten und Monitoransteuerungen.
- ADAPTERKOMPONENTEN sind für Konvertierungsoperationen im weiteren Sinne zuständig. Beispiele sind das Aufsammeln von allen Token eines Signalsegments, klassische Typkonvertierung und Resampling.

Tokenmaschine	Komponentenklassifizierung		
	Signalverarbeitung	Adapter	Kontrollfluß
Colored SDF (Nicht turing- äquivalent)	Fast-Fourier-Transformation Filter-Operationen Fuzzy-Klassifizierer PID-Regler Bildverarbeitungsoperationen Matrixoperationen Statistikgenerierung Visualisierung Digitale Ein/Ausgabe	Upsample Downsample Interpolation Extrapolation Typkonvertierung	–
Colored BDF (Turing- äquivalent)	–	Collector Splitter Cutter Filler	CSwitch CSelect
Colored DDF (Turingäqui- valent, Nicht- deterministisch)	–	Decoupler (1) Decoupler (2)	CMerge

Tabelle 4.1: Klassifikation von Datenflußkomponenten

- **KONTROLLFLUSSKOMPONENTEN** steuern den Fluß der Token durch den Datenflußgraphen. **Switch**, **Select** und **Merge** sind dafür Beispiele. Mit Hilfe dieser Datenflußkomponenten können komplexere Konstrukte wie **if-then-else**, **for-next**- und **while**-Schleifen konstruiert werden.

Tabelle 4.1 veranschaulicht die beiden Arten der Klassifikation und deren Beziehung untereinander.

4.3.3 Gefärbte Datenflußparadigmen

Dieser Abschnitt erweitert die klassischen Datenflußparadigmen SDF, BDF und DDF (siehe Abschnitt 2.2) zu den gefärbten Datenflußparadigmen Colored SDF, Colored BDF und Colored DDF. Aufbauend auf das in Abschnitt 4.3.2 vorgestellte Klassifikationsschema wird sichergestellt, daß der Anwendungsprogrammierer eine ausreichende Kontrolle über die Komplexität der Tokenmaschinen der von ihm verwendeten Datenflußkomponenten besitzt.

4.3.3.1 Colored SDF

Werden die einfachen Schaltregeln einer SDF-Komponente (siehe Abschnitt 2.2.3) derart erweitert, daß es physikalische Signale gemäß dem neuen Signalmodell verarbeiten kann, erhält man

folgende formale Beschreibung¹.

$$\mathbf{csdf} \begin{cases} S^m \rightarrow S^n \cup \{\perp\} & (m, n \in \mathbb{N}) \\ (\varepsilon_S, \dots, \varepsilon_S) \mapsto (\varepsilon_S, \dots, \varepsilon_S) \\ x \bullet r \mapsto f_{\text{token}}(x) \bullet \mathbf{csdf}(r) \\ r \mapsto \perp & \text{sonst} \end{cases} \quad (4.9)$$

$$f_{\text{token}} \begin{cases} T^m \rightarrow T^n \cup \{\perp\} \\ (\varepsilon_T, \dots, \varepsilon_T) \mapsto (\varepsilon_T, \dots, \varepsilon_T) \\ ((c_1, b_1), \dots, (c_m, b_m)) \mapsto \\ ((f_{\text{color}_1}(c_1, \dots, c_m), f_{\text{block}_1}(b_1, \dots, b_m)), \dots, \\ (f_{\text{color}_n}(c_1, \dots, c_m), f_{\text{block}_n}(b_1, \dots, b_m))) \\ (t_1, \dots, t_m) \mapsto \perp & \text{sonst} \end{cases} \quad (4.10)$$

$$f_{\text{block}_i} \begin{cases} B^m \rightarrow B \cup \{\perp\} & (i \in [1, \dots, n]) \\ (\varepsilon_B, \dots, \varepsilon_B) \mapsto \varepsilon_B \\ (p_1 \bullet b'_1, \dots, p_m \bullet b'_m) \mapsto f_{\text{point}_i}(p_1, \dots, p_m) \bullet f_{\text{block}}(b'_1, \dots, b'_m) \\ (b_1, \dots, b_m) \mapsto \perp & \text{sonst} \end{cases} \quad (4.11)$$

$$f_{\text{point}_i} \begin{cases} P^m \rightarrow P \cup \{\perp\} & (i \in [1, \dots, n]) \\ (p_1, \dots, p_m) \mapsto f_{\text{point}_i}(p_1, \dots, p_m) \end{cases} \quad (4.12)$$

Dabei wird durch f_{stream} aus einem m -stelligen Strom von Token ein n -stelliger Strom erzeugt. Dies geschieht mit Hilfe von der auf Tupeln von Token definierten Funktion f_{token} . Die Funktion f_{token} basiert selber auf den Funktionen f_{color_i} , welche die farbigen Markierungen der Ausgabefolgen aus den Farben der Eingabetoken ermitteln und den Funktionen f_{block_i} , mit deren Hilfe aus den Signalblöcken der Eingabetoken die Signalblöcke der Ausgabefolgen bestimmt werden. f_{block_i} wiederum wendet zur Berechnung der Ausgabepunkte sukzessive die Funktion f_{point_i} auf die Tupel der Eingabepunkte an. Wird m beziehungsweise n gleich 0 gesetzt, so erhält man das Modell einer QUELLEN- beziehungsweise SENKENKOMPONENTE. Der Fall $m = n = 0$ ist sinnlos und wird nicht weiter betrachtet. Die übrigen Datenflußkomponenten werden auch als TRANSFORMER bezeichnet.

Die Funktion f_{point_i} wird bei der punktweisen Verknüpfung von Signalen eingesetzt und repräsentiert alle unären, binären, ternären (und so weiter) Operationen auf Bild- und Signaldaten (vergleiche Abschnitt 5.2). Eine weitere wichtige Art von Funktionen auf Signalblöcken stellen blockweise Extraktionen dar. So wird beispielsweise bei der Fast-Fourier-Transformation jeweils einem Block von Signaldaten ein komplexwertiger Vektor zugeordnet. Solche Funktionen wer-

¹Die Bezeichnung von Datenflußkomponenten ist wie in folgendem Beispiel veranschaulicht strukturiert:

- **cswitch** bezeichnet die Tokenmaschine.
- **CSwitch** bezeichnet die gesamte Datenflußkomponente bestehend aus Tokenmaschine und gekapseltem Algorithmus.
- **Switch** bezeichnet entweder die dem klassischen Datenflußparadigma BDF entnommene Datenflußkomponente oder – falls aus dem Kontext ersichtlich – die entsprechende gefärbte Datenflußkomponente **CSwitch**.

den wie folgt beschrieben:

$$\hat{f}_{\text{point}} \begin{cases} B^m \rightarrow P \cup \{\perp\} \\ (b_1, \dots, b_m) \mapsto \hat{f}_{\text{point}}(b_1, \dots, b_m) \end{cases} \quad (4.13)$$

4.3.3.2 Colored BDF

Fügt man zu Colored SDF die erweiterten Datenflußkomponenten Colored Switch (**cswitch**, **CSwitch**) und Colored Select (**cselect**, **CSelect**) hinzu, so erhält man das gefärbte Datenflußparadigma Colored BDF. Dabei gibt es sowohl Kontrollflußkomponenten und mit deren Hilfe konstruierte Kontrollflußstrukturen als auch Adapterkomponenten, welche diesem Datenflußparadigma angehören.

Kontrollflußkomponenten: Die beiden Datenflußkomponenten **cswitch** und **cselect** reichen aus, um alle deterministischen Kontrollflußstrukturen wie zum Beispiel **while**-Schleifen (siehe unten) konstruieren zu können.

- **COLORED SWITCH** konsumiert ein Signalsegment, welches einen einzigen Booleschen Wert enthält, von seinem Steuereingang. Abhängig von diesem Steuerwert werden alle Token eines Signalsegmentes vom Dateneingang zu dem entsprechenden Datenausgang durchgeschaltet. Dabei wird beispielsweise die Funktion CSW_{true} dazu benutzt, alle Token eines Signalsegmentes zum mit **T** (**true**) markierten Ausgang zu übertragen. In analoger Weise schaltet $\text{CSW}_{\text{false}}$ alle Token eines Signalsegmentes zum Ausgang **F** (**false**) durch. Durch diese einheitliche Behandlung aller Token eines Signalsegmentes wird die Struktur eines physikalischen Signals bewahrt.

$$\text{cswitch} \begin{cases} S^2 \rightarrow S^2 \cup \{\perp\} \\ (\varepsilon_S, \varepsilon_S) \mapsto (\varepsilon_S, \varepsilon_S) \\ ((\mathbf{o}, [(t, \text{true})]) \bullet r_1, r_2) \mapsto \text{CSW}_{\text{true}}(r_1, r_2) \\ ((\mathbf{o}, [(t, \text{false})]) \bullet r_1, r_2) \mapsto \text{CSW}_{\text{false}}(r_1, r_2) \\ (r_1, r_2) \mapsto \perp \quad \text{sonst} \end{cases} \quad (4.14)$$

$$\text{CSW}_{\text{true}} \begin{cases} S^2 \rightarrow S^2 \cup \{\perp\} \\ (r_1, (\mathbf{s}, b) \bullet r_2) \mapsto ((\mathbf{s}, b), \varepsilon_S) \bullet \text{CSW}_{\text{true}}(r_1, r_2) \\ (r_1, (\mathbf{m}, b) \bullet r_2) \mapsto ((\mathbf{m}, b), \varepsilon_S) \bullet \text{CSW}_{\text{true}}(r_1, r_2) \\ (r_1, (\mathbf{e}, b) \bullet r_2) \mapsto ((\mathbf{e}, b), \varepsilon_S) \bullet \text{cswitch}(r_1, r_2) \\ (r_1, (\mathbf{o}, b) \bullet r_2) \mapsto ((\mathbf{o}, b), \varepsilon_S) \bullet \text{cswitch}(r_1, r_2) \\ (r_1, r_2) \mapsto \perp \quad \text{sonst} \end{cases} \quad (4.15)$$

$$\text{CSW}_{\text{false}} \begin{cases} S^2 \rightarrow S^2 \cup \{\perp\} \\ (r_1, (\mathbf{s}, b) \bullet r_2) \mapsto (\varepsilon_S, (\mathbf{s}, b)) \bullet \text{CSW}_{\text{false}}(r_1, r_2) \\ (r_1, (\mathbf{m}, b) \bullet r_2) \mapsto (\varepsilon_S, (\mathbf{m}, b)) \bullet \text{CSW}_{\text{false}}(r_1, r_2) \\ (r_1, (\mathbf{e}, b) \bullet r_2) \mapsto (\varepsilon_S, (\mathbf{e}, b)) \bullet \text{cswitch}(r_1, r_2) \\ (r_1, (\mathbf{o}, b) \bullet r_2) \mapsto (\varepsilon_S, (\mathbf{o}, b)) \bullet \text{cswitch}(r_1, r_2) \\ (r_1, r_2) \mapsto \perp \quad \text{sonst} \end{cases} \quad (4.16)$$

In folgendem Zahlenbeispiel wird die Funktion von **cswitch** veranschaulicht, indem mittels zweier Steuertoken je ein Signalsegment an beide Ausgänge transferiert wird.

$$\begin{aligned}
& \mathbf{cswitch}([(o, [(0, \text{true})]), (o, [(1, \text{false})])], [(s, b), (m, b'), (m, b''), (e, b'''), (o, b''')]) \\
&= \mathbf{csw}_{\text{true}}([(o, [(1, \text{false})])], [(s, b), (m, b'), (m, b''), (e, b'''), (o, b''')]) \\
&= ([[(s, b)], \varepsilon_S] \bullet \mathbf{csw}_{\text{true}}([(o, [(1, \text{false})])], [(m, b'), (m, b''), (e, b'''), (o, b''')]) \\
&= ([[(s, b), (m, b')], \varepsilon_S] \bullet \mathbf{csw}_{\text{true}}([(o, [(1, \text{false})])], [(m, b''), (e, b'''), (o, b''')]) \\
&= ([[(s, b), (m, b'), (m, b'')], \varepsilon_S] \bullet \mathbf{csw}_{\text{true}}([(o, [(1, \text{false})])], [(e, b'''), (o, b''')]) \\
&= ([[(s, b), (m, b'), (m, b''), (e, b''')], \varepsilon_S] \bullet \mathbf{csw}_{\text{true}}([(o, [(1, \text{false})])], [(o, b''')]) \\
&= ([[(s, b), (m, b'), (m, b''), (e, b''')], \varepsilon_S] \bullet \mathbf{cswitch}([(o, [(1, \text{false})])], [(o, b''')]) \\
&= ([[(s, b), (m, b'), (m, b''), (e, b''')], \varepsilon_S] \bullet \mathbf{csw}_{\text{false}}(\varepsilon_S, [(o, b''')]) \\
&= ([[(s, b), (m, b'), (m, b''), (e, b''')], [(o, b''')]) \bullet \mathbf{cswitch}(\varepsilon_S, \varepsilon_S) \\
&= ([[(s, b), (m, b'), (m, b''), (e, b''')], [(o, b''')])
\end{aligned} \tag{4.17}$$

- **COLORED SELECT** überträgt alle Token eines Signalsegmentes von dem durch einen Steuerwert des einwertigen Signalsegments ausgewählten Dateneingang zum Datenausgang. Die Hilfsfunktionen $\mathbf{cse}_{\text{true}}$ und $\mathbf{cse}_{\text{false}}$ übernehmen analog zu $\mathbf{csw}_{\text{true}}$ und $\mathbf{csw}_{\text{false}}$ die Aufgabe, jeweils alle Token eines Signalsegmentes von den mit **true** beziehungsweise **false** markierten Eingängen an den Ausgang zu übertragen.

$$\mathbf{cselect} \left\{ \begin{array}{l} S^3 \rightarrow S \cup \{\perp\} \\ (\varepsilon_S, \varepsilon_S, \varepsilon_S) \mapsto \varepsilon_S \\ ((o, [(t, \text{true})]) \bullet r_1, r_2, r_3) \mapsto \mathbf{cse}_{\text{true}}(r_1, r_2, r_3) \\ ((o, [(t, \text{false})]) \bullet r_1, r_2, r_3) \mapsto \mathbf{cse}_{\text{false}}(r_1, r_2, r_3) \\ (r_1, r_2, r_3) \mapsto \perp \quad \text{sonst} \end{array} \right. \tag{4.18}$$

$$\mathbf{cse}_{\text{true}} \left\{ \begin{array}{l} S^3 \rightarrow S^3 \cup \{\perp\} \\ (r_1, (s, b) \bullet r_2, r_3) \mapsto (s, b) \bullet \mathbf{cse}_{\text{true}}(r_1, r_2, r_3) \\ (r_1, (m, b) \bullet r_2, r_3) \mapsto (m, b) \bullet \mathbf{cse}_{\text{true}}(r_1, r_2, r_3) \\ (r_1, (e, b) \bullet r_2, r_3) \mapsto (e, b) \bullet \mathbf{cselect}(r_1, r_2, r_3) \\ (r_1, (o, b) \bullet r_2, r_3) \mapsto (o, b) \bullet \mathbf{cselect}(r_1, r_2, r_3) \\ (r_1, r_2, r_3) \mapsto \perp \quad \text{sonst} \end{array} \right. \tag{4.19}$$

$$\mathbf{cse}_{\text{false}} \left\{ \begin{array}{l} S^3 \rightarrow S^3 \cup \{\perp\} \\ (r_1, r_2, (s, b) \bullet r_3) \mapsto (s, b) \bullet \mathbf{cse}_{\text{false}}(r_1, r_2, r_3) \\ (r_1, r_2, (m, b) \bullet r_3) \mapsto (m, b) \bullet \mathbf{cse}_{\text{false}}(r_1, r_2, r_3) \\ (r_1, r_2, (e, b) \bullet r_3) \mapsto (e, b) \bullet \mathbf{cselect}(r_1, r_2, r_3) \\ (r_1, r_2, (o, b) \bullet r_3) \mapsto (o, b) \bullet \mathbf{cselect}(r_1, r_2, r_3) \\ (r_1, r_2, r_3) \mapsto \perp \quad \text{sonst} \end{array} \right. \tag{4.20}$$

Kontrollflußstrukturen: Mit Hilfe der Kontrollflußkomponenten **cswitch** und **cselect** lassen sich nun komplexere Kontrollflußstrukturen erstellen.

- **if-then-else:** In diesem Konstrukt werden abhängig von einem einwertigen Steuersegment alle Token eines Signalsegmentes entweder in den einen oder anderen Zweig geleitet (siehe Abbildung 4.5 (a))

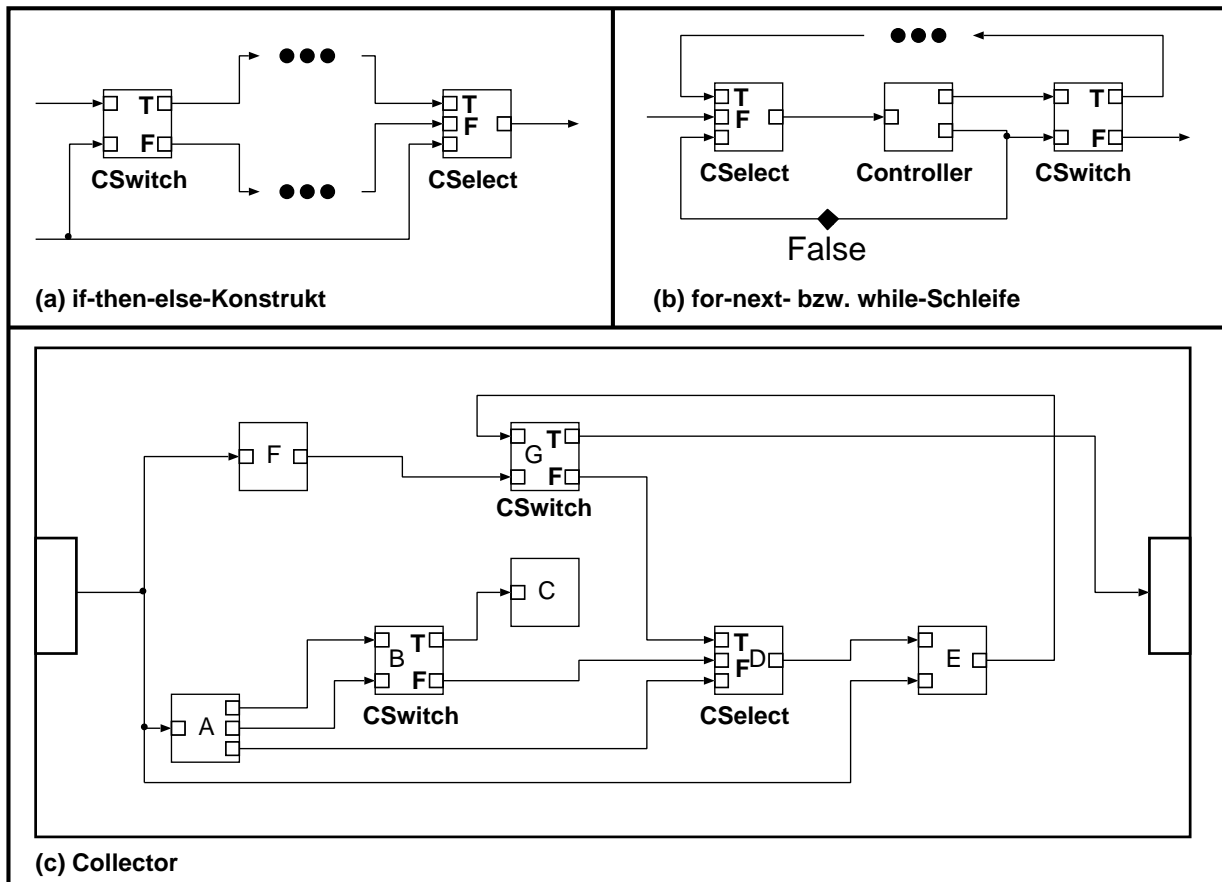


Abbildung 4.5: Komplexere Datenflußstrukturen

- **for-next-SCHLEIFE:** Im Falle der for-next-Schleife werden alle Token eines Signalsegmentes solange rückgekoppelt, bis eine bestimmte zum Beispiel durch den Benutzer festgelegte Anzahl von Durchläufen erreicht ist. Die Struktur einer for-next-Schleife ist in Abbildung 4.5 (b) dargestellt. Interessant ist die Verwendung eines Initialisierungstokens, welches mittels einer schwarzen Raute gekennzeichnet ist. Dieses Initialisierungstoken dient dazu, daß bei Beginn der Abarbeitung das erste Datentoken in die Schleife übernommen wird. Dadurch daß der in dem Initialisierungstoken gespeicherte Wert gleich false ist, wird das bei der CSelect-Komponente am mit F markierten Eingang, welcher auch den Eingang in die Schleife darstellt, anliegende Token an den Ausgang weitergeleitet.
- **while-SCHLEIFE:** Bei dieser Kontrollflußstruktur hängt die Anzahl der Durchläufe von einem von der Datenflußkomponente Controller überprüften Booleschen Ausdruck ab, der nach Durchschleusen aller Token eines Signalsegmentes jeweils neu ausgewertet wird (vergleiche Abbildung 4.5 (b)).

Adapterkomponenten: Um Signalverarbeitungskomponenten, die alle in Colored SDF modelliert sind, auch auf Signalsegmente anwenden zu können, sind verschiedene Adapterkomponenten notwendig, die im folgenden vorgestellt werden. Diese Adapterkomponenten lassen sich alle aus bereits eingeführten Datenflußkomponenten zusammenbauen und stellen somit ZUSAMMENGESETZTE DATENFLUSSKOMponentEN dar.

- **COLLECTOR:** Die **collector**-Komponente sammelt alle Token eines Signalsegmentes auf. Dabei werden die Signalblöcke der jeweiligen Token extrahiert und in einen einzigen Signalblock konkateniert. Das Ergebnistoken, welches ein ganzes Signalsegment beinhaltet, wird als einziges Token eines Signalsegmentes weitergeleitet. Diese Adapterkomponente wird beispielsweise eingesetzt, wenn eine nachgeschaltete Colored-SDF-Komponente auf alle Punkte eines Signalsegmentes auf einmal angewandt werden soll.

$$\text{collector} \left\{ \begin{array}{l} S \rightarrow S \cup \{\perp\} \\ \varepsilon_S \mapsto \varepsilon_S \\ r \mapsto \Pi_2 \text{collect}(\varepsilon_B, r) \end{array} \right. \quad (\Pi : \text{Projektion}) \quad (4.21)$$

$$\text{collect} \left\{ \begin{array}{l} B \times S \rightarrow (B \times S) \cup \{\perp\} \\ (b_1, (\mathbf{s}, b_2) \bullet r) \mapsto \text{collect}(b_1 \bullet b_2, r) \\ (b_1, (\mathbf{m}, b_2) \bullet r) \mapsto \text{collect}(b_1 \bullet b_2, r) \\ (b_1, (\mathbf{e}, b_2) \bullet r) \mapsto (\varepsilon_B, (\mathbf{o}, b_1 \bullet b_2) \bullet \text{collector}(r)) \\ (\varepsilon_B, (\mathbf{o}, b_2) \bullet r) \mapsto (\varepsilon_B, (\mathbf{o}, b_2) \bullet \text{collector}(r)) \\ (b, r) \mapsto \perp \quad \text{sonst} \end{array} \right. \quad (4.22)$$

- **SPLITTER:** Die **splitter**-Komponente ist das Gegenstück zur **collector**-Komponente. Ein Signalsegment, das aus einem einzigen Token besteht, wird in ein Signalsegment mit mehreren Token umgewandelt.

$$\text{splitter} \left\{ \begin{array}{l} S \rightarrow S \cup \{\perp\} \\ \varepsilon_S \mapsto \varepsilon_S \\ (\mathbf{o}, b) \bullet r \mapsto \Pi_4 \text{split}(n, b, \varepsilon_C, r) \\ r \mapsto \perp \quad \text{sonst} \end{array} \right. \quad (\Pi : \text{Projektion}) \quad (4.23)$$

$$\text{split} \left\{ \begin{array}{l} \mathbb{N} \times B \times (C \cup \{\varepsilon_C\}) \times S \rightarrow S \cup \{\perp\} \\ (n, p_1 \bullet \dots \bullet p_n \bullet \varepsilon_B, \varepsilon_C, r) \mapsto (\mathbf{o}, p_1 \bullet \dots \bullet p_n) \bullet \text{splitter}(r) \\ (n, p_1 \bullet \dots \bullet p_m \bullet \varepsilon_B, \varepsilon_C, r) \mapsto (\mathbf{o}, p_1 \bullet \dots \bullet p_m \bullet p_d \bullet \dots \bullet p_d) \bullet \text{splitter}(r) \\ (n, p_1 \bullet \dots \bullet p_n \bullet b, \varepsilon_C, r) \mapsto (\mathbf{s}, p_1 \bullet \dots \bullet p_n) \bullet \text{split}(n, b, \mathbf{s}, r) \\ (n, p_1 \bullet \dots \bullet p_n \bullet b, x, r) \mapsto (\mathbf{m}, p_1 \bullet \dots \bullet p_n) \bullet \text{split}(n, b, \mathbf{m}, r) \\ (n, p_1 \bullet \dots \bullet p_n \bullet \varepsilon_B, x, r) \mapsto (\mathbf{e}, p_1 \bullet \dots \bullet p_n) \bullet \text{splitter}(r) \\ (n, p_1 \bullet \dots \bullet p_m \bullet \varepsilon_B, x, r) \mapsto (\mathbf{e}, p_1 \bullet \dots \bullet p_m \bullet p_d \bullet \dots \bullet p_d) \bullet \text{splitter}(r) \\ (n, b, x, r) \mapsto \perp \quad \text{sonst} \end{array} \right. \quad (4.24)$$

$$\begin{array}{ll} n \in \mathbb{N} & \text{Länge der zu erzeugenden Blöcke} \\ \varepsilon_C & \text{leere Farbe} \\ p_d \in P & \text{Dummyspunkt} \\ x \in \{\mathbf{s}, \mathbf{m}\} \subset T_s & \text{Farbmarkierung} \end{array} \quad (4.25)$$

In der gegebenen funktionalen Beschreibung wird an die Hilfsfunktion `split` die gewünschte Blocklänge des Ausgabesignals als Parameter übergeben. Kann bei der Bearbeitung eines Signalsegmentes der letzte Signalblock nicht vollständig mit Signalwerten gefüllt werden, werden vom Benutzer zu spezifizierende Dummywerte p_d eingesetzt (vergleiche Abschnitt 3.3.4).

- **CUTTER**: Die `cutter`-Komponente paßt die Längen zweier Signalsegmente, die an den zwei Dateneingängen der Datenflußkomponente anliegen, an. Die Länge des jeweils bezüglich der Anzahl der gefärbten Token längeren Signalsegmentes wird der Länge des kürzeren Signalsegmentes angeglichen, indem die überzähligen Token verworfen werden und die Farben entsprechend angepaßt werden (vergleiche Abschnitt 3.3.4).

$$\text{cutter} \left\{ \begin{array}{l} S^2 \rightarrow S^2 \cup \{\perp\} \\ (\varepsilon_S, \varepsilon_S) \mapsto (\varepsilon_S, \varepsilon_S) \\ ((\mathbf{s}, b_1) \bullet r_1, (\mathbf{o}, b_2) \bullet r_2) \mapsto ((\mathbf{o}, b_1), (\mathbf{o}, b_2)) \bullet \text{cut}_{\text{first}}(r_1, r_2) \\ ((\mathbf{m}, b_1) \bullet r_1, (\mathbf{e}, b_2) \bullet r_2) \mapsto ((\mathbf{e}, b_1), (\mathbf{e}, b_2)) \bullet \text{cut}_{\text{first}}(r_1, r_2) \\ ((\mathbf{o}, b_1) \bullet r_1, (\mathbf{s}, b_2) \bullet r_2) \mapsto ((\mathbf{o}, b_1), (\mathbf{o}, b_2)) \bullet \text{cut}_{\text{second}}(r_1, r_2) \\ ((\mathbf{e}, b_1) \bullet r_1, (\mathbf{m}, b_2) \bullet r_2) \mapsto ((\mathbf{e}, b_1), (\mathbf{e}, b_2)) \bullet \text{cut}_{\text{second}}(r_1, r_2) \\ ((x, b_1) \bullet r_1, (x, b_2) \bullet r_2) \mapsto ((x, b_1), (x, b_2)) \bullet \text{cutter}(r_1, r_2), \\ \quad x \in T_S \\ (r_1, r_2) \mapsto \perp \quad \text{sonst} \end{array} \right. \quad (4.26)$$

$$\text{cut}_{\text{first}} \left\{ \begin{array}{l} S^2 \rightarrow S^2 \cup \{\perp\} \\ ((\mathbf{m}, b) \bullet r_1, r_2) \mapsto \text{cut}_{\text{first}}(r_1, r_2) \\ ((\mathbf{e}, b) \bullet r_1, r_2) \mapsto \text{cutter}(r_1, r_2) \\ (r_1, r_2) \mapsto \perp \quad \text{sonst} \end{array} \right. \quad (4.27)$$

$$\text{cut}_{\text{second}} \left\{ \begin{array}{l} S^2 \rightarrow S^2 \cup \{\perp\} \\ (r_1, (\mathbf{m}, b) \bullet r_2) \mapsto \text{cut}_{\text{second}}(r_1, r_2) \\ (r_1, (\mathbf{e}, b) \bullet r_2) \mapsto \text{cutter}(r_1, r_2) \\ (r_1, r_2) \mapsto \perp \quad \text{sonst} \end{array} \right. \quad (4.28)$$

Dabei werden die beiden Hilfsfunktionen `cutfirst` und `cutsecond` für die zwei Eingänge der Datenflußkomponente `cutter` jeweils für die Anpassung des längeren Signalsegmentes eingesetzt.

- **FILLER**: Die `filler`-Komponente arbeitet ähnlich wie die `cutter`-Komponente. Nur wird in diesem Fall das jeweils kürzere Signalsegment mit Dummyblöcken aufgefüllt (vergleiche

Abschnitt 3.3.4). Dazu werden die beiden Hilfsfunktionen $\text{fill}_{\text{first}}$ und $\text{fill}_{\text{second}}$ verwendet.

$$\text{filler} \left\{ \begin{array}{l} S^2 \rightarrow S^2 \cup \{\perp\} \\ (\varepsilon_S, \varepsilon_S) \mapsto (\varepsilon_S, \varepsilon_S) \\ ((\mathbf{s}, b_1) \bullet r_1, (\mathbf{o}, b_2) \bullet r_2) \mapsto ((\mathbf{s}, b_1), (\mathbf{s}, b_2)) \bullet \text{fill}_{\text{second}}(r_1, r_2) \\ ((\mathbf{m}, b_1) \bullet r_1, (\mathbf{e}, b_2) \bullet r_2) \mapsto ((\mathbf{m}, b_1), (\mathbf{m}, b_2)) \bullet \text{fill}_{\text{second}}(r_1, r_2) \\ ((\mathbf{o}, b_1) \bullet r_1, (\mathbf{s}, b_2) \bullet r_2) \mapsto ((\mathbf{s}, b_1), (\mathbf{s}, b_2)) \bullet \text{fill}_{\text{first}}(r_1, r_2) \\ ((\mathbf{e}, b_1) \bullet r_1, (\mathbf{m}, b_2) \bullet r_2) \mapsto ((\mathbf{m}, b_1), (\mathbf{m}, b_2)) \bullet \text{fill}_{\text{first}}(r_1, r_2) \\ ((x, b_1) \bullet r_1, (x, b_2) \bullet r_2) \mapsto ((x, b_1), (x, b_2)) \bullet \text{filler}(r_1, r_2), \\ \quad x \in T_S \\ (r_1, r_2) \mapsto \perp \quad \text{sonst} \end{array} \right. \quad (4.29)$$

$$\text{fill}_{\text{first}} \left\{ \begin{array}{l} S^2 \rightarrow S^2 \cup \{\perp\} \\ (r_1, (\mathbf{m}, b) \bullet r_2) \mapsto ((\mathbf{m}, b_d), (\mathbf{m}, b)) \bullet \text{fill}_{\text{first}}(r_1, r_2) \\ (r_1, (\mathbf{e}, b) \bullet r_2) \mapsto ((\mathbf{e}, b_d), (\mathbf{e}, b)) \bullet \text{filler}(r_1, r_2) \\ \quad (b_d \in B: \text{Dummyblock}) \\ (r_1, r_2) \mapsto \perp \quad \text{sonst} \end{array} \right. \quad (4.30)$$

$$\text{fill}_{\text{second}} \left\{ \begin{array}{l} S^2 \rightarrow S^2 \cup \{\perp\} \\ ((\mathbf{m}, b) \bullet r_1, r_2) \mapsto ((\mathbf{m}, b), (\mathbf{m}, b_d)) \bullet \text{fill}_{\text{second}}(r_1, r_2) \\ ((\mathbf{e}, b) \bullet r_1, r_2) \mapsto ((\mathbf{e}, b), (\mathbf{e}, b_d)) \bullet \text{filler}(r_1, r_2) \\ \quad (b_d \in B: \text{Dummyblock}) \\ (r_1, r_2) \mapsto \perp \quad \text{sonst} \end{array} \right. \quad (4.31)$$

Diese Adapterkomponenten können auch mit Hilfe der einfachen Kontrollflußkomponenten `cswitch` und `cselect` nachgebildet werden. So ist beispielsweise in Abbildung 4.5 (c) die `collector`-Komponente entsprechend modelliert. Bei Eintreffen eines Tokens an Datenflußkomponente **A** wird ein Token (im folgenden als Listentoken bezeichnet) mit Markierung **o**, das einen leeren Block enthält, erzeugt und an `cswitch` **B** weitergeleitet. Handelt es sich bei dem von Eingang eingetroffenen Token um ein Starttoken mit der Marke **s**, wird `cswitch` **B** mit `false` angesteuert, ansonsten mit `true`. Damit werden alle Listentoken verworfen, außer wenn deren Generierung durch ein Starttoken getriggert wurde. Wird das Listentoken nicht verworfen, so gelangt es zu `cselect` **D**, und wird dort in die Sammelschleife geleitet. In Datenflußkomponente **E** wird der Block jedes Signaltokens mit dem Block dieses Listentokens konkateniert. An `cswitch` **G** wird das Listentoken solange in diese Sammelschleife geleitet, bis an der Datenflußkomponente **F** das letzte Token eines Signalsegments anliegt. In diesem Fall wird das Listentoken an den Ausgang weitergeleitet, und der Vorgang beginnt von vorne.

In analoger Weise können alle vorgestellten Adapterkomponenten mit `cswitch` und `cselect` modelliert werden (siehe dazu [MSG02, MSG04]).

4.3.3.3 Colored DDF

Fügt man zu Colored BDF die Datenflußkomponente Colored Merge (**CMerge**) hinzu, so erhält man das Datenflußparadigma Colored DDF. Da **CMerge** nichtdeterministisch ist (vergleiche Definition 2.2.2), kann es nicht als Funktion auf Strömen modelliert werden.

Kontrollflußkomponente: COLORED MERGE (CMerge) überträgt analog zu CSwitch oder CSelect zuerst alle Token eines Signalsegmentes, bevor das nächste Signalsegment betrachtet wird. Trifft also ein Starttoken eines Signalsegmentes bei einem der beiden Eingänge ein, wird es an den Ausgang weitergeleitet. Nun werden nur noch Token, die zu diesem Signalsegment gehören, betrachtet. Treffen an den beiden Eingängen gleichzeitig Starttoken zweier Signalsegmente ein, so werden diese in zufälliger Reihenfolge weitergeleitet.

Adapterkomponenten: Zwei vergleichsweise komplexe Beispiele für Adapterkomponenten stellen die in Abschnitt 4.1 vorgestellten Modelle zur Entkopplung einer Datenflußkomponente von ihrer Vorgängerkomponente dar (vergleiche Abbildungen 4.2). Diese werden als Decoupler (1) und Decoupler (2) bezeichnet und lassen sich mit Hilfe von CMerge nachbilden. Diese Datenflußkomponenten stellen somit nur „syntaktischen Zucker“ dar. Da die in Abschnitt 4.1 geschilderten Nachteile gravierend sind, wird von der Verwendung dieser Datenflußkomponenten abgeraten.

4.3.3.4 Anwendungsbeispiel

Die Aufgabe des Datenflußgraphen in Abbildung 4.6 ist die Qualitätskontrolle von Bildschirmröhren [MSG04]. Ich danke an dieser Stelle Micro-Epsilon Meßtechnik GmbH & Co. KG für die Zurverfügungstellung des als Vorbild dienenden Iconnect-Datenflußgraphen². Ein fellverkleideter Hammer schlägt vorsichtig an die Bildschirmröhre. Das resultierende akustische Signal wird gemessen und an den Datenflußgraphen weitergeleitet, um Sprünge und Risse in der Bildschirmröhre zu erkennen. Die Datenflußkomponente SoundCard kapselt den Treiber für die Soundkarte, in welcher das analoge Akustiksignal digitalisiert wird. Beispiele für Signalverläufe einer defekten und einer funktionsfähigen Bildschirmröhre sind in Abbildung 4.6 dargestellt. Die Datenflußkomponente TrashCan läßt bei jedem Signalsegment das erste Token, das 4096 Signalwerte enthält, durch und verwirft die übrigen. Das resultierende Signalsegment wird in zwei verschiedene Zweige des Datenflußgraphen geleitet. Im ersten Zweig wird mit Hilfe der Datenflußkomponente Threshold-1 die absteigende Amplitude des Signals untersucht. Im zweiten Zweig wird das Signalsegment mittels der Datenflußkomponente FFT in den Frequenzbereich transferiert. Die Datenflußkomponente PeakFinder bestimmt den höchsten Peak im Spektrum. Anschließend werden die Ergebnisse der beiden Zweige als Eingabe für einen Fuzzyklassifizierer bestehend aus zwei Datenflußkomponenten Fuzzify-A und Fuzzify-B und einer Inferenzkomponente FuzzyLogic verwendet. Die Schwellwertbestimmungskomponente Threshold-2 bestimmt, ob der Zugehörigkeitsgrad des Fuzzyausgabeterms crack oberhalb einer bestimmten Schranke liegt. Dementsprechend wird die Bildschirmröhre als defekt (crack) oder nicht defekt (ok) eingestuft. Das Ergebnis wird durch die Datenflußkomponente BinaryDisplay auf einem Kontrollmonitor in Form eines Binärsignals ausgegeben. Außerdem wird ein Aktuator mit Hilfe von DigitalIO angesteuert, der die defekte Bildschirmröhre vom Fließband entfernt.

²Iconnect [MRSS01] ist ein von der Firma Micro-Epsilon Meßtechnik GmbH & Co. KG entwickeltes Bild- und Signalverarbeitungswerkzeug.

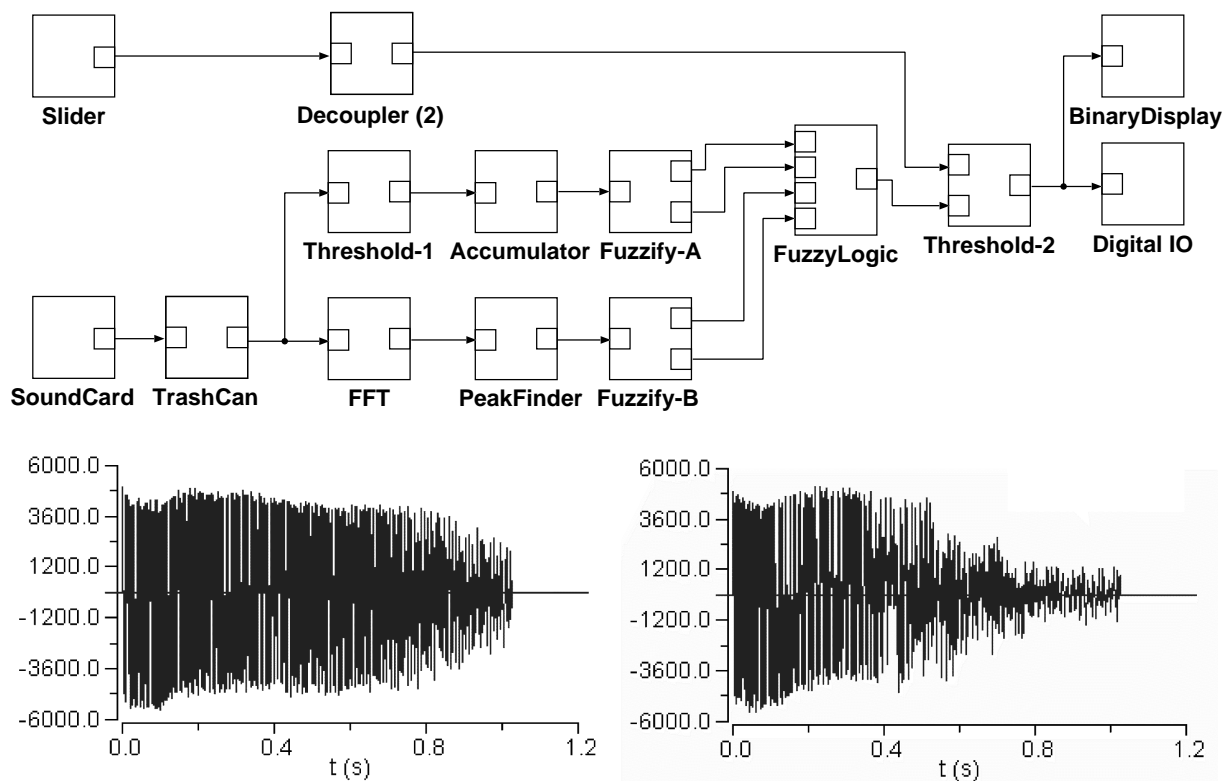


Abbildung 4.6: Datenflußgraph zur Klassifizierung von Bildschirmröhren

Der Benutzer ist in der Lage, den Schwellwert bezüglich des Zugehörigkeitsgrades des Fuzzy-terms mit Hilfe der **Slider**-Komponente zu ändern.

Dieses Beispiel veranschaulicht

- zum einen den Einsatz einer nichtdeterministischen **Decoupler**-Komponente zur Entkoppelung der Datenflußkomponente **Threshold-2** von der Benutzereingabe.
- zum anderen, daß es möglich ist, Bild- und Signalverarbeitungsanwendungen alleine mittels Signalverarbeitungskomponenten (vergleiche Abschnitt 4.3.2) ohne Verwendung von Adapter- und Kontrollflußkomponenten zu realisieren, wenn man leichte Einschränkungen im Anwendungskomfort in Kauf nimmt. So ist zum Beispiel die Verwendung der Datenflußkomponenten **Slider** und **Decoupler** nicht unbedingt erforderlich. Signalverarbeitungskomponenten gehören alle dem Datenflußparadigma **Colored SDF** an und besitzen somit eine **NICHTTURINGÄQUIVALENTE TOKENMASCHINE**. Damit sind solche Datenflußgraphen analysierbar.
- als drittes die **ABWÄGUNG ZWISCHEN ANALYSIERBARKEIT UND MODELLIERUNGSMÄCHTIGKEIT**. Der Anwendungsprogrammierer kann entscheiden, ob die Einstellung dieses Schwellwertes zur Laufzeit wirklich erforderlich ist oder ob dieser Schwellwert zur Laufzeit – nach einer Kalibrierphase – sowieso konstant bleibt.

4.3.4 Denotationelle Semantik

Aus den gegebenen funktionalen Beschreibungen der Datenflußkomponenten von Colored SDF und Colored BDF läßt sich eine denotationelle Semantik herleiten. Für Colored DDF ist dies nicht möglich, da Nichtdeterminismus sich nicht in Form von Funktionen sondern nur als Relationen beschreiben läßt. Dabei gilt [Sch97]:

DEFINITION 4.3.1: *Die BEDEUTUNG EINES WOHLDEFINIERTEN PROGRAMMS ist eine mathematische Funktion von den Eingabe- zu den Ausgabedaten. Die Schritte zur Berechnung der Ausgabe sind unwichtig; allein die Relation von Eingabe zu Ausgabe ist entscheidend.*

Die zugrundeliegende Theorie geht auf [Kah74] zurück (vergleiche auch Abschnitt 2.2.8) und basiert auf der Theorie von Fixpunkten [DP02].

DEFINITION 4.3.2:

1. Eine Funktion $f : S^m \rightarrow S^n$ heißt PRÄFIXMONOTON oder ORDNUNGSERHALTEND, falls

$$\forall x, y \in S^m : x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y) . \quad (4.32)$$

Mehr Information über die Eingabe führt zu mehr Information über die Ausgabe. Die Semantik eines Programms muß infolgedessen eine monotone Funktion sein.

2. Eine Funktion $f : S^m \rightarrow S^n$ ist STETIG, falls $f(C)$ ein Supremum $\bigsqcup f(C)$ für jede Kette $C \subset S^m$ besitzt und

$$f(\bigsqcup C) = \bigsqcup f(C) . \quad (4.33)$$

Eine endliche Ausgabe hängt somit nur von einer endlichen Eingabe ab. Die Ausgabe, die aus einer vollständigen Kenntniss der unendlichen Eingabedaten resultiert, enthält nicht mehr Informationen als die Ausgabe, die an endliche Eingaben gebunden ist.

3. Die MENGE DER STETIGEN FUNKTIONEN wird folgendermaßen dargestellt:

$$[S^m \rightarrow S^n] . \quad (4.34)$$

Falls S^n eine CPO (Complete Partial Order) ist, dann ist auch die Menge der stetigen Funktionen eine CPO. Die Funktion $\perp : S^m \rightarrow S^n$ liefert dabei immer ε_S^n .

4. Eine Abbildung von Funktionen auf Funktionen nennt man FUNKTIONAL.

5. Ein Element einer geordneten Menge S wird als FIXPUNKT einer Funktion $f : S \rightarrow S$ bezeichnet, wenn gilt:

$$f(s) = s . \quad (4.35)$$

6. Das CPO-FIXPUNKT-THEOREM I besagt: Falls S eine CPO ist und $f : S \rightarrow S$ stetig ist, dann existiert der kleinste Fixpunkt von f und ist gleich

$$\bigsqcup_{n \geq 0} f^n(\perp) . \quad (4.36)$$

Betrachtet man die gegebenen funktionalen Komponentenbeschreibungen, so kann man jeder Datenflußkomponente $f_{\text{stream}} : S^m \rightarrow S^n$ ein Funktional

$$\Phi : [S^m \rightarrow S^n] \rightarrow [S^m \rightarrow S^n] \quad (4.37)$$

zuordnen. Indem man nun Φ startend bei der undefinierten Funktion

$$\perp : S^m \rightarrow S^n \quad (4.38)$$

sukzessive angewendet wird eine immer bessere Approximation von f berechnet:

$$\Phi(\Phi(\dots(\Phi(\perp)\dots))) . \quad (4.39)$$

Angewandt auf einen Eingabestrom $r' = x \bullet r$ erhält man:

$$\Phi(f_{\text{stream}})(x \bullet r) = \begin{cases} f_{\text{token}}(x) \bullet f_{\text{stream}}(r) & \text{falls } x \in (S^f)^n \text{ (} S^f \text{: Menge endlicher Ströme)} \\ \varepsilon_S & \text{sonst} \end{cases} \quad (4.40)$$

Dabei ist die Semantik von f gegeben als kleinster Fixpunkt dieser Kette von Approximationen (CPO-Fixpunkt-Theorem I) [Lee97, LP95, DP02].

Will man nun die denotationelle Semantik eines Datenflußgraphen bestimmen, so ergibt sich diese kompositionell aus den Semantiken der Teilgraphen, indem die Hintereinanderschaltung, Parallelkomposition und Projektion auf Teilausgaben entsprechend berücksichtigt wird. So gilt beispielsweise [LP95, Lee97]:

$$\Phi(f_1 \circ f_2) = \Phi_1(f_1) \circ \Phi_2(f_2) \quad (4.41)$$

$$\Phi(f_1 || f_2) = \Phi_1(f_1) || \Phi_2(f_2) \quad (4.42)$$

4.3.5 Probleme und Lösungen

Betrachtet man die sechs Signalmerkmale (siehe Abschnitt 3.3.2), so kann man diese in zwei Gruppen unterteilen:

- **STATISCHE MERKMALE:** Abtastperioden, Blocklängen, Wertemengen, physikalische Einheiten
- **DYNAMISCHE MERKMALE:** Startzeitpunkte, farbige Blockmarkierungen

Aus Datenflußkomponenten lassen sich nun verschiedenste Constraints bezüglich dieser Signalmerkmale herleiten, welche die WOHLDEFINIERTHEIT dieser jeweiligen Datenflußkomponente garantieren.

Constraints, welche die statischen Merkmale betreffen, werden durch das in Kapitel 5 vorgestellte Interface-Typsystem überprüft. Constraints bezüglich des dynamischen Merkmals farbige Blockmarkierung, wie zum Beispiel die korrekte Abfolge solcher Markierungen, werden mittels dem in Kapitel 6 präsentierten Model-Checking-Verfahren getestet.

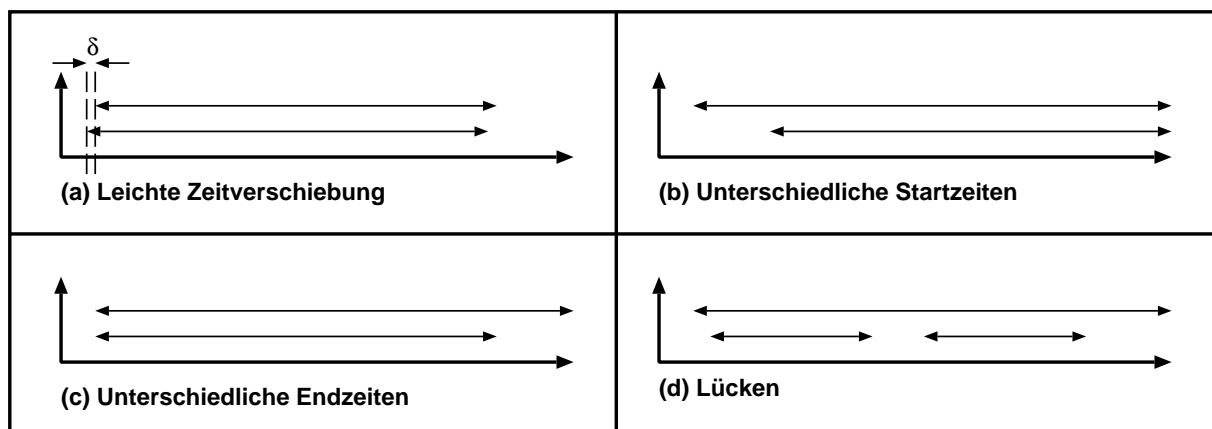


Abbildung 4.7: Unterschiedliche Kombinationen von Signalsegmenten (Zeitbereich)

Die verbleibenden Probleme resultieren aus den unterschiedlichen Zeitbereichen verschiedener Signalsegmente, was durch die dynamischen Merkmale Startzeitpunkte und farbige Blockmarkierungen charakterisiert ist. In Abbildung 4.7 sind die grundlegenden Szenarien dargestellt.

1. **LEICHTE ZEITVERSCHIEBUNG:** Häufig unterscheiden sich zwei Signalsegmente, die „gleichzeitig“ von zwei verschiedenen Sensoren erfaßt wurden, um eine kleine Konstante $\delta \in \mathbb{R}$ (siehe Abbildung 4.7 (a)). Dies liegt beispielsweise an kleinen Phasenverschiebungen der zur Abtastung verwendeten Taktsignale.
2. **UNTERSCHIEDLICHE STARTZEITEN:** Größere Unterschiede in den Startzeiten zweier Signalsegmente (siehe Abbildung 4.7 (b)) können sich ergeben, wenn bei einem Meßobjekt verschiedene Methoden zur Unterscheidung zwischen relevanten und irrelevanten Signalen angewandt werden (vergleiche Abschnitt 3.3.4). Sind die beiden Signalsegmente unterschiedlichen Meßobjekten zuzuordnen, dann sind solche Unterschiede in den Startzeiten ohne größeren Aufwand nicht zu vermeiden. Außerdem können sich die Längen zweier Signalsegmente unterscheiden.
3. **UNTERSCHIEDLICHE ENDZEITEN:** Ähnlich wie die Startzeiten können sich auch die Endzeiten zweier Signalsegmente unterscheiden (siehe Abbildung 4.7 (c)). Die Gründe sind ähnlich wie bei den unterschiedlichen Startzeiten.
4. **LÜCKEN:** In dem Zeitraum, der durch die Start- und Endzeit des einen Signalsegmentes bestimmt ist, sind mehrere Signalsegmente des anderen physikalischen Signals enthalten. Dieses Phänomen hat ähnliche Ursachen wie die unterschiedlichen Startzeiten zweier Signalsegmente.

Jede mögliche Kombination dieser Grundszenarien ist möglich. So könnte ein Signalsegment eine frühere Startzeit und eine spätere Endzeit haben als ein anderes Signalsegment.

Je nachdem, welche Constraints eine Datenflußkomponente definiert, können sich diese Szenarien als Probleme erweisen oder auch nicht. Erwartet zum Beispiel eine Datenflußkomponente, daß Signalsegmente mit gleichem Zeitbereich an ihren Eingängen anliegen, so sind geeignete Maßnahmen zu ergreifen:

1. **LEICHTE ZEITVERSCHIEBUNG:** Im Falle einer Zeitversetzung um einen konstanten Wert δ wird eine Adapterkomponente eingefügt, welche das Signalsegment mit dem früheren Startzeitpunkt zeitlich nach hinten verschiebt.
2. **UNTERSCHIEDLICHE STARTZEITEN UND ENDZEITEN:** Dieser Fall erfordert die Mitwirkung des Anwendungsentwicklers, da dem Entwurfssystem keine ausreichende Information zur automatischen Lösung vorliegt. Als Lösung bieten sich an – abhängig von der jeweiligen Anwendung –
 - sowohl eine Anpassung der Methodik zur Unterscheidung zwischen relevanten und irrelevanten Daten zum Beispiel mittels einer Lichtschranke
 - als auch die Verwendung von Adapterkomponenten beispielsweise zur zeitlichen Verschiebung, zur Extrapolation beziehungsweise zum Auffüllen des kürzeren Signalsegmentes mittels Dummywerten beziehungsweise zum Abschneiden des längeren Signalsegmentes.

Falls offensichtlich ist, welche Art von Adapterkomponenten in einer Anwendung eingefügt werden sollen, kann dies auch automatisch durch das in Kapitel 5 vorgestellte Interface-Typsystem erfolgen (vergleiche Abschnitt 5.4.6).

4.4 Innovative Aspekte

Die innovativen Aspekte des vorgestellten Komponentenmodells lassen sich wie folgt zusammenfassen:

- **Erweiterung der klassischen Datenflußparadigmen:** Die klassischen Datenflußparadigmen SDF, BDF und DDF wurden dergestalt erweitert, daß nun physikalische Signale entsprechend dem neuen Signalmodell (siehe Kapitel 3) verarbeitet werden können. Dabei wurde eine allgemeine Colored-SDF-Komponente definiert, indem für die Strom-, Token-, Block- und Punktmengen des neuen Signalmodells jeweils eine passende Funktion angegeben wurde. Analog wurde im Falle der Colored-BDF-Komponenten `cswitch` und `cselect` verfahren. Es wurden zudem die Schritte zur Herleitung einer denotationellen Semantik skizziert. Insgesamt betrachtet wurde damit die Grundlage für eine neue HIERARCHIE VON DOMÄNENSPEZIFISCHEN PROGRAMMIERSPRACHEN gelegt.
- **Unterstützung der Fehlererkennung:** Wichtig bei der Definition der Datenflußkomponenten waren die Bedingungen, die bei den einzelnen Teilfunktionen angegeben waren. Diese Regeln werden im folgenden Kapitel aufgegriffen und systematisiert, um dann darauf ein geeignetes Interface-Typsystem zur frühzeitigen Fehlererkennung zu definieren.

- **Explizite Kontrolle über Komplexität:** Ein weiterer wichtiger Aspekt bei der Modellbildung war die Unterteilung aller Datenflußkomponenten in drei Kategorien: Signalverarbeitungs-, Adapter- und Kontrollflußkomponenten. Da alle Signalverarbeitungskomponenten als Colored SDF modelliert sind, eröffnet dies dem Entwickler die WAHLMÖGLICHKEIT ZWISCHEN ANALYSIERBARKEIT UND MODELLIERUNGSMÄCHTIGKEIT. Je nach Bedarf kann ein Datenflußgraph bestehend aus Colored-SDF-Komponenten um weitere Adapterkomponenten wie Collector, Splitter, Cutter beziehungsweise Filler oder Kontrollflußkomponenten wie CSwitch, CSelect und CMerge und darauf aufbauenden Kontrollflußstrukturen wie if-then-else, while- beziehungsweise for-next-Schleifen angereichert werden.

Das vorgestellte neue Modell für Datenflußkomponenten bildet zusammen mit dem in Kapitel 3 entwickelten Modell physikalischer Signale die Grundlage für die in den folgenden Kapiteln eingeführten Verfahren der Interfacetypbestimmung und des Model Checkings.

Kapitel 5

Interface-Typsystem

Pacta sunt servanda.

Im Zentrum dieses Kapitels steht ein neues Interface-Typsystem für datenflußorientierte eingebettete Systeme (vergleiche [MG03, May04a]). Nach der Formulierung der Anforderungen an dieses neue Interface-Typsystem werden in einem nächsten Schritt an einem Beispiel die Möglichkeiten von Typconstraints erläutert. Anschließend wird dieses neue Typsystem vorgestellt. Dieses teilt sich in das neue Modell der Interfacetypen und das dazu passende Typbestimmungsverfahren auf. Eine Zusammenfassung der innovativen Aspekte rundet das Kapitel ab.

5.1 Anforderungen

Die Anforderungen an das neue Interface-Typsystem lauten:

- **ENTWURFSBEGLEITENDE ÜBERPRÜFUNG:** Die Typprüfung soll entwurfsbegleitend stattfinden. Während der Entwickler sukzessive Komponenten-Icons in der Zeichenfläche des Entwurfstools plaziert und die Schnittstellen dieser Icons mittels Kanten miteinander verbindet, soll unverzüglich die Einhaltung der Typconstraints geprüft werden. Im Fehlerfall ist eine sofortige Rückmeldung an den Benutzer zu gewährleisten, so daß dieser unmittelbar reagieren kann.
- **INKREMENTELLE TYPPRÜFUNG:** Die Typprüfung soll inkrementell erfolgen. Das bedeutet, daß Ergebnisse vorheriger Checks – soweit wie möglich – wiederverwendet werden, um unnötige Berechnungen zu vermeiden. Dies ist besonders interessant, da im interaktiven Entwurf die Reaktionszeit des Typprüfungsalgorithmus erheblich zum **BEDIENKOMFORT** beiträgt.
- **TYPCONSTRAINTS ZU SIGNALMERKMALEN:** Das Interface-Typsystem soll Typconstraints zu den in Abschnitt 3.3.2 genannten Merkmalen
 - Startzeitpunkte

- Abtastperioden
- Wertemengen (klassische Datentypen)
- physikalische Einheiten
- Blocklängen
- farbige Blockmarkierungen

überprüfen. Die Typconstraints sollen dabei die WOHLDEFINIERTHEIT der von den Komponenten gekapselten Algorithmen sicherstellen.

- **BERÜCKSICHTIGUNG DER KOMPONENTENPARAMETER:** Die vom Benutzer in den Dialogfenstern der Datenflußkomponenten einstellbaren Parameter sollen bei der Typprüfung mit berücksichtigt werden. So soll es möglich sein, Typconstraints zwischen Parametern und Interfacetypen definieren zu können.

5.2 Beispiele zu Typconstraints

In Abbildung 5.1 sind verschiedene Beispiele für Typconstraints dargestellt. Dabei kann man Regelmuster (im folgenden als Constraint-Pattern bezeichnet) erkennen, die bei verschiedenen Algorithmen aus dem Bereich der Bild- und Signalverarbeitung immer wieder auftauchen (vergleiche auch Abschnitt 4.3.3.1):

1. **PUNKTWEISE TRANSFORMATION:** Diese ist dadurch gekennzeichnet, daß eine unäre Funktion auf jeweils einen Signalpunkt angewandt wird. Der Definitionsbereich und die Signalmerkmale, die den Datentransport betreffen, sind bei einem Eingangs- und dem zugehörigen Ausgangssignal identisch. Abbildung 5.1 (a) listet die Typconstraints auf und gibt diverse Beispiele.
2. **PUNKTWEISE KOMBINATION:** Hier werden Funktionen auf Punkte von jeweils zwei oder mehr Eingangssignalen angewandt, um einen Punkt des Ausgangssignals zu berechnen. Auch hier sind die Typconstraints für Definitionsbereich und Datentransportmerkmale identisch. Lediglich bei den Typconstraints, die den Wertebereich betreffen, unterscheiden sich die in diese Kategorie fallenden Funktionen (Abbildung 5.1(b)).
3. **BLOCKWEISE EXTRAKTION:** In diesem Fall wird aus allen Werten eines Signalblockes jeweils ein Wert ermittelt. Dies ist zum Beispiel bei der Ermittlung vieler Statistiken gegeben (Abbildung 5.1(c)).
4. **SEGMENTWEISE EXTRAKTION:** Diese Form der Ermittlung eines Wertes aus allen Werten eines Signalsegmentes (siehe Abbildung 5.1 (d)) ist ein Spezialfall der blockweisen Extraktion. Es werden zuerst alle Token eines Signalsegmentes mittels einer Collector-Komponente (siehe Abschnitt 4.3.3.2) zu einem Token aufgesammelt und anschließend eine blockweise Extraktion durchgeführt.

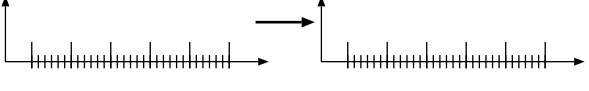
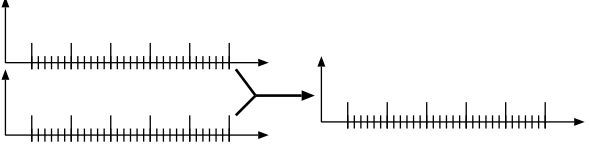
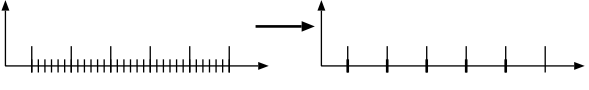
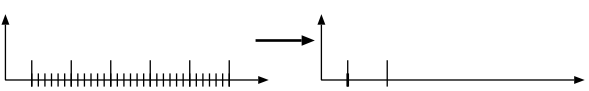
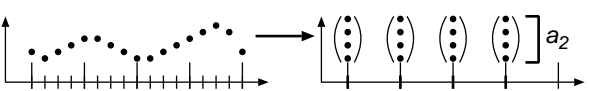
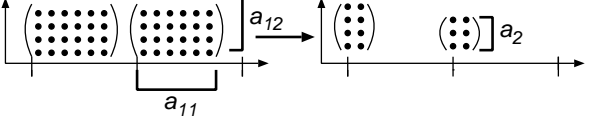
 <p>(a) Punktweise Transformation</p>	Typconstraints: <ul style="list-style-type: none"> - $t_{s2} = t_{s1}$ - $t_{p2} = t_{p1}$ - $bl_2 = bl_1$ - $bm_2 = bm_1$ 	Datenflußkomponenten: <ul style="list-style-type: none"> - Unäre Funktionen wie <ul style="list-style-type: none"> - $\log(x)$, x^n - Bildverarbeitungsalgorithmen (Kantendetektion, Glättung, etc.) - Finden eines Peaks im Spektrum
 <p>(b) Punktweise Kombination</p>	Typconstraints: <ul style="list-style-type: none"> - $t_{s3} = t_{s2} = t_{s1}$ - $t_{p3} = t_{p2} = t_{p1}$ - $bl_3 = bl_2 = bl_1$ - $bm_3 = bm_2 = bm_1$ 	Datenflußkomponenten: <ul style="list-style-type: none"> - Binäre Funktionen wie <ul style="list-style-type: none"> - Arithmetische Operationen auf Skalaren, Vektoren, usw. - Logische Operationen
 <p>(c) Blockweise Extraktion</p>	Typconstraints: <ul style="list-style-type: none"> - $t_{s2} = t_{s1}$ - $t_{p2} = bl_1 \cdot t_{p1}$ - $bl_2 = 1$ - $bm_2 = bm_1$ 	Datenflußkomponenten: <ul style="list-style-type: none"> - Ermittlung von Statistiken (Mittelwert, Minimum, Maximum, Abweichung, etc.) - FFT
 <p>(d) Segmentweise Extraktion</p>	Typconstraints: <ul style="list-style-type: none"> - $t_{s2} = t_{s1}$ - $bl_2 = 1$ - $bm_2 = e$ 	Datenflußkomponenten: <ul style="list-style-type: none"> - Ermittlung von Statistiken (Mittelwert, Minimum, Maximum, Abweichung, etc.)
 <p>(e) FFT (Blockweise Extraktion)</p>	Typconstraints: <ul style="list-style-type: none"> - $t_{s2} = t_{s1}$ - $t_{p2} = bl_1 \cdot t_{p1}$ - $x_1 = double$ - $x_2 = complex [a_2]$ - $arraysize a_2 = bl_1$ - $pu_2 = pu_1 = "--"$ - $bl_1 \in \{2^i \mid i \in \mathbb{N}\}$ - $bl_2 = 1$ - $bm_2 = bm_1$ 	Ähnliche Datenflußkomponenten: <ul style="list-style-type: none"> - Inverse FFT
 <p>(f) Kantendetektion (Punktweise Transformation)</p>	Typconstraints: <ul style="list-style-type: none"> - $t_{s2} = t_{s1}$ - $t_{p2} = t_{p1}$ - $x_1 = int [a_{11}][a_{12}]$ - $x_2 = int [2][a_2]$ - $a_2 \in [0; a_{11} \cdot a_{12}]$ - $pu_2 = pu_1 = "--"$ - $bl_2 = bl_1$ - $bm_2 = bm_1$ 	Ähnliche Datenflußkomponenten: <ul style="list-style-type: none"> - Bildverarbeitungsalgorithmen <ul style="list-style-type: none"> - Erzeugen von Bildern aus Bitmasken - Aufspüren von Zusammenhangskomponenten - Subsampling von Bildern

Abbildung 5.1: Constraint-Pattern

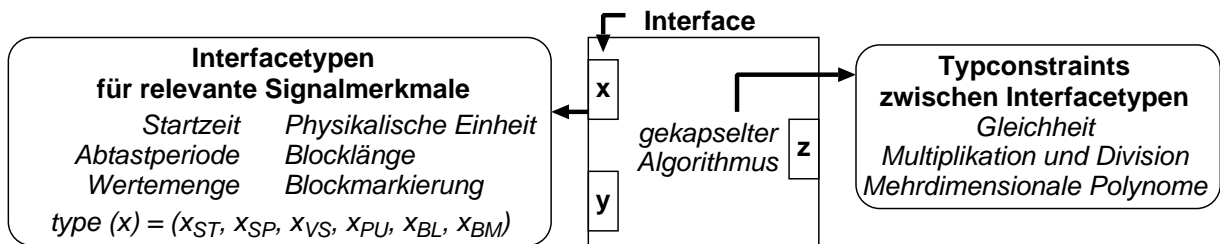


Abbildung 5.2: Interfacetypen und Typconstraints einer Datenflußkomponente

Ein prominentes Beispiel für blockweise Extraktion ist die Fast-Fourier-Transformation (siehe Abbildung 5.1 (e)). Es wird jeweils ein Signalblock aus `double`-Werten in einen komplexwertigen Vektor transferiert, dessen Größe der Blockgröße des Eingangssignals entspricht. Die Abtastperiode des Ausgangssignals ergibt sich dabei als Produkt aus Blocklänge und Abtastperiode eines Eingangssignals. Die Startzeitpunkte und die farbigen Blockmarkierungen sind bei Eingangs- und zugehörigem Ausgangssignal identisch.

Ein Beispiel für eine punktweise Transformation ist durch die Kantendetektion bei Bildern gegeben (siehe Abbildung 5.1 (f)). Aus Bildern, die zweidimensionalen Arrays fester Größe entsprechen, werden jeweils Kanten extrahiert. Jede Kante ist durch Start- und Endpunkt gekennzeichnet. Die Anzahl der Kanten kann aber von Bild zu Bild variieren. Startzeiten, Abtastperioden, Blocklängen und farbige Blockmarkierungen sind bei Ein- und Ausgangssignal identisch.

5.3 Neues Modell der Interfacetypen

Dieser Abschnitt beinhaltet ein neues Modell für Interfacetypen. Nach einigen einführenden Definitionen folgt die Beschreibung der Typdomäne bestehend aus Typtraits. In diesem Zusammenhang werden die mathematischen Modelle, die dem Interface-Typsystem zugrundeliegen, vorgestellt. Dabei spielt auch Polymorphismus eine zentrale Rolle. Am Ende dieses Abschnittes sind die Typconstraints erläutert.

5.3.1 Definitionen

In diesem Abschnitt werden einige für dieses neue Modell der Interfacetypen grundlegende Begriffe eingeführt, die für das Verständnis dieses Kapitels notwendig sind.

DEFINITION 5.3.1:

1. Ein `INTERFACE` ist eine Schnittstelle einer Datenflußkomponente [Xio02], zum Beispiel x in Abbildung 5.2.
2. Ein `INTERFACETYP` bestimmt die Menge der zulässigen Werte des jeweiligen Interfaces.
3. Eine `TYPDOMÄNE` ist die Menge aller möglichen Typen.

Abbildung 5.2 stellt eine Datenflußkomponente mit den Schnittstellen x , y und z dar. Der Interface-typ $\text{type}(x)$ des Interfaces x – das Analogon eines Datentyps – ist ein Element der Typdomäne

$$\begin{aligned} \text{DOM} &= \text{ST} \times \text{SP} \times \text{VS} \times \text{PU} \times \text{BL} \times \text{BM} & (5.1) \\ \text{ST} &: \text{Typtrait für Startzeiten} \\ \text{SP} &: \text{Typtrait für Abtastperioden} \\ \text{VS} &: \text{Typtrait für Wertemengen} \\ \text{PU} &: \text{Typtrait für physikalische Einheiten} \\ \text{BL} &: \text{Typtrait für Blocklängen} \\ \text{BM} &: \text{Typtrait für farbige Blockmarkierungen} \end{aligned}$$

Die verschiedenen Elemente dieses Kreuzprodukts werden als TYPTRAITS bezeichnet und im folgenden Abschnitt näher erläutert.

5.3.2 Typdomäne

Im folgenden werden die einzelnen Typtraits genauer betrachtet. Außerdem wird erläutert, in welcher Weise dieses Typsystem Polymorphismus unterstützt.

5.3.2.1 Typtraits

Als erstes werden die Mengen, welche die Typtraits darstellen, eingeführt. Anschließend wird gezeigt, daß auf allen Typtraits eine partielle Ordnung definiert ist. Alle in diesem Abschnitt vorgestellten Typtraits bilden vollständige Verbände [Bir84]. Vor der Definition eines Verbandes beziehungsweise eines vollständigen Verbandes sind aber die Begriffe Schranke, obere und untere Schranke und kleinste obere und größte untere Schranke zu klären. Dabei ist im Hinblick auf das Typbestimmungsverfahren (vergleiche Abschnitt 5.4) vor allem interessant, wie Infimum und Supremum für die einzelnen Typtraits ermittelt werden.

Typtraits als Mengen: Typtraits stellen die Bausteine der Typdomäne dar (siehe Abbildung 5.3). Zur Konstruktion einiger Typtraits ist das Funktional G erforderlich, welches aus einer gegebenen nichtleeren Menge Z eine Menge von Mengen erzeugt, in welcher Z , alle elementareren Teilmengen von Z und die leere Menge \emptyset enthalten sind.

$$G(Z) = \{Z, \emptyset\} \cup \{Z' \mid Z' \subseteq Z \wedge |Z'| = 1\} \quad (5.2)$$

- Die STARTZEITEN beziehungsweise die FARBIGEN BLOCKMARKIERUNGEN eines Signalsegmentes sind in der Regel bis zur Ausführung des Datenflußgraphen unbekannt. Erst dann werden die physikalischen Eingabesignale digitalisiert und anschließend die Startzeiten und farbigen Blockmarkierungen bestimmt. Nichtsdestotrotz erlaubt die Verwendung von symbolischen Werten einige Überprüfungen zur Entwurfszeit (wie zum Beispiel die Gleichheit der farbigen Blockmarkierungen zweier physikalischer Signale). Die Typtraits

ST und BM modellieren die möglichen Typbelegungen für die Startzeiten und die farbigen Blockmarkierungen:

$$\mathbf{ST} = G(\{s_i | 0 < i \leq k \wedge i, k \in \mathbb{N}\}) \quad (5.3)$$

$$\mathbf{BM} = G(\{b_i | 0 < i \leq l \wedge i, l \in \mathbb{N}\} \cup \{\mathbf{o}\}) \quad (5.4)$$

Dabei stellen s_i und b_i symbolische Werte dar. Die Werte für $k \in \mathbb{N}$ und $l \in \mathbb{N}$ repräsentieren die jeweilige Zahl unterschiedlicher Startzeitpunkte beziehungsweise farbiger Blockmarkierungen¹. Die farbige Blockmarkierung $\mathbf{o} \in T_s$ (vergleiche Abschnitt 3.3.3) wird als einziger nichtsymbolischer Wert in **BM** berücksichtigt, da manche Datenflußkomponenten wie beispielsweise **CSwitch** (vergleiche Abschnitt 4.3.3.2) an einer oder mehreren Schnittstellen Signalsegmente der Länge 1 erwarten. Für die Überprüfung von Kommunikationsprotokollen betreffs der farbigen Blockmarkierungen werden in Kapitel 6 eine eigene Modellierung in Form von Fifomaten und ein dezidiertes Model-Checking-Verfahren vorgestellt.

- **PHYSIKALISCHE EINHEITEN** werden als Sieben-Tupel modelliert, wobei jedes Element einen Exponenten einer SI-Einheit darstellt. Die Verwendung von SI-Einheiten garantiert dabei die Eindeutigkeit der Darstellung². So gilt zum Beispiel:

$$(1, 1, -2, 0, 0, 0, 0) \in \mathbb{Z}^7 \text{ entspricht } \text{m}^1 \cdot \text{kg}^1 \cdot \text{sec}^{-2} \cdot \text{A}^0 \cdot \text{K}^0 \cdot \text{mol}^0 \cdot \text{cd}^0 = \text{Newton} . \quad (5.5)$$

Der Typtrait **PU** modelliert alle möglichen Typbelegungen bezüglich der physikalischen Einheiten:

$$\mathbf{PU} = G(\mathbb{Z}^7) \quad (5.6)$$

- Die **BASISTYPEN BT** der **WERTEMENGE VS** sind gegeben als (vergleiche Abbildung 5.3 (e)):

$$\mathbf{BT} = \{\perp_{\mathbf{BT}}, \text{number}, \text{bool}, \text{int}, \text{long}, \text{float}, \text{double}, \text{complex}, \text{string}, \not\downarrow_{\mathbf{BT}}\} \quad (5.7)$$

Alle Typen inklusive der Aggregationstypen Records und Arrays können als kleinster Fixpunkt des Funktionals

$$F(\mathbf{Z}) = \underbrace{\mathbf{BT}}_{\text{Basistypen}} \cup \underbrace{\{\perp_{\mathbf{VS}}, \not\downarrow_{\mathbf{VS}}\}}_{\text{Bottom und Top}} \cup \mathbf{Z} \cup \underbrace{(\mathbf{Z} \times \mathbf{Z})}_{\text{Records}} \cup \underbrace{(\mathbf{Z} \times I(\mathbb{N}))}_{\text{Arrays}} \quad (5.8)$$

bestimmt werden. Dabei stellt $I(\mathbb{N})$ die Menge der Intervalle der natürlichen Zahlen dar. Die Menge der Basistypen inklusive Top- und Bottomelement ist das Ergebnis einer einmaligen Anwendung, wobei \mathbf{Z} auf \emptyset gesetzt wird:

$$F(\emptyset) = \mathbf{BT} \cup \{\perp_{\mathbf{VS}}, \not\downarrow_{\mathbf{VS}}\} . \quad (5.9)$$

¹Die Werte für k und l sind endlich, da diese durch die Gesamtanzahl der Schnittstellen aller im betrachteten Datenflußgraphen vorhandenen Datenflußkomponenten begrenzt sind.

²Diese eindeutige Darstellung wird im gesamten Datenflußgraphen aufrecht erhalten. Nur bei der Anzeige in Displays sind Umrechnungen in nichteindeutige Darstellungen, die weitere physikalische Einheiten wie beispielsweise Watt umfassen, sinnvoll.

Bei zweimaliger Anwendung von F in der Form $(F(F(\emptyset)))$ erhält man alle eindimensionalen Arrays von Basistypen und alle Records bestehend aus zwei Basistypen und so fort. Der Typtrait VS der Wertemengen wird in einem ersten Ansatz³ wie folgt konstruiert:

$$VS = \bigcup_{n=1}^{\infty} F^n(\emptyset) \quad (5.10)$$

- ABTASTPERIODEN SP und Blocklängen BL werden jeweils durch Intervalle von natürlichen Zahlen repräsentiert (vergleiche Abschnitt 5.3 (d)). Ein solches Intervall wird im Laufe des Typbestimmungsverfahrens solange verkürzt, bis es nur noch einen Wert enthält. Das Typbestimmungsverfahren ist in allen Details in Abschnitt 5.4 beschrieben. Arraygrößen AS werden für die Definition der Wertemengen benötigt und werden auch als Intervalle natürlicher Zahlen dargestellt.

Typtraits als partiell geordnete Mengen: Eine für die Typbestimmung grundlegende Voraussetzung ist, daß alle Typtraits partiell geordnete Mengen bilden.

DEFINITION 5.3.2: Sei X eine Menge. Eine PARTIELLE ORDNUNG auf X ist eine binäre Relation \leq auf X , so daß gilt:

$$\forall x, y, z \in X : \begin{cases} (i) & x \leq x & \text{Reflexivität} \\ (ii) & x \leq y \wedge y \leq x \Rightarrow x = y & \text{Antisymmetrie} \\ (iii) & x \leq y \wedge y \leq z \Rightarrow x \leq z & \text{Transitivität} \end{cases} \quad (5.11)$$

Eine Menge X mit einer Ordnungsrelation \leq bezeichnet man als PARTIELL GEORDNETE MENGE.

Eine weitere wichtige Eigenschaft der Typtraits ist das jeweilige Vorhandensein eines kleinsten und eines größten Elementes:

DEFINITION 5.3.3: Sei X eine partiell geordnete Menge. X hat ein kleinstes Element \perp_X , wenn gilt:

$$\forall x \in X : \perp_X \leq x . \quad (5.12)$$

X hat ein größtes Element \top_X , wenn gilt:

$$\forall x \in X : \top_X \geq x . \quad (5.13)$$

³Auf diese Weise können alle möglichen Arrays und Records konstruiert werden. Diese in der Theorie wünschenswerte Eigenschaft führt aber in der Praxis zu Terminierungsproblemen bei der Typbestimmung. Diese Typbestimmung basiert nämlich auf der Berechnung eines kleinsten Fixpunktes. Daher wird n auf einen endlichen Wert gesetzt. Dieses Problem und die zugehörige Lösung sind ausführlich in Abschnitt 5.4.5 behandelt.

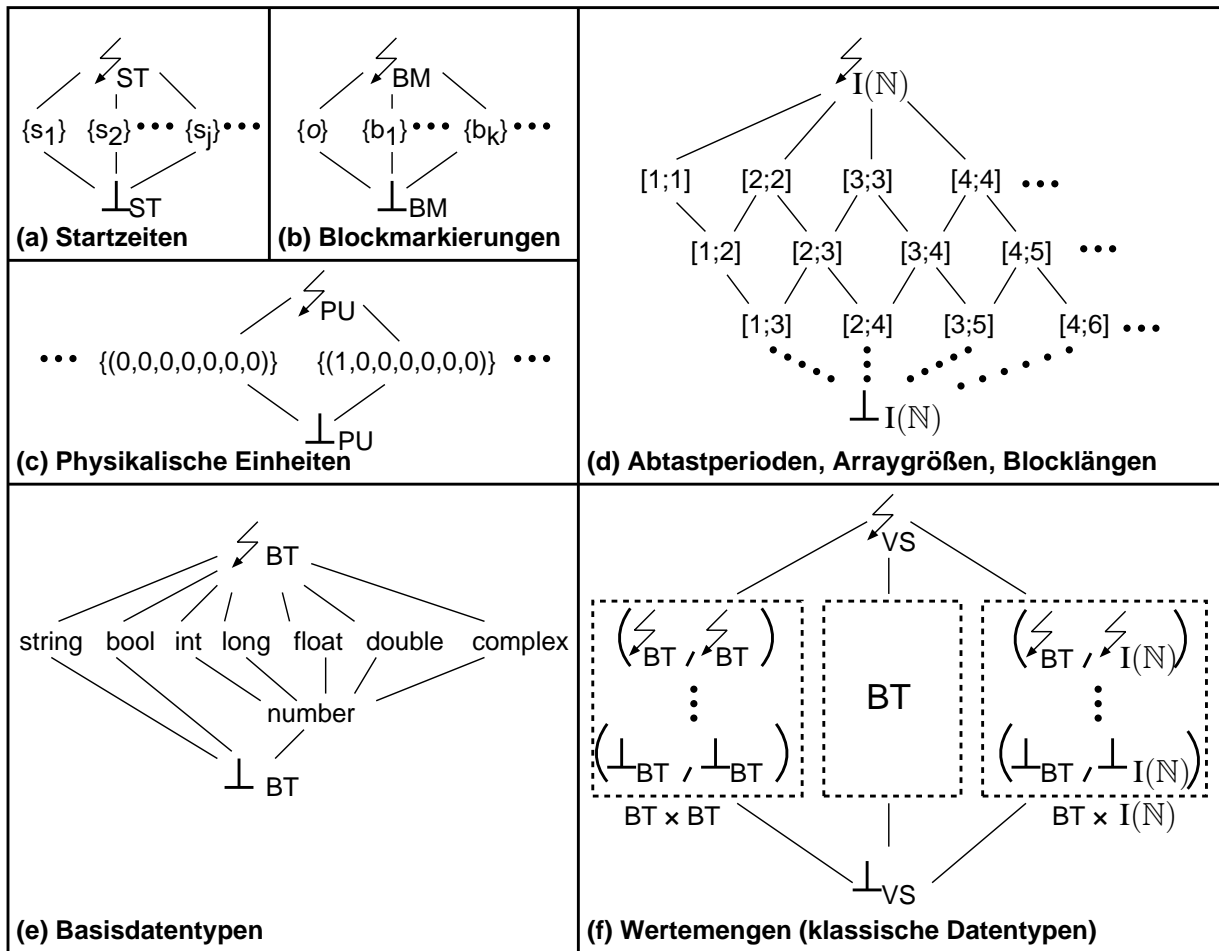


Abbildung 5.3: Hasse-Diagramme der Typtraits

Für den Nachweis obiger Eigenschaften für die jeweiligen Typtraits sind folgende Lemmata erforderlich.

LEMMA 5.3.1: *Das Kreuzprodukt der partiell geordneten Mengen X_1, \dots, X_n ist eine partiell geordnete Menge mit der Ordnung*

$$(x_1, \dots, x_n) \leq (y_1, \dots, y_n) \Leftrightarrow \forall i \in \{1, \dots, n\} : x_i \leq y_i, \quad (5.14)$$

wobei $x_i, y_i \in X_i$ sind.

Ein Beweis findet sich in [DP02]. Gibt es in allen an einem Kreuzprodukt beteiligten Mengen X_1, \dots, X_n ein kleinstes Element, dann entsteht das kleinste Element des Kreuzprodukts aus

diesen. Analog wird das größte Element des Kreuzprodukts erzeugt⁴.

LEMMA 5.3.2: Die Vereinigung $X \dot{\cup} Y$ disjunkter partiell geordneter Mengen X und Y ist partiell geordnet, wobei nur Elemente, die ursprünglich in derselben Menge waren, verglichen werden.

Ein Beweis findet sich in [DP02]. Die Menge besitzt aber kein gemeinsames kleinstes oder größtes Element, selbst wenn kleinste und größte Elemente in X und Y enthalten sind.

LEMMA 5.3.3: Sei X eine partiell geordnete Menge und seien \top und \perp keine Elemente von X . Dann ist

$$X' = X \dot{\cup} \{\top, \perp\} \quad (5.15)$$

mit der Ordnung

$$\forall x, y \in X' : x \leq y \Leftrightarrow x = \perp \vee y = \top \vee x \leq y \text{ in } X \quad (5.16)$$

eine partiell geordnete Menge mit \perp als kleinstem Element und \top als größtem Element.

Nach diesen mathematischen Einführungen kommt jetzt die erste zentrale Aussage bezüglich der Typtraits.

SATZ 5.3.1: Die Typtraits **ST**, **SP**, **VS**, **PU**, **BL**, **BM** sind partiell geordnete Mengen.

BEWEIS 5.3.1: Der Beweis gliedert sich in drei Teile. Zum einen werden alle Mengen einbezogen, welche mit dem Funktional G aus Gleichung 5.2 erzeugt wurden. Der zweite Teil behandelt die Menge aller Intervalle in \mathbb{N} . Der dritte Teil bezieht sich auf die durch das Funktional F aus Gleichung 5.8 definierte Menge **VS**.

1. Fall (**ST**, **PU** und **BM**):

Gegeben: Sei Y eine beliebige nichtleere Menge. Sei $G(Y)$ definiert durch die Gleichung 5.2.

Behauptung: Dann ist durch die Relation \supseteq eine partielle Ordnung auf $G(Y)$ definiert.

Beweis: Die Teilmengenbeziehung \subseteq definiert eine Ordnung auf $G(Y)$. Somit ist durch \supseteq die duale Ordnung gegeben [DP02].

\Rightarrow Behauptung.

Damit gilt auch, daß **ST**, **PU** und **BM** mit der Ordnung \supseteq partiell geordnete Mengen sind.

2. Fall (**SP** und **BL**):

Gegeben: Sei $I(\mathbb{N})$ die Menge aller Intervalle in \mathbb{N} .

Behauptung: Dann ist durch \supseteq eine partielle Ordnung auf $I(\mathbb{N})$ definiert.

⁴ $\dot{\cup}$ steht in dieser Arbeit für disjunkte Vereinigung.

Beweis: Analog zu Fall 1.

⇒ Behauptung.

Damit gilt auch, daß SP und BL mit der Ordnung \supseteq partiell geordnete Mengen sind.

3. Fall (VS):

Gegeben: Sei die Menge der Basistypen gegeben durch Gleichung 5.7. Sei auf BT die in Abbildung 5.3 (e) vorgegebene Ordnung \leq und auf $I(\mathbb{N})$ die Ordnung \supseteq definiert.

Behauptung: Dann ist auf VS induktiv eine Ordnung \leq definiert, welche eine Erweiterung der Ordnungen von BT und $I(\mathbb{N})$ darstellt. Es gibt zudem ein kleinstes Element \perp_{VS} und ein größtes Element ζ_{VS} .

Beweis:

(a) Induktionsanfang ($n = 1$): $F^1(\emptyset) = F(\emptyset) = \mathbf{BT} \cup \{\perp_{VS}, \zeta_{VS}\}$ ist aufgrund Lemma 5.3.3 partiell geordnet. Das kleinste Element ist \perp_{VS} , und das größte Element ist ζ_{VS} .

(b) Induktionsschritt ($n \rightarrow n + 1$):

$$\begin{aligned} & F^{n+1}(\emptyset) \\ &= F(F^n(\emptyset)) \\ &= \mathbf{BT} \cup \{\perp_{VS}, \zeta_{VS}\} \cup F^n(\emptyset) \cup (F^n(\emptyset) \times F^n(\emptyset)) \cup (F^n(\emptyset) \times I(\mathbb{N})) \\ &= (\{\perp_{VS}, \zeta_{VS}\} \cup F^n(\emptyset)) \dot{\cup} (F^n(\emptyset) \times F^n(\emptyset)) \dot{\cup} (F^n(\emptyset) \times I(\mathbb{N})) \end{aligned} \quad (5.17)$$

- i. $F^n(\emptyset) = X \dot{\cup} \{\perp_{VS}, \zeta_{VS}\}$ ist geordnet (Induktionsvoraussetzung)
- ii. $(F^n(\emptyset) \times F^n(\emptyset)) \dot{\cup} \{\perp_{VS}, \zeta_{VS}\}$ ist geordnet gemäß den Lemmata 5.3.1 und 5.3.3 mit \perp_{VS} als kleinstem und ζ_{VS} als größtem Element.
- iii. $(F^n(\emptyset) \times I(\mathbb{N})) \dot{\cup} \{\perp_{VS}, \zeta_{VS}\}$ ist geordnet gemäß den Lemmata 5.3.1 und 5.3.3 mit \perp_{VS} als kleinstem und ζ_{VS} als größtem Element.
- iv. Damit ist $X \dot{\cup} (F^n(\emptyset) \times F^n(\emptyset)) \dot{\cup} (F^n(\emptyset) \times I(\mathbb{N})) \dot{\cup} \{\perp_{VS}, \zeta_{VS}\}$ geordnet (vergleiche Lemmata 5.3.2 und 5.3.3). Fügt man \perp_{VS} als das kleinste Element und ζ_{VS} als das größte Element hinzu, folgt die Behauptung.

⇒ Behauptung.

⇒ Satz 5.3.1. □

Infima und Suprema der einzelnen Typtraits: Nach einer Reihe einleitender Definitionen werden das Infimum und das Supremum der jeweiligen Typtraits vorgestellt.

DEFINITION 5.3.4:

1. Sei X eine geordnete Menge und $Y \subseteq X$. Ein Element $x \in X$ ist eine OBERE SCHRANKE für Y , falls gilt:

$$\forall y \in Y : y \leq x \quad (5.18)$$

Die MENGE ALLER OBEREN SCHRANKEN von Y bezeichnet man mit Y_u .

2. Sei X eine geordnete Menge und $Y \subseteq X$. Ein Element $x \in X$ ist eine **UNTERE SCHRANKE** für Y , falls gilt:

$$\forall y \in Y : y \geq x \quad (5.19)$$

Die **MENGE ALLER UNTEREN SCHRANKEN** von Y bezeichnet man mit Y_l .

3. Falls Y_u ein **kleinstes Element** besitzt, dann bezeichnet man dieses als **KLEINSTE OBERE SCHRANKE** oder **SUPREMUM**.
4. Falls Y_l ein **größtes Element** besitzt, dann bezeichnet man dieses als **GRÖSSTE UNTERE SCHRANKE** oder **INFIMUM**.

Folgendes Lemma liefert Berechnungsvorschriften für Infimum und Supremum bezüglich der verschiedenen Typtraits. Dabei ist für den Typbestimmungsalgorithmus (vergleiche Abschnitt 5.4) besonders wichtig, daß die Berechnung des Supremums immer auf der **SCHNITTMENGENBILDUNG** basiert. Im Falle der Wertemenge **VS** wird dabei das Supremum beziehungsweise Infimum **KOMPONENTENWEISE** bestimmt.

LEMMA 5.3.4: *Es gibt BERECHNUNGSVORSCHRIFTEN FÜR INFIMUM UND SUPREMUM auf allen Typtraits. Diese lauten wie folgt:*

1. Für alle mit Hilfe des Funktionals G über der Menge Y definierten Typtraits:

$$\forall x, y \in G(Y) : \mathbf{sup}_{G(Y)}\{x, y\} = x \cap y \quad (5.20)$$

$$\forall x, y \in G(Y) : \mathbf{inf}_{G(Y)}\{x, y\} = \begin{cases} x & \text{falls } y = \emptyset \\ y & \text{falls } x = \emptyset \\ Y & \text{sonst} \end{cases} \quad (5.21)$$

2. Für die Intervalle $I(\mathbb{N})$ auf den natürlichen Zahlen:

$$\forall x, y \in I(\mathbb{N}) : \mathbf{sup}_{I(\mathbb{N})}\{x, y\} = x \cap y \quad (5.22)$$

$$\forall x, y \in I(\mathbb{N}) : \mathbf{inf}_{I(\mathbb{N})}\{x, y\} = \bigcap \{z \in I(\mathbb{N}) \mid x \cup y \subseteq z\} \quad (5.23)$$

3. Für die Wertemenge VS :

$$\forall x, y \in VS : \mathbf{sup}_{VS}\{x, y\} = \begin{cases} \mathbf{sup}_{BT}\{x, y\} & \text{falls } x, y \in BT \\ (\mathbf{sup}_{VS}\{x_1, y_1\}, \mathbf{sup}_{VS}\{x_2, y_2\}) & \text{falls } x = (x_1, x_2), \\ & y = (y_1, y_2) \text{ Records} \\ (\mathbf{sup}_{VS}\{x_1, y_1\}, \mathbf{sup}_{I(\mathbb{N})}\{x_2, y_2\}) & \text{falls } x = (x_1, x_2), \\ & y = (y_1, y_2) \text{ Arrays} \\ x & \text{falls } y = \perp_{VS} \\ y & \text{falls } x = \perp_{VS} \\ \frac{1}{2} VS & \text{sonst} \end{cases} \quad (5.24)$$

$$\forall x, y \in VS : \mathbf{inf}_{VS}\{x, y\} = \begin{cases} \mathbf{inf}_{BT}\{x, y\} & \text{falls } x, y \in BT \\ (\mathbf{inf}_{VS}\{x_1, y_1\}, \mathbf{inf}_{VS}\{x_2, y_2\}) & \text{falls } x = (x_1, x_2), \\ & y = (y_1, y_2) \text{ Records} \\ (\mathbf{inf}_{VS}\{x_1, y_1\}, \mathbf{inf}_{I(\mathbb{N})}\{x_2, y_2\}) & \text{falls } x = (x_1, x_2), \\ & y = (y_1, y_2) \text{ Arrays} \\ x & \text{falls } y = \frac{1}{2} VS \\ y & \text{falls } x = \frac{1}{2} VS \\ \perp_{VS} & \text{sonst} \end{cases} \quad (5.25)$$

BEWEIS 5.3.2:

1. Fall (ST, PU und BM):

Gegeben: Sei Y eine beliebige nichtleere Menge. Sei $G(Y)$ definiert durch die Gleichung 5.2, und sei auf $G(Y)$ die Ordnung \supseteq gegeben.

Behauptung: Dann sind durch die Gleichungen 5.21 Infimum und Supremum auf $G(Y)$ definiert.

Beweis: Seien x, y zwei beliebige Elemente von $G(Y)$, die nach der gegebenen partiellen Ordnung vergleichbar sind. Ohne Beschränkung der Allgemeinheit gilt somit $x \leq y$, was laut Definition $x \supseteq y$ bedeutet. Die für $G(Y)$ gegebene Definition des Supremums liefert $\sup_{G(Y)}\{x, y\} = x \cap y = y$. Das heißt, für $x \leq y$ liefert $\sup_{G(Y)}\{x, y\}$ den Wert y . Dies entspricht der Eigenschaft eines Supremums [DP02].

Seien x, y zwei beliebige Elemente von $G(Y)$, die nach der gegebenen partiellen Ordnung nicht vergleichbar sind. Dann gilt, daß die Menge der oberen Schranken sowohl für x als auch für y nur \emptyset enthält. Die kleinste obere Schranke für x und y ist somit auch \emptyset . Dies entspricht dem Ergebnis von $x \cap y$.

Somit ist in Gleichung 5.21 die Berechnungsvorschrift für das Supremum auf $G(Y)$ gegeben. Der Beweis für das Infimum verläuft analog. Daraus folgt die Behauptung.

2. Fall (SP und BL):

Gegeben: Sei $I(\mathbb{N})$ die Menge aller Intervalle in \mathbb{N} mit der Ordnung \supseteq .

Behauptung: Dann sind durch die Gleichungen 5.23 Infimum und Supremum auf $I(\mathbb{N})$ definiert.

Beweis: vergleiche [DP02].

3. Fall (VS):

Gegeben: Sei die Menge der Basistypen gegeben durch Gleichung 5.7 mit der in Abbildung 5.3 (e) vorgegebenen Ordnung. Supremum $\sup_{\mathbf{BT}}\{x, y\}$ und Infimum $\inf_{\mathbf{BT}}\{x, y\}$ für je zwei Basistypen x und y sind gleichfalls aus dieser Abbildung ablesbar. Sei $I(\mathbb{N})$ die Menge der Intervalle der natürlichen Zahlen mit der Ordnung \supseteq und der in Gleichung 5.23 vorgegebenen Berechnungsvorschrift für Supremum und Infimum.

Behauptung: Dann sind mit Hilfe der Gleichungen 5.24 und 5.25 Supremum und Infimum für die Wertemenge VS definiert.

Beweis:

(a) Induktionsanfang ($n = 1$): In $F^1(\emptyset) = F(\emptyset) = \mathbf{BT} \cup \{\perp_{\mathbf{VS}}, \downarrow_{\mathbf{VS}}\}$ ist für je zwei Elemente das Supremum definiert. Sind beide Elemente in \mathbf{BT} , dann wird das Supremum wie gewohnt berechnet. Ist mindestens eines der beiden Elemente gleich $\perp_{\mathbf{VS}}$, so ist das andere Element das Supremum, da $\perp_{\mathbf{VS}}$ das kleinste Element der Menge ist. Aufgrund der Symmetrie der Ordnung ist $\downarrow_{\mathbf{VS}}$ das Ergebnis der Supremumsbestimmung mit einem beliebigen anderen Element.

(b) Induktionsschritt ($n \rightarrow n + 1$):

$$\begin{aligned}
 & F^{n+1}(\emptyset) \\
 &= F(F^n(\emptyset)) \\
 &= \mathbf{BT} \cup \{\perp_{\mathbf{VS}}, \downarrow_{\mathbf{VS}}\} \cup F^n(\emptyset) \cup (F^n(\emptyset) \times F^n(\emptyset)) \cup (F^n(\emptyset) \times I(\mathbb{N})) \quad (5.26) \\
 &= \{\perp_{\mathbf{VS}}, \downarrow_{\mathbf{VS}}\} \cup (F^n(\emptyset) \dot{\cup} (F^n(\emptyset) \times F^n(\emptyset)) \dot{\cup} (F^n(\emptyset) \times I(\mathbb{N})))
 \end{aligned}$$

- i. In $F^n(\emptyset) = X \dot{\cup} \{\perp_{VS}, \zeta_{VS}\}$ gelte die gegebene Berechnungsvorschrift für das Supremum (Induktionsvoraussetzung).
- ii. Bei den gegebenen Kreuzprodukten wird die Berechnung des Supremums des Kreuzprodukts durch eine koordinatenweise Erweiterung der Berechnung der Suprema der beiden Teile definiert (vergleiche [DP02]).
- iii. Durch Hinzunahme desselben neuen Topelementes ζ_{VS} und desselben neuen Bottomelementes \perp_{VS} zu allen disjunkten Teilmengen wird die Bestimmung des Supremums entsprechend erweitert (vergleiche Induktionsanfang).
- iv. Vereinigt man nun diese einzelnen Mengen, so wird in jeder die vorgegebene Bestimmung des Supremums durchgeführt. Da außerdem allen Teilmengen das neue Topelement ζ_{VS} und das neue Bottomelement \perp_{VS} gemeinsam ist (siehe Punkt iii), sind Supremum und Infimum für zwei beliebige Elemente der gesamten Menge $F^{n+1}(\emptyset)$ definiert. Das Ergebnis ist die Berechnungsvorschrift von Gleichung 5.24.

Aufgrund der Symmetrie der partiell geordneten Menge läßt sich die Gültigkeit der Berechnungsvorschrift für das Infimum analog zeigen.

⇒ Behauptung.

⇒ Satz 5.3.4. □

Typtraits als vollständige Verbände: Aufbauend auf die bisher eingeführten Begriffe sind Verband und vollständiger Verband definiert:

DEFINITION 5.3.5:

1. Eine nichtleere partiell geordnete Menge bildet einen VERBAND, falls für je zwei Elemente der Menge Infimum und Supremum existieren.
2. Falls für alle Teilmengen einer nichtleeren partiell geordneten Menge Infimum und Supremum existieren, dann bezeichnet man diese Menge als VOLLSTÄNDIGEN VERBAND.

Folgender Satz bildet die Grundlage des Typbestimmungsalgorithmus:

SATZ 5.3.2: Die Typtraits *ST*, *SP*, *VS*, *PU*, *BL* und *BM* bilden vollständige Verbände.

BEWEIS 5.3.3: Der Beweis gliedert sich in drei Teile. Zum einen werden alle Mengen betrachtet, welche mit dem Funktional G aus Gleichung 5.2 erzeugt wurden. Der zweite Teil behandelt die Menge aller Intervalle in \mathbb{N} . Der dritte Teil bezieht sich auf die durch das Funktional F aus Gleichung 5.8 definierte Menge der Wertemengen *VS*.

1. Fall (ST, PU und BM):

Gegeben: Sei Y eine beliebige nichtleere Menge. Seien $y, y_i, y_j \in Y$ mit $y_i \neq y_j$ für $i \neq j$. Sei $G(Y)$ definiert durch die Gleichung 5.2.

Behauptung: Dann ist $(G(Y), \supseteq)$ ein vollständiger Verband, wobei die in Gleichung 5.21 gegebene Vorschrift zur Bestimmung von Infimum und Supremum verwendet wird:

Beweis:

(a) $G(Y)$ als Verband:

i. $\forall \{y_i\}, \{y_j\} \in G(Y) : \{y_i\} \cap \{y_j\} = \emptyset \in G(Y) \Rightarrow \sup_{G(Y)} \{\{y_i\}, \{y_j\}\} \in G(Y)$

ii. $\forall \{y\} \in G(Y) : \{y\} \cap \emptyset = \emptyset \in G(Y) \Rightarrow \sup_{G(Y)} \{\{y\}, \emptyset\} \in G(Y)$

iii. $\forall \{y\} \in G(Y) : \{y\} \cap Y = \{y\} \in G(Y) \Rightarrow \sup_{G(Y)} \{\{y\}, Y\} \in G(Y)$

Der Beweis, daß für je zwei Elemente von $G(Y)$ auch deren Infimum in $G(Y)$ enthalten ist, läuft analog. Damit ist $(G(Y), \supseteq)$ ein Verband.

(b) $G(Y)$ als vollständiger Verband: Jede in $G(Y)$ enthaltene total geordnete Teilmenge (Kette) hat aufgrund der Konstruktionsvorschrift von $G(Y)$ höchstens 3 Elemente und somit höchstens die Länge 3. Da also $(G(Y), \supseteq)$ ein Verband ist, der keine unendlichen Ketten enthält, handelt es sich dabei um einen vollständigen Verband [DP02].

\Rightarrow Behauptung.

2. Fall (SP und BL):

Gegeben: Sei $I(\mathbb{N})$ die Menge aller Intervalle in \mathbb{N} .

Behauptung: Dann ist $(I(\mathbb{N}), \supseteq)$ ein vollständiger Verband.

Beweis: siehe [DP02].

3. Fall (VS):

Gegeben: Sei die Menge der Basistypen gegeben durch Gleichung 5.7 mit der in Abbildung 5.3 (e) vorgegebenen Ordnung ein vollständiger Verband. Außerdem ist $(I(\mathbb{N}), \supseteq)$ ein vollständiger Verband.

Behauptung: Dann ist auch VS ein vollständiger Verband.

Beweis:

(a) Induktionsanfang ($n = 1$): In $F^1(\emptyset) = F(\emptyset) = \mathbf{BT} \cup \{\perp_{\text{VS}}, \not\perp_{\text{VS}}\}$ kann aufgrund der Endlichkeit dieser Menge für jede Teilmenge Supremum und Infimum ermittelt werden. Damit ist $F^1(\emptyset)$ ein vollständiger Verband.

(b) Induktionsschritt ($n \longrightarrow n + 1$):

$$\begin{aligned}
& F^{n+1}(\emptyset) \\
&= F(F^n(\emptyset)) \\
&= \mathbf{BT} \cup \{\perp_{\mathbf{VS}}, \not\downarrow_{\mathbf{VS}}\} \cup F^n(\emptyset) \cup (F^n(\emptyset) \times F^n(\emptyset)) \cup (F^n(\emptyset) \times I(\mathbb{N})) \quad (5.27) \\
&= \{\perp_{\mathbf{VS}}, \not\downarrow_{\mathbf{VS}}\} \cup (F^n(\emptyset) \dot{\cup} (F^n(\emptyset) \times F^n(\emptyset)) \dot{\cup} (F^n(\emptyset) \times I(\mathbb{N})))
\end{aligned}$$

- i. $F^n(\emptyset) = X \dot{\cup} \{\perp_{\mathbf{VS}}, \not\downarrow_{\mathbf{VS}}\}$ ist ein vollständiger Verband (Induktionsbehauptung).
- ii. Kreuzprodukte vollständiger Verbände bilden vollständige Verbände [Hor04].
- iii. Durch Hinzunahme des neuen Top-elementes $\not\downarrow_{\mathbf{VS}}$ und des neuen Bottom-elementes $\perp_{\mathbf{VS}}$ zu den einzelnen vollständigen Verbänden wird die Vollständigkeit nicht zerstört.
- iv. Vereinigt man nun diese einzelnen vollständigen Verbände, so wird in jedem die vorgegebene Bestimmung des Supremums beziehungsweise Infimums wie bisher durchgeführt. Da alle diese einzelnen vollständigen Verbände das Top-element $\not\downarrow_{\mathbf{VS}}$ und das Bottom-element $\perp_{\mathbf{VS}}$ gemeinsam haben, bildet auch die Vereinigung einen vollständigen Verband.

\Rightarrow Behauptung.

\Rightarrow Satz 5.3.2. □

Interpretation der Ordnung auf den Typtraits: Die Ordnung, die auf den Typtraits definiert ist, kann dabei wie folgt interpretiert werden: Ist ein Interfacetyp x im Hasse-Diagramm unterhalb eines anderen Interfacetypen y angeordnet und existiert in diesem Hasse-Diagramm ein Pfad zwischen beiden Interfacetypen, dann ist x „weniger definiert“ als y . Betrachtet man beispielsweise den vollständigen Verband der Intervalle von natürlichen Zahlen. In diesem ist die Ordnung \leq durch \supseteq gegeben. Das heißt, ein Intervall A ist „kleiner gleich“ beziehungsweise „weniger definiert“ als ein anderes Intervall B , wenn A eine Obermenge von B darstellt ($A \supseteq B$). Der undefinierte Typ ist somit gegeben durch

$$\perp_{\mathbb{N}} = \mathbb{N} \quad (\text{undefinierter Typ}) \quad (5.28)$$

und stellt die Menge aller möglichen Werte dar. Der Errortyp ist der spezifischste Typ, die leere Menge:

$$\not\downarrow_{\mathbb{N}} = \emptyset \quad (\text{Errortyp}) \quad (5.29)$$

Bei den verbleibenden Typtraits sind undefinierter Typ und Errortyp analog definiert.

5.3.2.2 Polymorphismus

In diesem Interface-Typsystem kann PARAMETRISCHER POLYMORPHISMUS (vergleiche Abschnitt 2.3.2.1) ausgedrückt werden. So stellt zum Beispiel das Tupel $(\mathbf{string}, \perp_{\mathbf{VS}})$ einen Record dar, der sich aus einem String und einem nicht weiter spezifizierten Basistyp oder Aggregations-typ der Wertemenge \mathbf{VS} zusammensetzt. Die Elemente der Wertemenge $(\mathbf{number}, [2; 17])$ sind

Arrays von `number`, wobei die Arraygrößen zwischen 2 und 17 liegen. Bildverarbeitungsalgorithmen wie beispielsweise Kantendetektion (siehe Abbildung 5.1 (f)) können solche Arrays, deren Größe von dem jeweiligen Eingabebild abhängen, produzieren. Zusätzlich zu diesem parametrischen Polymorphismus ist auch `OVERLOADING` möglich. Dabei werden entsprechend den ausgewählten Interfacetypen unterschiedliche Algorithmen zum Beispiel bei der Addition ausgeführt.

5.3.3 Typconstraints

Betrachtet man die in Abschnitt 5.2 vorgestellten Typconstraints, so kann man folgende Arten unterscheiden:

- `GLEICHHEITSCONSTRAINTS` werden bei allen Typtraits verwendet.
- `MULTIPLIKATIONS-` UND `DIVISIONSCONSTRAINTS` treten bei physikalischen Einheiten auf.
- `MEHRDIMENSIONALE POLYNOME` werden bei allen Typtraits, die auf natürlichzahligen Intervallen basieren, eingesetzt. Signalmerkmale, die von dieser Art von Typtraits beschrieben werden, sind Abtastperioden, Arraygrößen und Blocklängen.

Weiterhin kann man zwischen `EXPLIZITEN` und `IMPLIZITEN` Typconstraints unterscheiden. Jede Datenflußkomponente definiert explizite Typconstraints zwischen den Signalmerkmalen ihrer Interfaces. Diese expliziten Typconstraints können zu jeder oben angegebenen Art gehören. Demgegenüber stehen die impliziten Typconstraints, die durch Kanten zwischen Interfaces bedingt werden. Diese impliziten Typconstraints sind immer Gleichheitsconstraints. Weitere Typconstraints ergeben sich zwischen Parametern und Interfacetypen beziehungsweise zwischen verschiedenen Parametern. Da diese Typconstraints in der gleichen Weise wie Typconstraints zwischen Interfacetypen behandelt werden, wird darauf nicht gesondert eingegangen.

5.4 Neues Typbestimmungsverfahren

Der im folgenden vorgestellte Typbestimmungsalgorithmus basiert auf dem in Abschnitt 5.3 eingeführten Modell der Interfacetypen. Von besonderer Bedeutung ist die in Abschnitt 5.3.2.1 vorgestellte Eigenschaft, daß sich alle Typtraits als vollständige Verbände darstellen lassen. Nach einem kurzen Überblick über den Algorithmus werden die Constraint-Propagationsregeln im Detail beschrieben. Anhand eines Beispiels wird die Vorgehensweise des Algorithmus erläutert, bevor auf inhärente Probleme und deren Lösung eingegangen wird. Abgerundet wird der Abschnitt durch eine formale Beschreibung des Typbestimmungsalgorithmus.

5.4.1 Überblick über den Typbestimmungsalgorithmus

Der Typbestimmungsalgorithmus wird bei jeder Änderung im Datenflußgraphen aufgerufen (vergleiche Abbildung 5.4). Falls neue Datenflußkomponenten oder Kanten eingefügt werden, wer-

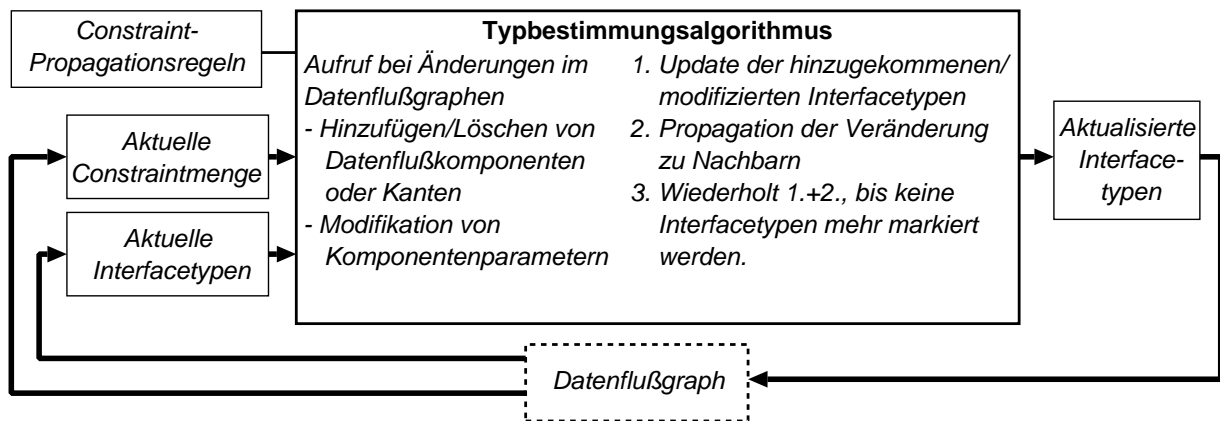


Abbildung 5.4: Typbestimmungsalgorithmus

den neue Typconstraints in die Constraintmenge aufgenommen. Der Typbestimmungsalgorithmus aktualisiert die Interfacetypen, indem Constraint-Propagationsregeln (siehe Abschnitt 5.4.2) auf die aktuellen Interfacetypen gemäß der Constraintmenge angewandt werden. Da die Aktualisierung eines Interfacetyps gegebenenfalls zu der Aktualisierung weiterer Interfacetypen führt, durchläuft der Typbestimmungsalgorithmus mehrere andere Interfaces – möglicherweise mehrmals –, bis sich keine Interfacetypen mehr ändern. Werden neue Datenflußkomponenten oder Kanten zum Datenflußgraphen hinzugefügt, dann können die vorher berechneten Interfacetypen als Ausgangspunkt der Aktualisierung verwendet werden. Das Verfahren ist somit INKREMENTELL. Löscht man Datenflußkomponenten oder Kanten, wird die Berechnung bei den anfänglichen Interfacetypen neu gestartet.

5.4.2 Constraint-Propagationsregeln

Unter einer CONSTRAINT-PROPAGATIONSREGEL versteht man eine Berechnungsvorschrift, die ausgehend von der aktuellen Typbelegung von Interfaces und einem Typconstraint die neue aktualisierte Typbelegung dieser Interfaces ermittelt. Ist x das betrachtete Interface. Dann gilt folgende Schreibweise:

- x^c : aktuelle Typbelegung,
- x^i : Zwischenergebnis,
- x^u : aktualisierte Typbelegung.

Die Constraint-Propagationsregel für ein Typconstraint, welches die GLEICHHEIT ZWEIER INTERFACETYPEN fordert, entspricht der Berechnung ihres Supremums. Angewandt auf die aktuellen Interfacetypen x^c und y^c berechnet das Gleichheitsconstraint

$$x = y \quad (5.30)$$

die aktualisierten Interfacetypen wie folgt⁵:

$$x^u = y^u = \sup\{x^c, y^c\}. \quad (5.31)$$

Das SUPREMUM von Startzeiten, Abtastperioden, Arraygrößen, physikalischen Einheiten, Blocklängen und farbigen Blockmarkierungen ist gegeben durch die Schnittmengenbildung der aktuellen Interfacetypen x^c und y^c (vergleiche Lemma 5.3.4). So ergibt beispielsweise $x_{\text{BL}}^c = [10; 20]$ und $y_{\text{BL}}^c = [15; 23]$ als Supremum $[15; 20]$. Das Supremum von Elementen der Wertemenge VS ist dabei komponentenweise definiert (vergleiche Lemma 5.3.4).

Tabelle 5.1 (a) zeigt die Constraint-Propagationsregeln für die Multiplikation von physikalischen Einheiten, die Addition von Intervallen und die Multiplikation von Intervallen. Dabei werden für jedes Interface die aktualisierten Interfacetypen ausgehend von den aktuellen Interfacetypen gegebenenfalls unter Verwendung von Zwischenergebnissen berechnet.

Das PRODUKT ZWEIER PHYSIKALISCHER EINHEITEN berechnet sich wie in folgendem Beispiel dargestellt:

$$\begin{array}{ll}
 x & = y \cdot z & \text{Typconstraint} \\
 x_{\text{PU}}^c & = \perp_{\text{PU}} & \text{Aktuelle Typbelegungen} \\
 y_{\text{PU}}^c & = (1, 1, -2, 0, 0, 0, 0) \\
 z_{\text{PU}}^c & = (1, 0, -1, 0, 0, 0, 0) \\
 x_{\text{PU}}^i & = y_{\text{PU}}^c + z_{\text{PU}}^c \\
 & = (1, 1, -2, 0, 0, 0, 0) + (1, 0, -1, 0, 0, 0, 0) \\
 & = (1 + 1, 1, -2 - 1, 0, 0, 0, 0) \\
 & = (2, 1, -3, 0, 0, 0, 0) \\
 x_{\text{PU}}^u & = \sup\{x_{\text{PU}}^c, x_{\text{PU}}^i\} & \text{Aktualisierte Typbelegung für } x \\
 & = \sup\{\perp_{\text{PU}}, (2, 1, -3, 0, 0, 0, 0)\} \\
 & = (2, 1, -3, 0, 0, 0, 0)
 \end{array} \quad (5.32)$$

Die Regeln für die DIVISION ZWEIER PHYSIKALISCHER EINHEITEN lassen sich daraus leicht ableiten.

Die Constraint-Propagationsregeln für ADDITION UND MULTIPLIKATION VON NATÜRLICHZÄHLIGEN INTERVALLEN sind in Tabelle 5.1 (b) und (c) dargestellt. Die Constraint-Propagationsregeln für SUBTRAKTION UND DIVISION können daraus leicht hergeleitet werden. MEHRDIMENSIONALE POLYNOME werden ausgewertet, indem die Constraint-

⁵Wenn aus dem Kontext ersichtlich ist, welche Art der Supremums- beziehungsweise Infimumsberechnung angewandt wird, werden in dieser Arbeit die jeweiligen Indizes zur Vereinfachung der betroffenen Ausdrücke weggelassen.

(a) Multiplikation physikalischer Einheiten	
Beispiel	Multiplikationskomponente: $x = y \cdot z$
Zwischen- ergebnis	$x_{\text{PU}}^i = \begin{cases} \perp_{\text{PU}} & \text{falls } y_{\text{PU}}^c = \perp_{\text{PU}} \text{ oder } z_{\text{PU}}^c = \perp_{\text{PU}} \\ y_{\text{PU}}^c + z_{\text{PU}}^c & \text{sonst} \end{cases}$
	$y_{\text{PU}}^i = \begin{cases} \perp_{\text{PU}} & \text{falls } x_{\text{PU}}^c = \perp_{\text{PU}} \text{ oder } z_{\text{PU}}^c = \perp_{\text{PU}} \\ x_{\text{PU}}^c - z_{\text{PU}}^c & \text{sonst} \end{cases}$
	z_{PU}^i analog
Aktualisiert	$x_{\text{PU}}^u = \sup\{x_{\text{PU}}^c, x_{\text{PU}}^i\}$, etc.
(b) Addition von Intervallen in \mathbb{N}	
Beispiel	Arraykonkatenation: $x = y \bullet z$ $x_{\text{VS}}^c = (\text{int}, [\alpha^c, \hat{\alpha}^c])$ $y_{\text{VS}}^c = (\text{int}, [\beta^c, \hat{\beta}^c])$ $z_{\text{VS}}^c = (\text{int}, [\gamma^c, \hat{\gamma}^c])$
Zwischen- ergebnis	$x_{\text{VS}}^i = (\text{int}, [\beta^c + \gamma^c, \hat{\beta}^c + \hat{\gamma}^c])$
	$y_{\text{VS}}^i = (\text{int}, [\alpha^c - \hat{\gamma}^c, \hat{\alpha}^c - \gamma^c])$
	$z_{\text{VS}}^i = (\text{int}, [\alpha^c - \hat{\beta}^c, \hat{\alpha}^c - \beta^c])$
Aktualisiert	$x_{\text{VS}}^u = \sup\{x_{\text{VS}}^c, x_{\text{VS}}^i\}$, etc.
(c) Multiplikation von Intervallen in \mathbb{N}	
Beispiel	$x = \text{FFT}(y)$; $x_{\text{SP}} = y_{\text{SP}} \cdot y_{\text{BL}}$ (vergleiche Abbildung 5.1 e) $x_{\text{SP}}^c = [\alpha^c, \hat{\alpha}^c]$ $y_{\text{SP}}^c = [\beta^c, \hat{\beta}^c]$ $y_{\text{BL}}^c = [\gamma^c, \hat{\gamma}^c]$
Zwischen- ergebnis	$x_{\text{SP}}^i = [\beta^c \cdot \gamma^c, \hat{\beta}^c \cdot \hat{\gamma}^c]$
	$y_{\text{SP}}^i = \left[\left[\frac{\alpha^c}{\hat{\gamma}^c}, \frac{\hat{\alpha}^c}{\gamma^c} \right] \right]$
	$y_{\text{BL}}^i = \left[\left[\frac{\alpha^c}{\hat{\beta}^c}, \frac{\hat{\alpha}^c}{\beta^c} \right] \right]$
Aktualisiert	$x_{\text{SP}}^u = \sup\{x_{\text{SP}}^c, x_{\text{SP}}^i\}$, etc.

Tabelle 5.1: Constraint-Propagationsregeln

Propagationsregeln sukzessive auf je zwei Interfaces angewandt werden. So liefert beispielsweise

$$\begin{aligned}
 x &= y^2 + 2 \cdot z && \text{Typconstraint} \\
 x_{\text{SP}}^c &= [10; 33], y_{\text{SP}}^c = [2; 3], z_{\text{SP}}^c = [4; 5] && \text{Aktuelle Typbelegungen} \\
 x_{\text{SP}}^i &= [2; 3] \cdot [2; 3] + [2; 2] \cdot [4; 5] \\
 &= [4; 9] + [2; 2] \cdot [4; 5] \\
 &= [4; 9] + [8; 10] \\
 &= [12; 19] && (5.33) \\
 x_{\text{SP}}^u &= \sup\{x_{\text{SP}}^c, x_{\text{SP}}^i\} \\
 &= \sup\{[10; 33], [12; 19]\} \\
 &= [10; 33] \cap [12; 19] \\
 &= [12; 19] && \text{Aktualisierte Typbelegung für } x .
 \end{aligned}$$

SATZ 5.4.1: Die vorgestellten Typpropagationsregeln bedingen für alle Interfaces x des zugehörigen Typconstraints

$$x^c \leq x^u \quad (5.34)$$

BEWEIS 5.4.1:

Gegeben: Sei x^c die aktuelle Typbelegung, x^i die als Zwischenergebnis berechnete Typbelegung und x^u die aktualisierte Typbelegung, welche das Ergebnis der Auswertung einer Typpropagationsregel darstellt.

Behauptung: Für eine beliebige Typbelegung gilt: $x^c \leq x^u$

Beweis: Da $x^u = \sup\{x^c, x^i\}$, folgt automatisch die Behauptung. □

5.4.3 Beispiel zur Typbestimmung

In dem Datenflußgraphen von Abbildung 5.5 (a) sind die Wertemengen der Interfaces bereits bis zu einem gewissen Grad definiert. Die Verbände zeigen die Interfacetypen für die Interfaces w (Kreis), x (Quadrat), y und z (Oval). Die Verbände für die Interfacetypen der Arrayelemente und für die Arraygrößen sind separat dargestellt, um die Anschaulichkeit zu erhöhen. Der Typ von w ist eine Menge von Arrays von Integerzahlen, wobei die Arraygröße entweder 1 oder 2 beträgt. In Abbildung 5.5 (b) wird eine Kante zwischen x und z eingefügt. Die Arraygrößen und Elementtypen werden aktualisiert, wie in den zugeordneten Verbänden gezeigt. Insbesondere werden die Interfacetypen von x und z zu $(\text{number}, [2; 3])$ geändert. Fügt man eine weitere Kante hinzu (siehe Abbildung 5.5 (c)), so werden die Interfacetypen letztlich auf $(\text{int}, [2; 2])$ gesetzt.

5.4.4 Formale Beschreibung des Typbestimmungsalgorithmus

Für eine formale Beschreibung des Typbestimmungsalgorithmus sind als erstes einige einführende Definitionen erforderlich. Dabei wird insbesondere zwischen TYPTRAITS MIT

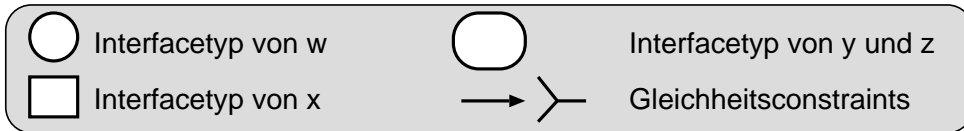
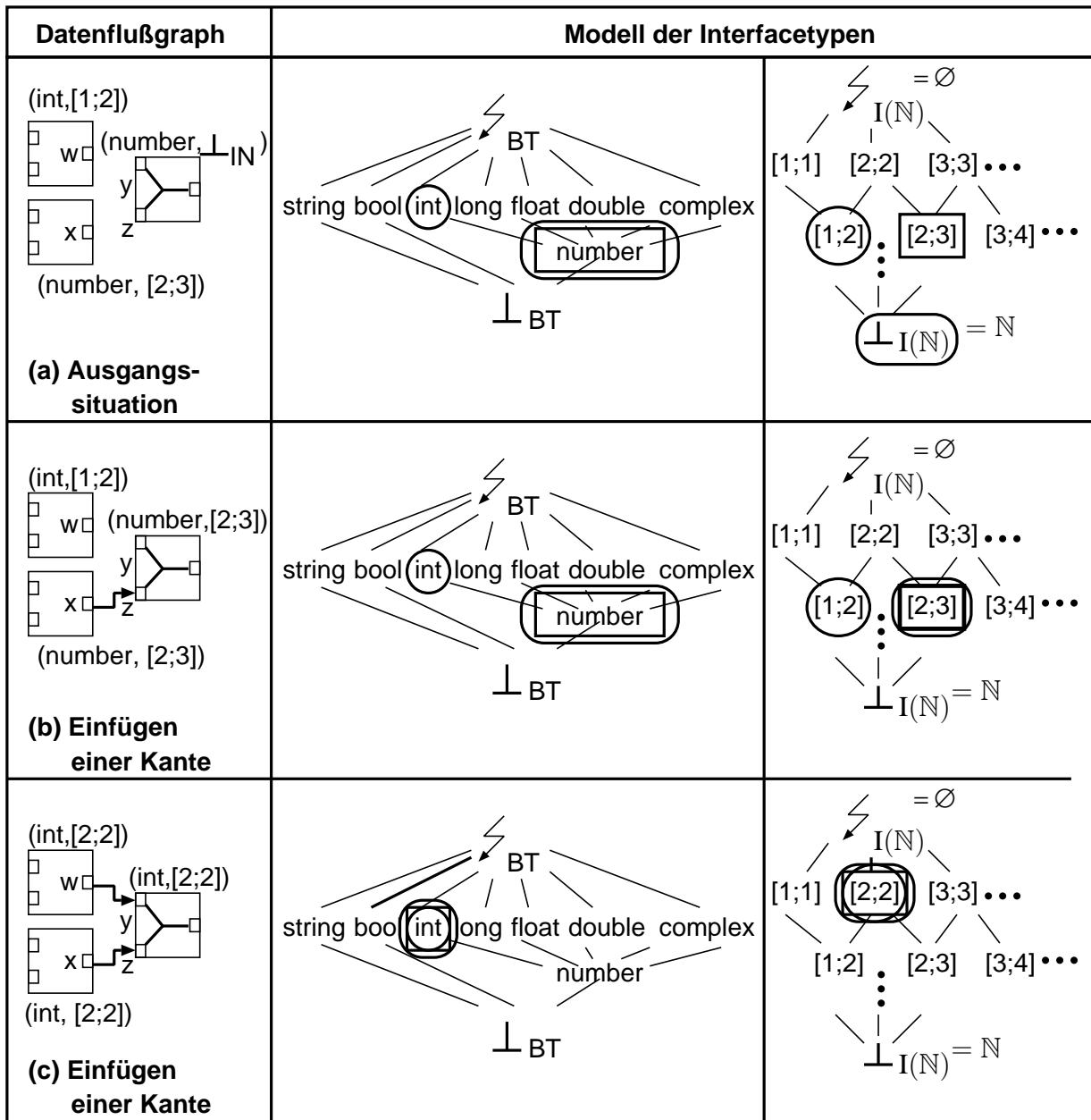


Abbildung 5.5: Beispiel zur Interface-Typbestimmung

GLEICHHEITSCONSTRAINTS und TYPTRAITS MIT ARITHMETISCHEN TYPCONSTRAINTS unterschieden.

DEFINITION 5.4.1:

1. ID : MENGE DER INTERFACEBEZEICHNER
2. DOM : TYPDOMÄNE (vergleiche Gleichung 5.1)
3. $TR = \{ST, SP, VS, PU, BL, BM\}$: TYPTRAITS
4. $TR_{Eq} = TR$: TYPTRAITS MIT GLEICHHEITSCONSTRAINTS
5. $TR_{Arith} = TR_{PU} \cup TR_{I(\mathbb{N})}$: TYPTRAITS MIT ARITHMETISCHEN TYPCONSTRAINTS
6. $TR_{PU} = \{PU\}$
7. $TR_{I(\mathbb{N})} = \{SP, AS, BL\}$
8. $TR_{united} = \bigcup_{tr \in TR} tr$: VEREINIGUNG ALLER TYPTRAITS
9. $X := ID \times TR$: MENGE DER ANNOTIERTEN IDENTIFIKATOREN
10. $OP = \{+, \cdot\}$: MENGE DER OPERATIONEN

Aufbauend auf diesen Definitionen wird die Typbelegung eines Interfaceidentifikators wie folgt definiert. Die dabei eingeführten Funktionen dienen zur formalen Beschreibung des Typbestimmungsalgorithmus.

DEFINITION 5.4.2: Eine TYPBELEGUNG ordnet einem Identifikator einen Typ zu:

$$\text{type} \begin{cases} ID \rightarrow DOM \\ id \mapsto (t_s, t_p, v, pu, bl, bm) . \end{cases} \quad (5.35)$$

Da häufig aber nicht der vollständige Typ, der alle sechs Signalmerkmale umfaßt, interessiert, sondern der einzelne Typtraut im Vordergrund steht, ist type überladen⁶:

$$\text{type} \begin{cases} X \rightarrow TR_{united} \\ (id, ST) \mapsto \Pi_1(\text{type}(id)) \text{ usw.} \end{cases} \quad (5.36)$$

Mit dieser Funktion kann man nun eine Funktion type_{tr} definieren, welche für jeden Identifikator die Typbelegung bezüglich eines Typtraits liefert:

$$\text{type}_{tr} \begin{cases} ID \rightarrow TR_{united} \\ id \mapsto \text{type}(id, tr) \end{cases} \quad (5.37)$$

⁶ Π_i steht in dieser Arbeit für die Projektion auf die i -te Komponente eines Tupels.

Aufbauend auf die bisherigen Definitionen wird die Menge der Typconstraints wie folgt beschrieben.

DEFINITION 5.4.3: Die Menge \mathbf{C} der auf den Typtraits definierbaren TYPCONSTRAINTS ist definiert als:

$$\mathbf{C} = \mathbf{C}_{Eq} \cup \mathbf{C}_{Arith} \quad (5.38)$$

Diese Definition baut auf den Mengen der Gleichheitsconstraints \mathbf{C}_{Eq} und der arithmetischen Typconstraints \mathbf{C}_{Arith} auf.

DEFINITION 5.4.4: Die GLEICHHEITSCONSTRAINTS sind durch folgende Gleichung gegeben:

$$\mathbf{C}_{Eq} = \{ \langle id, tr, id', tr' \rangle \mid id, id' \in ID, tr, tr' \in TR_{Eq} \} . \quad (5.39)$$

Dabei stellt $\langle id, tr, id', tr' \rangle$ eine andere Schreibweise des Typconstraints $type_{tr}(id) = type_{tr'}(id')$ dar.

DEFINITION 5.4.5: Die ARITHMETISCHEN TYPCONSTRAINTS sind in folgender Weise definiert:

$$\mathbf{C}_{Arith} = \mathbf{C}_{I(\mathbb{N})} \cup \mathbf{C}_{PU} \quad (5.40)$$

$$\mathbf{C}_{I(\mathbb{N})} = \{ \langle id, tr, id', tr', id'', tr'', op \rangle \mid id, id', id'' \in ID, tr, tr', tr'' \in TR_{I(\mathbb{N})}, op \in OP \} \quad (5.41)$$

$$\mathbf{C}_{PU} = \{ \langle id, tr, id', tr', id'', tr'', op \rangle \mid id, id', id'' \in ID, tr, tr', tr'' \in TR_{PU}, op \in OP \} \quad (5.42)$$

Dabei stellt $\langle id, tr, id', tr', id'', tr'', op \rangle$ eine andere Schreibweise des Typconstraints $type_{tr}(id) = type_{tr'}(id') \ op \ type_{tr''}(id'')$ dar.

DEFINITION 5.4.6: Für die heuristische Auswahl eines Elements aus der Menge der annotierten Interfaceidentifikatoren wird folgende Funktion verwendet:

$$\text{select} \begin{cases} P(X) \rightarrow X \\ M \mapsto x \text{ mit } x \in M . \end{cases} \quad (5.43)$$

DEFINITION 5.4.7: Um zu einem Element x in X alle Typconstraints aus \mathbf{C} zu ermitteln, in welchen der durch x beschriebene Identifikator mit dem zugehörigen Typtrait auftritt, wird die Funktion `determineConstraints()` eingesetzt:

$$\text{determineConstraints} \begin{cases} X \times P(\mathbf{C}) \rightarrow P(\mathbf{C}) \\ (x, \mathbf{C}') \mapsto \{ c \in \mathbf{C}' \mid c = \langle id, tr, id', tr' \rangle \wedge x \in \{ (id, tr), (id', tr') \} \\ \quad \vee c = \langle id, tr, id', tr', id'', tr'', op \rangle \\ \quad \wedge x \in \{ (id, tr), (id', tr'), (id'', tr'') \} \} \end{cases}$$

DEFINITION 5.4.8: Zur Ermittlung aller in einem Typconstraint $\langle id, tr, id', tr' \rangle$ beziehungsweise $\langle id, tr, id', tr', id'', tr'', op \rangle$ auftretenden annotierten Interfaceidentifikatoren aus X wird die Funktion $determineAnnotatedIdentifiers()$ eingesetzt:

$$determineAnnotatedIdentifiers \left\{ \begin{array}{l} \mathbf{C} \times P(\mathbf{X}) \rightarrow P(\mathbf{X}) \\ (\langle id, tr, id', tr' \rangle, X') \mapsto \{(id, tr), (id', tr')\} \cap X' \\ (\langle id, tr, id', tr', id'', tr'', op \rangle, X') \\ \mapsto \{(id, tr), (id', tr'), (id'', tr'')\} \cap X' \end{array} \right.$$

Den Kern des Typbestimmungsalgorithmus bildet die Aktualisierungsfunktion, welche aus einer gegebenen Typbelegung und einem Typconstraint eine neue Typbelegung für einen spezifizierten Interfaceidentifikator berechnet.

DEFINITION 5.4.9: Diese Funktion $update()$ ist überladen. Es existiert jeweils eine Variante für Gleichheits- und arithmetische Typconstraints.

$$update : \mathbb{N}_0 \times OP \times tr \times tr' \times tr'' \rightarrow TR_{united} \quad (5.44)$$

$$update : \mathbb{N}_0 \times tr \times tr' \times tr'' \rightarrow TR_{united} \quad (5.45)$$

Dabei findet die Aktualisierung der Typbelegung in $update()$ (siehe Gleichungen 5.44 und 5.45) mit Hilfe der in Tabelle 5.1 beziehungsweise in Abschnitt 5.4.2 angegebenen Constraint-Propagationsregeln statt. Die Zahl $i \in \mathbb{N}_0$, die als erster Parameter der jeweiligen Version von $update()$ übergeben wird, gibt dabei den Identifikator entsprechend seiner Position im zugehörigen Typconstraint an, für den der aktualisierte Typ berechnet werden soll.

Die beiden Varianten für Gleichheits- und arithmetische Typconstraints werden in der Funktion $update()$ genutzt.

$$update \left\{ \begin{array}{l} (X \times \mathbf{C} \times (ID \rightarrow DOM)) \rightarrow TR_{united} \\ ((id, tr), \langle id', tr', id'', tr'', id''', tr''', op \rangle, type) \\ \mapsto update(i, op, type_{tr'}(id'), type_{tr''}(id''), type_{tr'''}(id''')) \\ \text{mit } i = \begin{cases} 0 & \text{falls } id = id' \wedge tr = tr' \\ 1 & \text{falls } id = id'' \wedge tr = tr'' \\ 2 & \text{falls } id = id''' \wedge tr = tr''' \end{cases} \\ ((id, tr), \langle id', tr', id'', tr'' \rangle, type) \\ \mapsto update(i, type_{tr'}(id'), type_{tr''}(id'')) \\ \text{mit } i = \begin{cases} 0 & \text{falls } id = id' \wedge tr = tr' \\ 1 & \text{falls } id = id'' \wedge tr = tr'' \end{cases} \end{array} \right.$$

Beispiele zur Aktualisierung von Interfacetypen finden sich in Abschnitt 5.4.2.

Aufbauend auf den gegebenen Definitionen wird im folgenden der Typbestimmungsalgorithmus formal beschrieben.

ALGORITHMUS 5.4.1:

```

1  Algorithm typeResolution (C, X, Xmod, type) {
2      i := 0
3      Xmodi := Xmod
4      typei := type
5      while (Xmodi ≠ ∅) {
6          xmod := select(Xmodi)
7          Xmodi := Xmodi \ {xmod}
8          foreach (c ∈ determineConstraints(xmod, C)) {
9              Xtmp := determineAnnotatedIdentifiers(c, X)
10             typei+1(x) := { typei(x)           falls x ∉ Xtmp
                             update(x, c, typei)  sonst
11             Xmodi+1 := Xmodi ∪ {x ∈ Xtmp | typei(x) ≠ typei+1(x)}
12         }
13         i := i + 1
14     }
15     return typei
16 }
```

Ausgehend von einer Constraintmenge C , der Menge der annotierten Interfaceidentifikatoren X , der Menge von modifizierten annotierten Interfaceidentifikatoren X_{mod} und einer aktuellen Typbelegung $\text{type}()$ wird sukzessive die neue aktualisierte Typbelegung bestimmt. Dazu wird jeweils ein annotierter Interfaceidentifikator x_{mod} aus X_{mod}^i ausgewählt und entfernt. Danach werden mit Hilfe von $\text{determineConstraints}()$ alle Typconstraints bestimmt, in welchen x_{mod} auftritt. Für alle in diesen Typconstraints auftauchenden annotierten Interfaceidentifikatoren werden aktuelle Typbelegungen berechnet. Abschließend wird die Menge X_{mod} aktualisiert, wobei alle Interfaces berücksichtigt werden, bei denen sich die aktuelle Typbelegung von der vorherigen unterscheidet. Solange diese Menge X_{mod} nicht leer ist, werden die genannten Schritte wiederholt.

5.4.5 Probleme und Lösungen

Bei der Interfacetypbestimmung treten zwei Probleme hervor: das **PROBLEM DER TERMINIERUNG** und das **PROBLEM ZU SCHWACHER CONSTRAINT-PROPAGATIONSREGELN**.

Problem der Terminierung: Das Terminierungsproblem ergibt sich aus der Anwendung des Typbestimmungsalgorithmus auf Verbände mit unendlichen Ketten. Verbände mit unendlichen Ketten treten beispielsweise bei der Definition von Aggregationstypen wie Arrays oder Records auf (vergleiche Abbildung 5.3 (f)). Das Problem ist anhand zweier Beispiele in Abbildung 5.6

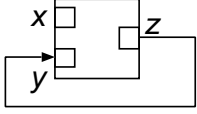
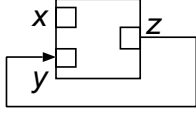
 Anfangs- belegung	$x_{VS}^c = y_{VS}^c = z_{VS}^c = \perp_{VS}$	 Anfangs- belegung	$x_{AS}^c = [1;1]; y_{AS}^c = z_{AS}^c = \perp_{I(\mathbb{N})}$
Typ- constraints	Datenflußkomponente: $z_{VS}^u = (x_{VS}^u, y_{VS}^u)$ Kante: $z_{VS}^u = y_{VS}^u$	Typ- constraints	Datenflußkomponente: $z_{AS}^u = x_{AS}^u + y_{AS}^u$ Kante: $z_{AS}^u = y_{AS}^u$
Rekursion	$y_{VS}^u = (x_{VS}^u, y_{VS}^u)$	Rekursion	$y_{AS}^u = x_{AS}^u + y_{AS}^u$
Algorithmus Iteration 1 Iteration 2 Iteration 3 etc.	$y_{VS}^u = (\perp_{VS}, \perp_{VS})$ $(y_{VS}^u)^u = (\perp_{VS}, (\perp_{VS}, \perp_{VS}))$ $((y_{VS}^u)^u)^u =$ $(\perp_{VS}, (\perp_{VS}, (\perp_{VS}, \perp_{VS})))$	Algorithmus Iteration 1 Iteration 2 Iteration 3 etc.	$y_{AS}^u = [1;1] + [0;\infty[= [1;\infty[$ $(y_{AS}^u)^u = [1;1] + [1;\infty[= [2;\infty[$ $((y_{AS}^u)^u)^u =$ $[1;1] + [2;\infty[= [3;\infty[$
(a) Wertemenge	(b) Intervalle natürlicher Zahlen		

Abbildung 5.6: Probleme bei der Typbestimmung (Terminierung)

((a) und (b)) veranschaulicht. Betrachtet man die Typconstraints im ersten Beispiel, die durch diesen Datenflußgraphen gegeben sind, so erkennt man, daß eine Rekursion existiert. Wendet man nun den Typbestimmungsalgorithmus auf diese Rekursion an, so sieht man, daß eine unendliche Folge von Zwischenergebnissen, die einer unendlich langen Kette im vollständigen Verband VS entsprechen, produziert wird und somit der Algorithmus niemals stoppt. Probleme dieser Art werden gelöst, indem ein Limit $n = 100$ für die Verschachtelungstiefe von Aggregationstypen eingeführt wird:

$$VS = \bigcup_{n=0}^{100} F^n(\emptyset) \quad (5.46)$$

Auf diese Weise ist es möglich, daß der Typbestimmungsalgorithmus, der den kleinsten Fixpunkt ermittelt, nach einer endlichen Zahl von Schritten terminiert (vergleiche dazu Satz 5.4.2).

Das zweite Beispiel zeigt, wie sich eine ähnliche Rekursion auf die Bestimmung der Arraygrenzen auswirkt. Es entsteht wiederum eine unendliche Folge von Zwischenergebnissen. Ein erster Lösungsansatz liegt in der Codierung natürlicher Zahlen als `unsigned int`, welche eine endliche Teilmenge der natürlichen Zahlen darstellen. Da die Bestimmung des Ergebnisses aber immer noch sehr lange dauern kann, ist durch die Überprüfung der Existenz solcher Zyklen eine Beschleunigung zu erreichen.

Mit den hier getroffenen Einschränkungen gilt folgende Aussage [DP02]:

SATZ 5.4.2: *Der Typbestimmungsalgorithmus terminiert in endlicher Zeit und liefert als Ergebnis den kleinsten Fixpunkt.*

BEWEIS 5.4.2:

Gegeben: Seien die Typtraits aus Abschnitt 5.3.2.1, welche vollständige Verbände bilden, gegeben. Sei die Funktion $\Phi_{C,X,X_{\text{mod}}}$ der Schleifenrumpf des Algorithmus 5.4.1:

$$\Phi_{C,X,X_{\text{mod}}} : \underbrace{(\text{ID} \rightarrow \text{DOM})}_{\text{Aktuelle Typen}} \rightarrow \underbrace{(\text{ID} \rightarrow \text{DOM})}_{\text{Aktualisierte Typen}} . \quad (5.47)$$

Behauptung: Dann liefert der Algorithmus den kleinsten Fixpunkt und terminiert in endlicher Zeit.

Beweis:

1. Die Typdomäne stellt als Kreuzprodukt der vollständigen Verbände der Typtraits einen vollständigen Verband dar [Hor04].
2. Jeder vollständige Verband ist eine CPO [DP02].
3. Die Typtraits sind mit den in diesem Abschnitt genannten Einschränkungen endlich.
4. Endliche Verbände erfüllen die Ascending Chain Condition [DP02].
5. Die Monotonie von $\Phi_{C,X,X_{\text{mod}}}$ ergibt sich aus der Monotonie der Constraint-Propagationsregeln (siehe Satz 5.4.1).
6. Eine monotone Abbildung $G : X \rightarrow Y$ ist stetig, wenn X die Ascending Chain Condition erfüllt [DP02]. Damit ist auch die durch den Schleifenrumpf definierte Abbildung $\Phi_{C,X,X_{\text{mod}}}$ stetig.
7. Damit gilt aufgrund des CPO-Fixpunkt-Theorems I (siehe Abschnitt 4.3.4), daß der kleinste Fixpunkt existiert und mittels

$$\bigsqcup_{n \geq 0} \Phi_{C,X,X_{\text{mod}}}^n(\perp) . \quad (5.48)$$

berechnet werden kann.

8. Aufgrund der Endlichkeit der Typtraits gilt $n < \infty$. Somit terminiert der Algorithmus nach endlich vielen Schritten.

⇒ Satz 5.4.2

□

Die Zeitkomplexität des Typbestimmungsalgorithmus verhält sich wie im folgenden Satz dargestellt.

SATZ 5.4.3: Sei $N \in \mathbb{N}$ die maximale Tiefe der verwendeten endlichen vollständigen Verbände und \mathbf{C} die Menge der Typconstraints. Dann beträgt die Zeitkomplexität

$$O(3 \cdot N \cdot |\mathbf{C}|) . \quad (5.49)$$

BEWEIS 5.4.3:

Gegeben: Sei $N < \infty$ die maximale Tiefe der verwendeten vollständigen Verbände. Sei $|\mathbf{C}|$ die Anzahl der Typconstraints der gegebenen Constraintmenge \mathbf{C} . Sei `typeResolution()` der in Algorithmus 5.4.1 gegebene Algorithmus zur Typbestimmung.

Behauptung: Dann ist die Zeitkomplexität des Algorithmus gegeben durch $O(3 \cdot N \cdot |\mathbf{C}|)$.

Beweis:

1. Jedes Typconstraint $c \in \mathbf{C}$ beinhaltet maximal 3 Interfaceidentifikatoren (vergleiche Definitionen 5.4.4 und 5.4.5).
2. Nur wenn sich die Belegung eines Interfaceidentifikators ändert, wird ein Typconstraint propagiert (vergleiche Algorithmus 5.4.1).
3. Die Belegung eines Identifikators kann maximal N mal modifiziert werden aufgrund der Monotonie der Typpropagationsregeln (vergleiche Satz 5.4.1).

⇒ Satz 5.4.3. □

Problem zu schwacher Constraint-Propagationsregeln: Der Fixpunkt, der von dem Typbestimmungsalgorithmus gefunden wurde, erfüllt zwar die gegebenen Constraint-Propagationsregeln. Diese Constraint-Propagationsregeln sind aber nicht stark genug. In Abbildung 5.7 ist die passende Lösung für Interface y der Wert \perp_{PU} und nicht \downarrow_{PU} , wie vom Typbestimmungsalgorithmus zurückgeliefert. Dieses Problem löst sich in der Regel während des weiteren Entwurfs durch die Einfügung zusätzlicher Datenflußkomponenten. Falls dennoch am Ende der Designphase unspezifizierte Interfacetypen verbleiben, wie zum Beispiel \perp_{PU} , dann muß der Benutzer entweder einige Parameter im Datenflußgraphen ändern oder es muß ein Algorithmus gestartet werden, der die verbleibenden Interfacetypen daraufhin untersucht, ob sie eine gültige Lösung darstellen. Der Typbestimmungsalgorithmus ist ein Kompromiß zwischen benötigter Berechnungszeit und Genauigkeit. Meist können die Interfacetypen aufgelöst werden. In seltenen Fällen muß der Benutzer eingreifen. Diese Problematik existiert auch für mehrwertig-

Anfangsbelegung	$w_{PU}^c = (1, 0, 0, 0, 0, 0, 0)$ $x_{PU}^c = y_{PU}^c = z_{PU}^c = \perp_{PU}$
Typconstraints	Datenflußkomponente: $z_{PU}^u = x_{PU}^u \cdot y_{PU}^u$ Kanten: $x_{PU}^u = w_{PU}^u; y_{PU}^u = z_{PU}^u$
Ergebnis des Algorithmus	$w_{PU}^u = (1, 0, 0, 0, 0, 0, 0)$ $x_{PU}^u = (1, 0, 0, 0, 0, 0, 0)$ $y_{PU}^u = \perp_{PU}$ $z_{PU}^c = \perp_{PU}$
Physikalische Einheiten	

Abbildung 5.7: Probleme bei der Typbestimmung (Zu schwache Constraint-Propagationsregeln)

ge Polynome. Daß dieses Problem systeminhärent ist, zeigen folgende Aussagen.

PROBLEM 5.4.1: HILBERTS 10. PROBLEM: *Hat ein gegebenes Polynom in mehreren Veränderlichen und natürlichzahligen Koeffizienten (auch diophantische Gleichung genannt) eine Nullstelle?*

Dieses Problem ist unentscheidbar, wie folgender von Matiyasevich im Jahr 1970 bewiesene Satz zeigt [Mat93].

SATZ 5.4.4: HILBERTS 10. PROBLEM IST NICHT LÖSBAR. *Es gibt keinen Algorithmus, der für jede diophantische Gleichung die Frage beantworten kann, ob diese Gleichung Lösungen in den natürlichen Zahlen besitzt oder nicht.*

Schränkt man das Problem, eine Lösung für eine gegebene Menge von Typconstraints zu finden, ein, indem man nur Ausdrücke erlaubt, die auf der Addition basieren, so gilt immer noch [HMU02]:

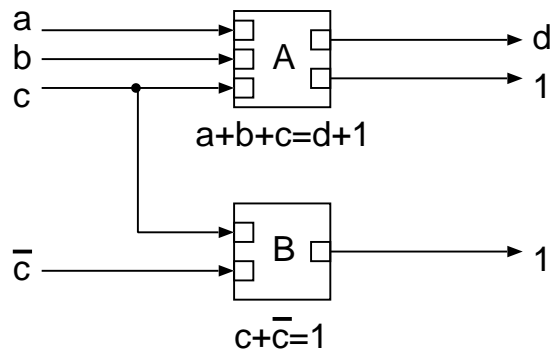


Abbildung 5.8: Beispiel für Additionsconstraints

SATZ 5.4.5: Das Problem der Bestimmung der Interfacetypen unter ausschließlicher Verwendung von Additionsconstraints ist NP-hart.

BEWEIS 5.4.4:

Gegeben: Seien a, b, c beliebige Boolesche Variablen. \bar{c} sei die Negation von c .

Behauptung: Beliebige Boolesche Terme bestehend aus drei Literalen lassen sich in dem Interface-Typsystem nachbilden.

Beweis:

$$1. a \vee b \vee c \Leftrightarrow a + b + c \geq 1 \Leftrightarrow \exists d \in \mathbb{N}_0 : a + b + c = 1 + d.$$

$$2. \bar{c} \Leftrightarrow \exists c \in \mathbb{N}_0 : \bar{c} = 1 - c \Leftrightarrow \exists c \in \mathbb{N}_0 : \bar{c} + c = 1$$

Abbildung 5.8 veranschaulicht dies.

\Rightarrow Behauptung.

Damit läßt sich 3SAT auf das Problem der Bestimmung der Interfacetypen reduzieren.

\Rightarrow Satz 5.4.5 □

5.4.6 Typkonvertierung

Einige Typfehler können mittels TYPKONVERTIERUNG gelöst werden. So werden beispielsweise bei den Signalmerkmalen Startzeitpunkt und farbige Blockmarkierung, die ja symbolisch behandelt werden (siehe Abschnitt 5.3.2.1), im Falle der Ungleichheit Konvertierungskomponenten (Adapterkomponenten; vergleiche Abschnitt 4.3.2) eingefügt. Für die Konvertierung der Wertemengen existieren in der Literatur bereits entsprechende Algorithmen (vergleiche Abschnitt 2.3.2.2).

5.5 Innovative Aspekte

Die innovativen Aspekte dieses neuen Interface-Typsysteams, welches auf einem NEUEN MODELL DER INTERFACETYPEN und einem neuen TYPBESTIMMUNGSSALGORITHMUS aufbaut, lassen sich in folgenden Punkten zusammenfassen:

- **Erweiterung der durch Typsysteme überprüften Eigenschaften:** Durch die Hinzunahme einer Reihe neuer Signalmerkmale, die den Definitionsbereich, den Wertebereich und den Datentransport umfassend beschreiben, ist es möglich, eine ganze Reihe zusätzlicher Typconstraints zu überprüfen. Diese Typconstraints stellen die Wohldefiniertheit der in den Datenflußkomponenten gekapselten Algorithmen sicher. Beispiele für vermiedene Fehler sind:
 - Speicherüberlauf aufgrund nicht zusammenpassender Abtastperioden
 - Laufzeitfehler aufgrund nicht zusammenpassender Blocklängen
 - Verknüpfung von physikalischen Signalen mit nicht zusammenpassenden Einheiten
 - Klassische Typfehler
- **Parametrischer Polymorphismus und Overloading:** Diese zwei Arten von Polymorphismus erhöhen die Wiederverwendbarkeit von Datenflußkomponenten, da diese sich damit leicht an eine Vielzahl unterschiedlicher von dem umgebenden Datenflußgraphen bereitgestellter Eingaben und geforderter Ausgaben anpassen.
- **Komplexere Typconstraints:** Im Gegensatz zum bisherigen Stand der Technik (siehe Abschnitt 2.3) werden nicht nur neue Signalmerkmale berücksichtigt, sondern es können auch deutlich komplexere Typconstraints formuliert werden. Anstelle von einfachen Ungleichungen treten beispielsweise mehrdimensionale Polynome. Die dabei gegebenenfalls entstehenden Probleme und deren Lösungen wurden ausführlich in Abschnitt 5.4.5 behandelt.
- **Berücksichtigung der Komponentenparameter:** Bei der Typbestimmung werden auch Komponentenparameter mit berücksichtigt, wenn sich deren Typen durch die vorgegebenen Signalmerkmale beschreiben lassen.
- **Entwurfsbegleitende Typprüfung:** Das vorgestellte Verfahren kann entwurfsbegleitend eingesetzt werden, was die hergeleitete Zeitkomplexität (siehe Satz 5.4.3) und die Testergebnisse (siehe Abschnitt 8.3) belegen.
- **Inkrementelle Typprüfung:** Der präsentierte Algorithmus ist inkrementell, da er bei Hinzunahme von Datenflußkomponenten beziehungsweise Kanten die Ergebnisse vorausgegangener Checks weiterverwendet.
- **Leichte Erweiterbarkeit:** Außerdem kann das vorgestellte Typprüfungsverfahren leicht erweitert werden, indem man zu jedem neuen Merkmal die zulässigen Typconstraints und Constraint-Propagationsregeln aufstellt.

Die vorgestellten Eigenschaften erlauben es, zur Entwurfszeit viele Fehler zu vermeiden. So kann man zusammenfassend feststellen, daß dieses Interface-Typsystem die Beschreibung zukünftiger Herausforderungen im Bereich der Typsysteme erfüllt [Ode96]: „Increasing the expressiveness of typed programming languages, broadening the information carried by types, and making type systems amenable to user definition and customization.“

Kapitel 6

Model Checking

Vertrauen ist gut, Kontrolle ist besser. [Wladimir Iljitsch Lenin]

Dieses Kapitel beschreibt einen neuartigen Ansatz des Model Checkings für datenflußorientierte eingebettete Systeme, der auf das in Kapitel 3 vorgestellte Signalmodell und das darauf aufbauende Komponentenmodell (siehe Kapitel 4) zugeschnitten ist. Auf eine Liste der Anforderungen an das neue Model-Checking-Verfahren folgt das Beispiel eines Kommunikationsprotokolls. Es wird ein neues Modell für Kommunikationsprotokolle basierend auf Fifomaten eingeführt. Daran schließt sich die detaillierte Beschreibung des Model-Checking-Verfahrens an. Dieses Verfahren unterscheidet dabei zwischen einem Kompositions- und einem Simulationsmodus. Den Abschluß bildet eine Zusammenfassung der wichtigsten innovativen Aspekte.

6.1 Anforderungen

Die Anforderungen an das neue Modell der Kommunikationsprotokolle und das neue Model-Checking-Verfahren lauten:

- **ENTWURFSBEGLEITENDE ÜBERPRÜFUNG:** Das Model Checking soll – soweit es die Komplexität des betrachteten Problems erlaubt – entwurfsbegleitend durchgeführt werden. Dies geschieht, analog zur Interface-Typprüfung (siehe Abschnitt 5.1), jedesmal, wenn der Entwerfer den Datenflußgraphen modifiziert. Modifikationen sind beispielsweise Einfügen oder Löschen von Datenflußkomponenten beziehungsweise von Kanten.
- **INKREMENTELLE VORGEHENSWEISE:** Das Model-Checking-Verfahren sollte inkrementell sein. Das heißt, Ergebnisse vorheriger Überprüfungen sollen soweit wie möglich wiederverwendet werden. Das Ziel ist, Mehrfachberechnungen, die unnötig Zeit kosten, zu vermeiden.
- **KOMMUNIKATIONSPROTOKOLLE AUF STRÖMEN GEFÄRBTER TOKEN:** Das Model-Checking-Verfahren soll die Kompatibilität der von den jeweiligen Datenflußkomponenten

implementierten Kommunikationsprotokolle auf Strömen gefärbter Token überprüfen. Dabei sollen insbesondere folgende Fragen, soweit die Berechnungskomplexität dies zuläßt, geklärt werden (vergleiche Abschnitt 1.1):

- Existiert ein ZYKLISCHER SCHEDULE?
- Treten DEADLOCKS auf?
- Wie groß ist der SPEICHERBEDARF für das Gesamtprogramm beziehungsweise für die einzelnen Fifos?

6.2 Beispiel eines Kommunikationsprotokolls

In Abbildung 6.1 ist die Abarbeitung zweier Eingabeströme durch eine Colored-Merge-Komponente (CMerge) dargestellt. Da bei gleichzeitigem Vorhandensein von Starttoken in den beiden Eingangsfifos eine NICHTDETERMINISTISCHE AUSWAHL getroffen wird, sind in der Abbildung zwei alternative Abarbeitungsschrittfolgen wiedergegeben. Dabei wird von CMerge, wie in Abschnitt 4.3.3.3 beschrieben, die Segmentstruktur der Eingabesignale beibehalten. Aus diesem Grund werden immer alle gefärbten Token eines Signalsegmentes sukzessive übertragen, bevor ein anderes Signalsegment behandelt wird. Dies kann sogar zur Folge haben, daß CMerge an dem einen Eingang auf weitere Token eines Signalsegmentes wartet, obwohl am anderen Eingang Token anliegen.

Dieses Beispiel der Abarbeitung von Eingabeströmen zeigt die Notwendigkeit einer allgemeinen Modellierungsmethodik, die alle möglichen Abläufe beschreiben kann. Dabei soll diese Modellierungsmethodik sowohl die DETERMINISTISCHEN als auch die NICHTDETERMINISTISCHEN Datenflußkomponenten aus Kapitel 4 auf der EBENE DER TOKENMASCHINEN beschreiben können. Von der Ebene der gekapselten Algorithmen (vergleiche Abbildung 4.4) wird abstrahiert, da diese für die Beschreibung des Kommunikationsverhaltens irrelevant ist. Wichtige Eigenschaften, welche diese Modellierungsmethodik unterstützen soll, sind beispielsweise:

- Gefärbte Token
- Fifos mit Schreib- und Leseoperationen
- Unbegrenzte Fifokapazitäten
- Rückkopplungen
- Nichtdeterminismus

In folgendem Abschnitt werden ausgehend von diesen Beobachtungen eine neue Modellierungsmethodik für das Kommunikationsverhalten von Datenflußkomponenten und dazu passende Analyseverfahren vorgestellt.

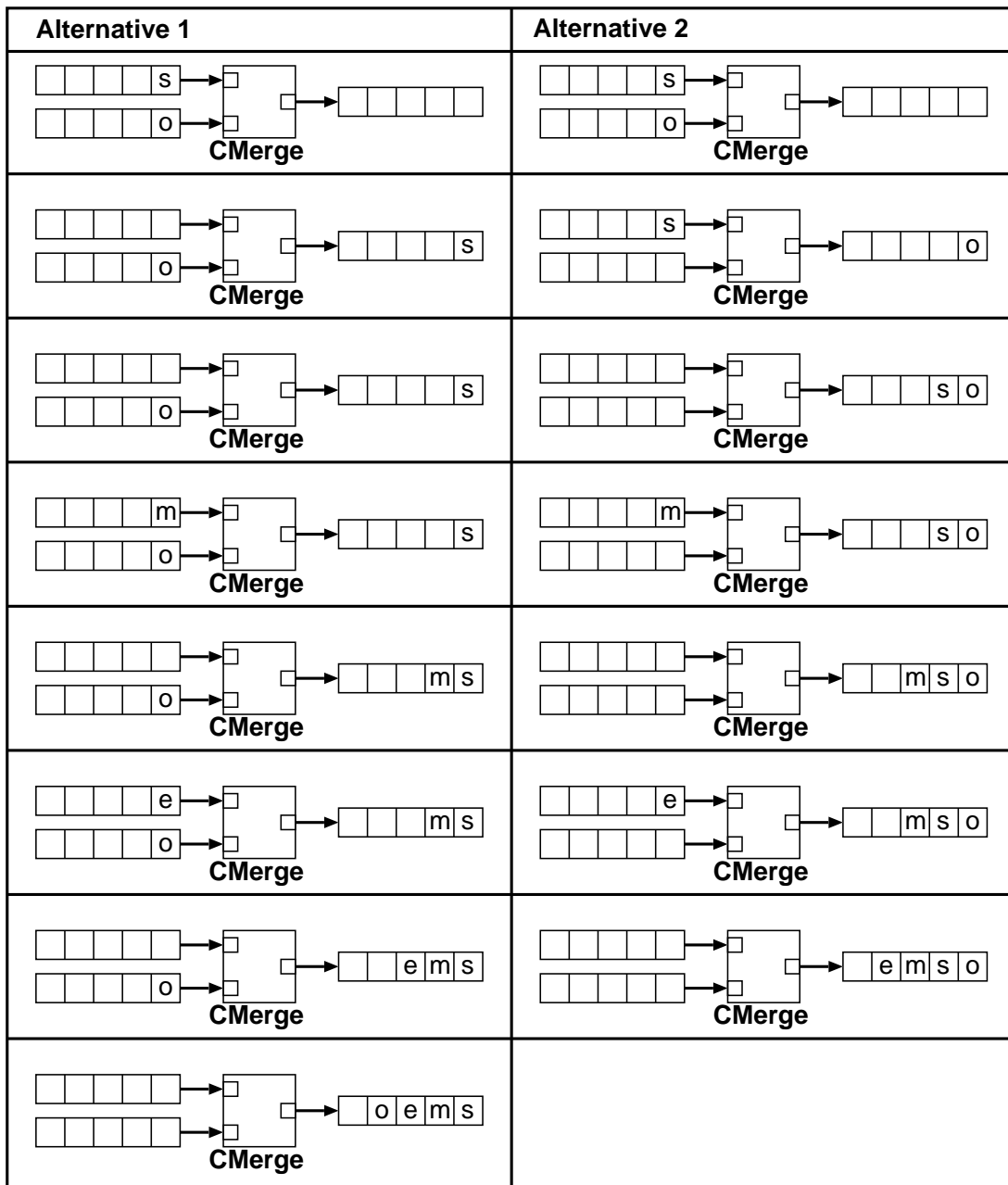


Abbildung 6.1: Beispielhafte Abarbeitung zweier Eingabeströme durch CMerge

6.3 Neues Modell der Kommunikationsprotokolle

Aufbauend auf eine Reihe einführender Definitionen werden Fifomaten als Modellierungsmethodik für Kommunikationsprotokolle von Datenflußkomponenten vorgestellt. Fifomaten stellen dabei eine Erweiterung des in Abschnitt 2.4.2.2 vorgestellten Modells der Communicating Finite State Machines dar.

6.3.1 Definitionen

In diesem Abschnitt werden einige grundlegende Definitionen eingeführt, die für das Verständnis dieses Kapitels wichtig sind:

DEFINITION 6.3.1:

1. Ein NACHRICHTENALPHABET M entspricht einer endlichen Menge von gefärbten Token (vergleiche Definition 3.3.4 der vereinfachten Tokenmenge).
2. Ein WORT stellt eine in der Regel endliche Sequenz von Token aus einem Nachrichtenalphabet M dar (vergleiche Definition 3.3.7 der vereinfachten Strommenge):

$$m \in M^* . \quad (6.1)$$

Das leere Wort wird mit ε bezeichnet¹.

3. Eine FIFO (KANAL) entspricht einem Tripel

$$c = (M_c, n_c, m_c) , \quad (6.2)$$

wobei M_c das Nachrichtenalphabet darstellt, $n_c \in \mathbb{N} \cup \{\infty\}$ die zugehörige Fifokapazität wiedergibt und $m_c \in M_c^*$ das INITIALISIERUNGSWORT der Fifo repräsentiert.

6.3.2 Fifomaten

Der Terminus FIFO-AUTOMAT wird in der Literatur für eine Reihe von Modellierungsverfahren verwendet, die Automaten mit Fifos kombinieren und sich dabei beispielsweise in der Modellierung von atomaren Operationen, Hinzunahme von Zählvariablen beziehungsweise in von der Fifo-Semantik abweichenden Zugriffsstrategien auf die Nachrichtenkanäle erheblich unterscheiden (vergleiche auch Abschnitt 2.4.2; [SU96, AABJ04, FM97, FS01, FIS00, BH99, Hol91, Hol04, VSVA04a, VSVA04b]). Zur Abgrenzung von diesen Definitionen wird im folgenden der neue Begriff FIFOMAT als Verkürzung von Fifo-Automat eingeführt. Dabei stellen Fifomaten eine Erweiterung von Communicating Finite State Machines dar [Hol91].

DEFINITION 6.3.2: Ein FIFOMAT ist wie folgt definiert:

$$\begin{array}{ll}
 F & = ((Q_m, Q_t), q_0, (C_i, C_o, C_h), T) \\
 Q_m, Q_t & \text{Mengen von HAUPT- und TRANSIENTEN ZUSTÄNDEN} \\
 Q & = Q_m \dot{\cup} Q_t, \text{ Menge von Zuständen} \\
 q_0 \in Q_m & \text{Anfangszustand} \\
 C_i, C_o, C_h & \text{Mengen von Eingabe-, Ausgabe- und verdeckten Fifos} \\
 C & = C_i \dot{\cup} C_o \dot{\cup} C_h, \text{ Menge der Fifos} \\
 M & = \bigcup_{c \in C} M_c, \text{ Nachrichtenalphabet} \\
 T & \subseteq Q \times ((\{?\} \times (C \rightarrow M^*)) \cup \\
 & \quad (\{!\} \times (C \rightarrow M^*))) \times Q, \\
 & \text{Menge von Transitionen}
 \end{array} \quad (6.3)$$

¹In Abbildungen wird für das leere Wort zugunsten einer besseren Visualisierung statt ε entweder „-“ verwendet oder nichts eingetragen.

Ein Fifomat wird als KANONISCH bezeichnet, falls

$$C_h = \emptyset . \quad (6.4)$$

DEFINITION 6.3.3:

1. **TRANSITIONEN:** Die Relation T liefert für einen aktuellen Zustand q und eine Aktion a den Nachfolgerzustand \hat{q} . Eine Aktion a ist dabei entweder eine Eingabe- oder Leseaktion, welche durch „?“ markiert ist, oder eine Ausgabe- oder Schreibaktion, die mittels „!“ gekennzeichnet ist. Mit Hilfe einer Funktion werden die zu den Fifos gehörenden zu lesenden oder schreibenden Wörter angegeben. Werden Fifos durch eine Aktion nicht verändert, so wird durch diese Aktion das leere Wort in die jeweilige Fifo geschrieben beziehungsweise aus dieser Fifo gelesen. Für eine anschauliche Darstellung beispielsweise in den Abbildungen werden aber als Aktionen Ausdrücke der Form $c?m$ oder $c!m$, wobei c die Fifo und m die zu lesende oder schreibende Nachricht bezeichnet, verwendet.
2. Eine Aktion a ist AUSFÜHRBAR, falls
 - a die leere Aktion darstellt,
 - a eine Eingabeaktion ist und m ein Präfix des entsprechenden Fifoinhaltes ist
 - a eine Ausgabeaktion bildet und der aktualisierte Fifoinhalt nicht die Kapazität n_c der entsprechenden Fifo überschreitet:

$$n_c \leq \text{size}(m_c) + \text{size}(m) . \quad (6.5)$$

3. Während der AUSFÜHRUNG werden zuerst alle Fifomaten und Fifos in ihren Anfangszustand versetzt. Anschließend wird, solange es möglich ist, ein beliebiger Fifomat i mit einer aktuell ausführbaren Transition ausgeführt. Andernfalls endet die Ausführung.
4. Jeder Fifomat definiert Folgen von Ein- und Ausgabeaktionen, die ein KOMMUNIKATIONSprotokoll repräsentieren. Damit dieses Kommunikationsprotokoll sich für endlose Ausführung eignet, darf der entsprechende Fifomat keine Sackgassen enthalten.

Es gibt Sequenzen von Lese- und Schreiboperationen, die atomar ausgeführt werden müssen, um beispielsweise das Verhalten von Datenflußkomponenten des Typs Colored SDF (vergleiche Abschnitt 4.3.3.1) nachbilden zu können. In einem Fifomaten ist in einem atomaren Schritt die Menge der Leseoperationen von der Menge der Schreiboperationen durch einen transienten Zustand getrennt. Das ist notwendig, um RÜCKKOPPLUNGSKANTEN in Datenflußgraphen modellieren zu können (vergleiche Abschnitt 2.2.3). Denn ansonsten wäre es möglich, in derselben atomaren Operation in eine Fifo zu schreiben und aus dieser zu lesen. Dies würde zu Problemen wie unterschiedlichen Ergebnissen bei unterschiedlichen Ausführungsreihenfolgen von Lese- und Schreiboperationen führen, was eine Nachbildung des Verhaltens von gefärbten SDF-Komponenten erheblich erschweren würde.

6.3.3 Kommunikationsprotokolle

Betrachtet man das in Kapitel 3 eingeführte Modell eines physikalischen Signals, so wurden zwei verschiedene Abstraktionsebenen basierend auf der Strommenge S und der vereinfachten Strommenge S_s vorgestellt (vergleiche Abschnitt 3.3.3). Die detaillierte Beschreibung mittels der Strommenge S ist für die Bestimmung der Interfacetypen (vergleiche Kapitel 5) grundlegend. Auf die Beschreibung des Kommunikationsverhaltens mittels Fifomaten ist die abstraktere Beschreibung mittels der vereinfachten Strommenge S_s zugeschnitten. Richtet man den Blick auf das in Kapitel 4 vorgestellte formale Modell von Datenflußkomponenten, so basiert dieses auf der detaillierteren Beschreibung der Strommengen. Man kann aber aus jeder dieser Beschreibungen eine VEREINFACHTE DARSTELLUNG basierend auf den vereinfachten Strommengen herleiten. Zum Beispiel sieht eine vereinfachte Darstellung csdf_s von csdf (vergleiche Abschnitt 4.3.3.1) wie folgt aus:

$$\text{csdf}_s \begin{cases} S_s^k \rightarrow S_s^n & (k, n \in \mathbb{N}) \\ x \bullet r \mapsto f_{\text{prefix}_s}(x) \bullet \text{csdf}_s(r) \\ x \bullet r \mapsto \varepsilon_S \text{ sonst} \end{cases} \quad (6.6)$$

$$f_{\text{prefix}_s} \begin{cases} P_{S_s^k} \rightarrow P_{S_s^n} \\ \underbrace{(\text{se}, \text{se}, \dots, \text{se})}_{k \text{ mal}} \mapsto \underbrace{(\text{sme}, \text{sme}, \dots, \text{sme})}_{n \text{ mal}} \\ \underbrace{(\text{sm}, \text{sm}, \dots, \text{sm})}_{k \text{ mal}} \mapsto \underbrace{(\text{smm}, \text{smm}, \dots, \text{smm})}_{n \text{ mal}} \\ \underbrace{(\text{mm}, \text{mm}, \dots, \text{mm})}_{k \text{ mal}} \mapsto \underbrace{(\text{mmm}, \text{mmm}, \dots, \text{mmm})}_{n \text{ mal}} \\ \underbrace{(\text{me}, \text{me}, \dots, \text{me})}_{k \text{ mal}} \mapsto \underbrace{(\text{mme}, \text{mme}, \dots, \text{mme})}_{n \text{ mal}} \end{cases} \quad (6.7)$$

Mit $P_{S_s^k}$ und $P_{S_s^n}$ werden die Mengen der endlichen Präfixe von S_s^k beziehungsweise S_s^n angegeben, für welche die Funktion f_{prefix_s} definiert ist. Das sind die Mengen:

$$\begin{aligned} P_{S_s^k} &= \{(\text{se}, \text{se}, \dots, \text{se}), (\text{sm}, \text{sm}, \dots, \text{sm}), (\text{mm}, \text{mm}, \dots, \text{mm}), (\text{me}, \text{me}, \dots, \text{me})\} \\ P_{S_s^n} &= \{(\text{sme}, \text{sme}, \dots, \text{sme}), (\text{smm}, \text{smm}, \dots, \text{smm}), \\ &\quad (\text{mmm}, \text{mmm}, \dots, \text{mmm}), (\text{mme}, \text{mme}, \dots, \text{mme})\} \end{aligned} \quad (6.8)$$

Dabei werden jeweils Präfixe der Länge 2 konsumiert und Präfixe der Länge 3 produziert. Damit hat die entsprechende Datenflußkomponente ein Eingangsgewicht von 2 und ein Ausgangsgewicht von 3. Die Farbe 0, welche ja Signalsegmente bestehend aus einem einzigen Block repräsentiert, tritt in diesem Fall nicht auf. Statt dessen werden Eingangssignale der Länge 2 durch **se** und Ausgangssignale der Länge 3 durch **sme** wiedergegeben.

Hier wird die Funktion f_{prefix_s} anstelle von f_{token_s} verwendet, da nicht nur jeweils ein Token sondern ein endliches Präfix des betreffenden Eingabestroms konsumiert wird. Die Funktion f_{prefix_s} stellt eine auf endliche Präfixe von unendlichen Strömen von gefärbten Token erweiterte Funktion f_{color} dar (siehe Abschnitt 4.3.3.1). Die vereinfachte Strommenge S_s kann mittels eines regulären Ausdrucks beschrieben werden (vergleiche Abschnitt 3.3.3).

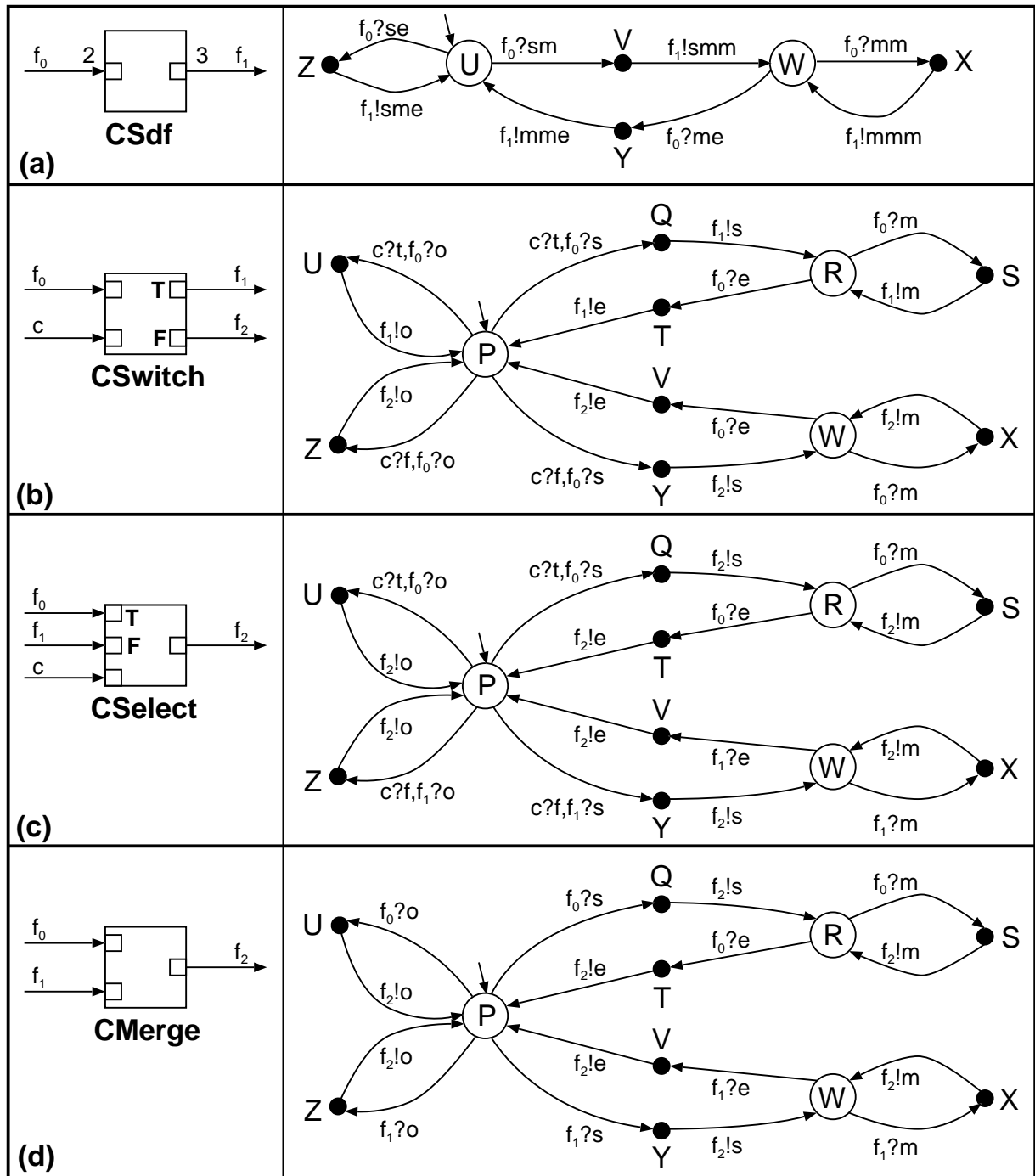


Abbildung 6.2: Fifomatenmodelle einiger Datenflußkomponenten

Abbildung 6.2 zeigt die Fifomatenmodelle² einer Colored-SDF-Komponente mit Eingangsgewicht 2 und Ausgangsgewicht 3, der Colored-BDF-Komponenten Colored Switch (CSwitch) beziehungsweise Colored Select (CSelect) und der Colored-DDF-Komponente Colored Merge (CMerge). Man sieht, wie Transitionen mit Eingabeaktionen, die durch „?“ markiert sind, von Transitionen mit Ausgabeaktionen, welche mittels „!“ gekennzeichnet sind, mit Hilfe von transienten Zuständen getrennt werden (vergleiche Abschnitt 6.3.2).

Im Falle der Datenflußkomponente CSwitch (siehe Abbildung 6.2 (b)) ist besonders die Bewahrung der Struktur von Signalsegmenten zu erwähnen. Alle gefärbten Token, die zu einem Signalsegment gehören, werden sukzessive in die gleiche Ausgabefifo übertragen. Dabei gibt es jeweils nur ein aus einem Signalblock bestehendes Steuersegment, in dem wiederum nur ein Signalwert gekapselt ist. Dieser Wert kennzeichnet den Ausgang, zu dem die Token eines Signalsegments transferiert werden sollen (vergleiche Abschnitt 4.3.3.2).

Daß auch NICHTDETERMINISMUS mittels Fifomaten einfach zu modellieren ist, zeigt das Beispiel der Datenflußkomponente CMerge (siehe Abbildung 6.2 (c)). Liegen an beiden Eingängen dieser Datenflußkomponente die Starttoken zweier Signalsegmente an, dann wird zufällig eines davon ausgewählt und an den Ausgang transferiert. Dies wird im Fifomatenmodell durch zwei von demselben Zustand ausgehende Lesetransitionen ausgedrückt, die gleichzeitig ausführbar sind. Einem Starttoken folgen alle weiteren Token des zugehörigen Signalsegments, da auch CMerge die Segmentstruktur bewahrt (siehe Abschnitt 6.2).

6.4 Neues Model-Checking-Verfahren

Im folgenden wird nach einem einführenden Überblick über das in dieser Arbeit entwickelte Model-Checking-Verfahren und seine Aufgabe auf die beiden Betriebsarten KOMPOSITION und SIMULATION eingegangen. Daran schließt sich die formale Beschreibung des Algorithmus an. Den Schluß bildet eine Betrachtung verschiedener Probleme und der dazugehörigen Lösungen.

6.4.1 Aufgabe

Die Aufgabe des Model Checkers besteht darin, einen ZYKLISCHEN SCHEDULE zu bestimmen (vergleiche Abschnitt 2.2.10), welcher SPEICHERÜBERLAUF AUSSCHLIESST. Falls ein solcher Schedule existiert, kann das Programm DEADLOCKFREI ausgeführt werden (siehe Abschnitt 1.1).

6.4.2 Überblick über den Model-Checking-Algorithmus

Der Model-Checking-Algorithmus (siehe Abbildung 6.3) wird entwurfsbegleitend bei jeder Änderung im Datenflußgraphen aufgerufen. Das Einfügen neuer Datenflußkomponenten oder Kanten in den Datenflußgraphen führt zur Hinzunahme eines neuen Fifomaten oder einer Verbindung zweier Fifos im zugehörigen Fifomatenmodell. Der Model-Checking-Algorithmus kann

²Der Begriff Fifomatenmodell wird synonym zu Modell der Kommunikationsprotokolle verwendet.

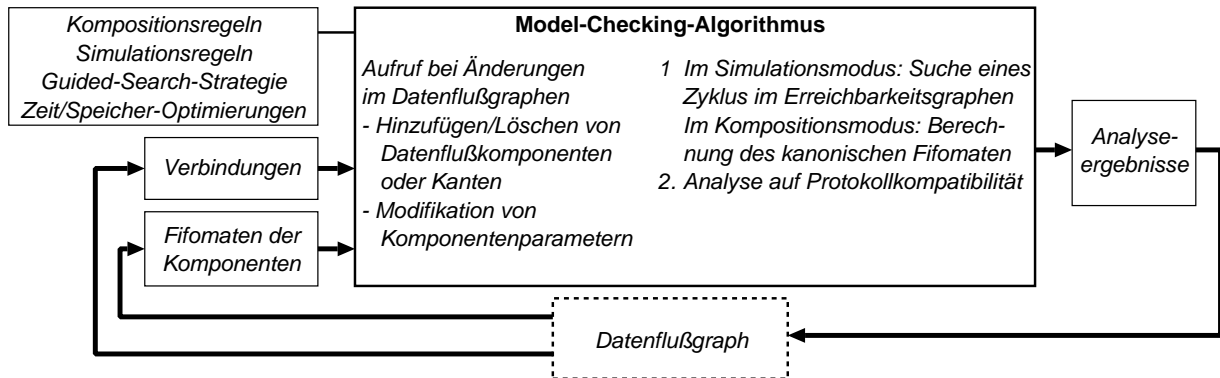


Abbildung 6.3: Model-Checking-Algorithmus

im Simulationsmodus, in welchem genau EIN ZYKLUS im Erreichbarkeitsgraphen gesucht wird, beziehungsweise im Kompositionsmodus, in welchem ein KANONISCHER FIFOMAT aus den beiden ursprünglichen Fifomaten bestimmt wird, betrieben werden. Im Kompositionsmodus können bei Hinzunahme von Datenflußkomponenten beziehungsweise Verbindungen in den Datenflußgraphen die bisherigen Ergebnisse des Fifomatenmodells, die in Form von in dem vorausgegangenen Schritt berechneten kanonischen Fifomaten vorliegen, weiterverwendet werden. Werden Datenflußkomponenten oder Verbindungen im Datenflußgraphen gelöscht, muß die Berechnung und Analyse des Fifomatenmodells neu gestartet werden.

6.4.3 Komposition

DEFINITION 6.4.1: Die KOMPOSITION $F^1 \otimes_J F^2$ zweier Fifomaten

$$\begin{aligned} F^1 &= ((Q_m^1, Q_t^1), q_0^1, (C_i^1, C_o^1, C_h^1), T^1) \\ F^2 &= ((Q_m^2, Q_t^2), q_0^2, (C_i^2, C_o^2, C_h^2), T^2) \end{aligned} \quad (6.9)$$

mit $C^1 \cap C^2 = \emptyset$ basierend auf einer VERBINDUNGSMENGE

$$J \subseteq (C_i^1 \dot{\cup} C_i^2) \times (C_o^1 \dot{\cup} C_o^2) \quad (6.10)$$

wird durch die folgenden drei Schritte definiert:

1. Berechnen des Produktfifomaten
2. Verbinden
3. Vereinfachen

Diese Schritte werden im folgendem im Detail erläutert.

Berechnen des Produktfifomaten: Das PRODUKT ZWEIER FIFOMATEN wird wie folgt berechnet:

$$\begin{aligned}
F^1 \otimes F^2 &= ((Q_m, Q_t), q_0, (C_i, C_o, C_h), T) \\
Q_m &= Q_m^1 \times Q_m^2 \\
Q_t &= (Q_m^1 \times Q_t^2) \cup (Q_t^1 \times Q_m^2) \\
q_0 &= (q_0^1, q_0^2) \\
C_i &= C_i^1 \cup C_i^2 \\
C_o &= C_o^1 \cup C_o^2 \\
C_h &= C_h^1 \cup C_h^2 \\
T &= \{((q^1, q^2), \text{op}, w, (\hat{q}^1, \hat{q}^2)) \mid \\
&\quad q^1 = \hat{q}^1 \in Q_m^1 \wedge (q^2, \text{op}, w, \hat{q}^2) \in T^2 \vee \\
&\quad q^2 = \hat{q}^2 \in Q_m^2 \wedge (q^1, \text{op}, w, \hat{q}^1) \in T^1\} \\
&\quad \text{mit op} \in \{!, ?\}
\end{aligned} \tag{6.11}$$

Verbinden: Der Fifomat $F^1 \otimes F^2$ wird im VERBINDUNGSSCHRITT zu $F^1 \otimes_J F^2$ aktualisiert, indem die gegebene Menge

$$J \subseteq C_i \times C_o \tag{6.12}$$

benutzt wird. Die aktualisierten Mengen der Eingabe-, Ausgabe- und verdeckten Fifos sind:

$$\begin{aligned}
C'_i &= C_i \setminus \{c \mid (c, c') \in J\} \\
C'_o &= C_o \setminus \{c' \mid (c, c') \in J\} \\
C'_h &= C_h \cup \{(M_c \cup M'_c, n_c + n'_c, m_c \bullet m'_c) \mid \\
&\quad ((M_c, n_c, m_c), (M'_c, n'_c, m'_c)) \in J\}
\end{aligned} \tag{6.13}$$

In diesem Schritt werden die Fifomengen aktualisiert, wobei die Menge der miteinander verbundenen Fifos der beiden Fifomaten in die aktualisierte Menge der verdeckten Fifos des Produktfifomaten aufgenommen wird. Zudem sind diese miteinander verbundenen Fifos in den aktualisierten Mengen der Eingabe- und Ausgabefifos nicht mehr vorhanden. Dieses Vorgehen wird im folgenden genauer erläutert.

Vereinfachen: Die Vereinfachung des Produktfifomaten gliedert sich in drei Teilschritte:

1. Im Rahmen der FIFOELIMINATION wird eine Fifo mit begrenzter Kapazität in die Zustandsmenge hineincodiert. Indem die Ausführung des Fifomaten simuliert wird, wird ein neuer Zustand durch Konkatenation des aktuellen Zustandes und des Fifoinhaltes erzeugt. Der so entstehende Fifomat entspricht dem im folgenden vorgestellten Erreichbarkeitsgraphen (siehe Abschnitt 6.4.4).
2. Während des PRUNING werden alle Sackgassen des Fifomaten (beziehungsweise Erreichbarkeitsgraphen) entfernt.
3. Den letzten Schritt bildet die ENTFERNUNG DER LEEREN TRANSITIONEN. Transitionen, die nur leere Aktionen beinhalten, werden entfernt, indem ihre Start- und Endzustände – falls möglich – verschmolzen werden. Eine solche Verschmelzung ist beispielsweise nicht

erlaubt, wenn vom Startzustand aus eine oder mehrere Transitionen zu anderen Zuständen führen, welche vom Endzustand aus in dieser Form nicht erreichbar sind. Dann würde eine Verschmelzung auch vom Endzustand aus diese Transitionen ermöglichen, die im ursprünglichen Fifomaten nicht vorgesehen waren.

Die Kompatibilität zweier Kommunikationsprotokolle ist wie folgt definiert:

DEFINITION 6.4.2: *Zwei Kommunikationsprotokolle heißen KOMPATIBEL, wenn der zugehörige Produktfifomat nach Durchführung aller Vereinfachungsschritte nicht leer ist.*

Beispiel zur Komposition: Abbildung 6.4 zeigt ein Beispiel zur Veranschaulichung der Komposition. Die funktionale Beschreibung des Verhaltens der beiden Datenflußkomponenten A und B lautet³:

$$\text{oneToTwo}_{\text{stream}_s} \begin{cases} S_s^1 \rightarrow S_s^2 \\ x \bullet r \mapsto \text{oneToTwo}_{\text{prefix}_s}(x) \bullet \text{oneToTwo}_{\text{stream}_s}(r) \end{cases} \quad (6.14)$$

$$\text{oneToTwo}_{\text{prefix}_s} \begin{cases} P_{S_s^1} \rightarrow P_{S_s^2} \\ \mathbf{o} \mapsto (\mathbf{o}, \mathbf{o}) \end{cases} \quad (6.15)$$

$$\text{twoToOne}_{\text{stream}_s} \begin{cases} S_s^2 \rightarrow S_s^1 \\ x \bullet r \mapsto \text{twoToOne}_{\text{prefix}_s}(x) \bullet \text{twoToOne}_{\text{stream}_s}(r) \end{cases} \quad (6.16)$$

$$\text{twoToOne}_{\text{prefix}_s} \begin{cases} P_{S_s^2} \rightarrow P_{S_s^1} \\ (\mathbf{o}, \mathbf{o}) \mapsto \mathbf{o} \end{cases} \quad (6.17)$$

Zu jeder Änderung im Datenflußgraphen ist die entsprechende Maßnahme im Fifomatenmodell dargestellt. Dabei werden nur Schritte, die eine Veränderung mit sich bringen, aufgezeigt.

1. Abbildung 6.4 a(1) zeigt zwei Datenflußkomponenten. Die beiden zugehörigen Fifomaten implementieren das durch die funktionale Beschreibung gegebene Verhalten der beiden Datenflußkomponenten. In einem ersten Schritt wird der Produktfifomat berechnet (siehe Abbildung 6.4 a(2)). Die Schritte Verbinden und Vereinfachen verändern dieses Ergebnis nicht.
2. Fügt man eine Kante in den Datenflußgraphen ein (siehe Abbildung 6.4 b(1)), so führt dies zu einer Aktualisierung der Mengen der Eingabe-, Ausgabe- und verdeckten Fifos (siehe Abbildung 6.4 b(2)). Im Vereinfachungsschritt wird eine Fifo eliminiert (siehe Abbildung 6.4 b(3)).
3. Die Umwandlung dieses Datenflußgraphen in eine hierarchische Datenflußkomponente (siehe Abbildung 6.4 c(1)) bringt keine Veränderungen im Fifomatenmodell mit sich.

³Um Beispiele so klein als möglich zu halten, wurden in dieser Arbeit zur Veranschaulichung gelegentlich klassische SDF-Komponenten verwendet und entsprechend mittels Fifomaten modelliert. Ähnliche aber umfangreichere Beispiele lassen sich auch mit Datenflußkomponenten der gefärbten Datenflußparadigmen erstellen.

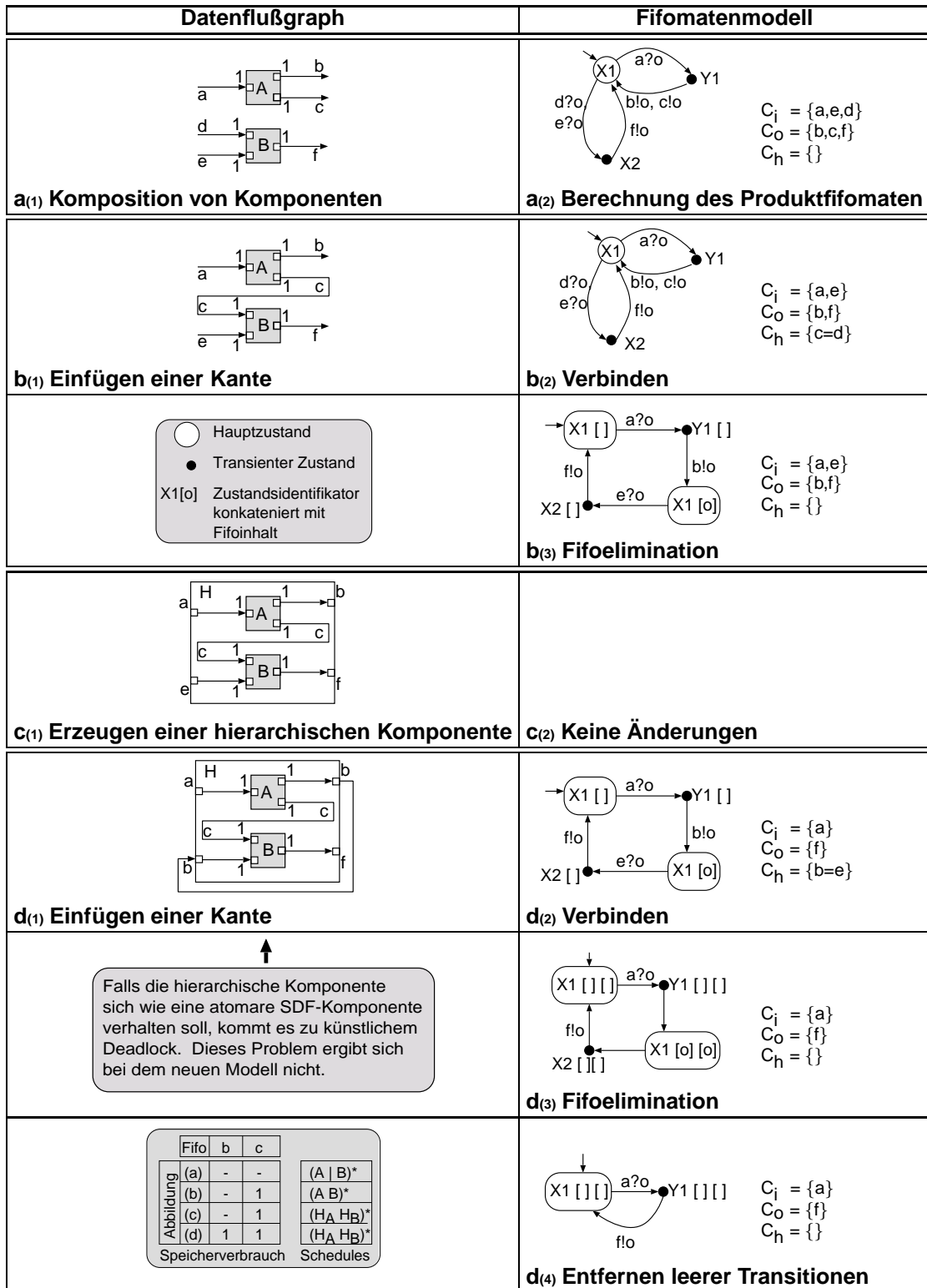


Abbildung 6.4: Beispiel zur Komposition

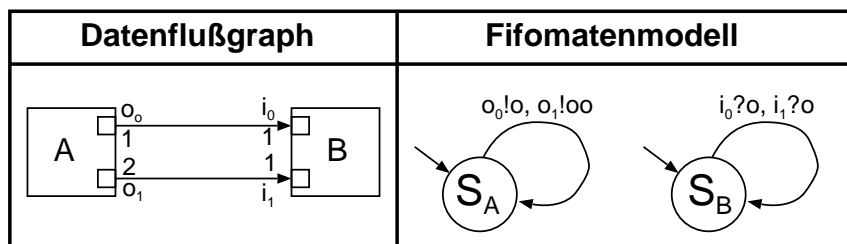


Abbildung 6.5: Datenflußgraph für Simulation

4. In Abbildung 6.4 d(1) wird eine Rückkopplungskante eingefügt. Würde man die operationelle Semantik des klassischen Datenflußparadigmas SDF zugrunde legen, so wäre dieser Datenflußgraph im Deadlock (vergleiche Abschnitt 2.5). Im Fifomatenmodell werden die Mengen der Fifos aktualisiert (siehe Abschnitt 6.4 d(2)), eine Fifo wird eliminiert (siehe Abbildung 6.4 d(3)), und es werden leere Transitionen entfernt (siehe Abbildung 6.4 d(4)).

Die Ergebnisse der Analyse sind in der unteren linken Ecke der Abbildung dargestellt.

6.4.4 Simulation

Im nachfolgenden Abschnitt wird der Simulationsmodus und der zugrundeliegende Algorithmus `guidedSearch()` schrittweise eingeführt.

Brute Force: Abbildung 6.5 zeigt einen Datenflußgraphen bestehend aus zwei Datenflußkomponenten, die durch zwei Kanten miteinander verbunden sind. Die funktionale Beschreibung der Quelle ist gemäß den Gleichungen 6.6 und 6.7

$$\text{source}_{\text{stream}_s} \begin{cases} S_s^0 \rightarrow S_s^2 \\ () \mapsto \text{source}_{\text{prefix}_s}() \bullet \text{source}_{\text{stream}_s}() \end{cases} \quad (6.18)$$

$$\text{source}_{\text{prefix}_s} \begin{cases} P_{S_s^0} \rightarrow P_{S_s^2} \\ () \mapsto (o, oo) \end{cases} \quad (6.19)$$

wohingegen die Senke folgendermaßen beschrieben wird:

$$\text{sink}_{\text{stream}_s} \begin{cases} S_s^2 \rightarrow S_s^0 \\ x \bullet r \mapsto \text{sink}_{\text{prefix}_s}(x) \bullet \text{sink}_{\text{stream}_s}(r) \end{cases} \quad (6.20)$$

$$\text{sink}_{\text{prefix}_s} \begin{cases} P_{S_s^2} \rightarrow P_{S_s^0} \\ (o, o) \mapsto () \end{cases} \quad (6.21)$$

Die Senkenkomponente hat unterschiedliche Gewichte an ihren Eingängen. Da es sich um eine klassische SDF-Komponente (siehe oben) handelt, resultiert das Gewicht 2 in `oo` und nicht in `se`. Baut man in einer Simulation sukzessive den zu dem Fifomatenmodell gehörenden Erreichbarkeitsgraphen auf, so werden folgende Schritte abgearbeitet:

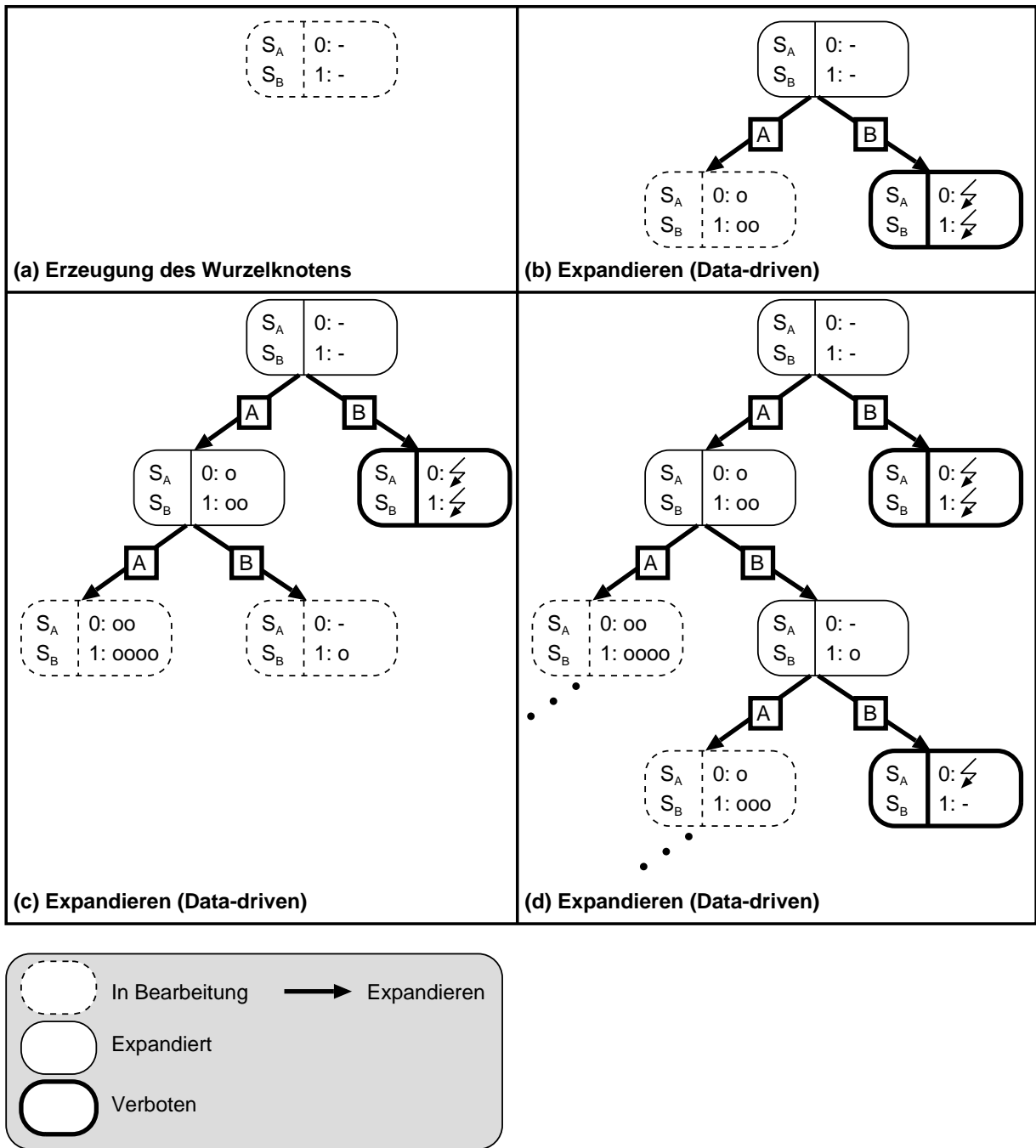


Abbildung 6.6: Simulation (Brute Force)

1. ERZEUGEN DES WURZELKNOTENS: Als erstes wird der Wurzelknoten des Erreichbarkeitsgraphen angelegt (siehe Abbildung 6.6 (a)). Dieser beinhaltet sowohl die Anfangszustände der beiden Fifomaten als auch die Anfangsbelegungen der Fifos, die in diesem

Beispiel leer sind. Bei den Fifos werden nur die Indizes 0 und 1 der jeweiligen Kantenbeschriftung angegeben.

2. EXPANDIEREN (DATA-DRIVEN): Im nächsten Schritt wird der Wurzelknoten des Erreichbarkeitsgraphen expandiert⁴. Das heißt, alle seine Söhne werden erzeugt. Diese ergeben sich durch die Ausführung jeweils eines Zustandsüberganges eines der beteiligten Fifomaten. In diesem Fall gibt es zwei potentielle Söhne. Zum einen kann die Quellkomponente versuchen zu schreiben. Dies ist erlaubt, da dadurch die Kapazität der beteiligten Fifos (die hier als unendlich angenommen ist) nicht überschritten wird. Zum anderen versucht die Senkenkomponente zu lesen. Dies ist nicht möglich, da die zu lesenden Wörter keine Präfixe der aktuellen Fifoinhalte darstellen. Aus diesem Grund resultiert diese Operation in einem verbotenen Knoten. Das Symbol ζ kennzeichnet die Fifos, deren Inhalte ursächlich für die Nichtausführbarkeit der entsprechenden Transition des zugehörigen Fifomaten sind.
3. EXPANDIEREN (DATA-DRIVEN): Daraufhin wird ein zurückgestellter Knoten heuristisch ausgewählt und expandiert.

Dieser Brute-Force-Ansatz zur Bestimmung des Erreichbarkeitsgraphen terminiert in diesem Beispiel nicht, da kein Zyklus in den Wurzelknoten des Erreichbarkeitsgraphen existiert und außerdem durch das Mißverhältnis von geschriebenen und gelesenen Token die Anzahl der Token auf der unteren Kante des Datenflußgraphen beliebig anwächst.

Blockieren/Deblockieren: In den Abbildungen 6.7 und 6.8 wird der Brute-Force-Ansatz um die Mechanismen BLOCKIEREN und LAZY DEBLOCKIEREN ergänzt:

1. ERZEUGEN DES WURZELKNOTENS: Der erste Schritt besteht in der Erzeugung des Wurzelknotens des Erreichbarkeitsgraphen analog zum Brute-Force-Ansatz (siehe Abbildung 6.7 (a)).
2. EXPANDIEREN (DATA-DRIVEN): Als nächstes wird wie beim Brute-Force-Ansatz der Wurzelknoten des Erreichbarkeitsgraphen expandiert, wobei ein verbotener Knoten und ein weiter bearbeitbarer Knoten erzeugt werden (siehe Abbildung 6.7 (b)).
3. BLOCKIEREN: Der weiter bearbeitbare Knoten wird vorerst blockiert (siehe Abbildung 6.7 (b)), da seine Konfiguration mit der des blockierenden Wurzelknotens bezüglich der Fifomatenzustände identisch ist und bezüglich der Fifoinhalte eine Erweiterung der Fifoinhalte des Wurzelknotens darstellt. Die Idee hinter diesem Blockieren besagt, daß zuerst untersucht werden soll, ob man nicht ausgehend von den „kürzeren“ Fifoinhalten des blockierenden Knotens den Zyklus zurück in den Wurzelknoten des Erreichbarkeitsgraphen finden kann.

⁴Diese Expansion entspricht der als data-driven gekennzeichneten Abarbeitungsstrategie von Datenflußgraphen (vergleiche Abschnitt 2.2.9).

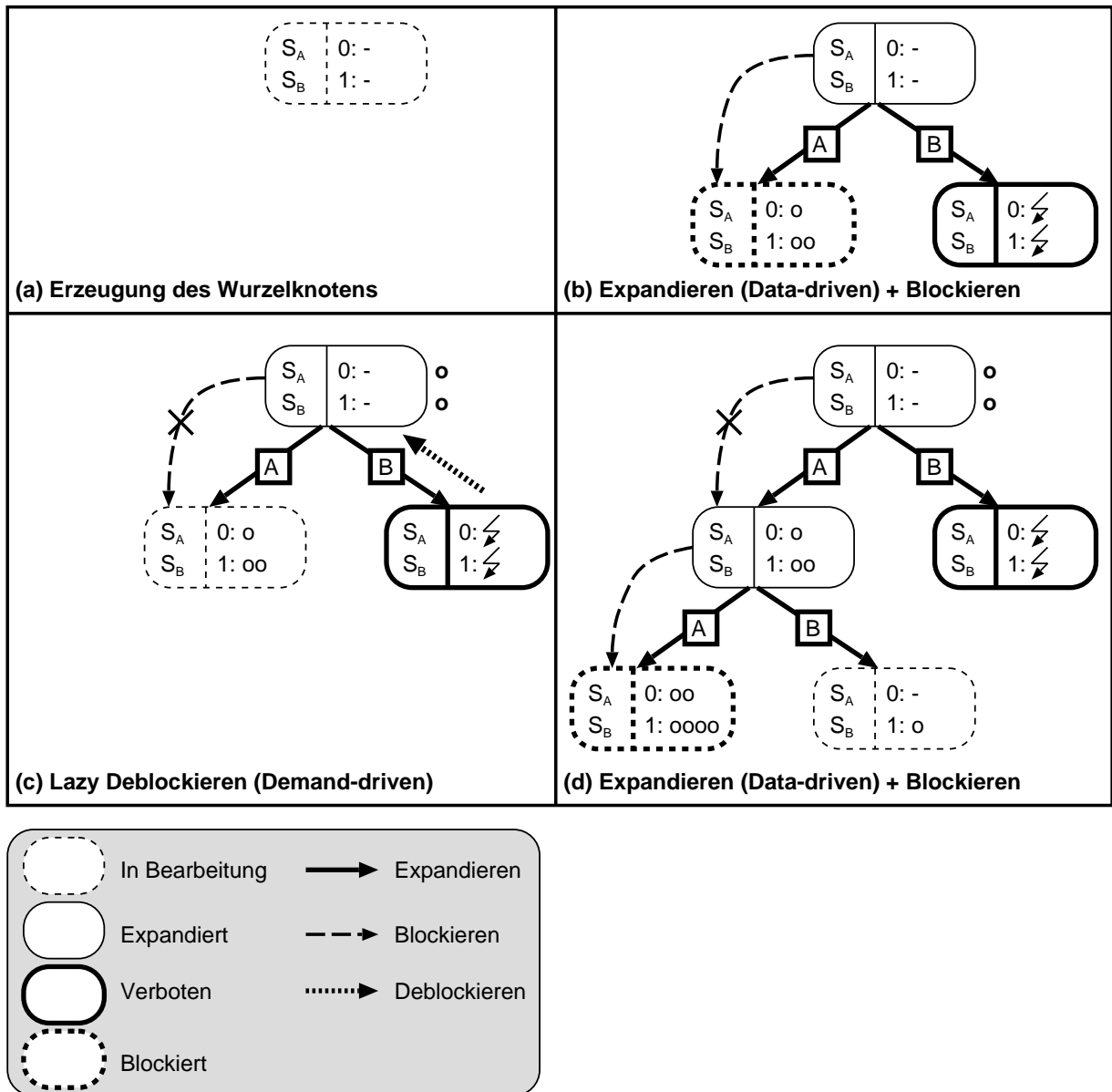


Abbildung 6.7: Simulation (Blockieren/Deblockieren) (1)

4. LAZY DEBLOCKIEREN (DEMAND-DRIVEN): Da allerdings keine zurückgestellten Knoten mehr vorhanden sind, wird als nächstes (lazy) deblockiert⁵ (siehe Abbildung 6.7 (c)). Das heißt, es gibt keinen begehbaren Pfad mehr im Erreichbarkeitsgraphen. Nur durch die Aufhebung einer Blockade kann weitergesucht werden. Die Idee hinter dem Lazy Deblockieren ist folgende: Der verbotene Knoten wäre erreichbar, wenn seine Leseaktionen

⁵Dieses Deblockieren mit anschließender Expansion entspricht der als demand-driven gekennzeichneten Abarbeitungsstrategie von Datenflußgraphen (vergleiche Abschnitt 2.2.9).

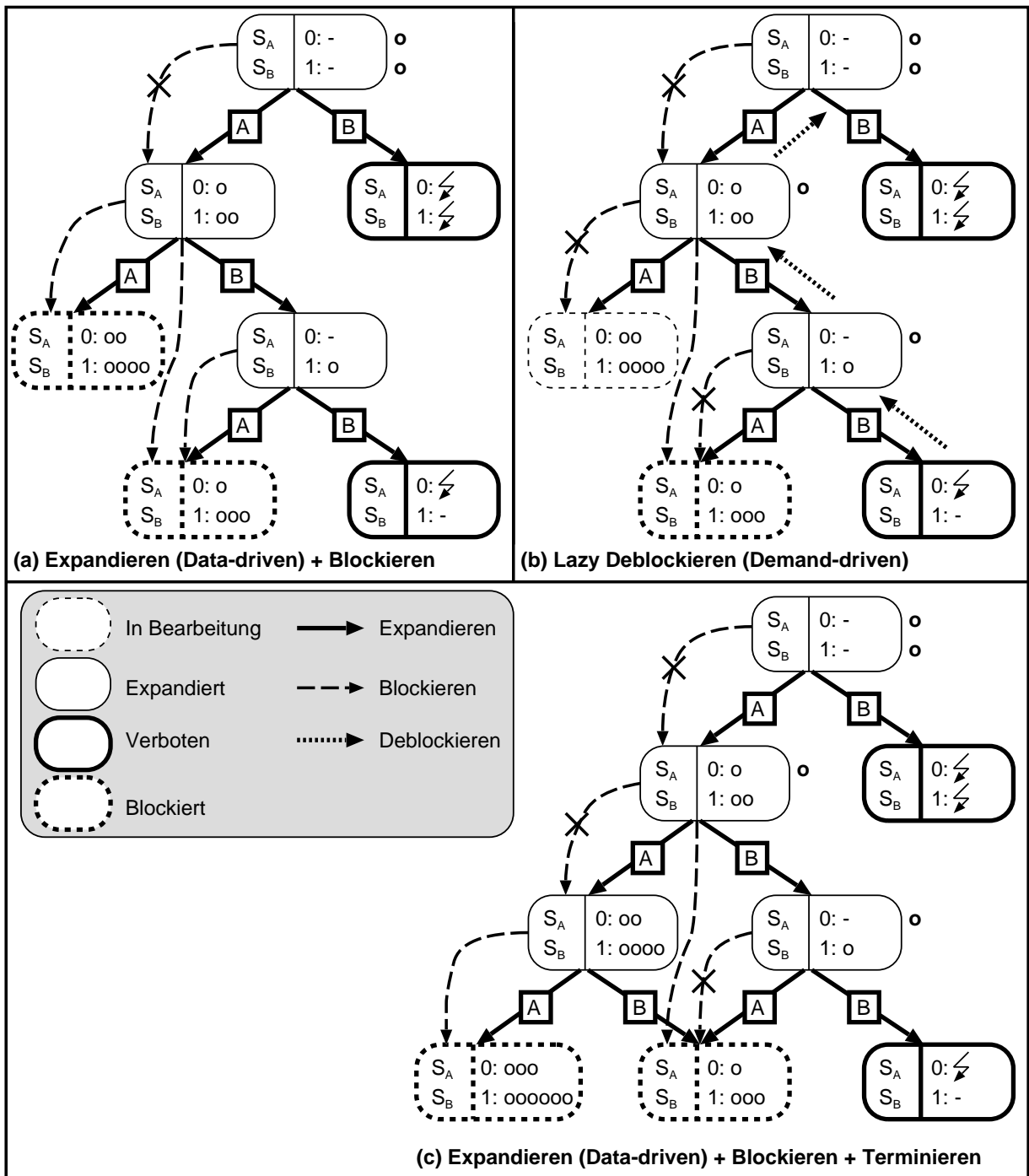


Abbildung 6.8: Simulation (Blockieren/Deblockieren) (2)

durch eine entsprechende Fifobelegung ermöglicht würde. In dem Beispiel wäre also der verbotene Knoten erreichbar, wenn vor Ausführen der zugehörigen Leseaktionen in jeder der beiden Fifos ein 0 gespeichert wäre. Diese gewünschte Fifobelegung ist der Startpunkt für eine Reihe von Deblockierschritten, die bei dem Wurzelknoten des Erreichbarkeitsgraphen enden. In jedem dieser Schritte wird die entsprechende Lese- oder Schreibtransition umgekehrt auf die jeweilige Wunschfifobelegung angewandt. In der Abbildung ist die gewünschte Fifobelegung, soweit diese sich von der aktuellen Fifobelegung des Knotens unterscheidet, neben dem Knoten dargestellt. Bei jedem besuchten Knoten wird dabei überprüft, ob nicht ein Knoten blockiert ist, dessen Fifobelegungen entweder diese Anforderungen erfüllen oder zumindest einen Schritt in Richtung Anforderungserfüllung darstellen. Erfüllt sind diese Anforderungen, wenn die gewünschten Fifobelegungen Präfixe der Fifobelegungen des blockierten Knotens darstellen. Eine Verbesserung liegt vor, wenn die Fifobelegungen des blockierenden Knotens Präfixe der Fifobelegungen des blockierten Knotens und diese wiederum Präfixe der gewünschten Fifobelegungen sind. Falls dem so ist, wird der entsprechende Knoten deblockiert und kann – falls keine anderen Blockierungen durch weitere Knoten des Erreichbarkeitsgraphen vorliegen – weiterbearbeitet werden. Auf diese Weise ist es möglich, einen ähnlichen Pfad des Erreichbarkeitsgraphen zu finden, der im ersten Anlauf aufgrund der „zu langen“ Fifobelegung abgeschnitten wurde.

5. EXPANDIEREN (DATA-DRIVEN): Da aufgrund des Lazy Deblockierens ein Knoten vorhanden ist, der expandiert werden kann, wird dieser expandiert (siehe Abbildung 6.7 (d)).
6. BLOCKIEREN: Einer der Nachfolger wird blockiert (siehe Abbildung 6.7 (d)).
7. EXPANDIEREN (DATA-DRIVEN): Da aber noch ein nicht blockierter zurückgestellter Knoten des Erreichbarkeitsgraphen vorhanden ist, wird dieser expandiert (siehe Abbildung 6.8 (a)).
8. BLOCKIEREN: Der eine Nachfolger ist verboten, der andere wird von zwei Vorgängerknoten blockiert (siehe Abbildung 6.8 (a)).
9. LAZY DEBLOCKIEREN (DEMAND-DRIVEN): Dies führt zu einer weiteren Deblockierung (siehe Abbildung 6.8 (b)). Man erkennt, daß nur ein Knoten vollständig deblockiert wird, da bei dem anderen in Frage kommenden Knoten noch eine zusätzliche Blockierkante existiert.
10. EXPANDIEREN (DATA-DRIVEN): Der vollständig deblockierte Knoten wird expandiert, wobei unter anderem ein redundanter Knoten erzeugt wird (siehe Abbildung 6.8 (c)). Ein Knoten ist redundant, wenn bereits ein gleicher Knoten existiert. Zwei Knoten sind dabei gleich, wenn ihre Konfigurationen bestehend aus Fifomatenzuständen und Fifoinhalten identisch sind. Redundante Knoten werden bei der weiteren Bearbeitung nicht berücksichtigt, da die von diesen Knoten aus erreichbaren Teile des Erreichbarkeitsgraphen auch von ihrem identischen Knoten aus erreicht werden können.
11. BLOCKIEREN: Der verbleibende Nachfolger des gerade expandierten Knotens wird blockiert (siehe Abbildung 6.8 (c)).

12. **TERMINIERUNG:** Da die Menge der bei der Expansion zurückgestellten Knoten leer ist und auch kein weiterer verbotener Knoten vorhanden ist, der mittels Deblockierung diese Menge um weitere Knoten ergänzen könnte, terminiert das Verfahren (siehe Abbildung 6.8 (c)). Dabei ist zu beachten, daß das Lazy Deblockieren nur durch verbotene Knoten ausgelöst wird.

Die **VORTEILE** des hier beispielhaft erläuterten Verfahrens, welches einen den Wurzelknoten beinhaltenden Zyklus im Erreichbarkeitsgraphen sucht, sind:

- **VERHINDERN DES DIVERGIERENS DER SUCHE:** Es werden im ersten Ansatz alle Knoten **BLOCKIERT**, wo bereits ein Knoten gefunden wurde, der eine identische Zustandsbelegung der Fifomaten und kürzere Fifobelegungen hinsichtlich der Präfixordnung aufweist. Damit werden alle Pfade abgeschnitten, die eine unnötige Zunahme der Fifolängen aufweisen.
- **SELEKTIVES ÖFFNEN VON PFADEN IN RICHTUNG WURZELKNOTEN:** Ist der durch die blockierten Knoten abgegrenzte Erreichbarkeitsgraph vollkommen durchsucht und kein Zyklus in den Wurzelknoten gefunden, dann werden weitere Pfade mit Hilfe des **LAZY DEBLOCKIERENS** geöffnet. Man betrachtet jeweils einen verbotenen Knoten, der aufgrund einer verbotenen **LESETRANSITION** nicht erreicht wurde, und sucht solche Knoten zu deblockieren, deren Fifobelegungen diese Lesetransition erlauben. Damit werden selektiv Pfade geöffnet, die in Richtung Wurzelknoten führen. Ein Knoten, dessen Fifobelegung kürzer ist als die seines Vorgängerknotens, liegt näher an der Wurzel des Erreichbarkeitsgraphen, wenn alle Fifobelegungen der Wurzel leer sind. Selbst wenn die Fifobelegungen der Wurzel endliche Initialisierungswörter enthalten, ist dieses Verfahren erfolgreich, da alle Knoten des Erreichbarkeitsgraphen mit kürzeren Fifobelegungen schnell erzeugt sind und anschließend das Verfahren greift. Demgegenüber bestünde bei einer Suche, die tendenziell längere Fifobelegungen bevorzugt, die Gefahr des Divergierens. Zusätzlich werden durch die heuristische Auswahl Deblocier bevorzugt, die näher am Wurzelknoten liegen.
- **BEWEIST EXISTENZ/NICHTEXISTENZ VON ZYKLEN (BEHAUPTUNG):** Falls es einen den Wurzelknoten beinhaltenden Zyklus des Erreichbarkeitsgraphen gibt, dann findet der Algorithmus diesen und terminiert. Gibt es keinen solchen Zyklus, wird dies gemeldet oder der Algorithmus terminiert nicht. Dies ist eine **NICHTBEWIESENE BEHAUPTUNG** (vergleiche Abschnitt 6.4.5).

Partial Order Reduction: Abbildung 6.9 zeigt einen Datenflußgraphen und zwei dazugehörige Erreichbarkeitsgraphen. In diesem Datenflußgraphen sind eine Quelle, mehrere interne Datenflußkomponenten und eine Senke enthalten. Die Quelle entspricht folgender funktionalen Beschreibung:

$$\text{source}_{\text{stream}_s} \begin{cases} S_s^0 \rightarrow S_s^2 \\ () \mapsto \text{source}_{\text{prefix}_s}() \bullet \text{source}_{\text{stream}_s}() \end{cases} \quad (6.22)$$

$$\text{source}_{\text{prefix}_s} \begin{cases} P_{S_s^0} \rightarrow P_{S_s^2} \\ () \mapsto (0, 0) \end{cases} \quad (6.23)$$

Die internen Datenflußkomponenten entsprechen

$$\text{internal}_{\text{stream}_s} \begin{cases} S_s^1 \rightarrow S_s^1 \\ x \bullet r \mapsto \text{internal}_{\text{prefix}_s}(x) \bullet \text{internal}_{\text{stream}_s}(r) \end{cases} \quad (6.24)$$

$$\text{internal}_{\text{prefix}_s} \begin{cases} P_{S_s^1} \rightarrow P_{S_s^1} \\ \mathbf{o} \mapsto \mathbf{o} \end{cases} \quad (6.25)$$

Die Senke ist definiert als:

$$\text{sink}_{\text{stream}_s} \begin{cases} S_s^2 \rightarrow S_s^0 \\ x \bullet r \mapsto \text{sink}_{\text{prefix}_s}(x) \bullet \text{sink}_{\text{stream}_s}(r) \end{cases} \quad (6.26)$$

$$\text{sink}_{\text{prefix}_s} \begin{cases} P_{S_s^2} \rightarrow P_{S_s^0} \\ (\mathbf{o}, \mathbf{o}) \mapsto () \end{cases} \quad (6.27)$$

Betrachtet man Abbildung 6.9 (b), so erkennt man eine rautenförmige Anordnung von Knoten des Erreichbarkeitsgraphen, die auf unterschiedlichen Pfaden vom Start- zum Endknoten liegen. Diese große Anzahl von Knoten entsteht, da im Laufe der Simulation alle möglichen Reihenfolgen, in denen die beiden Token durch den Datenflußgraphen traversieren können, berücksichtigt werden. Es ist allerdings so, daß nur ein beliebiger Pfad vom Start- zum Zielknoten von Interesse ist (siehe Abbildung 6.9 (c)). Diese Reduzierung der Zustandsmenge auf die Knoten des interessierenden Pfades bezeichnet man als PARTIAL ORDER REDUCTION (siehe Abschnitt 2.4.4).

Diese Reduktion basiert auf folgender Beobachtung: Falls es einen Fifomaten gibt, der im aktuellen Zustand eine atomare Operation durchführen kann, ohne daß eine Aktion irgendeiner Transition in dieser atomaren Operation nicht ausführbar ist aufgrund

- begrenzter Kapazitäten bei Schreiboperationen oder
- weil nur ein Präfix des zu lesenden Wortes bei Leseoperationen vorhanden ist,

dann wird beim Expandieren nur dieser Fifomaten berücksichtigt. Der Grund ist, daß die Expansion dieses Fifomaten KOMMUTATIV zur Expansion aller anderen Fifomaten ist (vergleiche Abschnitt 6.4.5).

6.4.5 Formale Beschreibung des Model-Checking-Algorithmus

Nach einer formalen Beschreibung des zu einem Datenflußgraphen gehörenden Fifomatenmodells, des Erreichbarkeitsgraphen und notwendiger Hilfsfunktionen fokussiert sich dieser Abschnitt auf die Suchstrategie im Erreichbarkeitsgraphen als Kern des Model-Checking-Verfahrens.

Fifomatenmodell: In der nachfolgenden Definition werden verschiedene Begriffe, welche die Menge der von dem Model-Checking-Verfahren betrachteten Fifomaten charakterisieren, vorgestellt. Dabei wird auf der Definition eines Fifomaten und seiner Bestandteile (vergleiche Definition 6.3.2) aufgebaut. Der Index f wird verwendet, um die Zugehörigkeit zu einem Fifomaten f auszudrücken. So ist

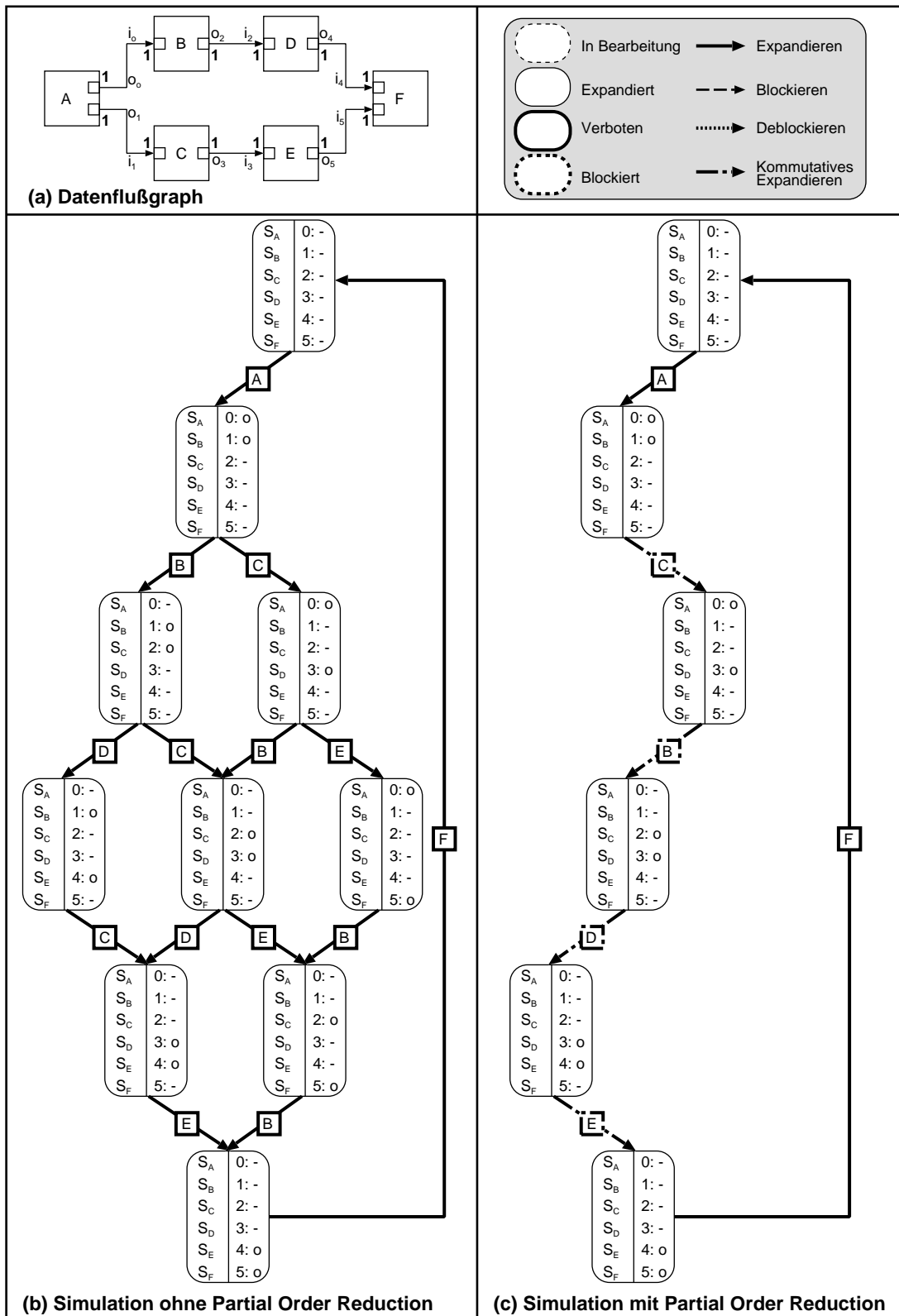


Abbildung 6.9: Simulation (Partial Order Reduction)

- Q_f die Zustandsmenge,
- $Q_{m,f}$ die Menge der Hauptzustände,
- $Q_{t,f}$ die Menge der transienten Zustände,
- T_f die Menge der Transitionen und
- C_f die Menge der Fifos

von f . Dabei sind die Mengen der Zustände Q_f und $Q_{f'}$ zweier beliebiger Fifomaten f und f' disjunkt. Das gleiche gilt für die Mengen der Transitionen. Einzig bei den Fifos gibt es Überschneidungen, wenn beispielsweise Fifomat f in eine Fifo schreibt und Fifomat f' aus derselben Fifo liest.

DEFINITION 6.4.3:

1. Die MENGE DER VOM MODEL-CHECKING-ALGORITHMUS UNTERSUCHTEN FIFOMATEN wird mit \mathbb{F} bezeichnet.
2. Die MENGE ALLER ZUSTÄNDE DER FIFOMATEN ist als disjunkte Vereinigung der Zustandsmengen der Fifomaten $f \in \mathbb{F}$ definiert:

$$Q_{\mathbb{F}} = \dot{\bigcup}_{f \in \mathbb{F}} Q_f \quad (6.28)$$

3. Die MENGE ALLER TRANSITIONEN DER FIFOMATEN ist als disjunkte Vereinigung der Transitions Mengen der Fifomaten $f \in \mathbb{F}$ definiert:

$$T_{\mathbb{F}} = \dot{\bigcup}_{f \in \mathbb{F}} T_f \quad (6.29)$$

4. Die MENGE DER FIFOS (KANÄLE) DER FIFOMATENMENGE ist als Vereinigung der Kanal Mengen der Fifomaten $f \in \mathbb{F}$ definiert:

$$C_{\mathbb{F}} = \bigcup_{f \in \mathbb{F}} C_f \quad (6.30)$$

Erreichbarkeitsgraph: In folgender Definition werden einige grundlegende Begriffe zum Verständnis des Aufbaus der hier betrachteten Erreichbarkeitsgraphen eingeführt.

DEFINITION 6.4.4:

1. Eine Abbildung

$$s \left\{ \begin{array}{l} \mathbb{F} \rightarrow Q_{\mathbb{F}} \\ f \mapsto q, \end{array} \right. \quad (6.31)$$

die jedem Fifomaten f der Fifomatenmenge \mathbb{F} einen Zustand q zuordnet, heißt STATEMAP.

2. Eine Abbildung

$$w \begin{cases} C_{\mathbb{F}} \rightarrow M^* \\ c \mapsto m, \end{cases} \quad (6.32)$$

die jeder Fifo c der Fifomenge $C_{\mathbb{F}}$ ein Wort $m \in M^*$ zuordnet, heißt WORDMAP.

3. Die KNOTENMENGE DES ERREICHBARKEITSGRAPHEN ist definiert als⁶

$$V^* \subseteq (\mathbb{F} \rightarrow Q_{\mathbb{F}}) \times (C_{\mathbb{F}} \rightarrow M^*). \quad (6.33)$$

Ein KNOTEN DES ERREICHBARKEITSGRAPHEN besteht somit aus einer Statemap und einer Wordmap:

$$v = (s, w). \quad (6.34)$$

V^i bezeichnet die KNOTENTEILMENGE DES ERREICHBARKEITSGRAPHEN, welche bis zum i -ten Schritt des Algorithmus berechnet wurde. Weitere Bezeichner sind:

$$\begin{aligned} V_E & \text{ Endliche Menge der EXPANDIERTEN KNOTEN} \\ V_P & \text{ Endliche Menge der ZURÜCKGESTELLTEN KNOTEN} \\ V_B & \text{ Endliche Menge der BLOCKIERTEN KNOTEN} \\ V_N & \text{ Potentiell unendliche Menge der NOCH NICHT EXISTIERENDEN KNOTEN} \end{aligned} \quad (6.35)$$

Dabei gilt:

$$\begin{aligned} V^i & = V_E^i \dot{\cup} V_P^i \dot{\cup} V_B^i \\ V^* & = V^i \dot{\cup} V_N^i \end{aligned} \quad (6.36)$$

4. Die ANNOTIERTE KANTENMENGE DES ERREICHBARKEITSGRAPHEN ist definiert als

$$E^* \subseteq (V^* \times T_{\mathbb{F}} \times V^*). \quad (6.37)$$

Eine Kante des Erreichbarkeitsgraphen

$$e = (v, t, v') \quad (6.38)$$

gibt an, welche Transition t eines Fifomaten $f \in \mathbb{F}$ ausgeführt wurde, um vom Knoten v zum Knoten v' zu gelangen.

⁶Das Superskript * wird in dieser Arbeit verwendet, um

- die Wortmenge M^* , die Wörter unterschiedlicher Länge beinhaltet, zu kennzeichnen beziehungsweise
- bei allen anderen Mengen hervorzuheben, daß in diesen Mengen alle durch den jeweiligen Algorithmus berechenbaren Elemente enthalten sind. Dies grenzt diese Mengen beispielsweise von den mit Superskript i gekennzeichneten bis zum i -ten Schritt des jeweiligen Algorithmus berechneten Teilmengen ab.
- bei Funktionen eine beliebig häufige Verkettung von Aufrufen zu symbolisieren.

5. Der ERREICHBARKEITSGRAPH einer Menge von Fifomaten ist somit definiert als

$$RT \subseteq V^* \times E^* \quad (6.39)$$

In Abbildung 6.6 (a) ist ein Knoten eines Erreichbarkeitsgraphen mit zugehöriger Statemap und Wordmap abgebildet. Die Kantenmenge E^* wird in dem hier vorgestellten Verfahren nicht gesondert abgespeichert. Der Algorithmus ermittelt die Kanten mit Hilfe der Knoten des Erreichbarkeitsgraphen und der Menge der Fifomaten.

Vergleichsoperationen: Um die Knoten eines Erreichbarkeitsgraphen miteinander vergleichen zu können, sind verschiedene Operationen erforderlich.

DEFINITION 6.4.5:

1. GLEICHHEIT VON STATEMAPS: *Die Gleichheit von Statemaps*

$$s, s' : \mathbb{F} \rightarrow Q_{\mathbb{F}} \quad (6.40)$$

ist wie bei Funktionen üblich punktweise definiert:

$$s = s' \Leftrightarrow (\forall f \in \mathbb{F} : s(f) = s'(f)) . \quad (6.41)$$

2. GLEICHHEIT VON WORDMAPS: *Die Gleichheit von Wordmaps*

$$w, w' : C_{\mathbb{F}} \rightarrow M^* \quad (6.42)$$

ist wie bei Funktionen üblich punktweise definiert:

$$w = w' \Leftrightarrow (\forall c \in C_{\mathbb{F}} : w(c) = w'(c)) . \quad (6.43)$$

3. VERGLEICH VON WORDMAPS: *Zwei Wordmaps*

$$w, w' : C_{\mathbb{F}} \rightarrow M^* \quad (6.44)$$

stehen in einer Kleiner-Gleich- beziehungsweise Kleiner-Relation, wenn gilt:

$$\begin{aligned} w \leq w' &\Leftrightarrow (\forall c \in C_{\mathbb{F}} : w(c) \sqsubseteq w'(c)) \\ w < w' &\Leftrightarrow (w \leq w' \wedge w \neq w') \end{aligned} \quad (6.45)$$

Dabei stellt (M^*, \sqsubseteq) die Präfixordnung dar (siehe Abschnitt 3.3.3).

4. VERGLEICH VON KNOTEN: *Seien $v = (s, w)$ und $v' = (s', w')$. Dann gilt:*

$$\begin{aligned} v \leq v' &\Leftrightarrow (s = s' \wedge w \leq w') \\ v < v' &\Leftrightarrow (s = s' \wedge w < w') \end{aligned} \quad (6.46)$$

Auf Statemaps ist nur die Gleichheit definiert, da hier nur eine Unterscheidung zwischen gleich und ungleich möglich ist. Die Präfixordnung auf Wörtern wird hier auf Wordmaps erweitert. Diese Vergleichsoperationen bilden die Grundlage für den Vergleich von Knoten des Erreichbarkeitsgraphen.

Lese- und Schreiboperationen: Zusätzlich sind noch verschiedene Operationen erforderlich, um den Model-Checking-Algorithmus beschreiben zu können.

DEFINITION 6.4.6:

1. Die Lese- und Schreiboperationen auf Wörtern $m, m', m'' \in M^*$ sind wie folgt definiert:

$$\begin{aligned}
m \oplus_l m' = m'' &\Leftrightarrow m' \bullet m = m'' && \text{LINKSSEITIGES SCHREIBEN} \\
m \oplus_r m' = m'' &\Leftrightarrow m \bullet m' = m'' && \text{RECHTSSEITIGES SCHREIBEN} \\
m \ominus_l m' = m'' &\Leftrightarrow m = m' \bullet m'' && \text{LINKSSEITIGES LESEN} \\
m \ominus_r m' = m'' &\Leftrightarrow m = m'' \bullet m' && \text{RECHTSSEITIGES LESEN}
\end{aligned} \tag{6.47}$$

Falls die angegebene Konkatenation nicht definiert ist, ist auch die entsprechende Schreib-beziehungsweise Leseoperation nicht definiert.

2. Die Lese- und Schreiboperationen auf Wordmaps $w, w', w'' \in (C_{\mathbb{F}} \rightarrow M^*)$ sind wie folgt punktweise definiert.

$$\begin{aligned}
w \oplus_l w' = w'' &\Leftrightarrow \forall_{c \in C_{\mathbb{F}}} : w(c) \oplus_l w'(c) = w''(c) && \text{LINKSSEITIGES SCHREIBEN} \\
w \oplus_r w' = w'' &\Leftrightarrow \forall_{c \in C_{\mathbb{F}}} : w(c) \oplus_r w'(c) = w''(c) && \text{RECHTSSEITIGES SCHREIBEN} \\
w \ominus_l w' = w'' &\Leftrightarrow \forall_{c \in C_{\mathbb{F}}} : w(c) \ominus_l w'(c) = w''(c) && \text{LINKSSEITIGES LESEN} \\
w \ominus_r w' = w'' &\Leftrightarrow \forall_{c \in C_{\mathbb{F}}} : w(c) \ominus_r w'(c) = w''(c) && \text{RECHTSSEITIGES LESEN}
\end{aligned} \tag{6.48}$$

Falls mindestens eine Operation auf den betrachteten Wörtern undefiniert ist, dann ist auch die dazugehörige Operation auf den zugehörigen Wordmaps undefiniert.

3. Die MENGE DER TRANSITIONSOPERATIONEN ist gegeben durch

$$OP = \{!, ?\}, \tag{6.49}$$

wobei „!“ Schreiboperationen bezeichnet und „?“ eine Leseoperation kennzeichnet.

Auf Wörtern sind jeweils zwei Lese- und zwei Schreiboperationen definiert, welche für das Erzeugen einer Wordmap aus einer anderen verwendet werden. Dabei gilt, daß \oplus_r und \ominus_l bei der Erzeugung eines neuen Knotens des Erreichbarkeitsgraphen verwendet werden, wohingegen \oplus_l und \ominus_r bei der Rückwärtspropagation von Wordmaps durch den Erreichbarkeitsgraphen (vergleiche Funktion `unapply()` in Definition 6.4.8) eingesetzt werden. Diese Operationen auf Wörtern sind auf Wordmaps erweitert. Auf Transitionen sind dann jeweils Lese- beziehungsweise Schreiboperationen mittels „!“ beziehungsweise „?“ (siehe Definition 6.3.2) angegeben.

Anwenden einer Transition: Die vorgestellten Schreib- und Leseoperationen werden unter anderem dazu verwendet, aus einem Knoten v des Erreichbarkeitsgraphen durch Anwendung einer Transition t einen neuen Knoten v' zu bestimmen. Im folgenden werden die für diese Operation notwendigen Definitionen eingeführt.

DEFINITION 6.4.7: Die MENGE DER FIFOS EINER TRANSITION t beinhaltet alle Fifos, deren Inhalt durch die Transition $t = (q_1, \mathbf{op}, w_t, q_2)$ eines Fifomaten f verändert wird:

$$C_t = \{c \in C_{\mathbb{F}} \mid w_t(c) \neq \varepsilon\} \tag{6.50}$$

Dabei umfaßt die der Transition t zugeordnete Wordmap w_t alle Wörter, welche in der Transition t aus den einzelnen Fifos gelesen beziehungsweise in diese Fifos geschrieben werden. Fifos, deren Belegungen nicht verändert werden, wird durch w_t das leere Wort ε zugewiesen.

Mit Hilfe der Funktion $\text{apply}()$ wird aus einer Wordmap eine neue Wordmap erzeugt, indem alle Aktionen einer vorgegebenen Transition t durchgeführt werden. Die Funktion $\text{unapply}()$ macht die Auswirkungen einer Transition wieder rückgängig.

DEFINITION 6.4.8:

1. Die Funktion $\text{apply}()$ berechnet aus einer Wordmap und einer Transition eine neue Wordmap:

$$\text{apply} \left\{ \begin{array}{l} (C_{\mathbb{F}} \rightarrow M^*) \times T_{\mathbb{F}} \rightarrow (C_{\mathbb{F}} \rightarrow M^*) \\ (w, t = (q_1, \text{op}, w_t, q_2)) \mapsto \begin{cases} w \oplus_r w_t & \text{falls } \text{op} = ! \\ w \ominus_l w_t & \text{falls } \text{op} = ? \end{cases} \end{array} \right. \quad (6.51)$$

2. Die Funktion $\text{unapply}()$ bestimmt zu einer Wordmap und einer Transition die Vorgängerwordmap:

$$\text{unapply} \left\{ \begin{array}{l} (C_{\mathbb{F}} \rightarrow M^*) \times T_{\mathbb{F}} \rightarrow (C_{\mathbb{F}} \rightarrow M^*) \\ (w, t = (q_1, \text{op}, w_t, q_2)) \mapsto \begin{cases} w \ominus_r w_t & \text{falls } \text{op} = ! \\ w \oplus_l w_t & \text{falls } \text{op} = ? \end{cases} \end{array} \right. \quad (6.52)$$

Falls die verwendeten Schreib- beziehungsweise Leseoperationen undefiniert sind, ist auch das Ergebnis von $\text{apply}()$ beziehungsweise $\text{unapply}()$ undefiniert. Eine Leseoperation ist beispielsweise undefiniert, wenn die zugehörige Fifo leer beziehungsweise mit einem nicht zur Lesetransition passenden Inhalt gefüllt ist. Eine Schreiboperation ist undefiniert, wenn die zugehörige Fifo eine Kapazitätsbeschränkung besitzt, die durch die Ausführung der Schreiboperation überschritten würde⁷.

Dabei muß gelten, daß sich $\text{apply}()$ und $\text{unapply}()$ gegenseitig aufheben, wenn beide definierte Ergebnisse liefern.

BEMERKUNG 6.4.1: Es gilt:

$$\text{apply}(w, t) = w' \Leftrightarrow \text{unapply}(w', t) = w \quad (6.53)$$

Die Funktion $\text{apply}()$ ist überladen. Neben der oben vorgestellten Version für Wordmaps wird nachfolgend eine Version für Statemaps eingeführt.

⁷Die Kapazität n_c einer Fifo c ist dabei wie in Definition 6.3.1 angegeben ein Element der Menge $\mathbb{N} \cup \{\infty\}$. In der Regel ist die Kapazität unbekannt und daher ∞ . Der Benutzer kann allerdings Kapazitätsbegrenzungen einführen.

DEFINITION 6.4.9: Die Funktion $\mathbf{apply}()$ liefert für eine Statemap s und eine Transition t die Statemap s' , welche durch den in der Transition festgelegten Zustandsübergang entsteht.

$$\mathbf{apply} \left\{ \begin{array}{l} (\mathbb{F} \rightarrow Q_{\mathbb{F}}) \times T_{\mathbb{F}} \rightarrow (\mathbb{F} \rightarrow Q_{\mathbb{F}}) \\ (s, (q_1, \mathbf{op}, w_t, q_2)) \mapsto s' \text{ mit } \begin{cases} s'(f) = q_2 & \text{falls } s(f) = q_1 \\ s'(f) = s(f) & \text{sonst} \end{cases} \end{array} \right. \quad (6.54)$$

Faßt man nun die Varianten von $\mathbf{apply}()$ für Wordmaps und Statemaps zusammen, so erhält man folgende für Knoten des Erreichbarkeitsgraphen definierte Version.

DEFINITION 6.4.10: Die Funktion $\mathbf{apply}()$ liefert für einen Knoten v des Erreichbarkeitsgraphen und eine Transition t den Knoten v' , der durch Anwendung der Transition t entsteht:

$$\mathbf{apply} \left\{ \begin{array}{l} V^* \times T_{\mathbb{F}} \rightarrow V^* \\ ((s, w), t) \mapsto (\mathbf{apply}(s, t), \mathbf{apply}(w, t)) \end{array} \right. \quad (6.55)$$

Zulässige und nicht zulässige Operationen: Die Funktion $\mathbf{enabled}()$ überprüft, ob bei einem durch einen Knoten des Erreichbarkeitsgraphen beschriebenen Zustand des Systems bestehend aus Statemap und Wordmap eine Transition eines Fifomaten ausgeführt werden kann. Eine Leseoperation ist dabei ausführbar, wenn die zu lesenden Worte Präfixe der in den Fifos gespeicherten Worte sind. Eine Schreiboperation ist ausführbar, wenn durch die Schreibaktion die Kapazität n_c der jeweiligen Fifo c nicht überschritten wird.

DEFINITION 6.4.11:

1. Die Funktion $\mathbf{readingEnabled}()$ bestimmt, ob bei gegebenem Knoten des Erreichbarkeitsgraphen eine ausgewählte Transition ihre Leseoperation durchführen kann:

$$\mathbf{readingEnabled} \left\{ \begin{array}{l} V^* \times T_{\mathbb{F}} \rightarrow \mathbb{B} \\ ((s, w), (q_1, \mathbf{op}, w_t, q_2)) \mapsto \exists f \in \mathbb{F} : \begin{array}{l} (q_1, \mathbf{op}, w_t, q_2) \in T_f \\ \wedge s(f) = q_1 \wedge \mathbf{op} = ? \\ \wedge w_t \leq w \end{array} \end{array} \right. \quad (6.56)$$

2. Die Funktion $\mathbf{writingEnabled}()$ bestimmt, ob bei gegebenem Knoten des Erreichbarkeitsgraphen eine ausgewählte Transition ihre Schreiboperation durchführen kann:

$$\mathbf{writingEnabled} \left\{ \begin{array}{l} V^* \times T_{\mathbb{F}} \rightarrow \mathbb{B} \\ ((s, w), (q_1, \mathbf{op}, w_t, q_2)) \mapsto \exists f \in \mathbb{F} : \begin{array}{l} (q_1, \mathbf{op}, w_t, q_2) \in T_f \\ \wedge s(f) = q_1 \wedge \mathbf{op} = ! \wedge \\ \forall c \in C_t : |w_t(c)| + |w(c)| \leq n_c \end{array} \end{array} \right. \quad (6.57)$$

3. Die Funktion $\text{enabled}()$ bestimmt, ob bei gegebenem Knoten des Erreichbarkeitsgraphen eine ausgewählte Transition ihre Operation durchführen kann:

$$\text{enabled} \begin{cases} V^* \times T_{\mathbb{F}} \rightarrow \mathbb{B} \\ (v, t) \mapsto \text{readingEnabled}(v, t) \vee \text{writingEnabled}(v, t) \end{cases} \quad (6.58)$$

Mit Hilfe der Funktion $\text{enabled}()$ können ausführbare von nicht ausführbaren Transitionen unterschieden werden. Dabei baut $\text{enabled}()$ auf $\text{readingEnabled}()$ und $\text{writingEnabled}()$ auf, welche diese Unterscheidung für lesende beziehungsweise schreibende Transitionen treffen. Die Funktion $\text{readingEnabled}()$ liefert true , wenn die betrachtete Transition von einem aktuellen Zustand eines Fifomaten ausgeht und die zu lesenden Wörter Präfixe der aktuellen Fifoinhalte sind. Die Funktion $\text{writingEnabled}()$ liefert true , wenn die betrachtete Transition von einem aktuellen Zustand eines Fifomaten ausgeht und die zu schreibenden Wörter die Kapazitäten der jeweiligen Fifos nicht überschreiten.

Deblocker: Im folgenden werden grundlegende Begriffe zum Verständnis der Deblockieroperation eingeführt.

DEFINITION 6.4.12: Die MENGE DER DEBLOCKER ist definiert als

$$D^* \subseteq V^* \times (C_{\mathbb{F}} \rightarrow M^*) \times T_{\mathbb{F}} \quad (6.59)$$

D^i bezeichne die Menge der Deblocker, die bis zum i -ten Schritt des Algorithmus bestimmt wurden. Weitere Bezeichner sind:

$$\begin{aligned} D_E &: \text{Endliche Menge der EXPANDIERTEN DEBLOCKER} \\ D_P &: \text{Endliche Menge der ZURÜCKGESTELLTEN DEBLOCKER} \\ D_N &: \text{Menge der NOCH NICHT BESTIMMTEN DEBLOCKER} \end{aligned} \quad (6.60)$$

Dabei gilt:

$$\begin{aligned} D^i &= D_E^i \dot{\cup} D_P^i \\ D^* &= D^i \dot{\cup} D_N^i \end{aligned} \quad (6.61)$$

Priorität eines Knotens beziehungsweise Deblockers: Die Priorität eines Knotens basiert auf der Länge seiner Wordmap und seiner Entstehungsnummer.

DEFINITION 6.4.13:

1. Die LÄNGE EINER WORDMAP ist gegeben durch

$$l \begin{cases} (C_{\mathbb{F}} \rightarrow M^*) \rightarrow \mathbb{N}_0 \\ w \mapsto \sum_{c \in C_{\mathbb{F}}} |w(c)| \end{cases} \quad (6.62)$$

2. Die ENTSTEHUNGSNUMMER EINES KNOTENS ist wie folgt definiert:

$$nr \begin{cases} V^* \rightarrow \mathbb{N} \\ v_i \mapsto i \end{cases} \quad (6.63)$$

Es gilt für die Knotenmengen im n -ten Schritt des Algorithmus:

$$\forall v \in V_N^n : \forall v' \in V^n : nr(v) > nr(v') \quad (6.64)$$

Dabei geschieht die Nummernvergabe nach der Entstehungszeit.

3. Die PRIORITÄT $\rho(v)$ eines Knotens $v = (s, w)$ des Erreichbarkeitsgraphen ist wie folgt definiert:

$$\rho \begin{cases} V^* \rightarrow \mathbb{Z} \times \mathbb{Z} \\ v \mapsto (-l(w), -nr(v)) \end{cases} \quad (6.65)$$

Dabei werden die Prioritäten zweier Knoten $v = (s, w)$ und $v' = (s', w')$ lexikographisch verglichen:

$$\rho(v) > \rho(v') \Leftrightarrow (-l(w) > -l(w')) \vee (l(w) = l(w') \wedge -nr(v) > -nr(v')) \quad (6.66)$$

Die Länge einer Wordmap ergibt sich aus der Summe der Längen der zugeordneten Wörter. Es kann jeweils mehrere Knoten des Erreichbarkeitsgraphen geben, welche die gleiche Länge besitzen. Um durch die Priorität eine eindeutige Anordnung der Knoten zu erhalten, wird bei deren Berechnung noch zusätzlich die Entstehungsnummer des Knotens, das heißt dessen Entstehungszeit, berücksichtigt. Ein Knoten ist daher höher priorisiert als ein anderer, falls er eine kürzere Wordmap hat beziehungsweise – wenn deren Wordmaps gleich lang sind – falls er früher erzeugt wurde. Letzteres bevorzugt tendentiell näher bei der Wurzel liegende Knoten des Erreichbarkeitsgraphen.

Die Tiefe eines Knotens, das heißt sein Abstand zum Wurzelknoten v_{root} , wird mittels der Funktion $\text{succ}()$ ermittelt, welche in Definition 6.4.23 beschrieben ist:

DEFINITION 6.4.14:

1. Die TIEFE EINES KNOTENS ist gegeben durch:

$$\text{depth} \begin{cases} V^* \rightarrow \mathbb{N}_0 \\ v \mapsto \min\{n \in \mathbb{N}_0 \mid v \in \text{succ}^n(v_{\text{root}})\} \end{cases} \quad (6.67)$$

Die Anzahl n der Aufrufe von $\text{succ}()$ entspricht dem Abstand des Knotens v vom Wurzelknoten v_{root} .

2. Die PRIORITÄT EINES DEBLOCKERS ist gegeben durch:

$$\rho \begin{cases} D^* \mapsto \mathbb{Z} \times \mathbb{Z} \\ (v, w, t) \mapsto (-\text{depth}(v), -nr(v)) \end{cases} \quad (6.68)$$

Dabei werden die Prioritäten der Deblocker $d = (v, w_v, t)$ und $d' = (v', w_{v'}, t')$ lexikographisch verglichen.

$$p(d) > p(d') \Leftrightarrow (-\text{depth}(v) > -\text{depth}(v') \vee (\text{depth}(v) = \text{depth}(v') \wedge (-nr(v) > -nr(v')))) \quad (6.69)$$

Ein Deblocker d ist damit höher priorisiert als ein Deblocker d' , wenn die Tiefe des zu d gehörenden Knotens v geringer ist als die Tiefe des zu d' gehörenden Knotens v' . Sind diese Tiefen gleich, entscheidet die Entstehungsreihenfolge der Knoten.

Auswahl eines Knotens beziehungsweise eines Deblockers: Die Funktion $\text{select}()$ ist überladen. Eine auf Knotenmengen definierte Variante von $\text{select}()$ dient zur Auswahl eines Knotens aus einer Menge von Knoten des Erreichbarkeitsgraphen, wobei jeweils der Knoten mit der höchsten Priorität ausgewählt wird. Daneben gibt es auch eine gleichnamige Auswahlfunktion für Deblocker.

DEFINITION 6.4.15:

1. Die Funktion $\text{select}()$ zur Auswahl eines Knotens aus einer Knotenmenge ist definiert durch:

$$\text{select} \begin{cases} P(V^*) \setminus \emptyset \rightarrow V^* \\ V \mapsto v \text{ mit } \forall v' \in V \setminus \{v\} : p(v) > p(v') . \end{cases} \quad (6.70)$$

2. Die Funktion $\text{select}()$ zur Auswahl eines Deblockers aus einer Menge von Deblockern ist definiert durch:

$$\text{select} \begin{cases} P(D^*) \setminus \emptyset \rightarrow D^* \\ D \mapsto d \text{ mit } \forall d' \in D \setminus \{d\} : p(d) > p(d') . \end{cases} \quad (6.71)$$

Die Auswahlfunktion für Deblocker wählt den Deblocker aus, dessen Knoten näher bei der Wurzel liegt.

Kommutativität und Notwendigkeit der Anwendung einer Transition: Dieser Unterabschnitt beschreibt die für das Verständnis des in dem Model-Checking-Algorithmus enthaltenen PARTIAL-ORDER-REDUCTION-Verfahrens notwendigen Grundlagen.

DEFINITION 6.4.16: Die Funktion $\text{isWaiting}()$ ermittelt für eine Transition t , einen Knoten $v = (s, w)$ und einen Fifomaten f , ob durch eine beliebige Lese- oder Schreiboperation die Wordmap w so angepaßt werden kann, daß die Transition t ihre Operation durchführen kann.

$$\text{isWaiting} \begin{cases} V^* \times T_{\mathbb{F}} \rightarrow \mathbb{B} \\ ((s, w), t) \mapsto \neg \text{enabled}((s, w), t) \wedge \\ \exists w' \in (C_{\mathbb{F}} \rightarrow M^*) : \text{readingEnabled}((s, w \oplus_r w'), t) \vee \\ \text{writingEnabled}((s, w \ominus_l w'), t) \end{cases} \quad (6.72)$$

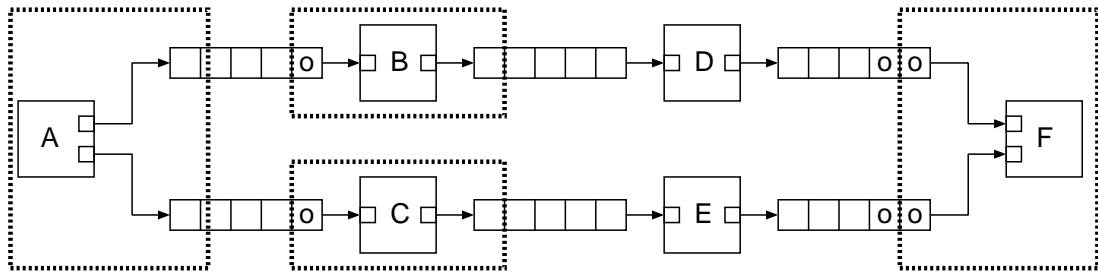


Abbildung 6.10: Beispiel zur Kommutativität von Fifomaten

Eine Transition t ist also wartend, wenn durch eine Schreib- oder Leseaktion die entsprechenden Fifoinhalte so verändert werden können, daß t angewandt werden kann.

Die folgende Funktion $\text{isCommutative}()$ ermittelt für einen gegebenen Knoten des Erreichbarkeitsgraphen $v = (s, w)$ und einen Fifomaten f , ob irgendeine Aktion in einer Transition der aktuellen atomaren Operationen von f wartend ist. Ist dies nicht der Fall, so wird dieser Fifomat f als KOMMUTATIV bezeichnet, da seine Expansion mit der aller anderen Fifomaten vertauscht werden kann, ohne am Endergebnis etwas zu ändern. Bei atomaren Operationen wird $\text{isCommutative}()$ für alle enthaltenen Transitionen rekursiv aufgerufen.

DEFINITION 6.4.17: Die Funktion $\text{isCommutative}()$ ermittelt, ob ein Fifomat kommutativ ist:

$$\text{isCommutative} \left\{ \begin{array}{l} V^* \times \mathbb{F} \rightarrow \mathbb{B} \\ ((s, w), f) \mapsto \forall t = (s(f), \text{op}, w_t, q') \in T_f : \neg \text{isWaiting}((s, w), t) \\ \quad \wedge (q' \in Q_{m,f} \vee \text{isCommutative}(\text{apply}((s, w), t), f)) \end{array} \right. \quad (6.73)$$

Abbildung 6.10 zeigt einen Datenflußgraphen, in welchem vier Datenflußkomponenten A, B, C und F rechenbereit sind. Die Rechenbereitschaft wird durch die gepunktete Umrandung verdeutlicht. Der zur Datenflußkomponente B gehörende Fifomat kann seine atomare Operation, welche das Lesen eines Tokens o von der Eingabefifo und das Schreiben eines Tokens o in die Ausgabefifo beinhaltet, ausführen. Da in den vorliegenden Datenflußparadigmen jeweils genau eine Datenflußkomponente aus einer Fifo liest beziehungsweise in eine Fifo schreibt und keine dieser Aktionen wartend ist, können die vier Datenflußkomponenten in beliebiger Reihenfolge ausgeführt werden, ohne sich gegenseitig zu beeinflussen. Damit sind die Fifomaten dieser vier Datenflußkomponenten – und somit die Datenflußkomponenten selbst – KOMMUTATIV zueinander.

Die Funktionen $\text{isPrefix}()$ und $\text{isSuffix}()$ sind Hilfsfunktionen, die überprüfen, ob eine Wordmap Präfix beziehungsweise Suffix einer anderen Wordmap ist.

DEFINITION 6.4.18: Die Funktion $\text{isPrefix}()$ ermittelt, ob eine Wordmap w Präfix einer anderen Wordmap w' ist:

$$\text{isPrefix} \begin{cases} (C_{\mathbb{F}} \rightarrow M^*) \rightarrow \mathbb{B} \\ (w, w') \mapsto w \leq w' \end{cases} \quad (6.74)$$

Die Funktion $\text{isSuffix}()$ ermittelt, ob eine Wordmap w Suffix einer anderen Wordmap w' ist:

$$\text{isSuffix} \begin{cases} (C_{\mathbb{F}} \rightarrow M^*) \rightarrow \mathbb{B} \\ (w, w') \mapsto \exists w'' \in (C_{\mathbb{F}} \rightarrow M^*) : w' = w'' \oplus_r w \end{cases} \quad (6.75)$$

DEFINITION 6.4.19: Die Funktion $\text{isNecessary}()$ ermittelt, ob die Ausführung eines Fifomaten f notwendig ist, um vom aktuellen Knoten v des Erreichbarkeitsgraphen in den Wurzelknoten zu gelangen. Dabei sind s_{root} und w_{root} die Statemap beziehungsweise die Wordmap des Wurzelknotens v_{root} des Erreichbarkeitsgraphen.

$$\text{isNecessary} \begin{cases} V^* \times \mathbb{F} \rightarrow \mathbb{B} \\ ((s, w), f) \mapsto s(f) \neq s_{\text{root}}(f) \vee \\ \exists t = (q_1, \text{op}, w_t, q_2) \in T_f : \exists c \in C_t : \\ \text{op} = ! \wedge \neg \text{isSuffix}(w_{\text{root}}, w) \vee \text{op} = ? \wedge \neg \text{isPrefix}(w, w_{\text{root}}) \end{cases} \quad (6.76)$$

Die Arbeitsweise von $\text{isNecessary}()$ ist in Abbildung 6.11 anhand eines zufällig ausgewählten Knotens des zu dem in Teilabbildung (a) dargestellten Datenflußgraphen gehörenden Erreichbarkeitsgraphen veranschaulicht. Ist als Zielknoten der Wurzelknoten des Erreichbarkeitsgraphen v_{root} vorgegeben, so ermittelt $\text{isNecessary}()$ für einen Fifomaten f zum einen, ob dieser seinen Zustand verändern muß, um ausgehend von der aktuellen Statemap die Statemap des Wurzelknotens zu erreichen. Zum anderen wird überprüft, welcher Fifomat in welche Fifo schreiben oder aus dieser lesen muß, damit der Unterschied zwischen der Wordmap des aktuellen Knotens und der Wordmap des Wurzelknotens des Erreichbarkeitsgraphen kleiner wird. Dabei gilt, daß in jede Fifo genau ein Fifomat schreibt und genau ein Fifomat aus dieser Fifo liest.

Die Funktion $\text{isNecessary}()$ stellt eine wichtige Ergänzung von $\text{isCommutative}()$ dar. Denn die Ausführung des Fifomaten einer Quellkomponente wie der Datenflußkomponente A in Abbildung 6.11 ist beispielsweise immer kommutativ mit der Ausführung der Fifomaten aller anderen rechenbereiten Datenflußkomponenten. Wird aber eine Quellkomponente beliebig oft ausgeführt – vorausgesetzt die Kapazitäten der zugehörigen Fifos sind unendlich –, erreicht man den Wurzelknoten des Erreichbarkeitsgraphen nie wieder. Die Funktion $\text{isNecessary}()$ ermittelt, ob die Ausführung eines Fifomaten notwendig zur Erreichung des Wurzelknotens ist. So erkennt beispielsweise $\text{isNecessary}()$, daß nach der Ausführung der Quellkomponente A

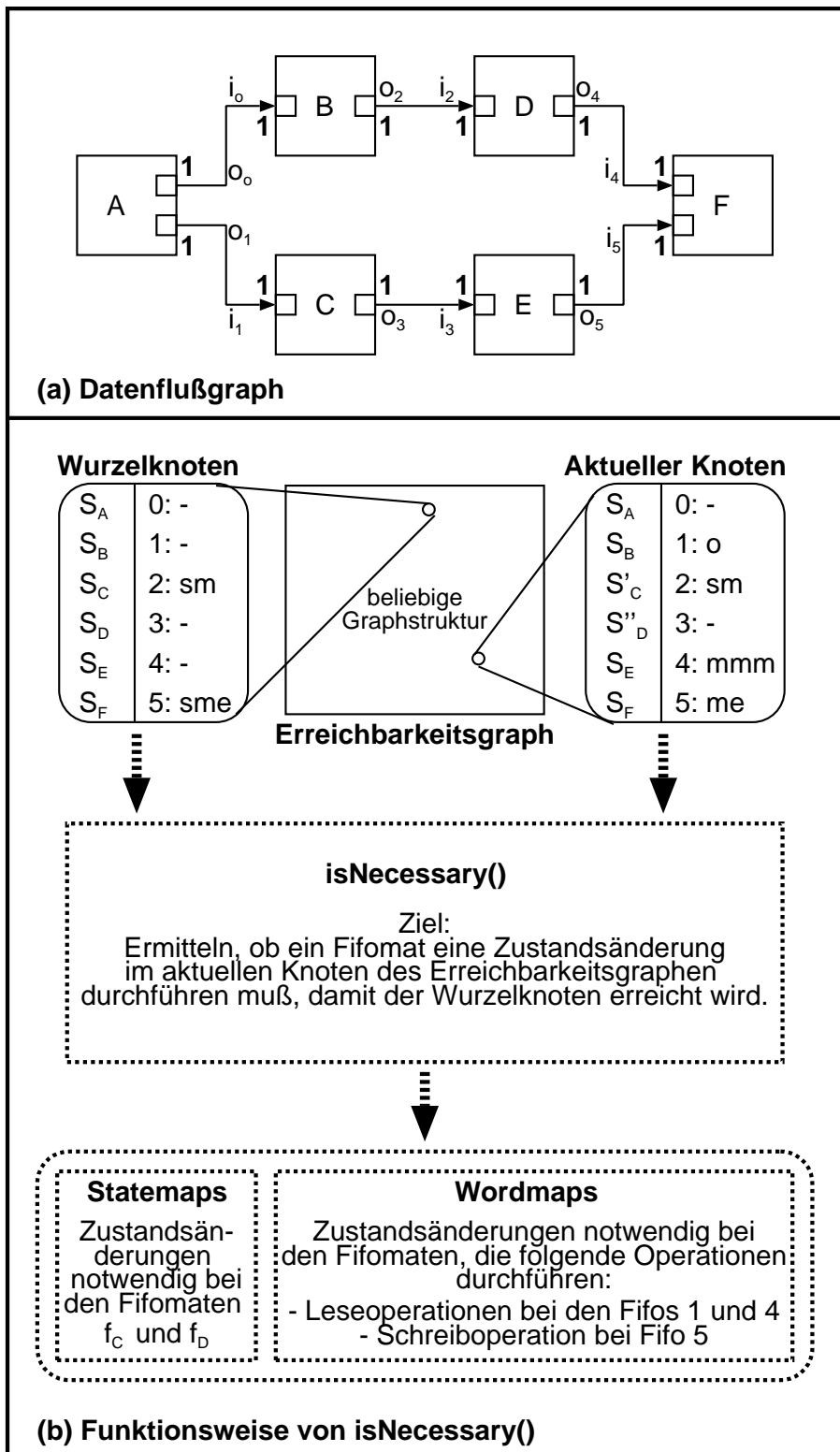


Abbildung 6.11: Beispiel zur Bestimmung der notwendigerweise auszuführenden Fifomaten

in Abbildung 6.11 zuerst die rechenbereiten internen Datenflußkomponenten und die Senkenkomponenten auszuführen sind, um den Wurzelknoten zu erreichen. Es werden zum einen die Zustände der Fifomaten betrachtet, wobei man in dem Beispiel erkennt, daß die Fifomaten der Datenflußkomponenten **C** und **D** sich in von der Statemap des Wurzelknotens unterschiedlichen Zuständen befinden (S'_C statt S_C beziehungsweise S''_D statt S_D). Zum anderen ergibt der Vergleich der Wordmap des aktuellen Knotens und der Wordmap des Wurzelknotens, daß aus den Fifos mit den Indizes 1 und 4 gelesen werden muß und in die Fifo mit dem Index 5 geschrieben werden muß, damit ausgehend von der aktuellen Wordmap die Wordmap des Wurzelknotens erreicht wird.

DEFINITION 6.4.20: Die Funktion *isCommutativeAndNecessary* ermittelt, ob ein Fifomat f in einem Knoten v des Erreichbarkeitsgraphen kommutativ und ob die Ausführung von f notwendig ist:

$$\text{isCommutativeAndNecessary} \begin{cases} V^* \times \mathbb{F} \rightarrow \mathbb{B} \\ (v, f) \mapsto \text{isCommutative}(v, f) \wedge \text{isNecessary}(v, f) \end{cases} \quad (6.77)$$

Auswahl eines Fifomaten: Im folgenden werden eine Reihe von Auswahlfunktionen definiert, welche zur Bestimmung der bei der Expansion eines Knotens des Erreichbarkeitsgraphen zu berücksichtigenden Fifomaten benötigt werden.

DEFINITION 6.4.21:

1. Die Funktion *selectFifomatonInTransientState()* überprüft, ob es im aktuellen Knoten (s, w) des Erreichbarkeitsgraphen einen Fifomaten gibt, der sich in einem transienten Zustand befindet.

$$\text{selectFifomatonInTransientState} \begin{cases} V^* \rightarrow \mathbb{F} \cup \{\emptyset\} \\ (s, w) \mapsto \begin{cases} f \in \mathbb{F} \text{ mit } s(f) \in Q_{t,f} \\ \emptyset \text{ sonst} \end{cases} \end{cases} \quad (6.78)$$

Dabei gilt, daß entweder genau ein oder kein Fifomat mit dieser Eigenschaft existiert.

2. Die Funktion *selectComAndNec()* wählt heuristisch einen Fifomaten aus, der sich in Bezug zum aktuellen Knoten des Erreichbarkeitsgraphen kommutativ verhält und dessen Ausführung zum Erreichen des Zielknotens notwendig ist. Es ist möglich, daß kein solcher Fifomat existiert. In diesem Fall wird die leere Menge zurückgeliefert.

$$\text{selectComAndNec} \begin{cases} V^* \rightarrow \mathbb{F} \cup \{\emptyset\} \\ (s, w) \mapsto \begin{cases} f \in \mathbb{F} \text{ mit } \text{isCommutativeAndNecessary}((s, w), f) \\ \emptyset \text{ sonst} \end{cases} \end{cases} \quad (6.79)$$

3. Die Funktion $\text{selectFifomaton}()$ wählt einen Fifomaten aus, wobei zuerst die sich in einem transienten Zustand befindlichen Fifomaten berücksichtigt werden.

$$\text{selectFifomaton} \begin{cases} V^* \rightarrow \mathbb{F} \cup \{\emptyset\} \\ (s, w) \mapsto \begin{cases} \text{selectFifomatonInTransientState}((s, w)) \\ \text{falls } \text{selectFifomatonInTransientState}((s, w)) \neq \emptyset \\ \text{selectComAndNec}((s, w)) \\ \text{sonst} \end{cases} \end{cases} \quad (6.80)$$

Expandieren eines Knotens: Die Expansion eines Knotens v des Erreichbarkeitsgraphen, bei welcher ein Sohn oder mehrere Söhne von v erzeugt werden, geschieht mittels der Funktion $\text{expand}()$. Diese Funktion ist überladen, wobei unterschieden wird, ob die Expansion anhand eines ausgewählten Fifomaten beziehungsweise einer ausgewählten Transition erfolgt. Eine weitere Variante erzeugt alle Söhne eines Knoten des Erreichbarkeitsgraphen unter Berücksichtigung einer eventuell gerade in Bearbeitung befindlichen atomaren Operation.

DEFINITION 6.4.22:

1. Die Funktion $\text{expand}()$ erzeugt zu einem Knoten v alle Söhne, deren Statemaps und Wordmaps durch Ausführen des Fifomaten f aus der Statemap beziehungsweise Wordmap von v hervorgehen:

$$\text{expand} \begin{cases} V^* \times \mathbb{F} \rightarrow P(V^*) \\ (v, f) \mapsto \{v' \in V^* \mid \exists t \in T_f : \text{enabled}(v, t) \wedge v' = \text{apply}(v, t)\} \end{cases} \quad (6.81)$$

2. Die Funktion $\text{expand}()$ erzeugt zu einem Knoten v alle Söhne, deren Statemaps und Wordmaps durch Ausführen der Transition t aus der Statemap beziehungsweise Wordmap von v hervorgehen:

$$\text{expand} \begin{cases} V^* \times T_{\mathbb{F}} \rightarrow P(V^*) \\ (v, t) \mapsto \{v' \in V^* \mid \text{enabled}(v, t) \wedge v' = \text{apply}(v, t)\} \end{cases} \quad (6.82)$$

3. Die Funktion $\text{expand}'()$ erzeugt zu einem Knoten des Erreichbarkeitsgraphen alle Söhne⁸.

$$\text{expand}' \begin{cases} V^* \rightarrow P(V^*) \\ v \mapsto \{v' \in V^* \mid \exists t \in T_{\mathbb{F}} : \text{enabled}(v, t) \wedge v' = \text{apply}(v, t)\} \end{cases} \quad (6.83)$$

⁸Der Funktionsbezeichner ist mit dem Zeichen „'“ versehen, da keine Unterscheidung zu $\text{expand} : V^* \rightarrow P(V^*)$ anhand der Funktionssignatur möglich ist. Diese Form der Unterscheidung von Funktionen mit identischer Signatur und verwandter Aufgabe wird in der gesamten Arbeit eingesetzt.

4. Die Funktion $\text{expand}()$ erzeugt zu einem Knoten des Erreichbarkeitsgraphen dessen Söhne, wobei im Falle des Vorhandenseins eines Fifomaten in einem transienten Zustand nur dieser bei der Expansion berücksichtigt wird, um seine atomare Operation nicht zu unterbrechen. Ist stattdessen ein kommutativer Fifomat vorhanden, wird allein dieser Fifomat zu expandieren versucht. In allen anderen Fällen werden alle Fifomaten bei der Expansion betrachtet.

$$\text{expand} \begin{cases} V^* \rightarrow P(V^*) \\ v \mapsto \begin{cases} \text{expand}(v, \text{selectFifomaton}(v)) \text{ falls } \text{selectFifomaton}(v) \neq \emptyset \\ \text{expand}'(v) \text{ sonst} \end{cases} \end{cases} \quad (6.84)$$

Mit Hilfe von $\text{expand}()$ lassen sich die Mengen der direkten und indirekten Nachfolger eines Knotens des Erreichbarkeitsgraphen beschreiben.

DEFINITION 6.4.23: Die Funktion $\text{succ}()$ bestimmt die MENGE DER DIREKTEN NACHFOLGER eines Knotens v des Erreichbarkeitsgraphen:

$$\text{succ} \begin{cases} V^* \rightarrow P(V^*) \\ v \mapsto \text{expand}(v) \end{cases} \quad (6.85)$$

Die MENGE DER INDIREKTEN NACHFOLGER ist gegeben durch

$$\text{succ}^+(v) = \bigcup_{n>1} \text{succ}^n(v). \quad (6.86)$$

Somit ist die MENGE ALLER NACHFOLGER gegeben durch:

$$\text{succ}^*(v) = \text{succ}(v) \cup \text{succ}^+(v). \quad (6.87)$$

Die Zahl $n \in \mathbb{N}_0$ stellt dabei die Anzahl der Aufrufe von $\text{succ}()$ und zugleich die Länge des Pfades zwischen den Knoten plus 1 dar.

Mit Hilfe der Nachfolger eines Knotens lassen sich Zyklen in einer Knotenmenge erkennen.

DEFINITION 6.4.24: Die Funktion $\text{existsCycle}()$ ermittelt, ob in einer Menge von Knoten des Erreichbarkeitsgraphen ein Zyklus existiert:

$$\text{existsCycle} \begin{cases} P(V^*) \rightarrow \mathbb{B} \\ V \mapsto \exists v \in V : v \in \text{succ}^*(v) \end{cases} \quad (6.88)$$

Insbesondere interessieren in dieser Arbeit Zyklen im Erreichbarkeitsgraphen, die den Wurzelknoten beinhalten.

DEFINITION 6.4.25: Die Funktion $\text{existsRootCycle}()$ ermittelt, ob es einen Zyklus gibt, welcher den Wurzelknoten beinhaltet:

$$\text{existsRootCycle} \begin{cases} P(V^*) \rightarrow \mathbb{B} \\ V \mapsto v_{\text{root}} \in V \wedge v_{\text{root}} \in \text{succ}^*(v_{\text{root}}) \end{cases} \quad (6.89)$$

Blockieren und Deblokieren: Die im folgenden vorgestellten Funktionen beschreiben den Blockier/Deblockier-Mechanismus des Model-Checking-Algorithmus.

DEFINITION 6.4.26: Die Funktion $\mathit{determineTransitions}()$ bestimmt zu einem gegebenen Startknoten v und einem Zielknoten v' die Menge der Transitionen von v nach v' .

$$\mathit{determineTransitions} \begin{cases} V^* \times V^* \rightarrow P(T_{\mathbb{F}}) \\ (v, v') \mapsto \{t \in T_{\mathbb{F}} \mid v' \in \mathit{expand}(v, t) \cap \mathit{expand}(v)\} \end{cases} \quad (6.90)$$

In $\mathit{determineTransitions}()$ tauchen zwei Varianten von $\mathit{expand}()$ auf. $\mathit{expand}(v, f, t)$ liefert den Nachfolgeknoten v' , wenn man im Knoten v den Fifomaten f mit Hilfe der Transition t expandiert. $\mathit{expand}(v)$ zeigt an, ob diese Expansion zulässig ist unter Berücksichtigung der Kommutativität und eventuell vorhandener Fifomaten in transienten Zuständen, die sich inmitten einer atomaren Operation befinden.

DEFINITION 6.4.27: Die Funktion $\mathit{determineWordmaps}()$ bestimmt zu einem Knoten v , einem Deblocker $(v', w_{v'}, t)$ und einer Debblockermenge D die Menge aller Wordmaps w_v , die durch Ausführen von $\mathit{unapply}()$ entlang aller vorhandenen Kanten von v' nach v rückwärts propagiert werden. Falls bereits ein Deblocker bei dem Knoten v existiert, der w_v enthält, dann wird w_v nicht in die Rückgabemenge aufgenommen.

$$\mathit{determineWordmaps} \begin{cases} V^* \times D^* \times P(D^*) \rightarrow P(C_{\mathbb{F}} \rightarrow M^*) \\ (v, (v', w_{v'}, t), D) \mapsto \{w \in C_{\mathbb{F}} \rightarrow M^* \mid \\ \quad \forall t' \in \mathit{determineTransitions}(v, v') : \\ \quad w_v = \mathit{unapply}(w_{v'}, t') \\ \quad \wedge (v, w_v, t) \notin D\} \end{cases} \quad (6.91)$$

Die Funktion $\mathit{determineDeblocker}()$ ist überladen. Die verschiedenen Varianten dieser Funktion werden zur Aktualisierung der Debblockermengen verwendet.

DEFINITION 6.4.28:

1. Die Funktion $\mathit{determineDeblocker}()$ bestimmt die Menge der Deblocker, die bei der Expansion eines Knotens v des Erreichbarkeitsgraphen entstehen. Dabei werden zu dem Knoten v alle Fifomaten mit aktuell nicht ausführbaren Lesetransitionen betrachtet. Ist es möglich, die Fifoinhalte von w zu w' zu ergänzen, so daß die Transition ausführbar wird, so wird aus w' ein Deblocker erzeugt.

$$\mathit{determineDeblocker} \begin{cases} V^* \rightarrow P(D^*) \\ v = (s, w) \mapsto \{(v, w_v, t) \in D^* \mid \\ \quad \exists f \in \mathbb{F} : \exists t = (s(f), ?, w_t, q_2) \in T_f : \\ \quad \neg \mathit{readingEnabled}(v, t) \\ \quad \wedge (\exists w' \in (C_{\mathbb{F}} \mapsto M^*) : w_t = w \oplus_r w' \\ \quad \wedge w_v = \mathit{unapply}(w', t))\} \end{cases} \quad (6.92)$$

2. Die Funktion **determineDeblocker**() bestimmt zu einem Knoten v , einer Knotenmenge V und einer Deblockermenge D , ob es einen Nachfolger v' von v gibt, dem ein Deblocker $(v', w_{v'}, t) \in D$ zugeordnet ist. Mit Hilfe von **determineWordmaps**() werden die Wordmaps der zu v gehörenden und noch nicht erzeugten Deblocker bestimmt.

$$\mathbf{determineDeblocker} \left\{ \begin{array}{l} V^* \times P(V^*) \times P(D^*) \rightarrow P(D^*) \\ (v, V, D) \mapsto \{(v, w_v, t) \in D^* \mid \exists v' \in \mathbf{succ}(v) \cap V : \\ \exists (v', w_{v'}, t) \in D : \\ w_v \in \mathbf{determineWordmaps}(v, (v', w_{v'}, t), D)\} \end{array} \right. \quad (6.93)$$

3. Die Funktion **determineDeblocker**() bestimmt zu einem Deblocker (v, w_v, t) und insbesondere dem Knoten v , dem der Deblocker zugeordnet ist, und einer Deblockermenge D alle Deblocker, die zu dessen direkten Vorgängern gehören und die noch nicht in D enthalten sind.

$$\mathbf{determineDeblocker} \left\{ \begin{array}{l} D^* \times P(D^*) \rightarrow P(D^*) \\ ((v, w_v, t), D) \mapsto \{(v', w_{v'}, t) \in D^* \mid v \in \mathbf{succ}(v') \wedge w_{v'} \in \\ \mathbf{determineWordmaps}(v', (v, w_v, t), D)\} \end{array} \right. \quad (6.94)$$

4. Die Funktion **determineDeblocker'**() wird in **heuristicSearch**() (siehe Algorithmus 6.4.3) eingesetzt. Die Funktion bestimmt zu einem Deblocker (v, w_v, t) und insbesondere dem Knoten v , dem der Deblocker zugeordnet ist, und einer Deblockermenge D den Deblocker, der zu dem direkten Vorgänger von v mit der kleinsten Entstehungsnummer gehört und der nicht in D enthalten ist.

$$\mathbf{determineDeblocker}' \left\{ \begin{array}{l} D^* \times P(D^*) \rightarrow P(D^*) \\ ((v, w_v, t), D) \mapsto \{(v', w_{v'}, t) \mid v \in \mathbf{succ}(v') \wedge \\ \forall v'' \in \mathbf{succ}(v') \setminus \{v \mid nr(v) < nr(v'')\} \wedge w_{v'} \in \\ \mathbf{determineWordmaps}(v', (v, w_v, t), D)\} \end{array} \right. \quad (6.95)$$

Bei der Erzeugung aller Nachfolgerknoten eines Knotens v des Erreichbarkeitsgraphen entsteht ein Deblocker, wenn eine Lesetransition eines Fifomaten nicht ausgeführt werden kann. Der Deblocker setzt sich aus dem aktuellen Knoten v , der verbotenen Transition t und der Desired Wordmap w_v zusammen (vergleiche Definition 6.4.29).

DEFINITION 6.4.29: Sei $d = (v, w_v, t) \in D^*$. Dann wird die Wordmap w_v , welche die Fifobelegung für den Knoten v des Erreichbarkeitsgraphen beinhaltet, die ein Erreichen eines verbotenen Knotens ermöglicht hätte, als **DESIRED WORDMAP** bezeichnet.

DEFINITION 6.4.30: Die Funktion **deblock**() ermittelt zu einem blockierten Knoten v und einen diesen blockierenden Knoten v' , ob v die Anforderungen einer Desired Wordmap $w_{v'}$ bezüglich

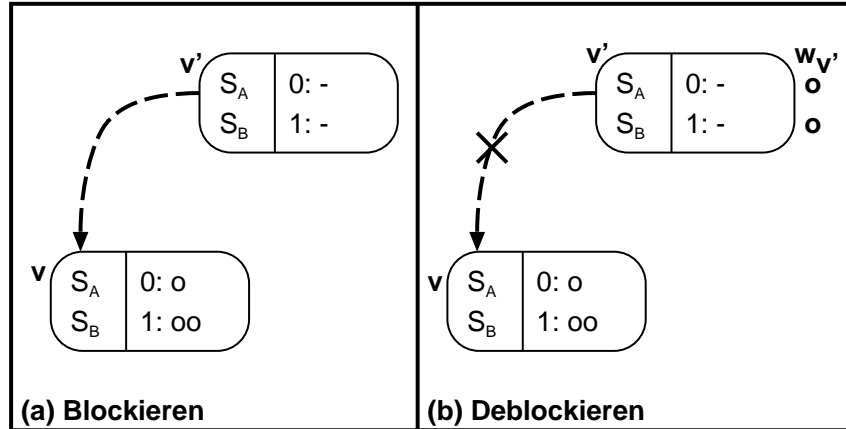


Abbildung 6.12: Beispiel zum Blockier/Deblockier-Mechanismus

der in der blockierten Transition t enthaltenen Fifos entspricht und damit zu deblockieren ist.

$$\text{deblock} \left\{ \begin{array}{l} V^* \times V^* \times (C_{\mathbb{F}} \rightarrow M^*) \times T_{\mathbb{F}} \rightarrow \mathbb{B} \\ (v = (s, w), v' = (s', w'), w_{v'}, t) \mapsto \left(\left(\forall c \in C_t : w_{v'}(c) \begin{array}{l} \geq \\ \leq \end{array} w(c) \right) \right. \\ \left. \wedge (\exists c \in C_t : w'(c) < w_{v'}(c) \wedge w'(c) < w(c)) \right) \end{array} \right. \quad (6.96)$$

Abbildung 6.12 (b) veranschaulicht die Arbeitsweise der Funktion $\text{deblock}()$. Der Knoten des Erreichbarkeitsgraphen v' blockiert v . Aber da die Desired Wordmap $w_{v'}$ vergleichbar ist mit der Wordmap w von v und es mindestens eine Fifo gibt, wo die Fifobelegung von w' Präfix der gewünschten Belegung in $w_{v'}$ und der Belegung des blockierten Knotens in w ist, ermöglicht entweder die Wordmap w die Durchsuchung eines Pfades, der ausgehend von w' abgeschnitten war, oder w stellt zumindest eine Verbesserung im Hinblick auf die Desired Wordmap $w_{v'}$ im Vergleich zu w' dar. Daher wird v deblockiert.

DEFINITION 6.4.31: Die Funktion $\text{block}()$ ermittelt zu einem Knoten v , ob dieser durch den Knoten v' blockiert wird und ob ein dem Knoten v' zugeordneter Deblocier in der Menge D enthalten ist, welcher v deblockiert.

$$\text{block} \left\{ \begin{array}{l} V^* \times V^* \times P(D^*) \rightarrow \mathbb{B} \\ (v = (s, w), v' = (s', w'), D) \mapsto v \neq v' \wedge s = s' \wedge w' \leq w \\ \wedge \nexists d = (v', w_{v'}, t) \in D : \\ \text{deblock}(v, v', w_{v'}, t) \end{array} \right. \quad (6.97)$$

Abbildung 6.12 (a) veranschaulicht die Funktionsweise von $\text{block}()$. Die Statemaps von v und v' sind identisch und die Wordmap w' von v' ist ein Präfix der Wordmap w von v . Damit sind die

in $\text{block}()$ aufgeführten Bedingungen erfüllt und der Knoten v wird blockiert.

DEFINITION 6.4.32: Die Funktion $\text{determineBlockedSet}()$ ermittelt zu einem Knoten v , die Menge aller Knoten v' , die von v blockiert werden, wobei diese Blockade durch keinen Deblocker der Menge D aufgehoben wird.

$$\text{determineBlockedSet} \left\{ \begin{array}{l} V^* \times P(V^*) \times P(D^*) \rightarrow P(V^*) \\ (v, V, D) \mapsto \{v' \in V \mid \text{block}(v', v, D)\} \end{array} \right. \quad (6.98)$$

DEFINITION 6.4.33: Die Funktion $\text{determineDeblockedSet}()$ ermittelt zu einem Deblocker $d = (v', w_v, t)$ die Menge der von dem Knoten v blockierten Knoten, die deblockiert wird.

$$\text{determineDeblockedSet} \left\{ \begin{array}{l} D^* \times P(V^*) \times P(V^*) \times P(D^*) \rightarrow P(V^*) \\ ((v, w_v, t), V, V', D) \mapsto \{v' \in V \mid \text{deblock}(v', v, w_v, t) \\ \wedge \nexists v'' \in V' \setminus \{v\} : \\ \text{block}(v', v'', D)\} \end{array} \right. \quad (6.99)$$

Suchalgorithmen: Ziel des Model-Checking-Verfahrens ist es, einen Zyklus in den Wurzelknoten des Erreichbarkeitsgraphen zu finden. Grundlegend für den Erfolg dieses Vorgehens ist folgende Aussage.

SATZ 6.4.1: Die Gesamtknotenmenge V^* ist aufzählbar.

BEWEIS 6.4.1:

Gegeben: Sei \mathbb{F} die Menge der betrachteten Fifomaten.

Behauptung: Die Anzahl der Knoten des Erreichbarkeitsgraphen ist aufzählbar.

Beweis: Da jeweils nur eine endliche Anzahl von Fifomaten mit jeweils einer endlichen Anzahl von Zuständen betrachtet wird, ist auch die Anzahl der Statemaps $\mathbb{F} \rightarrow Q_{\mathbb{F}}$ begrenzt. Ebenso ist die Anzahl der Wordmaps $C_{\mathbb{F}} \rightarrow M^*$ einer bestimmten Länge begrenzt. Man kann also die Knoten des Erreichbarkeitsgraphen $V^* \subseteq (\mathbb{F} \rightarrow Q_{\mathbb{F}}) \times (C_{\mathbb{F}} \rightarrow M^*)$ wie folgt aufzählen:

1. Aufzählen aller Knoten $v \in V^*$ mit Wordmaps der Länge 0.
2. Aufzählen aller Knoten $v \in V^*$ mit Wordmaps der Länge 1.
3. und so fort.

Jeder dieser Teilschritte ist durchführbar, da die jeweils betrachteten Teilmengen von V^* endlich sind.

\Rightarrow Satz 6.4.1. □

Ein erster Ansatz für einen Suchalgorithmus wurde bereits in Abschnitt 6.4.4 und Abbildung 6.6 informell vorgestellt. Im folgenden ist dieser Algorithmus formal beschrieben.

Dabei gilt für alle in diesem Kapitel vorgestellten Suchalgorithmen, daß diese die Menge der Fifomaten \mathbb{F} als Eingabe erhalten. Da aber diese Menge von den Suchalgorithmen nicht verändert wird und somit konstant ist, wurde \mathbb{F} nicht als eigener Parameter durch die einzelnen Funktionen durchgereicht. Dies dient zur Erhöhung der Übersichtlichkeit.

ALGORITHMUS 6.4.1:

```

1  Algorithm bruteForceSearch() {
2       $V_P^0 := \{v_{\text{root}}\}; V_E^0 := \emptyset; i := 0$ 
3      while ( $V_P^i \neq \emptyset$ ) {
4           $v_{\text{best}} := \text{select}(V_P^i)$ 
5           $V_P^i := V_P^i \setminus \{v_{\text{best}}\}$ 
6           $V_E^{i+1} := V_E^i \cup \{v_{\text{best}}\}$ 
7           $V_{\text{valid}}^i := \text{expand}'(v_{\text{best}})$ 
8           $V_P^{i+1} := V_P^i \cup (V_{\text{valid}}^i \setminus V_E^{i+1})$ 
9           $i := i + 1$ 
10     }
11     return  $V_E^i$ 
12 }
```

Dabei besitzen `select()` und `expand()` folgende Eigenschaften:

LEMMA 6.4.1: *Jeder Knoten in V_P wird von `select()` irgendwann ausgewählt.*

BEWEIS 6.4.2:

Gegeben: Sei `select()` wie in Definition 6.4.15 vorgegeben. Sei $v = (s, w)$ Element der Menge der zurückgestellten Knoten V_P beliebig.

Behauptung: Dann wird jeder Knoten v in V_P von `select()` irgendwann ausgewählt.

Beweis:

1. $l(w) = \sum_{c \in C_{\mathbb{F}}} |w(c)| < \infty$

2. $\text{nr}(v) < \infty$

3. $\mathbf{p}(v) = (-l(w), -\text{nr}(v))$

4. $\mathbf{p}(v') > \mathbf{p}(v) \Leftrightarrow (l(w') < l(w)) \vee (l(w') = l(w) \wedge \text{nr}(v') < \text{nr}(v))$

\Rightarrow 5. $\{v' \in V \mid \mathbf{p}(v') > \mathbf{p}(v)\} < \infty$. Die Menge der Knoten mit höherer Priorität als v ist endlich.

\Rightarrow Lemma 6.4.1. □

LEMMA 6.4.2: *Jeder erreichbare Nachfolger v' eines ausgewählten Knotens v wird in V_P^i eingefügt, falls er nicht bereits Element von V_P^j ($j \leq i$) war.*

BEWEIS 6.4.3:

Gegeben: Seien $v \in V_P^i$ und $v' \in \text{succ}^*(v)$ beliebig.

Behauptung: Jeder erreichbare Nachfolger von v wird in V_P^i eingefügt, falls er nicht bereits Element von V_P^j ($j \leq i$) war.

Beweis:

1. Jeder Knoten in V_P wird irgendwann ausgewählt (Lemma 6.4.1)
2. Jeder ausgewählte Knoten wird expandiert (siehe Algorithmus 6.4.1).

⇒ Lemma 6.4.2. □

LEMMA 6.4.3: Ein aus V_P^i entfernter Knoten wird später ($j \geq i$) nicht mehr in V_P^i eingefügt.

BEWEIS 6.4.4:

Gegeben: Sei v Element der Menge der zurückgestellten Knoten V_P . Sei `bruteForceSearch()` der in Algorithmus 6.4.1 definierte Suchalgorithmus.

Behauptung: Dann wird ein aus V_P entfernter Knoten v nicht mehr in V_P eingefügt.

Beweis:

1. Jeder aus V_P entfernte Knoten v wird in V_E eingefügt (siehe Zeilen 4–6 des Algorithmus `bruteForceSearch()`).
2. Beim Einfügen neuer Knoten in V_P werden solche, die in V_E enthalten sind, nicht berücksichtigt (siehe Zeile 8 des Algorithmus `bruteForceSearch()`).

⇒ Lemma 6.4.3. □

Diese Aussagen bilden die Grundlagen für folgenden Satz:

SATZ 6.4.2: `bruteForceSearch()` zählt den Erreichbarkeitsgraphen G auf.

BEWEIS 6.4.5:

Gegeben: Sei ein Erreichbarkeitsgraph G gegeben.

Behauptung: Dann zählt `bruteForceSearch()` G auf.

Beweis: Aus den Lemmata 6.4.1, 6.4.2 und 6.4.3 folgt: jeder vom Wurzelknoten aus erreichbare Knoten wird betrachtet.

⇒ Satz 6.4.2. □

Der vorgestellte Brute-Force-Ansatz für eine Suche im Erreichbarkeitsgraphen wird nun zum Algorithmus `guidedSearch()` ausgebaut, der dem hier vorgestellten Model-Checking-Verfahren zugrundeliegt.

Der Algorithmus `guidedSearch()` konstruiert als erstes den Wurzelknoten v_{root} des Erreichbarkeitsgraphen RT . Die Statemap von v_{root} beinhaltet die Anfangszustände aller Fifomaten. Die

Wordmap von v_{root} beinhaltet alle Anfangsbelegungen der Fifos. Solange es zu bearbeitende Knoten in V_P oder Deblocker in D_P gibt und kein Zyklus zu v_{root} gefunden wurde, sucht der Algorithmus weiter.

ALGORITHMUS 6.4.2:

```

1  Algorithm guidedSearch() {
2       $V_E^0 := \emptyset; V_B^0 := \emptyset; V_P^0 := \{v_{\text{root}}\}; D_E^0 := \emptyset; D_P^0 := \emptyset; i := 0$ 
3      while  $((V_P^i \neq \emptyset \vee D_P^i \neq \emptyset) \wedge \neg \text{existsRootCycle}(V_E^i))$  {
4          if  $(V_P^i \neq \emptyset)$  {
5               $v_{\text{best}} := \text{select}(V_P^i)$ 
6               $V_E^{i+1} := V_E^i \cup \{v_{\text{best}}\}$ 
7               $V_{\text{tmp}} := \bigcup_{v \in \text{expand}(v_{\text{best}})} \text{determineBlockedSet}(v, \text{expand}(v_{\text{best}}), D_E^i \cup D_P^i)$ 
8                   $\cup \bigcup_{v \in \text{expand}(v_{\text{best}})} \text{determineBlockedSet}(v, V_E^i \cup V_B^i \cup V_P^i, D_E^i \cup D_P^i)$ 
9                   $\cup \bigcup_{v \in V_E^i \cup V_B^i \cup V_P^i} \text{determineBlockedSet}(v, \text{expand}(v_{\text{best}}), D_E^i \cup D_P^i)$ 
10              $V_B^{i+1} := (V_B^i \cup V_{\text{tmp}}) \setminus V_E^{i+1}$ 
11              $V_P^{i+1} := (V_P^i \cup \text{expand}(v_{\text{best}})) \setminus (V_E^{i+1} \cup V_B^{i+1})$ 
12              $D_P^{i+1} := (D_P^i \cup \text{determineDeblocker}(v_{\text{best}}, V_E^{i+1} \cup V_B^{i+1} \cup V_P^{i+1}, D_E^i \cup D_P^i)$ 
13                  $\cup \text{determineDeblocker}(v_{\text{best}}) \setminus D_E^i$ 
14         }
15         elseif  $(D_P^i \neq \emptyset)$  {
16              $(v, w_v, t) := \text{select}(D_P^i)$ 
17              $V_{\text{tmp}} := \text{determineDeblockedSet}((v, w_v, t), V_B^i, V_E^i \cup V_B^i \cup V_P^i, D_E^i \cup D_P^i)$ 
18              $V_B^{i+1} := V_B^i \setminus V_{\text{tmp}}$ 
19              $V_P^{i+1} := V_P^i \cup V_{\text{tmp}}$ 
20              $D_E^{i+1} := D_E^i \cup \{(v, w_v, t)\}$ 
21              $D_P^{i+1} := (D_P^i \cup \text{determineDeblocker}(v, D_E^{i+1} \cup D_P^{i+1})) \setminus D_E^{i+1}$ 
22         }
23          $i := i + 1$ 
24     }
25     return  $V_E^i \cup V_P^i \cup V_B^i$ 
26 }

```

Im einzelnen werden dabei folgende Schritte ausgeführt⁹:

- **Expandieren von zu bearbeitenden Knoten (Zeilen 4–14):**

- **AUSWAHL EINES KNOTENS (ZEILE 5):** Die zurückgestellten Knoten in V_P sind in einer Prioritätswarteschlange angeordnet. Je kürzer die Wordmap desto höher priorisiert ist der Knoten. Sind die Wordmaps identisch, so wird derjenige Knoten höher priorisiert, der früher erzeugt wurde (vergleiche Definition 6.4.13 der Funktion $\mathfrak{p}()$). Diese Prioritätsvergabe bedingt, daß der Suchalgorithmus Knoten bevorzugt, die

⁹Bei der Darstellung von Algorithmen werden in dieser Arbeit gelegentlich Funktionen mit denselben Parametern zur Veranschaulichung mehrmals aufgerufen. In der Implementierung werden die Ergebnisse der Funktionsaufrufe aber zwischengespeichert und Mehrfachaufrufe vermieden.

einen geringen Speicherverbrauch der zu untersuchenden Fifomaten repräsentieren. Außerdem wird die Entdeckung eines Zyklus beschleunigt, wenn der Algorithmus kürzere Wordmaps höher priorisiert. Auf diese Weise werden Pfade in Richtung des Wurzelknotens bevorrechtet.

– AKTUALISIERUNG DER KNOTENMENGEN (ZEILEN 6–11):

1. MENGE DER EXPANDIERTEN KNOTEN (ZEILE 6): Im nächsten Schritt wird der ausgewählte Knoten v_{best} in die Menge der expandierten Knoten V_E eingefügt.
2. MENGE DER BLOCKIERTEN KNOTEN (ZEILEN 7–10): Es wird überprüft, ob
 - * die expandierten Knoten sich gegenseitig blockieren,
 - * die expandierten Knoten bereits vorhandene Knoten des Erreichbarkeitsgraphen blockieren,
 - * die vorhandenen Knoten die gerade durch die Expansion erzeugten Knoten blockieren.

Dabei werden jeweils die aktuell vorhandenen Mengen der Debloccker berücksichtigt. Die so ermittelten blockierten Knoten werden in der Menge V_{tmp} zwischengespeichert und anschließend zur Gesamtmenge der blockierten Knoten hinzugefügt. Die Menge der expandierten Knoten ist dabei disjunkt von der Menge der blockierten Knoten.

ZIEL DES BLOCKIERENS ist es, die Größe des zu untersuchenden Erreichbarkeitsgraphen weiter einzuschränken. Blockierte Knoten werden bei der weiteren Abarbeitung nicht expandiert, was die Konstruktion uninteressanter Teile des Erreichbarkeitsgraphen verhindert. Dabei blockiert ein Knoten v einen anderen Knoten v' , wenn diese dieselbe Statemap haben und die Wordmap von v ein Präfix der Wordmap von v' darstellt. Die Idee dahinter ist, daß es keinen Grund gibt, die Nachfolger von v' zu untersuchen, solange es keinen Nachfolger von v gibt, der aufgrund einer nichtausführbaren Lesetransition nicht konstruiert werden kann.

3. MENGE DER ZURÜCKGESTELLTEN KNOTEN (ZEILE 11): In die Menge der zurückgestellten Knoten V_P werden alle bei der Expansion von v_{best} entstandenen Knoten, die nicht blockiert oder bereits expandiert sind, aufgenommen. Für die Funktion `expand()` gibt es drei Möglichkeiten:
 - (a) Falls der aktuelle Knoten einen Fifomaten enthält, der sich in einem transienten Zustand befindet, dann wird nur dieser Fifomat bei der Expansion berücksichtigt. Transiente Zustände markieren atomare Operationen.
 - (b) Gibt es einen Fifomaten, dessen aktuelle atomare Operation ausführbar ist, dann wird nur dieser Fifomat bei der Expansion berücksichtigt. Dabei muß gelten, daß keine Transition in dieser atomaren Operation blockiert ist, weil bei Schreiboperationen eine Fifokapazität überschritten würde oder weil im Falle einer Leseoperation nur ein Präfix des zu lesenden Wortes in der betreffenden Fifo vorhanden ist. Der Grund für die alleinige Expansion dieses Fifomaten liegt darin, daß dieser sich KOMMUTATIV zu der Expansion aller

anderen Fifomaten verhält (vergleiche Definition 6.4.17). Es handelt sich damit um eine spezielle Form der PARTIAL ORDER REDUCTION (vergleiche Abschnitt 6.4.4).

(c) In allen anderen Fällen werden alle Fifomaten bei der Expansion berücksichtigt.

– AKTUALISIERUNG DER MENGE DER ZURÜCKGESTELLTEN DEBLOCKER (ZEILEN 12–13): Die überladene Funktion `determineDeblocker()` bestimmt alle Debblocker, die durch

* verbotene Transitionen (`determineDeblocker(v_{best})`)

* Rückwärtspropagation der Debblocker der Zielknoten aktuell hinzugekommener redundanter Kanten (`determineDeblocker(v_{best} , $V_E^{i+1} \cup V_B^{i+1} \cup V_P^{i+1}$, $D_E^i \cup D_P^i$)`)

entstehen. Diese so ermittelten Debblocker werden in die Gesamtmenge der zurückgestellten Debblocker D_P aufgenommen, falls diese Debblocker nicht bereits in der Menge der expandierten Debblocker enthalten sind.

• **Lazy Deblockieren (Zeilen 15–22):**

– AUSWAHL EINES DEBLOCKERS (ZEILE 16): Ist die Menge der zurückgestellten Knoten V_P leer, wird ein Debblocker (v, w_v, t) aus D_P ausgewählt. Dabei sind die zurückgestellten Debblocker D_P in einer Prioritätswarteschlange organisiert. Je näher der Knoten eines Debblockers am Wurzelknoten liegt, desto höher wird der zugehörige Debblocker priorisiert.

– AKTUALISIERUNG DER KNOTENMENGEN (ZEILEN 17–19):

1. MENGE DER BLOCKIERTEN KNOTEN (ZEILEN 17–18): Mit Hilfe der Funktion `determineDeblockedSet()` werden alle Knoten ermittelt, die durch den ausgewählten Debblocker deblockiert werden. Diese ermittelte Knotenmenge wird in V_{tmp} zwischengespeichert und aus der Menge der blockierten Knoten entfernt.

2. MENGE DER ZURÜCKGESTELLTEN KNOTEN (ZEILE 19): Die in V_{tmp} gespeicherten deblockierten Knoten werden in die Menge der zurückgestellten Knoten aufgenommen.

– AKTUALISIERUNG DER DEBLOCKERMENGEN (ZEILEN 20–21):

1. MENGE DER EXPANDIERTEN DEBLOCKER (ZEILE 20): Der ausgewählte Debblocker wird in die Menge der expandierten Debblocker aufgenommen.

2. MENGE DER ZURÜCKGESTELLTEN DEBLOCKER (ZEILE 21): Der ausgewählte Debblocker (v, w_v, t) wird zu allen direkten Vorgängern von v propagiert, wobei für jeden dieser Vorgängerknoten jeweils ein eigener Debblocker erzeugt wird. Dies geschieht mit Hilfe der Funktion `determineDeblocker()`. Die so erzeugten Debblocker werden in die Gesamtmenge der zurückgestellten Debblocker D_P aufgenommen.

Die genannten Schritte werden solange durchgeführt, bis entweder die Menge der zurückgestellten Knoten des Erreichbarkeitsgraphen und die Menge der zurückgestellten Deblocker leer sind oder bis ein Zyklus im Erreichbarkeitsgraphen, der den Wurzelknoten beinhaltet, gefunden wurde.

NICHTBEWIESENE BEHAUPTUNG 6.4.1: *Der Algorithmus `guidedSearch()` erhält als Eingabe eine gegebene Menge von Fifomaten und ist in dem folgenden Sinne korrekt: Falls es einen den Wurzelknoten beinhaltenden Zyklus des Erreichbarkeitsgraphen gibt, dann findet der Algorithmus diesen und terminiert. Gibt es keinen solchen Zyklus, wird dies gemeldet oder der Algorithmus terminiert nicht.*

6.4.6 Probleme und Lösungen

Problem der Terminierung: Im allgemeinen Fall kann nicht entschieden werden, ob die Simulation eines Fifomaten bei einer beliebigen Eingabe terminiert.

SATZ 6.4.3: *Das Fifomatenmodell basierend auf Definition 6.3.2 unter Verwendung von Fifos c mit unendlichen Kapazitäten n_c ist turingvollständig.*

BEWEIS 6.4.6:

Gegeben: Sei f ein Fifomat mit mindestens vier Fifos unendlicher Kapazität.

Behauptung: Dann besitzt f zwei Stacks.

Beweis:

1. Mit Hilfe zweier Fifos kann ein Stack simuliert werden.
 - (a) Das Schreiben in den Stack, die `push`-Operation, entspricht dem Schreiben in die Fifo.
 - (b) Das Lesen aus dem Stack, die `pop`-Operation, wird wie in Abbildung 6.13 dargestellt, simuliert.
2. Ein endlicher Automat mit einem Stack entspricht einer Kellermaschine [JLR97].
3. Eine Kellermaschine mit zwei Stacks ist turingvollständig [JLR97].

⇒ Satz 6.4.3. □

Aus der Turingvollständigkeit des Fifomatenmodells folgt:

SATZ 6.4.4: *Es ist nicht entscheidbar, ob die Simulation einer Menge miteinander kommunizierender Fifomaten terminiert.*

Schreiben (push)	Lesen (pop)		
<div style="border: 1px solid black; display: inline-block; padding: 2px;"> b a </div>	1) Einfügen einer Top-Markierung 2) Auslesen der Zeichen: - Entfernen des Zeichens aus der Hauptfifo - Falls das gelesene Zeichen der Top-Markierung entspricht, Verwerfen des Zeichens. In der Hilfsfifo liegt das oberste Element des Stacks. - Anderenfalls Einfügen des aus der Hauptfifo gelesenen Zeichens in die Hilfsfifo. Auslesen des in der Hilfsfifo liegenden Zeichens und Einfügen desselben in die Hauptfifo	Hauptfifo <div style="border: 1px solid black; display: inline-block; padding: 2px;"> c b a </div>	Hilfsfifo <div style="border: 1px solid black; display: inline-block; padding: 2px;"> </div>
<div style="border: 1px solid black; display: inline-block; padding: 2px;"> c b a </div>		<div style="border: 1px solid black; display: inline-block; padding: 2px;"> T c b a </div>	<div style="border: 1px solid black; display: inline-block; padding: 2px;"> </div>
		<div style="border: 1px solid black; display: inline-block; padding: 2px;"> T c b </div>	<div style="border: 1px solid black; display: inline-block; padding: 2px;"> a </div>
		<div style="border: 1px solid black; display: inline-block; padding: 2px;"> a T c </div>	<div style="border: 1px solid black; display: inline-block; padding: 2px;"> b </div>
		<div style="border: 1px solid black; display: inline-block; padding: 2px;"> b a T </div>	<div style="border: 1px solid black; display: inline-block; padding: 2px;"> c </div>
		<div style="border: 1px solid black; display: inline-block; padding: 2px;"> b a </div>	<div style="border: 1px solid black; display: inline-block; padding: 2px;"> c </div>

Abbildung 6.13: Simulation eines Stacks mittels zweier Fifos

BEWEIS 6.4.7:

Gegeben: Sei \mathbb{F} eine Menge zu simulierender Fifomaten.

Behauptung: Dann ist nicht entscheidbar, ob die Simulation von \mathbb{F} terminiert

Beweis: Da das Fifomatenmodell turingvollständig ist, ist es aufgrund der Unentscheidbarkeit des Halteproblems nicht möglich, daß eine korrekte Simulation immer terminiert.

⇒ Satz 6.4.4. □

Eine Lösung dieses Problems besteht darin, die Kapazitäten der Fifos zu beschränken. Dann ist die Turingvollständigkeit nicht mehr gegeben und die Terminierung kann garantiert werden. Dies ergibt sich daraus, daß der Erreichbarkeitsgraph dann endlich ist und somit in endlicher Zeit abgezählt werden kann.

Eine andere Lösung besteht darin, den Algorithmus 6.4.2 so zu modifizieren, daß eine Heuristik entsteht, welche zwar immer terminiert aber nicht immer einen Zyklus findet, obwohl ein solcher existiert.

Dies geschieht, indem beim Lazy Deblockieren nicht mehr alle Vorgänger berücksichtigt werden, sondern nur derjenige mit der kleinsten Knotennummer. Dabei unterscheidet sich `heuristicSearch()` von `guidedSearch()` an genau zwei Stellen:

1. Entstehen redundante Kanten bei der Expansion zurückgestellter Knoten, so werden Deblocker nicht über diese Kanten rückwärts propagiert. Die dafür verantwortliche Funktion `determineDeblocker()` wurde entfernt (Zeile 12).
2. In dem für das Lazy Deblockieren zuständigen Teil werden Deblocker nur noch zu dem Vorgängerknoten mit der kleinsten Entstehungsnummer propagiert. Dies geschieht mit einer modifizierten Funktion `determineDeblocker'()` (Zeile 20).

ALGORITHMUS 6.4.3:

```

1  Algorithm heuristicSearch() {
2       $V_E^0 := \emptyset; V_B^0 := \emptyset; V_P^0 := \{v_{\text{root}}\}; D_E^0 := \emptyset; D_P^0 := \emptyset; i := 0$ 
3      while  $((V_P^i \neq \emptyset \vee D_P^i \neq \emptyset) \wedge \neg \text{existsRootCycle}(V_E^i))$  {
4          if  $(V_P^i \neq \emptyset)$  {
5               $v_{\text{best}} := \text{select}(V_P^i)$ 
6               $V_E^{i+1} := V_E^i \cup \{v_{\text{best}}\}$ 
7               $V_{\text{tmp}} := \bigcup_{v \in \text{expand}(v_{\text{best}})} \text{determineBlockedSet}(v, \text{expand}(v_{\text{best}}), D_E^i \cup D_P^i)$ 
8                   $\cup \bigcup_{v \in \text{expand}(v_{\text{best}})} \text{determineBlockedSet}(v, V_E^i \cup V_B^i \cup V_P^i, D_E^i \cup D_P^i)$ 
9                   $\cup \bigcup_{v \in V_E^i \cup V_B^i \cup V_P^i} \text{determineBlockedSet}(v, \text{expand}(v_{\text{best}}), D_E^i \cup D_P^i)$ 
10              $V_B^{i+1} := (V_B^i \cup V_{\text{tmp}}) \setminus V_E^{i+1}$ 
11              $V_P^{i+1} := (V_P^i \cup \text{expand}(v_{\text{best}})) \setminus (V_E^{i+1} \cup V_B^{i+1})$ 
12              $D_P^{i+1} := (D_P^i \cup \text{determineDeblocker}(v_{\text{best}})) \setminus D_E^i$ 
13         }
14         elseif  $(D_P^i \neq \emptyset)$  {
15              $(v, w_v, t) := \text{select}(D_P^i)$ 
16              $V_{\text{tmp}} := \text{determineDeblockedSet}((v, w_v, t), V_B^i, V_E^i \cup V_B^i \cup V_P^i, D_E^i \cup D_P^i)$ 
17              $V_B^{i+1} := V_B^i \setminus V_{\text{tmp}}$ 
18              $V_P^{i+1} := V_P^i \cup V_{\text{tmp}}$ 
19              $D_E^{i+1} := D_E^i \cup \{(v, w_v, t)\}$ 
20              $D_P^{i+1} := (D_P^i \cup \text{determineDeblocker}'(v, D_E^{i+1} \cup D_P^{i+1})) \setminus D_E^{i+1}$ 
21         }
22          $i := i + 1$ 
23     }
24     return  $V_E^i \cup V_P^i \cup V_B^i$ 
25 }

```

SATZ 6.4.5: Der Algorithmus *heuristicSearch()* terminiert immer.

BEWEIS 6.4.8:

Gegeben: Sei \mathbb{F} eine endliche Menge von Fifomaten.

Behauptung: Dann terminiert die oben geschilderte Heuristik.

Beweis:

1. Die Anzahl der Statemaps ist endlich.
2. Die Anzahl der Deblocker ist endlich. Es gibt nämlich zwei Arten, wie Deblocker entstehen können:
 - (a) Zum einen entstehen Deblocker, wenn eine Transition leseblockiert ist. Da die Anzahl der Wordmaps, bei denen eine Transition leseblockiert ist, endlich ist, ist auch die Menge der so entstehenden Deblocker endlich.

(b) Zum anderen entstehen Deblocker bei der Rückwärtspropagation durch den bereits erzeugten Erreichbarkeitsgraphen. Dabei wurden folgende Einschränkungen in `heuristicSearch()` getroffen:

- Durch Entfernen der Funktion `determineDeblocker()` (siehe Zeile 12) werden Deblocker nicht mehr rückwärts über redundante Kanten, die bei der Expansion von zurückgestellten Knoten entstehen, propagiert.
- In Zeile 20 werden durch die eingeschränkte Funktion `determineDeblocker'()` Deblocker nur noch zu dem Vorgängerknoten mit der kleinsten Entstehungsnummer propagiert.

Aufgrund dieser Einschränkungen werden Deblocker auf dem kürzesten Weg zur Wurzel propagiert. Damit ist die Anzahl der `unapply()`-Schritte endlich und somit auch die Anzahl der so entstehenden Deblocker.

3. Angenommen, es existiert ein unendlich langer Pfad im vom Algorithmus erzeugten Erreichbarkeitsgraphen.
4. Dann wiederholt sich eine Statemap auf dem Pfad unendlich oft.
5. Zu dieser Statemap gibt es unendlich viele Wordmaps. Darunter findet man eine monoton steigende Folge.
6. Das heißt, jedes Element der Folge blockiert alle darauffolgenden Elemente.
7. Durch einen Deblocker kann man nur endlich viele blockierte Elemente deblockieren.
8. Da die Anzahl der Deblocker endlich ist, geschieht dies nur endlich oft. Irgendein Element der Folge blockiert somit den Pfad.
9. Dies ist ein Widerspruch zur Annahme. Daraus folgt, es gibt keinen unendlich langen Pfad. Somit ist der Erreichbarkeitsgraph endlich.

⇒ Satz 6.4.5.

□

Zustandsraumexplosion (State Space Explosion): Das zentrale Problem beim Model Checking ist die Explosion der Anzahl der Knoten des Erreichbarkeitsgraphen (vergleiche Abschnitt 2.4.2.2). Aufgrund des hohen Parallelitätsgrades, der in einem Datenflußgraphen enthalten ist, führt die hohe Zahl der möglichen Abarbeitungsreihenfolgen zu einer sehr großen Anzahl von Knoten des Erreichbarkeitsgraphen. Um nun dieses Problem in den Griff zu bekommen, wurden sowohl Maßnahmen vorgesehen, welche die Anzahl der Knoten begrenzen, als auch Maßnahmen, welche den Speicherbedarf pro Knoten eingrenzen. In die erste Gruppe von Maßnahmen fallen eine spezielle Methode zur PARTIAL ORDER REDUCTION und der BLOCKIER/DEBLOCKIER-MECHANISMUS. Die zweite Gruppe von Maßnahmen zur Milderung

der Auswirkungen der Zustandsraumexplosion ist in Abschnitt 7.3.2.2 beschrieben und beinhaltet REDUNDANZFREIE ABSPEICHERUNG DER KNOTEN, indem alle Knotenbestandteile wie beispielsweise Statemaps und Wordmaps in Depositories abgelegt werden, welche die Eindeutigkeit garantieren.

SATZ 6.4.6: Die Funktion `expand()`, die mit Hilfe von `isCommutativeAndNecessary()` nur eine Teilmenge der möglichen Nachfolgerknoten eines Knotens des Erreichbarkeitsgraphen erzeugt, bewirkt eine PARTIAL ORDER REDUCTION im Sinne von Definition 2.4.4.

BEWEIS 6.4.9:

Gegeben: Sei f ein Fifomat und \mathbb{F} die Menge der zu simulierenden Fifomaten. Die Funktionen `expand()` und `isCommutativeAndNecessary()` seien wie oben definiert gegeben.

Behauptung: Die mit Hilfe von `isCommutativeAndNecessary()` erzeugten Teilmengen bei der Expansion von Knoten mittels `expand()` stellen eine PARTIAL ORDER REDUCTION dar.

Beweis:

1. Die Funktion `expand()` unterscheidet drei Fälle bei der Expansion eines Knotens v :
 - (a) Ein Fifomat befindet sich in einem transienten Zustand. Dann wird nur dieser bei der Expansion berücksichtigt.
 - (b) Die Funktion `isCommutativeAndNecessary()` liefert für einen Fifomaten f den Wert `true`. Dann wird nur dieser Fifomat bei der Expansion berücksichtigt.
 - (c) Ansonsten werden alle Fifomaten bei der Expansion berücksichtigt.
2. Die Funktion `isCommutativeAndNecessary()` überprüft mit Hilfe der Funktionen `isCommutative()` und `isNecessary()`, ob ein Fifomat f folgende Eigenschaften besitzt.
 - f kann alle seine aktuellen atomaren Operationen ausführen, ohne daß eine darin enthaltene Aktion wartend ist (`isWaiting()`). Da nur f in die betroffenen Fifos schreiben beziehungsweise aus diesen lesen kann, ist die Ausführung von f kommutativ mit der Ausführung aller anderen Fifomaten, die im Knoten v eine atomare Operation durchführen können.
 - Liefert die Funktion `isNecessary()` den Wert `true` für f , so besitzt v in Kombination mit f folgende Eigenschaften:
 - Die Statemap von $v = (s, w)$ unterscheidet sich von der Zielstatemap von $v_{\text{root}} = (s_{\text{root}}, w_{\text{root}})$ in dem Zustand des Fifomaten f .
 - Die Wordmap w unterscheidet sich von w_{root} und f muß diese Wordmap durch Lesen verändern, wenn die aktuelle Wordmap kein Präfix der Wurzelwordmap ist. Die Statemap muß durch Schreiben verändert werden, wenn die Wurzelwordmap kein Suffix der aktuellen Wordmap ist.

In diesen Fällen ist eine Ausführung von f notwendig, um den Zielknoten v_{root} zu erreichen.

Modus	Komposition	Simulation	
Anwendung	Colored-SDF-(Teil-)Graphen	Colored-SDF-Graphen	Colored-BDF-, -DDF-Graphen
Beschreibung	Bestimmt kanonischen Fifomaten	Sucht Zyklus im Erreichbarkeitsgraphen	
Analytische Fähigkeiten	Zyklische Schedules, Speicherverbrauch, Deadlock	Zyklische Schedules, Speicherverbrauch, Deadlock	Analyse eines einzigen Zyklus (Gegenbeispiele)
Vorteile	Inkrementell (wiederverwendbare Ergebnisse), kompositionelle operationelle Semantik	Schnell	

Tabelle 6.1: Komposition vs. Simulation

Somit gilt, daß `expand()`, wenn dies möglich ist, jeweils nur einen Fifomaten ausführt, der KOMMUTATIV zu allen anderen ausführbaren Fifomaten ist und dessen Ausführung notwendig zum Erreichen des Zielknotens ist. Auf diese Weise wird aus einer Menge paralleler Pfade im Erreichbarkeitsgraphen, die vom Wurzelknoten v_{root} ausgehen und in diesem enden, nur ein Pfad durchsucht. Damit ist die Definition der Partial Order Reduction erfüllt.

⇒ Satz 6.4.6. □

Berechnungszeit: Mit der Zustandsraumexplosion einher geht ein erhöhter Bedarf an Rechenzeit für die Verwaltung dieser Datenmenge. Als Lösung ist hier eine INKREMENTELLE ABSPEICHERUNG DER DATEN in Form von speziellen geordneten Entscheidungsbäumen vorgesehen. Details dazu finden sich in Abschnitt 7.3.2.2.

6.4.7 Komposition vs. Simulation

Tabelle 6.1 stellt die Vorteile der zwei Modi des Model-Checking-Verfahrens einander gegenüber. Den kanonischen Fifomaten eines Datenflußgraphen durch wiederholte Anwendung der KOMPOSITION (vergleiche Abschnitt 6.4.3) zu bestimmen, ist allgemeiner, da keine Informationen verloren gehen. Aus diesem Grund kann der kanonische Fifomat an Stelle der Originalfifomaten verwendet werden. Dies ist im SIMULATIONSMODUS (siehe Abschnitt 6.4.4) nicht möglich, da die Bestimmung eines einzigen Zyklus zu spezifisch ist. Dadurch würde bei Verwendung des durch den Algorithmus ermittelten Fifomaten anstelle der originalen Fifomaten die Entdeckung weiterer Zyklen verhindert. Aus diesem Grund muß die Simulation immer auf den ursprünglichen oder auf mittels Komposition erzeugten Fifomaten ausgeführt werden.

6.5 Innovative Aspekte

Die innovativen Aspekte des NEUEN MODELLS FÜR KOMMUNIKATIONSPROTOKOLLE (FIFOMATENMODELL) und des MODEL-CHECKING-VERFAHRENS lassen sich wie folgt zusammenfassen:

- **Skalierbare Modellierung des Kommunikationsverhaltens:** Fifomaten stellen eine Erweiterung von Communicating Finite State Machines (siehe Abschnitt 2.4.2.2) dar. Damit wurde eine Modellierungsmethodik entwickelt, die sich hervorragend für die Beschreibung des Kommunikationsverhaltens von Datenflußkomponenten eignet. In derselben Modellierungssprache können sowohl einfache Colored-SDF-Komponenten als auch komplizierte Colored-BDF-Komponenten wie Colored Switch (CSwitch) oder Colored Select (CSelect), die den Datenfluß steuern, modelliert werden. Auch die Beschreibung NICHTDETERMINISTISCHER DATENFLUSSKOMPONENTEN wie zum Beispiel Colored Merge (CMerge) ist problemlos möglich. Damit ist die Basis geschaffen für eine DURCHGÄNGIGE ANALYSEMETHODIK für alle betrachteten gefärbten Datenflußparadigmen. Dies wird hier auch als SKALIERBARKEIT bezeichnet. Die vorgestellte Modellierungsmethodik stellt damit eine wesentliche Verbesserung zu vorhandenen Analysemethoden dar (vergleiche Abschnitt 2.2.10).
- **Dediziertes Model-Checking-Verfahren:** Das Model-Checking-Verfahren ist auf dieses Fifomatenmodell und die zugehörigen Datenflußparadigmen zugeschnitten. Es kann sowohl im SIMULATIONSmodus als auch im KOMPOSITIONSMODUS betrieben werden. Die Komposition von Fifomaten kann nur für nichtturingäquivalente Datenflußgraphen garantiert werden. Im Falle von Colored BDF oder Colored DDF kann also keine allgemeingültige Erfolgsaussage getroffen werden. Der Hauptvorteil des Kompositionsmodus ist die Wiederverwendung der Zwischenergebnisse, wohingegen der Simulationsmodus sich durch seine Schnelligkeit auszeichnet.
- **Effiziente Guided-Search-Strategie:** Kern des Model-Checking-Verfahrens ist eine neuartige Suchstrategie `guidedSearch()`. Diese verwendet Methoden zum BLOCKIEREN und LAZY DEBLOCKIEREN von Knoten des Erreichbarkeitsgraphen und ein Verfahren zur PARTIAL ORDER REDUCTION, um schnellstmöglich Zyklen im Erreichbarkeitsgraphen zu finden. Wie die Ergebnisse in Abschnitt 8.4 zeigen, können auf diese Weise SEHR SCHNELL auch sehr große Datenflußgraphen untersucht werden.
- **Entwurfsbegleitende Analyse:** Die Analyse kann ENTWURFSBEGLEITEND durchgeführt werden, wobei bei Datenflußgraphen, die dem Datenflußparadigma Colored SDF angehören, ZYKLISCHE SCHEDULES, DEADLOCKS und SPEICHERVERBRAUCH entscheidbar beziehungsweise berechenbar sind. Bei Colored BDF und Colored DDF können zumindest einzelne Zyklen untersucht werden. Außerdem ist das Verfahren INKREMENTELL, wenn es im Kompositionsmodus eingesetzt wird.

Damit handelt es sich bei dem hier vorgestellten Model-Checking-Verfahren um eine SEHR EFFIZIENTE ANALYSEMETHODIK für Datenflußgraphen. Diese Arbeit stellt deshalb einen WESENT-

LICHEN BEITRAG hinsichtlich entwurfsbegleitender Qualitätssicherung bei der Erstellung von Datenflußgraphen dar. Damit wurde die in Abschnitt 1.4.1 vorgegebene Zielsetzung voll und ganz erreicht.

Kapitel 7

Bild- und Signalverarbeitungswerkzeug Skylla

Incidit in Scyllam, qui vult vitare Charybdim.

Dieses Kapitel beschreibt das Bild- und Signalverarbeitungswerkzeug SKYLLA. Nach einer Aufstellung der Anforderungen an die Implementierung findet sich ein Überblick über die einzelnen Bestandteile von **Skylla**. Dabei werden einige interessante Punkte besonders beleuchtet. Abschließend sind die innovativen Aspekte zusammenfassend dargestellt. Eine detaillierte Beschreibung findet sich in [Le01, Lau02, Sto03, Abs04].

7.1 Anforderungen

Die Anforderungen an ein Werkzeug zur Erstellung und Ausführung von Datenflußgraphen für die Bild- und Signalverarbeitung lassen sich anhand der im folgenden vorgestellten Benutzergruppen ([MRSS01]; siehe Abbildung 7.1) unterteilen:

- **ANWENDER:** Der Anwender interagiert mit dem Werkzeug **Skylla** mit Hilfe von Ein- und Ausgabefenstern. Falls keine graphische Oberfläche zur Verfügung steht beziehungsweise keine Interaktion notwendig ist, kann das Werkzeug ohne Oberfläche gestartet werden. Das eigentliche Programm in Form eines Datenflußgraphen wird vor dem Anwender verborgen. Für den Anwender stehen unter anderem folgende Gesichtspunkte im Vordergrund [Bal96]:
 - Ausführungsgeschwindigkeit
 - Zuverlässigkeit
 - Visualisierung der Ergebnisse

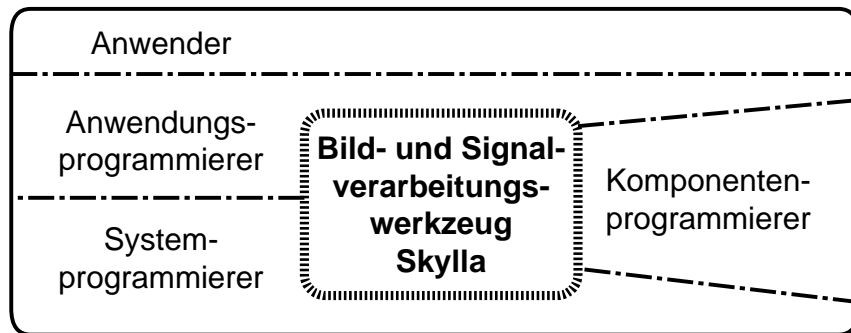


Abbildung 7.1: Benutzergruppen des Bild- und Signalverarbeitungswerkzeuges Skylla

- **ANWENDUNGSPROGRAMMIERER:** Der Anwendungsprogrammierer erstellt Datenflußgraphen mit Hilfe eines graphischen Editors. Dieses Vorgehen bezeichnet man als **VISUELLE PROGRAMMIERUNG** [Sch98a]. Dabei besteht ein Graph aus Datenflußkomponenten und Kanten. Für den Anwendungsprogrammierer sind folgende Punkte besonders wichtig:
 - Einblick in die Berechnungskomplexität der Tokenmaschinen von Datenflußkomponenten (vergleiche Abschnitt 4.2.3).
 - Unterstützung bei der Fehlererkennung und -vermeidung.
- **KOMPONENTENPROGRAMMIERER:** Der Komponentenprogrammierer erstellt mit Hilfe einer textuellen Programmiersprache wie zum Beispiel **C++** Datenflußkomponenten und mittels einer Schnittstellenbeschreibungssprache dazu passende Schnittstellenbeschreibungen für die automatische Typprüfung beziehungsweise zur Bestimmung der Komponentenkompatibilität. Die Anforderungen des Komponentenprogrammierers lauten:
 - Einfache Implementierung neuer Datenflußkomponenten und einfache Erstellung dazu passender Schnittstellenbeschreibungen.
 - Unkomplizierte Integration neuer Datenflußkomponenten in das Bild- und Signalverarbeitungswerkzeug **Skylla**.
- **SYSTEMPROGRAMMIERER:** Die Aufgaben des Systemprogrammierers sind die Konzeption und Realisierung beziehungsweise Wartung des Werkzeuges **Skylla**. Die Hauptanforderung dieser Benutzergruppe lautet:
 - Einfache Wartbarkeit

Wie diese Anforderungen durch das Bild- und Signalverarbeitungswerkzeug **Skylla** erfüllt werden, wird im folgenden skizziert.

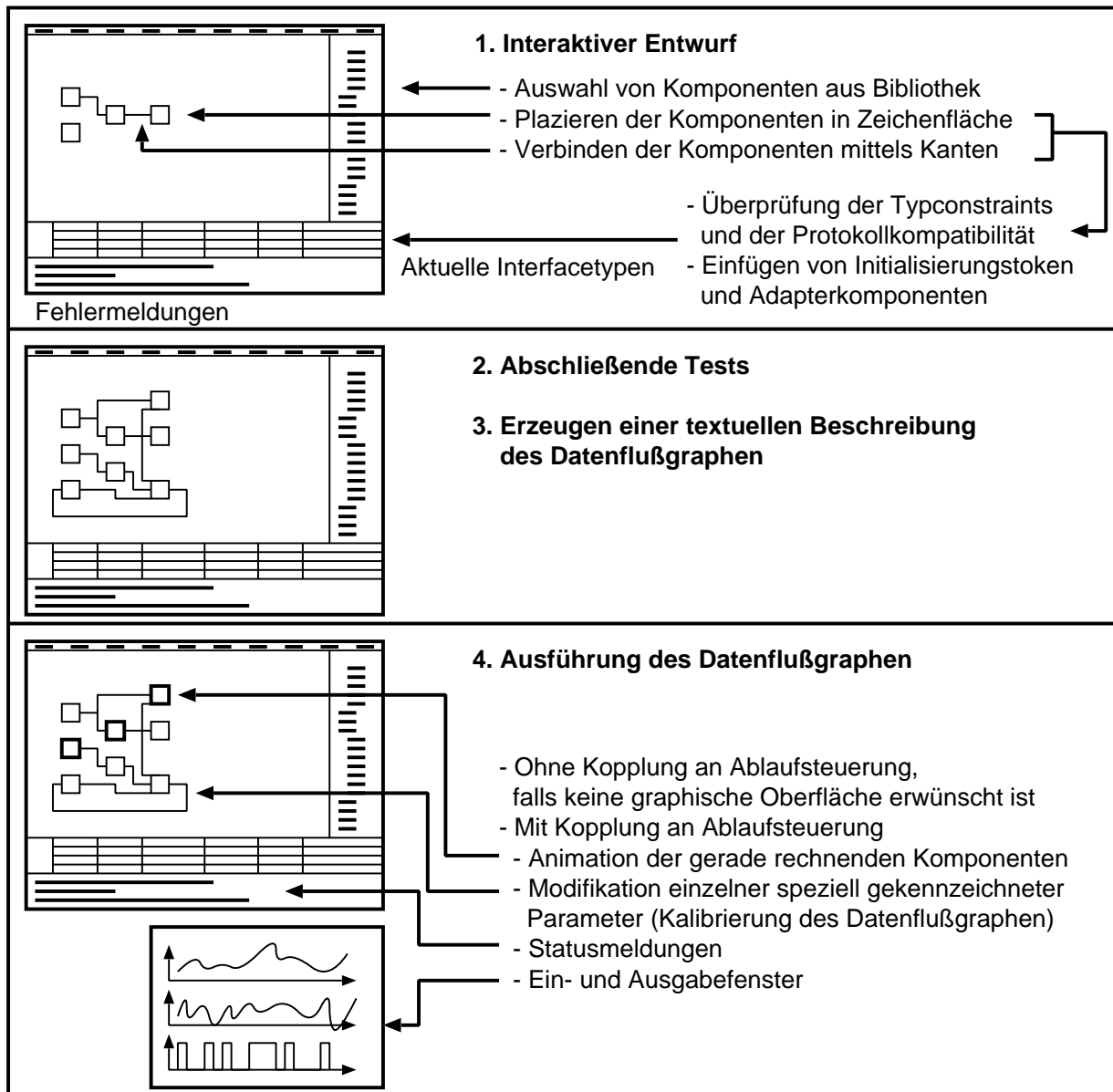


Abbildung 7.2: Sicht des Anwendungsprogrammierers auf Skylla

7.2 Beispiel eines Entwicklungsablaufs

In Abbildung 7.2 ist der Entwicklungsablauf eines Datenflußgraphen aus Sicht des Anwendungsprogrammierers dargestellt. Dabei unterscheidet man zwischen INTERAKTIVEM ENTWURF, ABSCHLIESSENDEN TESTS und AUSFÜHRUNG DES DATENFLUSSGRAPHEN.

1. INTERAKTIVER ENTWURF: Während des interaktiven Entwurfs des Datenflußgraphen wählt der Anwendungsprogrammierer Komponenten-Icons, die beispielsweise Kompo-

nenten zur Fast-Fourier-Transformation, zur Glättung von Bildern oder Filterkomponenten repräsentieren, aus einer Bibliothek aus und platziert diese in einer Zeichenfläche. Er verbindet die Schnittstellen der Datenflußkomponenten mittels Kanten. Eine Anpassung der Datenflußkomponenten an den jeweiligen Verwendungszweck erfolgt durch Parameter-einstellungen. Entwurfsbegleitend werden dabei TYPCONSTRAINTS ZWISCHEN INTER-FACETYPEN UND PARAMETERN überprüft. Die jeweiligen Interfacetypen der einzelnen Datenflußkomponenten sind in einer Tabelle dargestellt. Kommt es zu Typfehlern wird dies entsprechend ausgegeben. Durch einen Mausklick in die Tabelle wird im Editor die dazugehörige Datenflußkomponente gesucht und mittig platziert. Klickt man umgekehrt auf die Datenflußkomponente, so wird der entsprechende Tabelleneintrag zentriert dargestellt. Im Falle von Rückkopplungskanten innerhalb von SDF-Teilgraphen fügt das Programm Initialisierungstoken ein.

2. **ABSCHLIESSENDE TESTS:** Am Ende des Entwurfs des Datenflußgraphen fügt **Skylla** (semi-) automatisch Adapterkomponenten ein (vergleiche Abschnitt 4.3.2) und erzeugt eine textuelle Beschreibung des Datenflußgraphen, die als Eingabe für die Ablaufsteuerung dient.
3. **AUSFÜHRUNG DES DATENFLUSSGRAPHEN:** Der Datenflußgraph kann nun durch die Ablaufsteuerung ausgeführt werden. Dies ist je nachdem vom Editor aus steuerbar beziehungsweise erfolgt unabhängig von diesem. Im Falle der Kopplung zwischen Editor und Ablaufsteuerung erlaubt eine farbige Animation der gerade rechnenden Datenflußkomponenten dem Benutzer, den Programmablauf zu verfolgen. Zudem können Parameter, die vom Komponentenprogrammierer als zur Laufzeit veränderbar gekennzeichnet wurden, modifiziert werden. Dies ist insbesondere in der Kalibrierphase eines Datenflußgraphen interessant. Im unteren Teil des Editors erfolgt die Anzeige von Statusmeldungen. Die Ein- und Ausgaben des Datenflußgraphen erfolgen unabhängig vom Editor in eigenen Ein-/Ausgabefenstern. Dies ist notwendig, um dem Anwender eine vom Editor getrennte Schnittstelle zur Verfügung stellen zu können.

Aus der Sicht des Komponentenprogrammierers stellt sich die Entwicklung einer Datenflußkomponente wie folgt dar (vergleiche Abbildung 7.3):

1. **ERSTELLEN DER DATENFLUSSKOMPONENTE IN TEXTUELLER PROGRAMMIERSPRACHE:** Um der Ablaufsteuerung das dynamische Laden der Datenflußkomponente zu ermöglichen, wird die Datenflußkomponente von einer Basisklasse abgeleitet. Dabei stellt die Datenflußkomponente die Funktion `execute()` als Aufrufchnittstelle zur Verfügung. Für das Anfordern beziehungsweise die Weitergabe von Signalen ist eine im Rahmen der Ablaufsteuerung realisierte Speicherverwaltung verantwortlich.
2. **ERZEUGEN EINER SCHNITTSTELLENBESCHREIBUNGSDATEI:** Die Schnittstellen einer Datenflußkomponente zum Benutzer in Form eines Parameterdialoges und die Schnittstellen zu anderen Datenflußkomponenten und zum Scheduler sind mittels einer Schnittstellenbeschreibungdatei spezifiziert. In dieser können außerdem Regeln angegeben werden,

<p>1. Erstellen der Komponente in textueller Programmiersprache</p> <ul style="list-style-type: none"> - Ableiten von einer Basisklasse - Schnittstelle für dynamisches Laden: Funktion execute() - Anfordern und Weitergabe von Signaldaten mittels Speicherverwaltung
<p>2. Erzeugen einer Schnittstellenbeschreibungsdatei</p> <ul style="list-style-type: none"> - Schnittstelle zu Schedulingalgorithmen - Schnittstelle zum Benutzer: Parameterdialog - Schnittstelle zu anderen Datenflußkomponenten <ul style="list-style-type: none"> → - Typconstraints bezüglich der Signalmerkmale - Regeln für Änderbarkeit von Komponentenparametern in Abhängigkeit von der jeweiligen Entwurfs-/Ausführungsphase
<p>3. Generieren eines Icons</p>
<p>4. Integration in Skylla:</p> <ul style="list-style-type: none"> Plazieren <ul style="list-style-type: none"> - der Shared Library - der Schnittstellenbeschreibungsdatei - des Icons <p>in entsprechenden Verzeichnissen</p>

Abbildung 7.3: Sicht des Komponentenprogrammierers auf Skylla

in welcher Entwurfs- beziehungsweise Abarbeitungsphase es möglich ist, einen Parameter zu modifizieren. Weitere Regeln umfassen Typconstraints und Kommunikationsprotokolle.

3. GENERIEREN EINES ICONS: Damit die Datenflußkomponente im Editor ansprechend und leicht erkennbar dargestellt werden kann, ist ein geeignetes Icon als Repräsentant der Datenflußkomponente erforderlich.
4. INTEGRATION IN SKYLLA: Die Integration der Datenflußkomponente in Skylla erfolgt durch einfaches Plazieren der Icons und der Schnittstellenbeschreibungsdatei beziehungsweise der aus dem textuellen Programm erzeugten Shared Library in dafür vorgesehene Verzeichnisse.

Betrachtet man die Entwicklungsabläufe bei der Komponenten- und Datenflußgraphprogrammierung, so ist in Skylla die MEET-IN-THE-MIDDLE-Strategie realisiert (vergleiche Abbildung 7.4). BOTTOM-UP werden die einzelnen Datenflußkomponenten zur Verfügung gestellt. TOP-DOWN wird der Datenflußgraph aus diesen Datenflußkomponenten konstruiert.

7.3 Das Werkzeug Skylla

Skylla ist ein Werkzeug zur Bild- und Signalverarbeitung, welches im wesentlichen aus den folgenden Bestandteilen besteht (vergleiche Abbildung 7.5):

- dem EDITOR als visuellem Benutzerinterface.

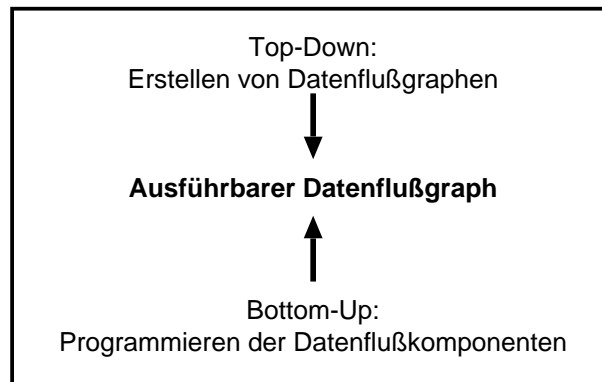


Abbildung 7.4: Meet-in-the-Middle-Strategie

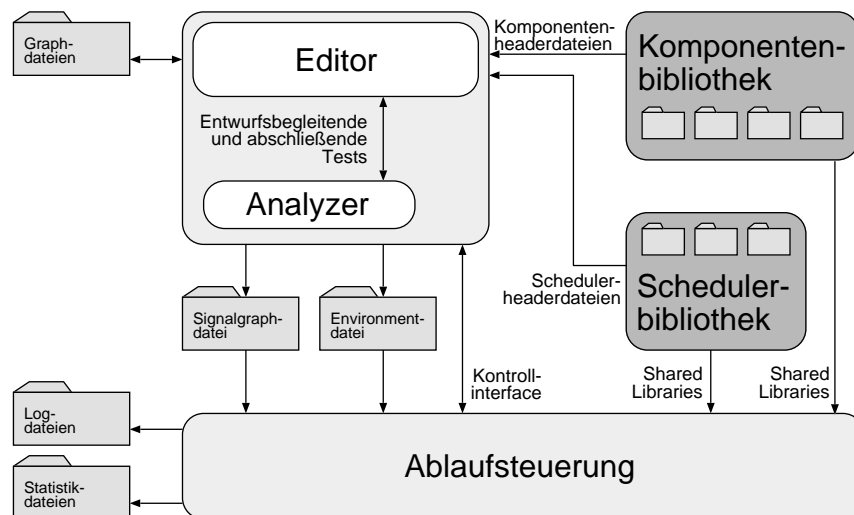


Abbildung 7.5: Überblick über das Gesamtsystem Skylla

- dem ANALYZER, welcher neben weiteren Plausibilitätsprüfungen die in Kapitel 5 vorgestellte Typprüfung und das in Kapitel 6 präsentierte Model-Checking-Verfahren beinhaltet.
- der KOMPONENTENBIBLIOTHEK, in der alle Datenflußkomponenten abgelegt sind.
- der SCHEDULERBIBLIOTHEK.
- der ABLAUFSTEUERUNG zur verteilten Ausführung der erstellten Datenflußgraphen.

Diese Bestandteile von Skylla werden im folgenden kurz beschrieben.

7.3.1 Editor

Der Editor (vergleiche Abbildungen 7.5 und 7.6) faßt die gesamte Funktionalität zur Erzeugung, Verwaltung und Ausführung von Datenflußgraphen in einer intuitiv und komfortabel bedienbaren graphischen Benutzeroberfläche zusammen. Die erstellten Datenflußgraphen sind in Form von GRAPHDATEIEN gespeichert, welche neben der Struktur eines Datenflußgraphen Informationen zu dessen visueller Darstellung enthalten. Ausführbare Datenflußgraphen werden als SIGNALGRAPHDATEIEN mit den für die Ausführung notwendigen Informationen an die Ablaufsteuerung übergeben. Über das KONTROLLINTERFACE erfolgt die Steuerung der Ablaufsteuerung sowie der Datenaustausch. Der Editor sendet beispielsweise PARAMETERÄNDERUNGEN von Datenflußkomponenten, die der Benutzer ZUR LAUFZEIT vornimmt, an die Ablaufsteuerung. In umgekehrter Richtung werden STATUSMELDUNGEN der einzelnen Datenflußkomponenten übermittelt.

Zwischen Status- und Menüleiste befindet sich der Arbeitsbereich, der sich in vier Teile gliedert (vergleiche Abbildung 7.6):

1. ZEICHENFLÄCHE: Die Zeichenfläche zeigt vom Benutzer eingefügte Datenflußkomponenten und Kanten an. Die Datenflußkomponenten werden durch Rechtecke, welche jeweils ein Icon beinhalten, dargestellt. Gegebenenfalls befinden sich am oberen und unteren Rand der Rechtecke Einkerbungen, welche die zugehörigen Datenflußkomponenten als Colored-BDF- (Dreieck) oder Colored-DDF-Komponente (Halbkreis) identifizieren.
2. KOMPONENTENBIBLIOTHEK: Diese wird beim Start des Editors eingelesen und dann mittels einer Baumstruktur, welche die Verzeichnisstruktur der Komponentenbibliothek wiedergibt, dargestellt.
3. ATTRIBUTFENSTER: Das Attributfenster listet für alle Komponenteninterfaces die Werte der jeweiligen Typtraits (vergleiche Kapitel 5) auf. Durch Anklicken eines Eintrages wird eine Datenflußkomponente im Datenflußgraphen selektiert und gegebenenfalls in den sichtbaren Bereich verschoben. Umgekehrt kann man eine Datenflußkomponente in der Zeichenfläche auswählen, wobei das Attributfenster die zugehörigen Schnittstellen und deren Typbelegungen anzeigt.
4. AUSGABEFENSTER: In diesem Fenster werden alle Meldungen des Editors, des Analyzers und der Ablaufsteuerung angezeigt.

7.3.2 Analyzer

Der Analyzer (vergleiche Abbildung 7.5) enthält die Analysemethodiken wie beispielsweise das Typsystem. Seine Aufgabe ist es, dem Benutzer entwurfsbegleitend bei der Erstellung der Datenflußgraphen zu helfen. Dazu stellt der Analyzer verschiedene Funktionen für die Typ- und Fehlerprüfung bereit. Außerdem enthält der Analyzer Routinen für die Erzeugung von Signalgraphdateien aus Datenflußgraphen.

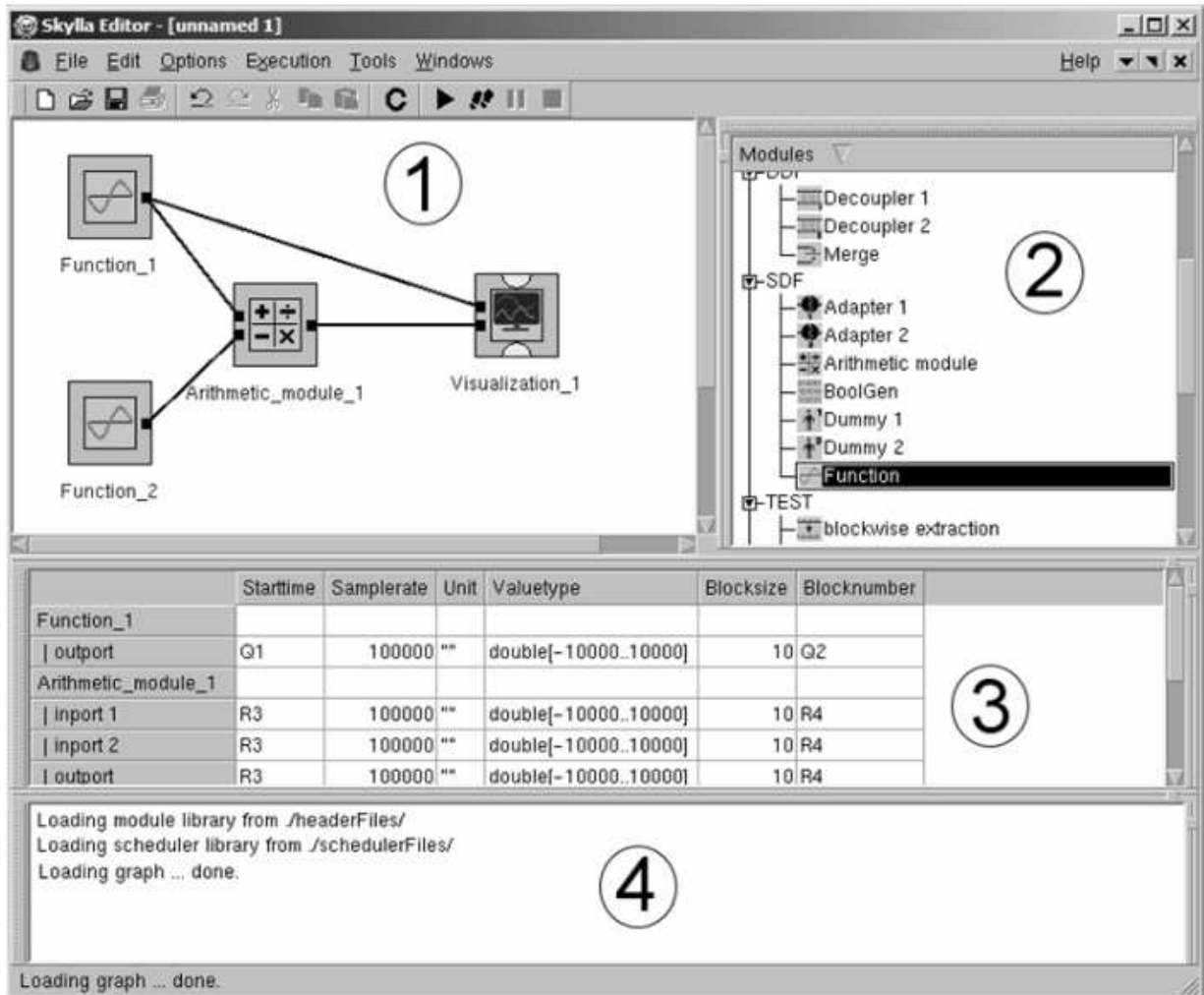


Abbildung 7.6: Editor von Skylla (Screenshot)

7.3.2.1 Interface-Typsystem

Die Implementierung des Interface-Typsystems ist detailliert in [Sto03] beschrieben. Die verwendeten Datenstrukturen sind der Standard Template Library (STL) von C++ entnommen [Str00].

7.3.2.2 Model Checker

Für die Effizienz des Model Checkers sind aufgrund des Problems der ZUSTANDSRAUMEXPLOSION (vergleiche Abschnitt 6.4.6) sowohl eine kompakte Ablegung der Daten als auch ein effizienter Zugriff erforderlich. Zur Erreichung dieses Zieles wurden verschiedene Maßnahmen ergriffen [May04b].

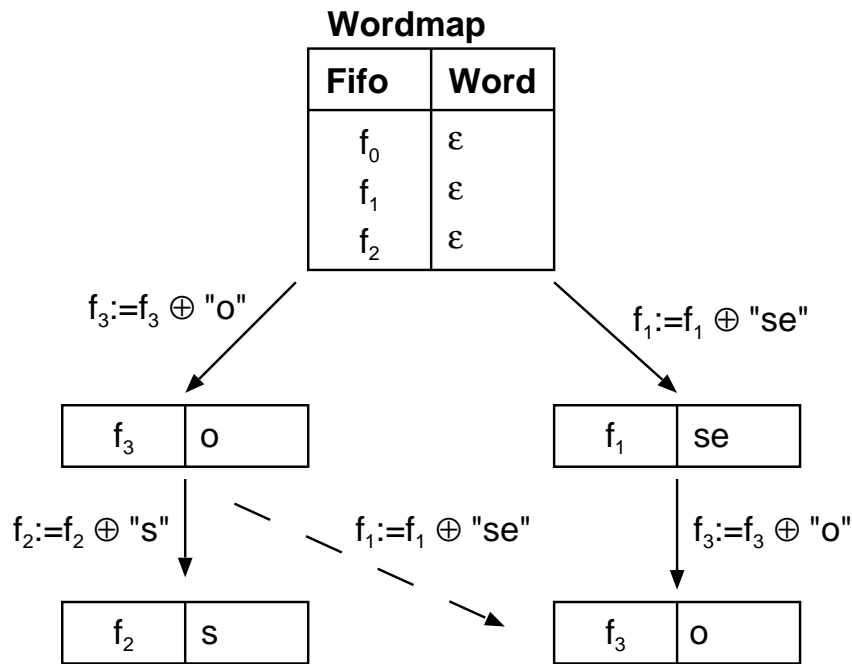


Abbildung 7.7: Incrementalmap

Redundanzfreie Datenhaltung: Jede Datenstruktur, die beispielsweise ein Token, ein Wort beziehungsweise eine Wordmap repräsentiert, wird **NUR EINMAL IN EINEM DEPOSITORY ABGESPEICHERT**. Wird eine derartige Datenstruktur erzeugt, wird überprüft, ob eine andere mit identischem Inhalt bereits existiert. Ist dies der Fall, wird die bereits existierende Datenstruktur verwendet. Dies ist möglich, da alle diese DATENSTRUKTUREN KONSTANT sind.

Abspeicherung von Änderungen: Änderungen, die beispielsweise entstehen, wenn eine Wordmap aus einer anderen durch Ausführen von Schreib- beziehungsweise Leseaktionen einer Transition erzeugt wird, müssen kompakt abgespeichert werden. Das zweite Ziel ist der schnelle Zugriff auf die abgespeicherten Daten. Um diese Ziele zu erreichen, wurden zwei alternative Implementierungen untersucht:

- **INCREMENTALMAPS:** Eine Incrementalmap ist eine Map [Str00], die nur Änderungen in Bezug zu ihrer Vorgängermap abspeichert (vergleiche Abbildung 7.7). Die vollständige Information erhält man, indem man von einer Incrementalmap rückwärts bis zur Wurzel sucht und dabei alle Dateneinträge aufsammelt. Demzufolge sind zwei Incrementalmaps identisch, wenn sie denselben Inhalt gemäß diesem Datenextraktionsmechanismus abgespeichert haben. Sind zwei Incrementalmaps identisch, so wird nur eine davon gespeichert (vergleiche oben vorgestellte Depositories). Um lange Suchpfade zu vermeiden, wird Caching verwendet. Caches werden Incrementalmaps in einem durch `cacheDistance` festgelegten Abstand zugewiesen. Der Parameter `cacheThreshold` legt fest, ab welcher Such-

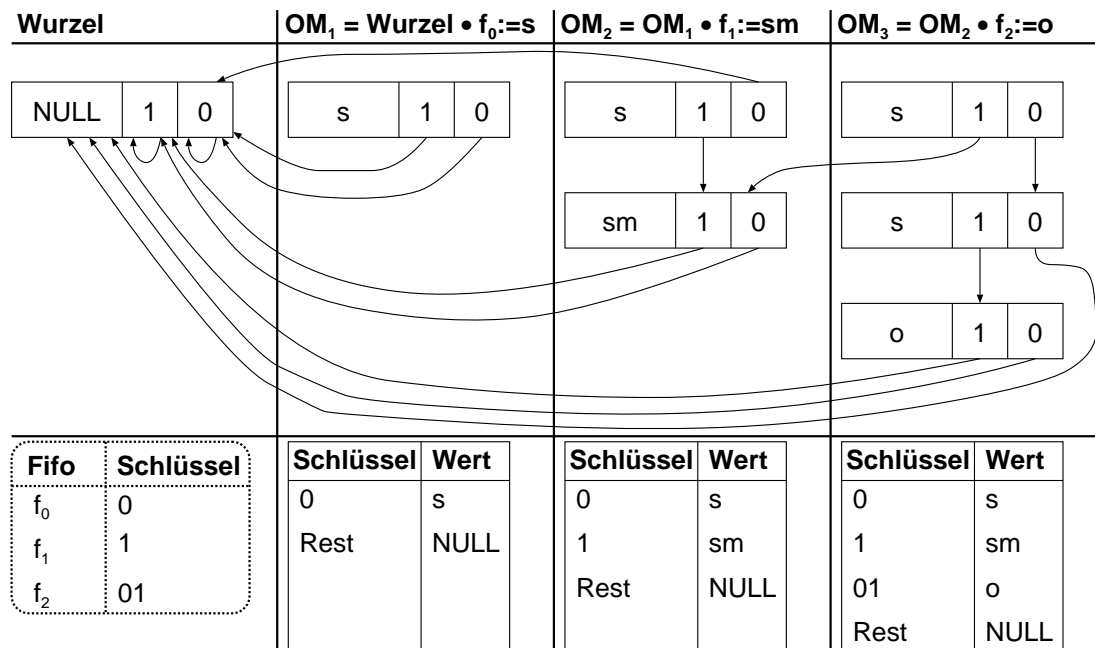


Abbildung 7.8: Ordered Map

tiefe ein Wert in den Cache eingetragen wird.

- ORDERED MAPS: Auf eine Ordered Map wird mit Hilfe eines binären Schlüssels zugegriffen. Dabei wird jeweils das niedrigste Bit betrachtet. Entsprechend seinem Wert wird entlang eines Zeigers die nächste Ordered Map erreicht. Bei jedem Übergang zu einer anderen Ordered Map wird der Schlüssel um ein Bit nach rechts verschoben. Ist der Schlüssel gleich 0, dann befindet sich in der entsprechenden Ordered Map der gesuchte Wert. Abbildung 7.8 veranschaulicht dieses Vorgehen. Dabei bedeutet

$$OM_2 = OM_1 \bullet f_1 := sm \quad (7.1)$$

daß die Ordered Map OM_2 aus der Ordered Map OM_1 erzeugt wird, indem der Fifo f_1 der Wert sm zugewiesen wird. Man kann natürlich auch mehr als zwei Zeiger in einer Ordered Map abspeichern. Die erlaubten Werte stellen Zweierpotenzen dar, da dann n Bits eines Schlüssels durch eine Ordered Map abgedeckt werden und somit 2^n Zeiger zu speichern sind.

7.3.3 Ablaufsteuerung

Die Ablaufsteuerung (vergleiche Abbildungen 7.5 und 7.9) ist für die gegebenenfalls VERTEILTE AUSFÜHRUNG der erstellten Datenflußgraphen zuständig. Um eine effiziente Abarbeitung von Datenflußgraphen gewährleisten zu können, besteht die Ablaufsteuerung aus einem Server und einen oder mehreren Clients.

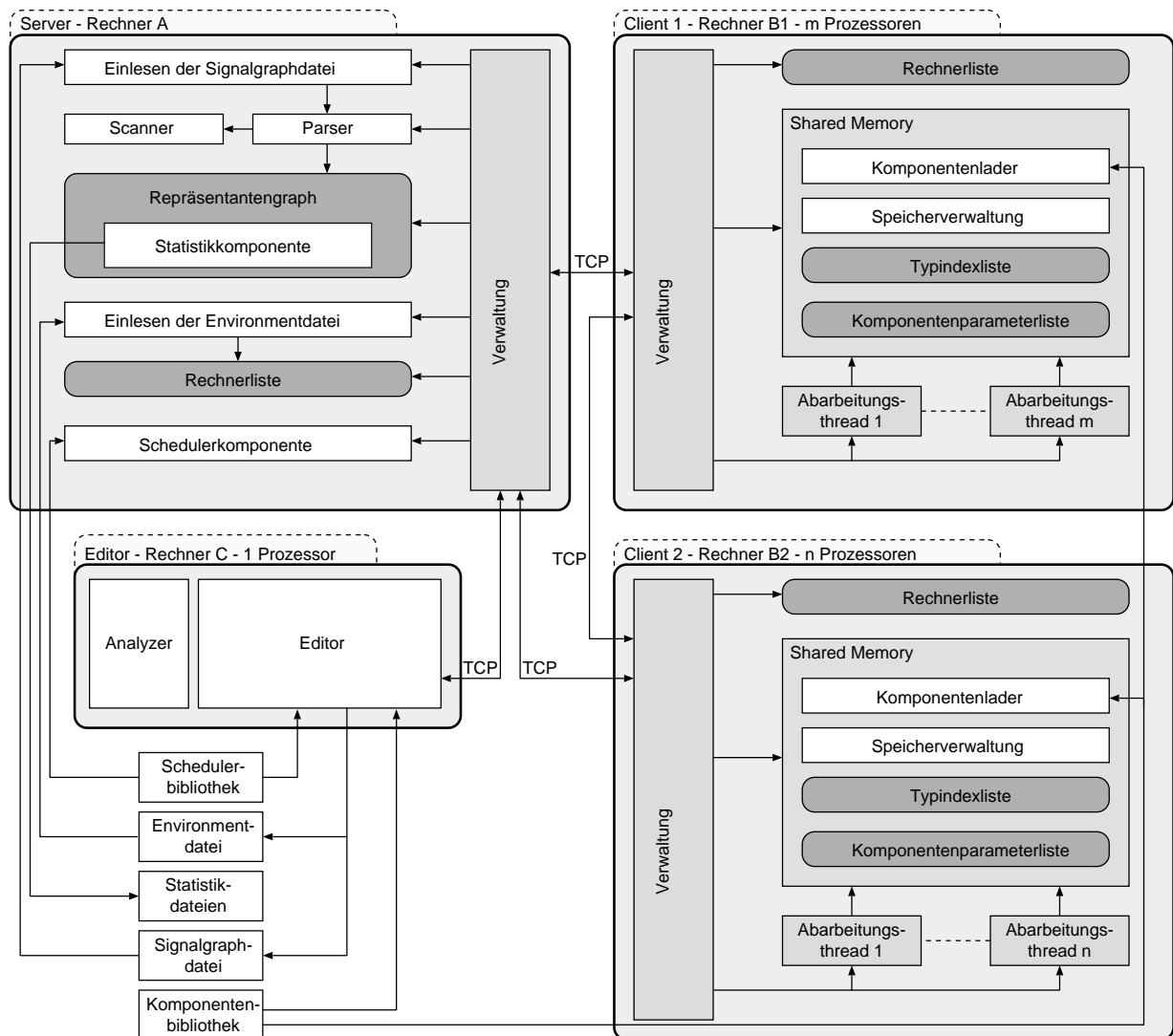


Abbildung 7.9: Überblick über die Ablaufsteuerung von Skylla

7.3.3.1 Server

Der Server ist die ZENTRALE VERWALTUNGSINSTANZ der Ablaufsteuerung, die durch den Editor oder manuell gestartet wird. Für die Ausführung eines Datenflußgraphen benötigt der Server folgende Daten:

- eine Signalgraphdatei, welche vom Analyzer durch Übersetzen eines Datenflußgraphen erzeugt wird.
- eine Environmentdatei, in der die zu verwendenden Rechner mit ihren charakteristischen Merkmalen wie beispielsweise die Anzahl der Prozessoren eingetragen sind.

Die Aufgaben des Servers sind:

- Gegebenenfalls Erstellen einer TCP/IP-Verbindung zum Editor.
- Einlesen der Signalgraphdatei.
- Einlesen der Environmentdatei und Starten eines Clients pro angegebenen Rechner.
- Herstellen einer TCP/IP-Verbindung zu den Clients.
- Laden des Schedulers aus der Schedulerbibliothek.
- Erteilen von Aufträgen zur Ausführung von Datenflußkomponenten an die Clients.
- Ermitteln der Rechenbereitschaft der Datenflußkomponenten. Dies wird, soweit möglich, vom Scheduler entschieden. Kann der Scheduler diese Entscheidung – zum Beispiel bei Colored-BDF-Komponenten – nicht treffen, wird die jeweilige Datenflußkomponente selbst nach ihrer Rechenbereitschaft befragt.
- Sammeln und Speichern von Statistikdaten.

Der Server hat also nur eine KONTROLL- UND VERWALTUNGSAUFGABE und führt selber keine Datenflußkomponenten aus. Der Server hält die Information, welche Datenflußkomponente auf welchem Client ausgeführt wird, welche Clients noch Aufträge annehmen können und wo Signaldaten liegen. Abbildung 7.9 zeigt eine schematische Darstellung der Ablaufsteuerung mit Server, zwei Clients und dem Editor. Pfeile stellen den Daten- und Informationsfluß zwischen den einzelnen Teilen dar.

7.3.3.2 Clients

Ein Client nimmt Aufträge vom Server entgegen. Zu diesem Zweck startet der Client in der Regel so viele Threads wie Prozessoren auf dem Rechner vorhanden sind. Ein Thread kann jeweils eine Datenflußkomponente abarbeiten. Die Anzahl der Threads pro Client ist in der Environmentdatei festgelegt und wird über den Server an die Clients übermittelt. Bevor die Abarbeitung einer Datenflußkomponente beginnt, werden – falls diese nicht vorliegen – alle für deren Abarbeitung nötigen Daten von auf anderen Rechnern ablaufenden Clients angefordert. Da der Client keine Informationen über den Datenflußgraphen besitzt, läuft diese Anforderung über den Server.

Die Aufgaben des Clients sind somit:

- Starten der vom Server übermittelten Anzahl von Abarbeitungsthreads.
- Entgegennehmen von Aufträgen zur Ausführung von Datenflußkomponenten vom Server.
- Gegebenenfalls Empfangen der Eingabedaten für die jeweilige Datenflußkomponente von einem anderen Client.
- Erteilen des Bearbeitungsauftrages an den Abarbeitungsthread, sobald alle benötigten Daten vorliegen. Dieser Abarbeitungsthread führt dann die Datenflußkomponente aus.

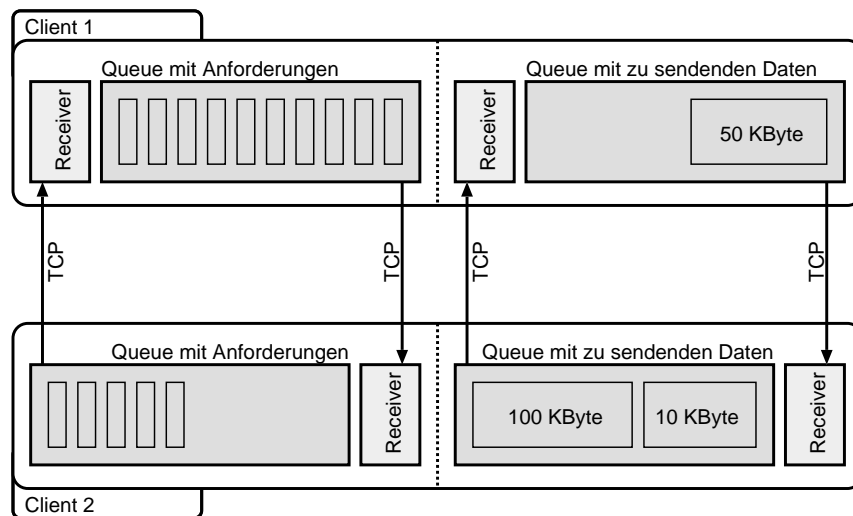


Abbildung 7.10: Kommunikationsverbindung zwischen zwei Clients

- Senden von Statusinformationen an den Server. Bei Ende der Abarbeitung der Datenflußkomponente wird dies dem Server zusammen mit der Anzahl der konsumierten beziehungsweise produzierten Anzahl von Signaltoken mitgeteilt.

7.3.3.3 Untersuchungen zur Kommunikation

Die Übertragung von Daten und Befehlen auf demselben Rechner geschieht mittels Shared Memory, wobei die Verwaltung dieser Speicherbereiche durch eine Speicherverwaltung erfolgt (siehe Abbildung 7.9).

Für die Kommunikation zwischen verschiedenen Rechnern stehen mehrere Alternativen zur Auswahl, deren Eignung in [Lau02, Abs04] untersucht wurde:

- Transmission Control Protocol (TCP) mit unterschiedlichen Parametereinstellungen,
- TCP für Transaktionen (T/TCP),
- User Datagram Protocol (UDP) ohne beziehungsweise mit einer eigenen Sicherungsschicht und
- Reliable User Datagram Protocol.

Den besten Datendurchsatz lieferte die Anordnung aus Abbildung 7.10. Es werden zwei TCP-Verbindungen zwischen zwei Clients aufgebaut. Dabei dient die eine Verbindung dazu, Daten anzufordern, während die zweite Verbindung für den Transport angeforderter Daten verwendet wird. Dies trägt dazu bei, Verzögerungszeiten zu vermeiden. Mit diesem Aufbau kann beispielsweise ein Client im gleichen Zeitraum nicht nur selber Daten von einem anderen Client anfordern, sondern auch Daten an diesen Client senden.

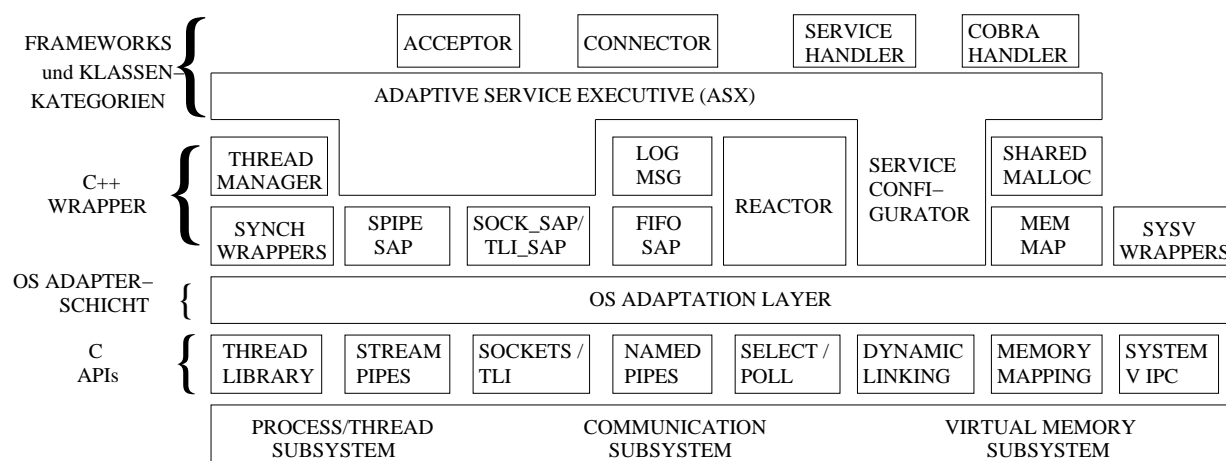


Abbildung 7.11: Adaptive Communication Environment (ACE)

7.3.3.4 Alternative Implementierung in ACE

Anstelle einer eigenen Implementierung können auch bereits vorhandene Realisierungen einer Middleware [Bal96] verwendet werden. Ein Beispiel dafür ist die Adaptive Communication Environment (ACE; siehe Abbildung 7.11) [Sch98b]. In [Le01] wurde eine auf ACE basierende Ablaufsteuerung entworfen und implementiert. Dabei wurden die von ACE zur Verfügung gestellte Funktionalität zur Erzeugung von Threads und zur Synchronisation des Zugriffs auf gemeinsame Speicherbereiche verwendet.

7.3.3.5 Alternative Implementierung in CORBA

Eine weitere Möglichkeit zur Verwendung bereits existierender Middleware-Implementierungen besteht in der Verwendung der Common Object Request Broker Architecture (CORBA) [Bal96]. In [Abs04] ist eine prototypische Implementierung der Ablaufsteuerung in CORBA beschrieben. Dabei wird der Datentransport zwischen den einzelnen Datenflußkomponenten mittels des von CORBA zur Verfügung gestellten Name- und Eventservice realisiert.

Schlußfolgerungen: Die Schlußfolgerungen aus den beiden alternativen Implementierungen lauten:

- Die Entwicklung einer geeigneten Ablaufsteuerung ist mit ACE und CORBA aufwendiger als eine direkte Umsetzung in C++. Der Mehraufwand ist an der zum Teil unzureichenden beziehungsweise unübersichtlichen Dokumentation festzumachen.
- Die entworfenen Ablaufsteuerungen auf der Basis von Middleware waren langsamer als die direkte Umsetzung. Dies liegt in der Optimierung der Datenübertragung für den speziellen Anwendungsfall begründet [Lau02].

- Ein weiterer Punkt, der für eine eigene Entwicklung der Ablaufsteuerung spricht, ist die Unabhängigkeit von komplexen Programmpaketen Dritter, was insbesondere bei dem Umstieg auf eine neue Betriebssystem- oder Compilerversion zu Problemen führen kann.

7.3.4 Komponentenbibliothek

Die Komponentenbibliothek beinhaltet alle Datenflußkomponenten. Bevor diese realisiert wurde, wurden Modelle der zugehörigen TOKENMASCHINEN (vergleiche Abschnitt 4.3.2) in Haskell (vergleiche Abschnitt 2.2.7) beziehungsweise mittels gefärbter Petrinetze (siehe auch Abschnitt 2.2.6) simuliert, um deren Praxistauglichkeit zu untersuchen.

7.3.4.1 Implementierung des Komponentenmodells in Haskell

Die Umsetzung der Tokenmaschinen in Haskell erlaubte die Erprobung der funktionalen Modelle der Datenflußkomponenten. Die dabei gewonnenen Erkenntnisse bildeten die Grundlagen der formalen Beschreibung in Kapitel 4, auf der wiederum die Implementierung der Datenflußkomponenten in C++ (siehe Abschnitt 7.3.4.3) beruht.

7.3.4.2 Implementierung des Komponentenmodells in Design/CPN

Die Implementierung der Tokenmaschinen in Design/CPN Version 3.1.2 für Sun Solaris 2.6 bot im Vergleich zu Haskell den Vorteil, daß neben den deterministischen Datenflußkomponenten auch das nichtdeterministische Colored Merge (CMerge) implementiert werden konnte. Allerdings wird die Darstellung in dem graphischen Editor sehr schnell SEHR UNÜBERSICHTLICH (vergleiche Abbildung 7.12). Auch die Analyse mittels Erreichbarkeitsgraphen scheiterte an fehlenden Optimierungsstrategien. Einzig mit Hilfe der schrittweisen Simulation konnten weitere Einsichten in das Verhalten der Tokenmaschinen gewonnen werden.

7.3.4.3 C++-Bibliothek

Die Komponentenbibliothek enthält alle Datenflußkomponenten, welche Bestandteile von Datenflußgraphen sind. Zum einen sind in der Bibliothek alle zur Ausführung von Datenflußkomponenten notwendigen Binärdaten, die ausschließlich von der Ablaufsteuerung benötigt werden, enthalten. Dabei liegt die Bibliothek in Form von Shared Libraries für verschiedene Betriebssysteme wie Linux und Solaris in getrennten Verzeichnissen vor. Die vom Server kontrollierten Clients der Ablaufsteuerung laden die Datenflußkomponenten aus diesen Shared Libraries und erzeugen daraus Komponenteninstanzen, welche aus den jeweiligen Eingabedaten Ausgabedaten produzieren. Zum anderen beinhaltet die Komponentenbibliothek die Schnittstellenbeschreibungen der Datenflußkomponenten in Form von HEADERDATEIEN. Diese werden ausschließlich vom Editor benötigt. Die abgelegte Information beinhaltet beispielsweise die Anzahl der Interfaces, die Interfacetypen und Interface-Typconstraints. Außerdem sind beschreibende Informationen aufgenommen, die etwa das Datenflußparadigma der Datenflußkomponente (Colored SDF, Colored BDF oder Colored DDF), die Darstellung der Datenflußkomponente im Editor mittels

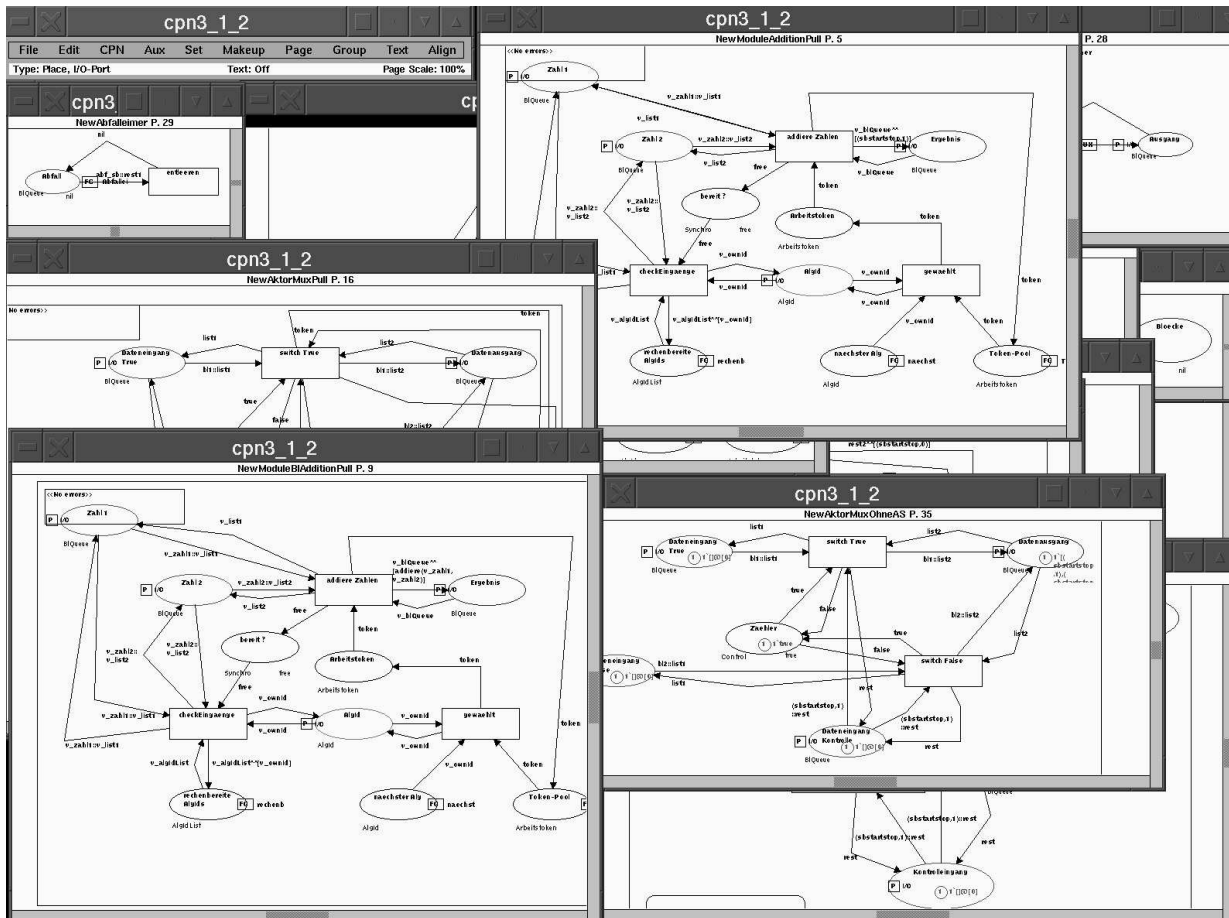


Abbildung 7.12: Komponentenmodelle in Design/CPN

eines Icons beziehungsweise den Aufbau des Parameterdialogs der Datenflußkomponente betreffen.

Die Schnittstellenbeschreibung für Datenflußkomponenten gliedert sich in drei Teile:

- **KOMPONENTENBESCHREIBUNG:** Dieser einfache Teil beinhaltet beschreibende Elemente wie Modulname und Zugehörigkeit zu dem jeweiligen Datenflußparadigma.
- **PARAMETERLISTE:** Diese Liste umfaßt die Komponentenparameter, die ein Benutzer über einen Dialog verändern kann. Die Parameter sind dabei typisiert und gegebenenfalls durch Regeln miteinander verknüpft.
 - Es ist möglich, eine Anpassung des Dialogfensters in Abhängigkeit von einem ausgewählten Parameter vorzunehmen (vergleiche Abbildung 7.13).
 - Zudem kann spezifiziert werden, ob ein Parameter nur zur Entwurfszeit oder auch zur Abarbeitungszeit modifizierbar ist.

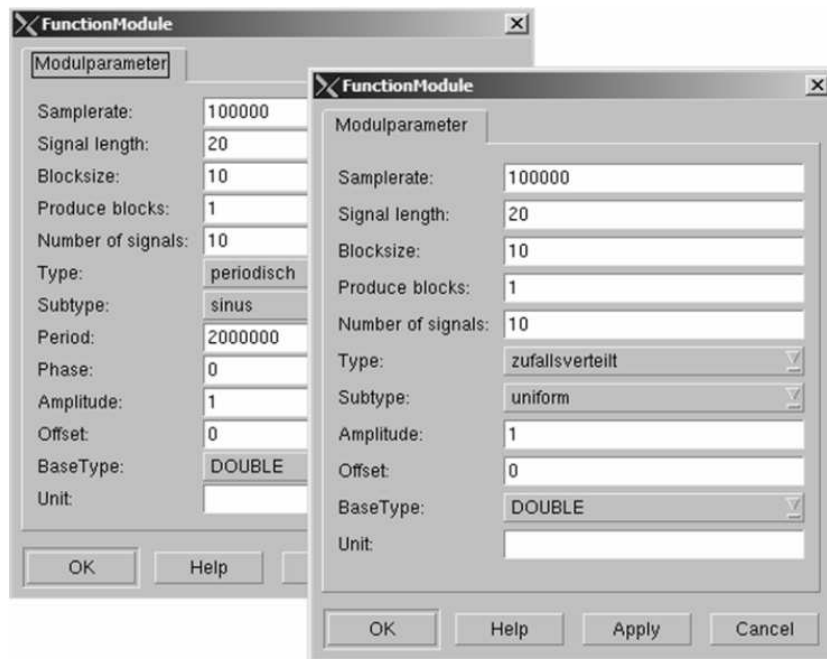


Abbildung 7.13: Kontextgesteuerte Dialogfenster einer Datenflußkomponente in Skylla

- Zwischen Parametertypen und Interfacetypen sind Typconstraints spezifizierbar, die durch das Interface-Typsystem überprüft werden.
- INTERFACELISTE: In dieser Liste sind die in Kapitel 5 beschriebenen Typen und die dazugehörigen Typconstraints enthalten. Außerdem sind Interfaceparameter definierbar, mit deren Hilfe beispielsweise der Benutzer die Anzahl bestimmter Schnittstellen im Dialogfenster einstellt.

Auf diese Weise lassen sich die Querbezüge zwischen Parametern und Interfacetypen einfach und flexibel beschreiben.

7.3.5 Schedulerbibliothek

Analog zur Komponentenbibliothek besteht die Schedulerbibliothek aus zwei Teilen. Zum einen liegen die Schedulingalgorithmen in Form von Shared Libraries vor, welche die Ablaufsteuerung exklusiv nutzt. Zum anderen ist die Schnittstellenbeschreibung der Schedulingalgorithmen in Schedulerheaderdateien abgelegt, welche ausschließlich der Editor verwendet. Es wurde eine ähnliche Beschreibungssprache wie bei den Datenflußkomponenten entwickelt, mit der beispielsweise die Parameterdialoge definiert werden, die der Benutzer im Editor verwendet.

7.4 Innovative Aspekte

Die innovativen Aspekte des für die Bild- und Signalverarbeitung konzipierten Werkzeuges *Skylla* lassen sich anhand der in Abbildung 7.1 vorgestellten Benutzergruppen veranschaulichen.

- **Anwender:** Für den Anwender steht der Einsatz von *Skylla* in einem eingebetteten System im Vordergrund. Daher sind für den Anwender folgende Punkte entscheidend:
 - Der Einsatz von *Skylla* mit beziehungsweise ohne graphischer Oberfläche erlaubt einen SEHR FLEXIBLEN EINSATZ in eingebetteten Systemen.
 - Für eine HOHE AUSFÜHRUNGSGESCHWINDIGKEIT sorgen die mögliche verteilte Ausführung von *Skylla* und die Verwendung einer für die Kommunikation optimierten Ablaufsteuerung.
 - Die ZUVERLÄSSIGKEIT DER ANWENDUNG wird durch die entwurfsbegleitenden Überprüfungen bezüglich Interfacetypen und Protokollkompatibilität entscheidend erhöht.
 - Die VISUALISIERUNG DER ERGEBNISSE geschieht mit Hilfe vom Editor unabhängigen Ausgabefenstern und ermöglicht somit auch einen vom Editor unabhängigen Einsatz.
- **Anwendungsprogrammierer:** Der Anwendungsprogrammierer erstellt Datenflußgraphen und legt auf folgende Merkmale erhöhten Wert:
 - Aufgrund der Klassifizierung der Datenflußkomponenten anhand ihrer Tokenmaschinen hat der Anwendungsprogrammierer bei deren Auswahl die ENTSCHEIDUNGSFREIHEIT ZWISCHEN ANALYSIERBARKEIT UND MODELLIERUNGSMÄCHTIGKEIT (vergleiche Abschnitt 4.2.3). Insbesondere die nicht turingäquivalenten Tokenmaschinen von Datenflußkomponenten, die Bild- und Signalverarbeitungsalgorithmen kapseln, ermöglichen die Programmierung großer analysierbarer Datenflußgraphen.
 - Die entwurfsbegleitende Überprüfung von Typconstraints und Protokollkompatibilität erlaubt die SCHNELLE UND FEHLERFREIE ERSTELLUNG VON DATENFLUSSGRAPHEN.
 - Die Möglichkeit, Initialisierungstoken in Rückkopplungen einzufügen, erlaubt beispielsweise die VERMEIDUNG VON NICHTDETERMINISTISCHEN ADAPTERKOMPONENTEN und bewahrt die Analysierbarkeit.
 - Die AUTOMATISCHE EINFÜGUNG VON ADAPTERKOMPONENTEN beschleunigt die Erstellung von großen Datenflußgraphen.
- **Komponentenprogrammierer:** Der Komponentenprogrammierer erstellt neue Datenflußkomponenten und legt Wert auf folgendes:
 - Die ERSTELLUNG VON DATENFLUSSKOMPONENTEN IST SEHR EINFACH. Die Implementierung geschieht durch Ableiten von einer Basisklasse. Für den dynamischen Lader muß als Schnittstelle die Funktion `execute()` implementiert werden.

Datenübertragung und Synchronisation des Datenzugriffs wird durch die Ablaufsteuerung übernommen. Außerdem muß der Komponentenprogrammierer noch eine Schnittstellenbeschreibung der Datenflußkomponente erstellen. Dabei sind die Schnittstellen zum Benutzer, zu anderen Datenflußkomponenten und zum Scheduler zu spezifizieren.

- Die EINFACHE INTEGRATION NEUER DATENFLUSSKOMPONENTEN erfolgt durch Kopieren der Shared Libraries und der Schnittstellenbeschreibungen in die entsprechenden Verzeichnisse. Der Editor kann dabei diese Beschreibungen dynamisch nachladen.
- **Systemprogrammierer:** Der Systemprogrammierer ist für die Konzeption, Realisierung und Wartung des Bild- und Signalverarbeitungswerkzeuges **Skylla** zuständig. Die bei den anderen Benutzergruppen aufgeführten innovativen Aspekte resultieren aus geeigneten Maßnahmen des Systemprogrammierers. Die leichte Wartbarkeit wird dabei unterstützt durch:
 - KLARE AUFTEILUNG DER AUFGABEN in einzelne Teilwerkzeuge wie Editor, Analyzer und Ablaufsteuerung.
 - EINDEUTIGE UND EINFACHE SCHNITTSTELLEN zwischen den einzelnen Teilwerkzeugen.

Damit stellt **Skylla** ein Werkzeug zur Bild- und Signalverarbeitung dar, welches neben den in Abschnitt 1.4.1 genannten Hauptzielen der entwurfsbegleitenden Überprüfung von Programmeigenschaften und der effizienten Kommunikation zwischen Datenflußkomponenten auf zusätzliche Anforderungen der in Abbildung 7.1 vorgestellten Benutzergruppen eingeht. **Skylla** eignet sich somit hervorragend für die komponentenbasierte Softwareentwicklung für datenflußorientierte eingebettete Systeme.

Kapitel 8

Ergebnisse

Gott würfelt nicht. [Albert Einstein]

In diesem Kapitel werden zuerst die Anforderungen an die Testläufe formuliert und die Testbedingungen vorgestellt. Daran schließt sich eine intensive Untersuchung des Interface-Typsysteams, des Model Checkers und der Ablaufsteuerung des Bild- und Signalverarbeitungswerkzeuges **Skylla** an. Am Ende des Kapitels findet sich eine Zusammenfassung der wesentlichen Ergebnisse. An dieser Stelle sei Herrn Klaus Schießl und Herrn Detlef Menzel für die Bereitstellung der für die Versuche notwendigen Rechenkapazität gedankt!

8.1 Anforderungen

Die Untersuchungen des Interface-Typsysteams, des Model Checkers und der Ablaufsteuerung verfolgen drei Ziele:

- **ÜBERPRÜFEN DER EFFIZIENZ:** Da das Interface-Typsysteamsystem und der Model Checker entwurfsbegleitend eingesetzt werden sollen, ist die Schnelligkeit, mit der einzelne Überprüfungen abgeschlossen werden, von entscheidender Bedeutung.
- **HERSTELLEN DER RELATION ZUM STAND DER TECHNIK:** Wo ein Vergleich mit existierenden Methoden möglich ist, wird ein solcher durchgeführt, um den Grad der Innovation der in dieser Arbeit vorgestellten Verfahren besser beurteilen zu können.
- **VERANSCHAULICHEN DES VERHALTENS AN FALLSTUDIEN:** Interessante Problemfälle werden exemplarisch herausgegriffen, um das Verhalten der einzelnen Verfahren zu verdeutlichen.

	Rechnerkategorien			
	A	B	C	D
Prozessor(en)	AMD K7	Intel Pentium 4	Intel Pentium 4	2 Intel Pentium III
Prozessortakt	665 MHz	2,8 GHz	1.2 GHz	733 MHz
Cache	512 KByte	512 KByte	256 KByte	256 KByte
Hauptspeicher	256 MByte	1 GByte	256 MByte	512 MByte
Netzanbindung	100 MBit/sec	100 MBit/sec	100 MBit/sec	100 MBit/sec
Betriebssystem(e)	Linux	Linux	Linux/Windows 2000	Linux
Linux-Kernel	2.4	2.4	2.4	2.4

Tabelle 8.1: Testplattformen

ident.	identisch
inkomp.	inkompatibel
Komp.	Komponente
komp.	kompatibel
Prot.	Kommunikationsprotokoll
Teilschl.	Teilschlüssel

Tabelle 8.2: Abkürzungen in Tabellen- und Abbildungsbeschriftungen

8.2 Testbedingungen

Es wurden im wesentlichen vier Arten von Rechnern für die Untersuchungen verwendet, die in Tabelle 8.1 beschrieben sind.

Die einzelnen Programmteile von **Skylla** sind in der Programmiersprache **C++** und mit Hilfe der Graphikbibliothek **QT 3.0.4** entwickelt und mit dem Compiler **gcc 2.96** übersetzt. Standen bei den Vergleichssystemen die Quellcodes zur Verfügung, so wurden diese ebenfalls mit **gcc 2.96** kompiliert. Es wurde jeweils über 10 Testläufe gemittelt.

Bei den in dieser Arbeit enthaltenen Histogrammen und Funktionsgraphen werden zur kurzen und prägnanten Beschreibung Ausdrücke der Form

$$y = F(x_1, \dots, x_n)(x_{n+1}, \dots, x_{n+m}) \quad (8.1)$$

verwendet. Dabei stellen x_1, \dots, x_n die auf der x -Achse aufgetragenen Parameter dar, wohingegen x_{n+1}, \dots, x_{n+m} weitere Parameter repräsentieren, die in dem betrachteten Versuch konstant waren beziehungsweise zur Erhaltung einer Kurvenschar variiert wurden. Parameter, die variiert werden, sind in der Regel im Funktionsgraphen oben links dargestellt.

Um die Abbildungsbeschriftungen kurz zu halten, werden in diesen eine Reihe von Abkürzungen verwendet, die in Tabelle 8.2 aufgelistet sind.

8.3 Interface-Typsystem

Das Interface-Typsystem wurde mittels einer Reihe systematischer Tests und einer Fallstudie zur Qualitätskontrolle von Getränkedosen untersucht. Dabei kamen ausschließlich Rechner vom Typ B zum Einsatz (siehe Tabelle 8.1; [Sto03]). In diesem Zusammenhang möchte ich mich bei Herrn Dipl.-Inform. Reiner Kickingeder, Prof. Dr. Donner und der Firma Micro-Epsilon Meßtechnik GmbH & Co. KG für die Zurverfügungstellung des zugehörigen Iconnect-Datenflußgraphen und für die Beantwortung zahlreicher Fragen bedanken.

8.3.1 Systematische Tests

Im Rahmen der systematischen Tests des Interface-Typsystems wurden drei Arten von Datenflußkomponenten mit unterschiedlicher Komplexität der Typconstraints verwendet:

- keine Typconstraints zwischen den Interfaces einer Datenflußkomponente (**simple**)
- Gleichheitsconstraints zwischen den Interfaces einer Datenflußkomponente (**medium**)
- Typconstraints in Form mehrwertiger Polynome (**difficult**)

Zusätzlich zu den Typconstraints der Datenflußkomponenten sind jeweils die durch die Kanten implizierten Gleichheitsconstraints vorhanden. Die Größe des Datenflußgraphen wird hier mit Hilfe der Anzahl der Kanten gemessen, da diese direkt die Anzahl der Aktualisierungsschritte des Typbestimmungsalgorithmus wiedergeben (siehe Abschnitt 5.4). Die Anzahl der Kanten wurde zwischen 25 und 200 variiert. Die Datenflußgraphen sind schichtenweise aufgebaut (vergleiche Abbildung 8.1).

Ein weiteres interessantes Kriterium zur Beurteilung der Effizienz des Typbestimmungsalgorithmus ist die Reihenfolge, in welcher die Kanten eingefügt wurden. Dabei wird unterschieden zwischen

- schichtenweisem Einfügen der Kanten beginnend bei den Quellen (**forward**),
- schichtenweisem Einfügen der Kanten beginnend bei den Senken (**backward**) und
- Konstruktion von vier Teilgraphen, die abschließend miteinander verbunden werden (**parts**).

Die letzte Variante simuliert die Wiederverwendung von Teilgraphen durch den Entwickler. Tabelle 8.3 faßt die untersuchten Merkmale zusammen.

8.3.1.1 Einfügereihenfolge

Abbildung 8.2 zeigt die Einfügezeiten für jede einzelne Kante eines Datenflußgraphen, wobei die Einfügereihenfolge variiert wurde. Fügt man alle Kanten beginnend bei den Quellen in den Datenflußgraphen ein (**forward**), so sind die Zeiten in der Größenordnung von unter einer Millisekunde. Beim Einfügen in umgedrehter Reihenfolge (**backward**) beobachtet man, daß

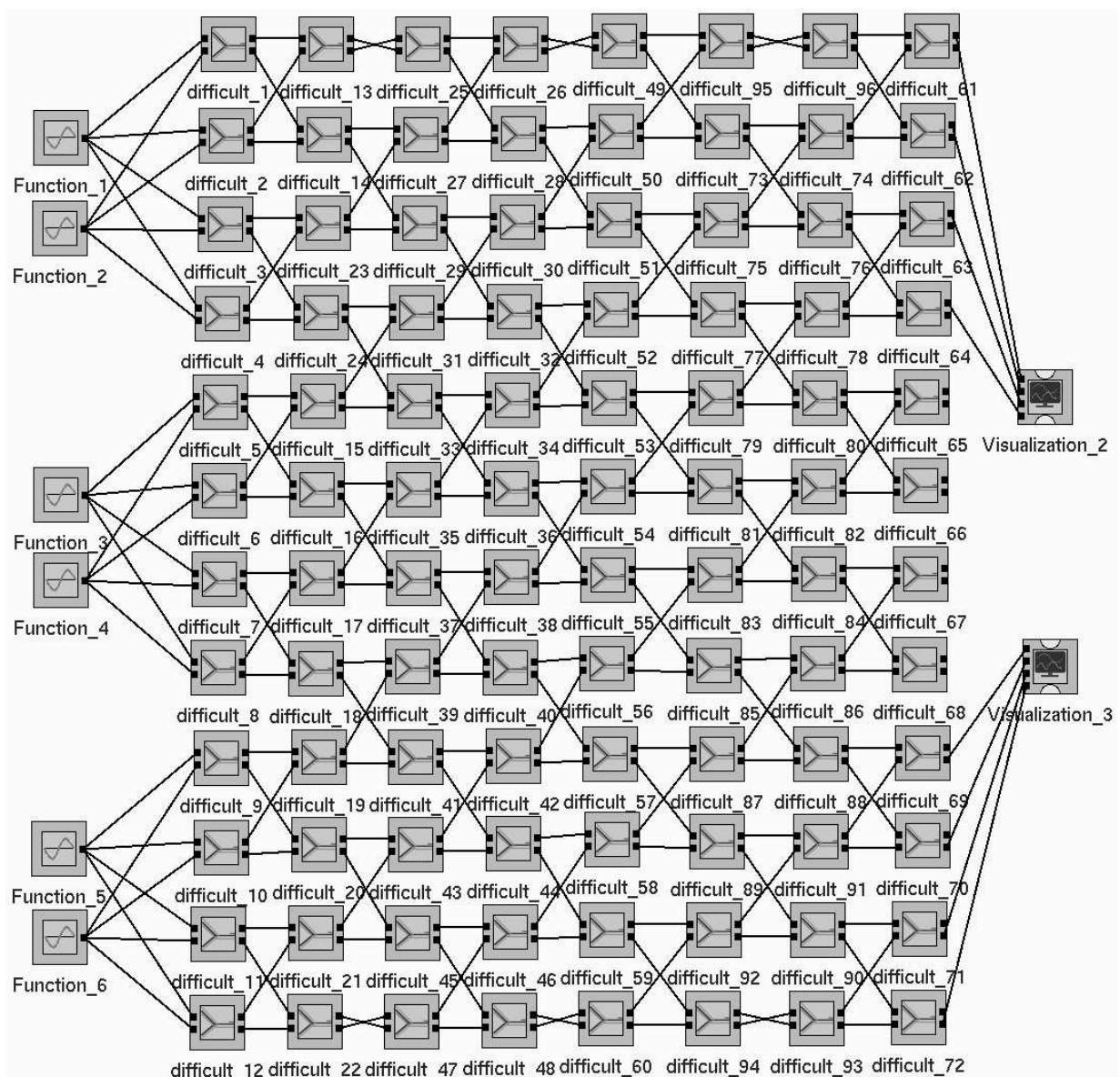


Abbildung 8.1: Beispiel eines Datenflußgraphen (systematische Tests)

Constraintkomplexität	simple, medium, difficult
Graphgröße (Anzahl Kanten)	25, 50, 100, 200
Einfügereihenfolge	forward, backward, parts

Tabelle 8.3: Systematische Tests (Interface-Typsystem)

beim Einfügen der Quellen die Rechenzeit erheblich ansteigt. Ursache hierfür ist, daß bei der Einfügereihenfolge forward die Typänderungen ausgehend von den Quellen immer mitpropa-

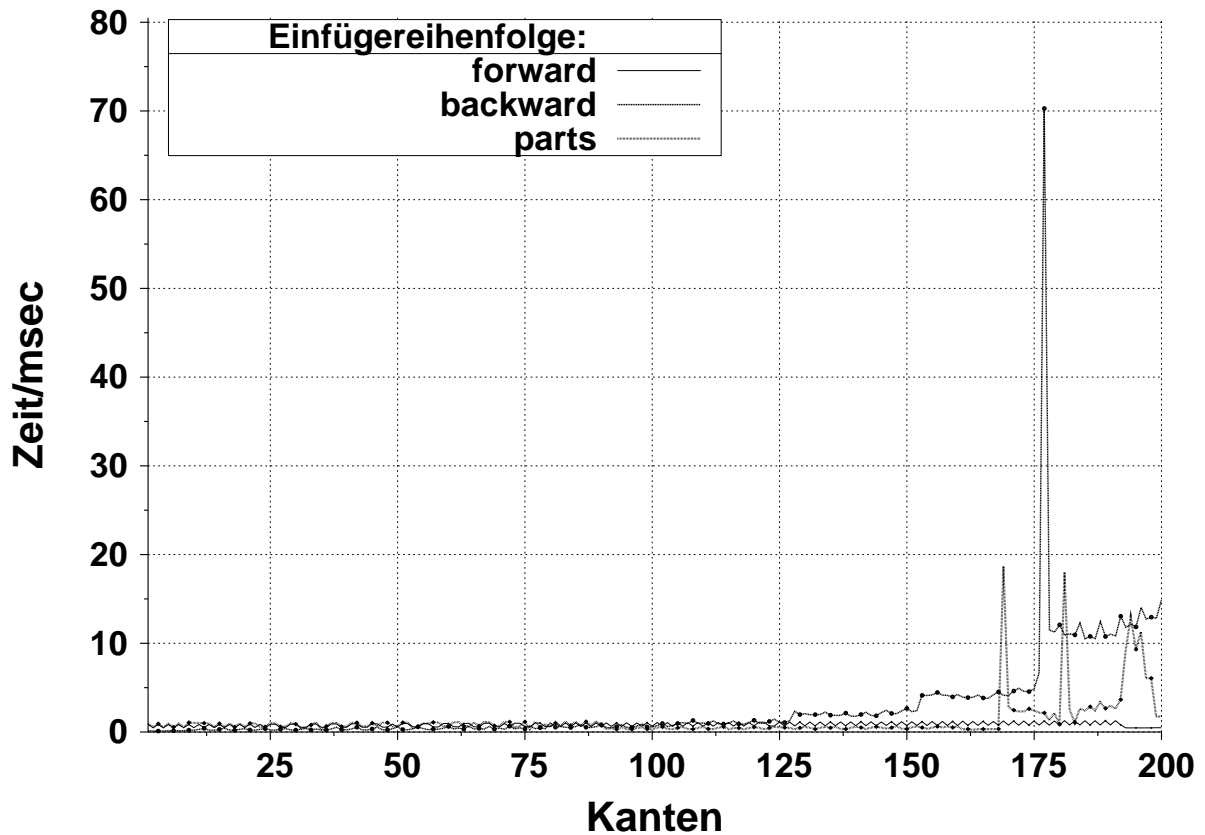


Abbildung 8.2: Einfügezeit = $F(\text{Kanten})$ (Datenflußgraph mit 200 Kanten; Constraintkomplexität: difficult)

giert wurden. Bei der Einfügereihenfolge **backward** werden die Interfacetypen aber erst gegen Ende durch die Quellen eingeschränkt. Dies hat zur Folge, daß Typänderungen durch den gesamten Datenflußgraphen propagiert werden müssen. Betrachtet man die dritte Einfügereihenfolge **parts**, so erkennt man, daß auch dort das Verbinden mit Teilgraphen, die Quellkomponenten enthalten, zu etwas kleineren Ausschlägen führt. Insgesamt kann aber festgehalten werden, daß selbst die größte Spitze in dem ermittelten Kurvenverlauf immer noch einen so kleinen Zeitwert (70 msec) darstellt, daß dieser vom Benutzer im interaktiven Entwurf nicht wahrgenommen wird.

8.3.1.2 Constraintkomplexität

Abbildung 8.3 zeigt die mittlere Einfügezeit für eine Kante in Abhängigkeit von der Komplexität der Typconstraints der Datenflußkomponenten. Dabei sieht man, daß selbst im Falle der Typconstraints der Art **difficult** die Einfügezeiten so klein sind, daß diese den Benutzer während des Programmentwurfs nicht stören.

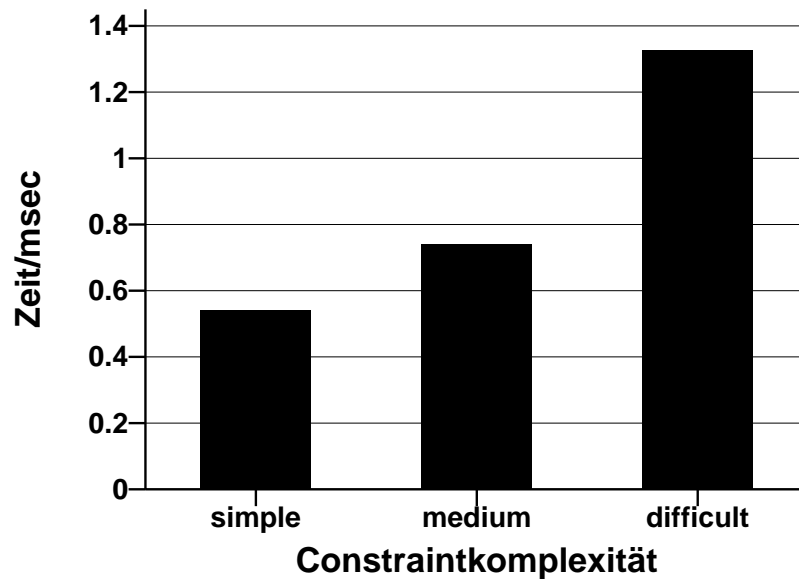


Abbildung 8.3: Mittlere Einfügezeit = F(Constraintkomplexität)

8.3.1.3 Graphgröße

Der dritte wichtige Parameter ist die Größe eines Datenflußgraphen. Abbildung 8.4 zeigt die mittlere Einfügezeit in Abhängigkeit von diesem Parameter. Auch hier ist zu erkennen, daß die Zeiten für das Einfügen von Kanten dem entwurfsbegleitenden Einsatz des Typprüfungsalgorithmus nicht im Wege stehen.

8.3.1.4 Löschen

Ein wichtiger Gesichtspunkt bei der Bewertung des Typprüfungsalgorithmus ist die Zeit, die zur Bestimmung der Interfacetypen benötigt wird, wenn eine Kante gelöscht wird. Im Gegensatz zum Einfügen einer Kante geschieht das Löschen NICHT INKREMENTELL. Das heißt, alle Typbelegungen des Datenflußgraphen müssen neu berechnet werden (vergleiche dazu Abschnitt 5.4). In Abbildung 8.5 sind die mittleren Zeiten, die zum Löschen einer Kante benötigt werden, in Abhängigkeit von der Graphgröße und aufgesplittet nach den drei betrachteten Constraintkomplexitäten simple, medium und difficult aufgezeigt. Selbst im Falle des Datenflußgraphen mit 200 Kanten und Typconstraints der Komplexität difficult sind die Zeiten ausreichend klein, so daß diese einem entwurfsbegleitenden Einsatz nicht im Wege stehen. Denn man kann davon ausgehen, daß in einer Anwendung Datenflußkomponenten unterschiedlicher Constraintkomplexitäten eingesetzt werden, so daß im Regelfall kleinere Zeiten zu erwarten sind.

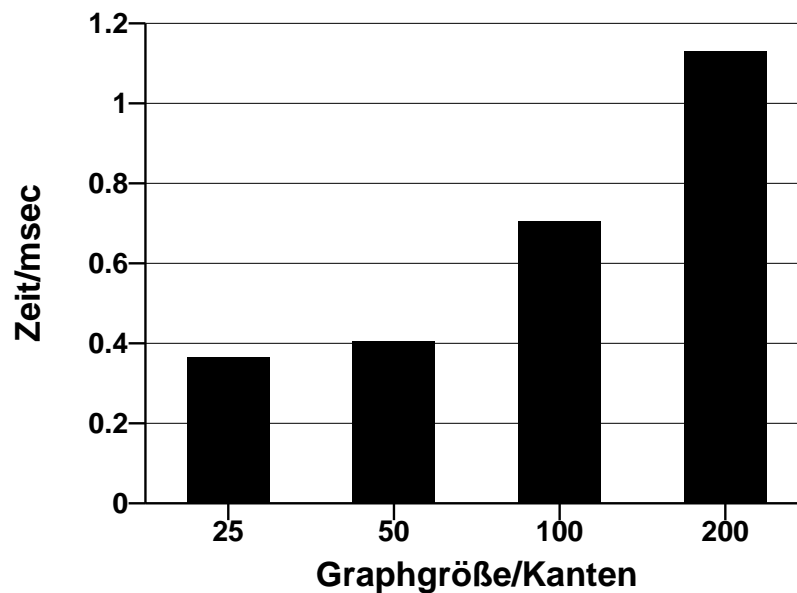


Abbildung 8.4: Mittlere Einfügezeit = $F(\text{Graphgröße})$

8.3.2 Fallstudie: Qualitätskontrolle von Getränkedosen

Als Fallstudie wurde ein Datenflußgraph für die Qualitätskontrolle von Getränkedosen, der bereits in der industriellen Fertigung eingesetzt wurde, implementiert [BFH⁺03, Sto03]. Der dazugehörige Iconnect-Datenflußgraph wurde dankenswerterweise von Herrn Dipl.-Inform. Reiner Kickingereder und Prof. Dr. Donner zur Verfügung gestellt. Eine detaillierte Beschreibung findet sich in [BFH⁺03], welche im folgenden zusammenfassend wiedergegeben ist. Ziel des Datenflußgraphen ist es, Getränkedosen aus dem Produktionsprozeß zu entfernen, die keine zuverlässige Dichtigkeit aufweisen. Dazu werden meßbare Eigenschaften einer halbfertigen Getränkedose mit Sollwerten verglichen. Bei diesem Zwischenprodukt handelt es sich um einen Dosenrumpf, der schematisch in Abbildung 8.6 dargestellt ist. Der Dosenrand wird dabei in einen inneren und einen äußeren Dosenrand sowie die dazwischenliegende Randfläche unterteilt. Diese Randfläche verläuft zunächst flach, dann steil abfallend zum inneren Rand hin. Die Meßaufgabe besteht darin, die Größe der Ränder sowie deren Kreisform zu ermitteln und auszuwerten.

Der Datenflußgraph gliedert sich in folgende Abschnitte:

1. **BILDAUFNAHME:** Der zu vermessende Dosenrumpf mit seinen Rändern wird zunächst von einer Grauwertkamera erfaßt (vergleiche Abbildung 8.6). Die Dosen laufen dabei auf einem Fließband unter der Kamera vorbei. Die Bildaufnahme erfolgt jedesmal, wenn eine Dose eine Lichtschranke passiert. Durch eine entsprechende Ausleuchtung der Dose wird dabei ein optimales Kontrastverhältnis im aufgenommenen Bild erreicht.
2. **POSITIONSBESTIMMUNG DER DOSE:** Für die weitere Verarbeitung ist zunächst eine Lagebestimmung der Dose im aufgenommenen Bild notwendig. Dazu wird der Grauwert-

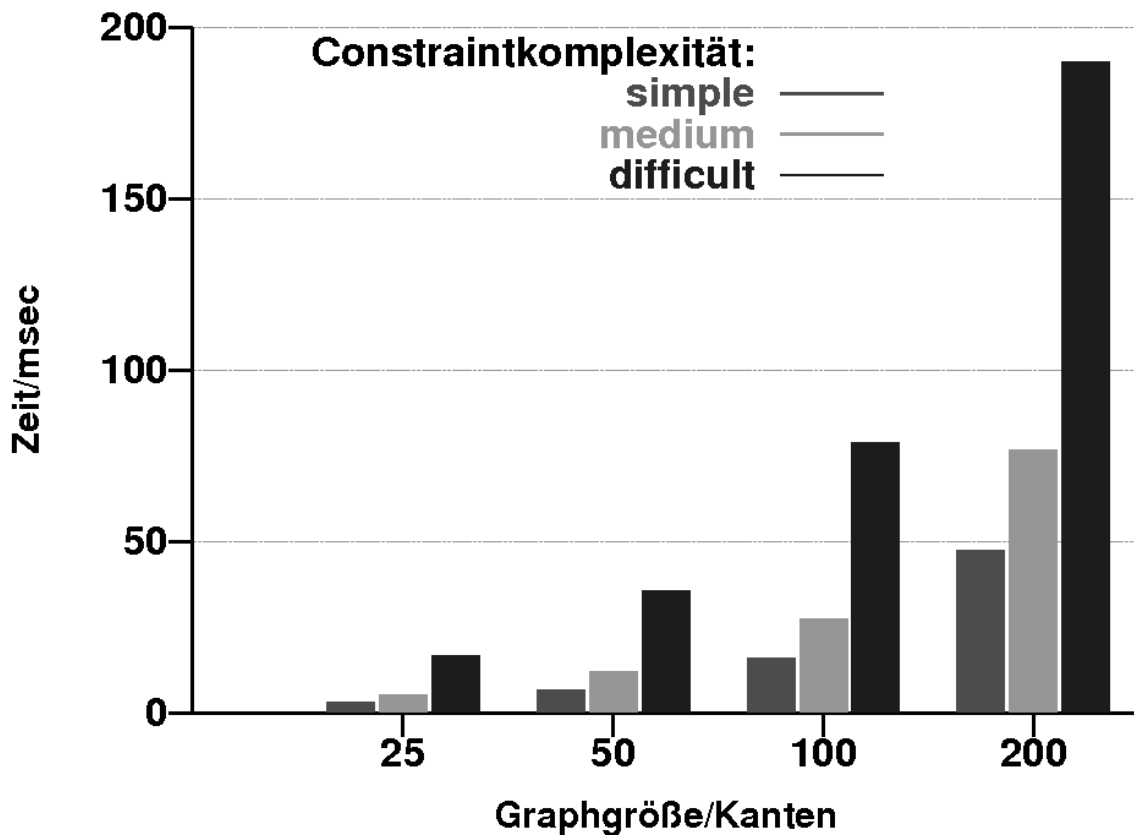


Abbildung 8.5: Mittlere Neuberechnungszeit = $F(\text{Graphgröße}, \text{Constraintkomplexität})$

schwerpunkt im Bild ermittelt. Um die Daten und die Komplexität zu reduzieren, reicht die Betrachtung eines vergrößerten Bildes aus, in dem nur jedes vierte Pixel in x - und y -Richtung verwendet wird.

3. **EXTRAKTION DER RANDPUNKTE:** Entlang von Strahlen, die von diesem ermittelten Grauwertschwerpunkt ausgehen und sternförmig in alle Richtungen verlaufen, werden die äußeren Randpunkte der Dose extrahiert. Um diese Randpunkte wird ein Ring mit festgelegter Breite gelegt. Auf den einzelnen Strahlen können nun die Punkte des inneren und äußeren Randes mit einem Schwellwertverfahren bestimmt werden. Die Anzahl der zu untersuchenden Randpunkte ist dabei vom Anwender vorgegeben.
4. **TRANSFORMATION IN METRISCHES SYSTEM:** Vor der Weiterverarbeitung der extrahierten Randpunkte ist eine Umrechnung in das metrische System notwendig. Denn die gemessenen Abstände und Positionen sind in Bildkoordinaten, das heißt Pixeln, angegeben. Dazu wird eine mittels einer Kamerakalibrierung ermittelte Transformation zur Umrechnung von Pixelkoordinaten in metrische Koordinaten verwendet.

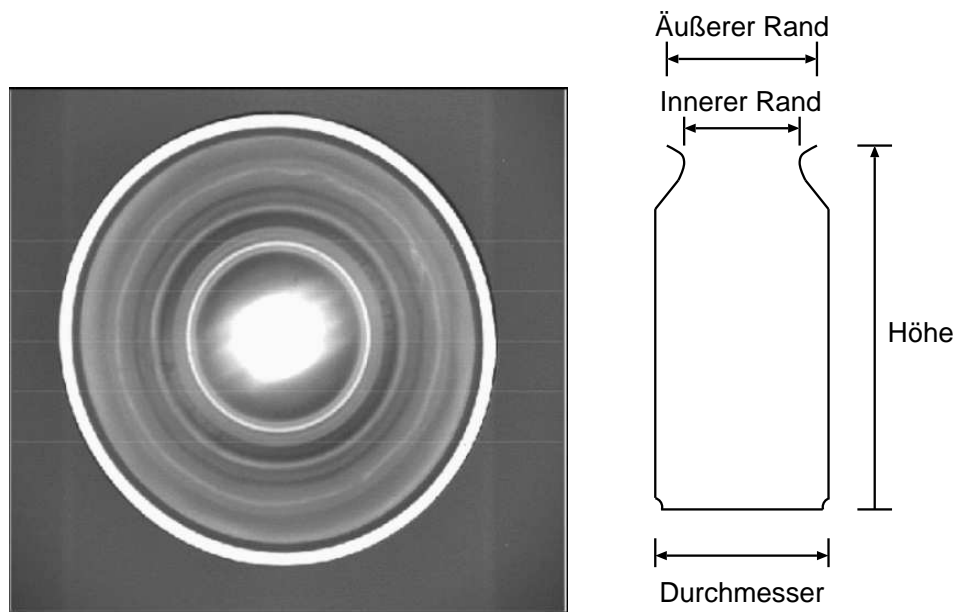


Abbildung 8.6: Grauwertbild und Seitenriß einer Getränkedose

5. **KREISSPASSUNG:** Für einen Vergleich mit der Spezifikation paßt man durch die gefundenen Randpunkte zwei Kreise, die den inneren und äußeren Dosenrand repräsentieren.

6. **AUSWERTUNG:** Die ermittelten Randpunkte, gepaßten Kreise, Kreismittelpunkte und Radien können nun in zweierlei Weise zur Klassifikation von Getränkedosen verwendet werden. Zum einen wird die Abweichung von idealen konzentrischen Kreisen ermittelt. Zum anderen wird der Abstand der Radien von den Vorgaben bestimmt. Anhand eines Vergleichs dieser gemessenen Istwerte mit vorgegebenen Sollwerten wird dann eine Dose gegebenenfalls aus dem Produktionsprozeß entfernt.

Der zu dieser Beschreibung gehörende Datenflußgraph besteht aus 204 Datenflußkomponenten und 336 Kanten. In Abbildung 8.7 sind die Zeiten aufgeführt, welche der Typbestimmungsalgorithmus beim Einfügen einer jeden Kante benötigt hat. Dabei war die Kanteneinfügereihenfolge zufällig, wie dies in einem normalen Anwendungsszenario üblich ist. Man erkennt wie bei den vorherigen Kurvenverläufen, daß das Einfügen von Quellmodulen zu Spitzen im Zeitverbrauch führt. Diese Spitzenwerte sind aber vernachlässigbar klein. Die Zeit für eine vollständige Neuberechnung, wie sie bei einer Löschooperation auftritt, beträgt 0.3 Sekunden. Damit gilt auch hier das Fazit, daß der Typbestimmungsalgorithmus sich uneingeschränkt für einen entwurfsbegleitenden Einsatz eignet.

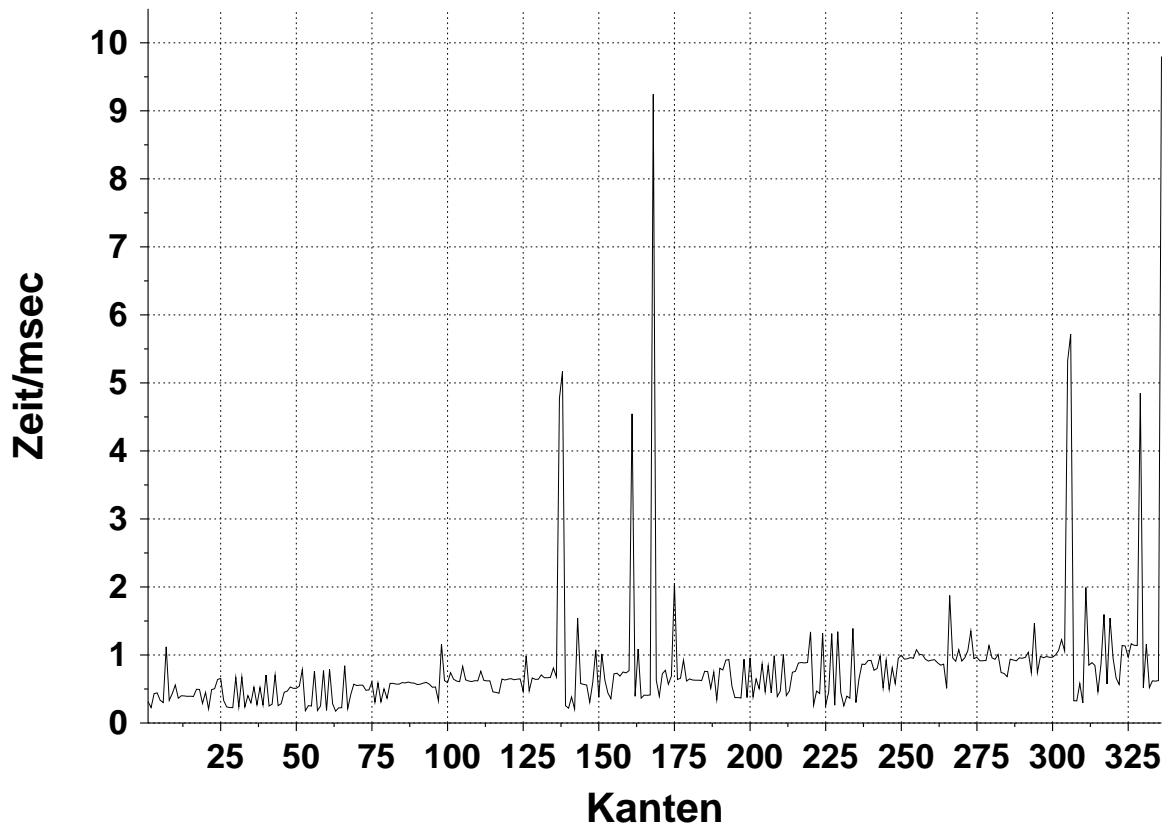


Abbildung 8.7: Einfügezeit = $F(\text{Kanten})$ (Qualitätskontrolle von Getränkedosen)

8.4 Model Checking

Das in Kapitel 6 eingeführte Model-Checking-Verfahren wurde in über 160000 systematischen Einzeltests, die mehrere Wochen lang nachts und am Wochenende auf 13 Rechnern vom Typ B (siehe Tabelle 8.1) liefen, untersucht. Neben diesen systematischen Tests, die mit Hilfe von Datenflußgraphen des Datenflußparadigmas Colored SDF durchgeführt wurden und welche vor allem die zeitliche Effizienz des Verfahrens testen sollten, wurden auch Untersuchungen anhand von Datenflußgraphen der Datenflußparadigmen Colored BDF und Colored DDF durchgeführt. Letztere Tests sollen die Eignung des Model Checkers für turingvollständige Datenflußparadigmen – mit all den damit verbundenen Einschränkungen (vergleiche Abschnitt 6.4.6) – belegen. Außerdem wurde das in dieser Arbeit entwickelte Model-Checking-Verfahren dem Model Checker Spin (vergleiche Abschnitt 2.4.5) gegenübergestellt. Die Beschreibung der diesem Abschnitt zugrundeliegenden Versuche und Ergebnisse ist im Vergleich zu den anderen Abschnitten dieses Kapitels sehr ausführlich gehalten, da die entsprechenden Ergebnisse an keiner anderen Stelle nachlesbar sind. Die in [May04a, May04b] präsentierten Resultate sind mittlerweile veraltet und durch bessere ersetzt worden.

8.4.1 Systematische Tests

Im Zuge der in dieser Arbeit durchgeführten Tests wurden eine Reihe von Parametern untersucht und deren Auswirkungen auf das Verhalten des Model-Checking-Verfahrens mit Hilfe diverser Ausgaben beobachtet.

Testparameter: Die bei der Untersuchung variierten Parameter kann man in GRAPHPARAMETER, KANTENPARAMETER, KOMPONENTENPARAMETER und ALGORITHMUSPARAMETER unterteilen.

- **Graphparameter:** Graphparameter beschreiben die Struktur und die Größe des betrachteten Datenflußgraphen.
 1. **STRUKTUR DES DATENFLUSSGRAPHEN:** Die betrachteten Graphstrukturen waren Rows, Net und Random. Die Graphstruktur Rows bezeichnet Datenflußgraphen mit jeweils einer Quelle und einer Senke und einer einstellbaren Anzahl von Komponentenreihen zwischen beiden (vergleiche Abbildung 8.8 (a)). Die zweite verwendete Graphstruktur Net ist in Abbildung 8.8 (b) dargestellt. Es handelt sich um eine schichtenweise aufgebaute Graphstruktur, wobei die Art der Verbindung der Schichten untereinander ebenfalls parametrisiert werden kann. Bei der Graphstruktur Random (vergleiche Abbildung 8.8 (e)) sind die Datenflußkomponenten auch in Reihen und Spalten angeordnet. Die Verbindungen zwischen den einzelnen Datenflußkomponenten sind jedoch zufällig erzeugt.
 2. **GRÖSSE DES DATENFLUSSGRAPHEN:** Neben der Struktur des Datenflußgraphen kann auch dessen Größe, das heißt die Anzahl seiner Datenflußkomponenten, eingestellt werden. Dabei wird diese Größe indirekt durch die Angabe der Breite und Tiefe der Graphstruktur bestimmt.
 3. **RÜCKKOPPLUNGEN:** Ein weiteres einzustellendes Merkmal ist die Verwendung von Rückkopplungen. Die Abbildungen 8.8 (c) und (d) veranschaulichen, wie die Graphstrukturen Rows und Net mit Rückkopplungen (RowsF und NetF) aussehen. Bei der Struktur Random sind Rückkopplungen generell erlaubt. Rückkopplungen treten in realen Datenflußgraphen vergleichsweise häufig auf, da diese beispielsweise eine Möglichkeit der Speicherung darstellen (vergleiche Abschnitt 2.2.3).
- **Kantenparameter:** Der einzige hier betrachtete Kantenparameter ist die KAPAZITÄT der durch die Kante repräsentierten Fifo. In der Regel ist dieser Wert auf 1 Million eingestellt, was eine UNENDLICH GROSSE KANTENKAPAZITÄT nachbilden soll.
- **Komponentenparameter:** Bei den Parametern, die eine Datenflußkomponente charakterisieren, unterscheidet man zwischen dem durch eine Datenflußkomponente realisierten KOMMUNIKATIONSPROTOKOLL, der ANZAHL DER SCHNITTSTELLEN einer Datenflußkomponente und den GEWICHTEN, die den jeweiligen Schnittstellen zugeordnet sind.

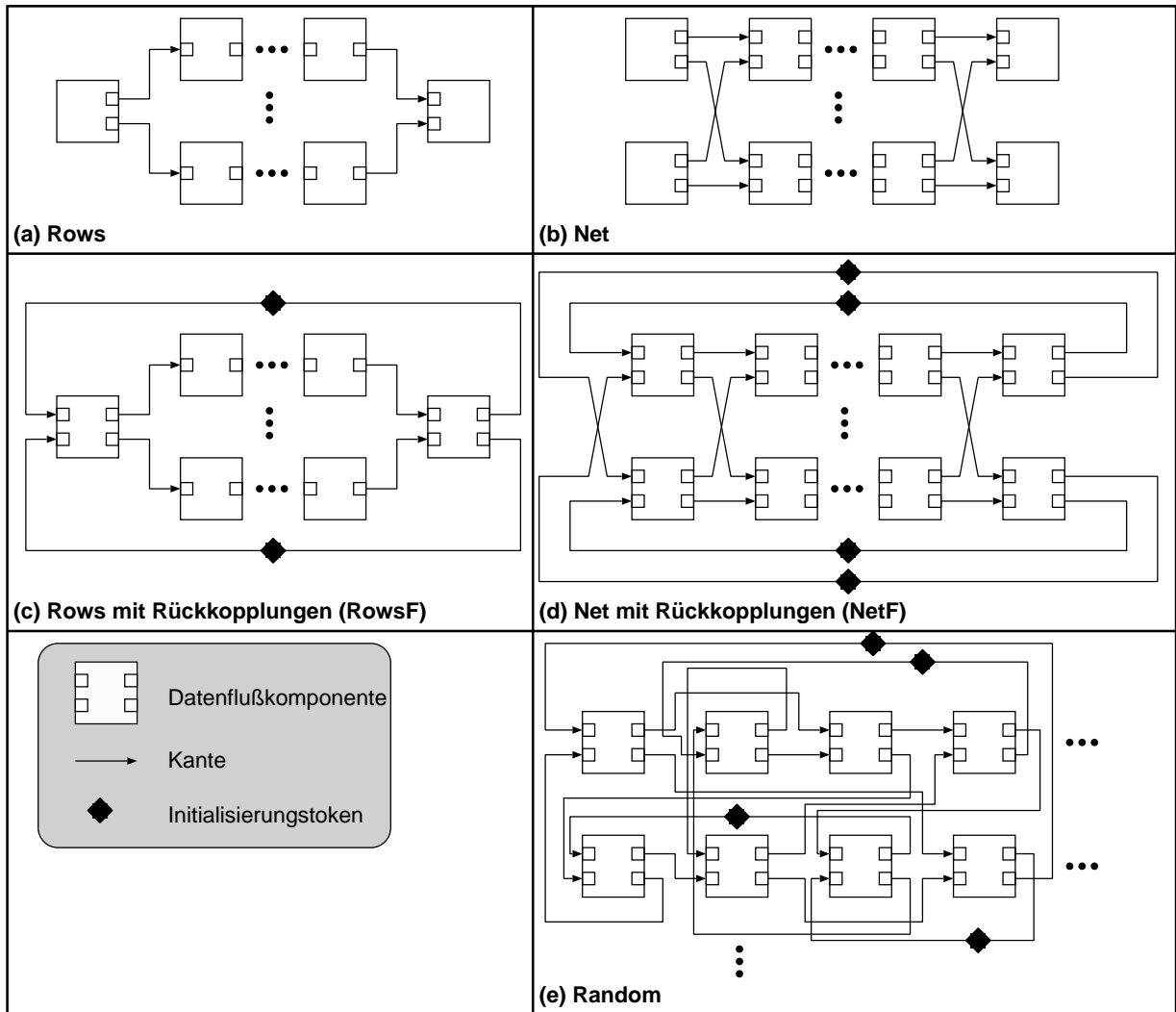


Abbildung 8.8: Graphstrukturen Rows, Net und Random

1. KOMMUNIKATIONSPROTOKOLL: Im Rahmen der systematischen Tests wurden mehrere Arten von Datenflußkomponenten untersucht. Die funktionale Beschreibung einer solchen Testkomponente sieht beispielsweise folgendermaßen aus (vergleiche Abschnitt 6.4):

$$\mathbf{test}_s \begin{cases} S_s^n \rightarrow S_s^n & (n \in \mathbb{N}) \\ x \bullet r \mapsto \mathbf{test}_{\text{prefix}_s}(x) \bullet \mathbf{test}_s(r) \\ x \bullet r \mapsto \varepsilon_S \text{ sonst} \end{cases} \quad (8.2)$$

$$\mathbf{test}_{\text{prefix}_s} \begin{cases} P_{S_s^n} \rightarrow P_{S_s^n} \\ \underbrace{(0, 0, \dots, 0)}_{n \text{ mal}} \mapsto \underbrace{(0, 0, \dots, 0)}_{n \text{ mal}} \end{cases} \quad (8.3)$$

Ein zentrales Merkmal einer Datenflußkomponente ist das realisierte Kommunikationsprotokoll. Da bei den vorliegenden Tests im wesentlichen Datenflußkomponenten verwendet wurden, deren konsumierte und produzierte Strommengen S_s durch denselben regulären Ausdruck beschrieben werden können (vergleiche Abschnitt 3.3.3), werden im folgenden diese regulären Ausdrücke zur abkürzenden Beschreibung der jeweiligen Datenflußkomponenten verwendet. Obige funktionale Beschreibung stellt eine klassische SDF-Komponente dar, welche durch den regulären Ausdruck o^* gekennzeichnet wird. Zwei relativ einfache Kommunikationsprotokolle sind durch $(se)^*$ und $(sme)^*$ gegeben. Gefärbte Datenflußkomponenten, die beliebig lange Signalsegmente verarbeiten können, sind mittels der Ausdrücke $(sm^*e)^*$, $(sm^+e)^*$ und $(sm^*e|o)^*$ beschrieben. Das Kommunikationsprotokoll $(s|m|e|o)^*$ beschreibt eine Datenflußkomponente, welche mit jeder der bisher vorgestellten Komponentenarten kombiniert werden kann. Demgegenüber beschreibt e^* eine Datenflußkomponente, deren Kombination mit den bisher vorgestellten Komponentenarten jeweils zu einem Fehler führt.

2. ANZAHL DER SCHNITTSTELLEN: Die Zahl der Ein- und Ausgangsschnittstellen der Datenflußkomponenten kann variiert werden.
 3. GEWICHTE: Die Gewichte sind entweder alle auf den gleichen Wert, welcher in der Regel 1 ist, eingestellt oder zufällig gewählt.
- **Algorithmusparameter:** Unter Algorithmusparametern versteht man Parameter, welche das Verhalten des Model-Checking-Verfahrens direkt beeinflussen.
 1. TIMEOUT: Zum einen wird ein zeitliches Limit vorgegeben, nach dessen Ablauf der Algorithmus unabhängig von seinem Ergebnis terminiert wird. Dieser Wert ist auf 1 Stunde eingestellt.
 2. SPEICHERLIMIT: Das Speicherlimit gibt den Schwellwert der Speicherbelegung an, bei dessen Überschreitung der Algorithmus terminiert wird. Dieses Limit ist auf 1 GByte eingestellt, da dies dem maximal nutzbaren Hauptspeicher der verwendeten Rechner vom Typ B (vergleiche Tabelle 8.1) entspricht. Ein größerer Wert macht keinen Sinn, da dann die meiste Zeit mit Swappen verschwendet würde.
 3. SUCHZIEL: Als Suchziel kann man angeben, daß der Algorithmus genau einen Zyklus, der den Wurzelknoten des Erreichbarkeitsgraphen beinhaltet, beziehungsweise alle solchen Zyklen suchen soll.
 4. BLOCKIEREN/DEBLOCKIEREN: Man kann einstellen, ob der Blockier/Deblockier-Mechanismus des Model-Checking-Verfahrens genutzt werden soll.
 5. PARTIAL ORDER REDUCTION: Außerdem kann man die Partial Order Reduction ein- beziehungsweise ausschalten.
 6. ANZAHL DER BITS FÜR ORDERED MAPS: Bei der verwendeten Datenstruktur Ordered Maps kann man einstellen, wieviele Bits für einen Teilschlüssel genutzt werden sollen.

Graphparameter	
Graphstrukturen	Rows, Net, Random
Graphgröße	Breite, Tiefe
Rückkopplungen	ohne/mit
Kantenparameter	
Kapazität	1000000/zufallsverteilt
Komponentenparameter	
Kommunikationsprotokolle	o^* , $(se)^*$, $(sme)^*$, $(sm^*e)^*$, $(sm^+e)^*$, $(sm^*e o)^*$, $(s m e o)^*$, e^* .
Anzahl der Schnittstellen	2/4
Gewichte	1/zufallsverteilt
Algorithmusparameter	
Timeout	1 Stunde
Speicherlimit	1 GByte
Suchziel	ein/alle Zyklen
BLOCKIEREN	aktiviert/deaktiviert
PARTIAL ORDER REDUCTION	aktiviert/deaktiviert
Anzahl der Bits für Ordered Maps	4/frei wählbar
Seed für Zufallszahlen	frei wählbar

Tabelle 8.4: Systematische Tests (Model Checker)

7. SEED FÜR ZUFALLSZAHLEN: Für die Generierung von Zufallszahlen zum Beispiel für Gewichte wird die Seed vorgegeben. Dies garantiert, daß die entsprechenden Tests reproduzierbar werden. So kann man beispielsweise Versuche durchführen, bei der dieselbe Graphstruktur mit unterschiedlichen Kommunikationsprotokollen untersucht wird.

In Tabelle 8.4 sind die Parameter des Model-Checking-Verfahrens überblicksartig zusammengestellt.

Ausgaben des Model Checkers: Während der Ausführung des Suchalgorithmus beziehungsweise nach der Terminierung der Suche werden diverse Statistikdaten ausgegeben.

- **Zeitverbrauch:** Es wird der Zeitverbrauch gemessen, wobei diverse von Linux zur Verfügung gestellte Funktionen zur Zeitmessung genutzt werden. Außerdem werden im Falle von Intel-Prozessoren die Prozessorzyklen mittels einer Assembleroutine hochgenau ausgelesen.
- **Speicherverbrauch:** Der Speicherverbrauch wird ausgegeben.
- **Zyklen:** Die Anzahl der gefundenen Zyklen im Erreichbarkeitsgraphen, die den Wurzelknoten beinhalten, wird abgespeichert.

- **Datenflußgraph:** Die Anzahl der Datenflußkomponenten, die in einem Datenflußgraphen mit vorgegebener Graphstruktur enthalten sind, wird ausgegeben.
- **Knoten:** Die Anzahl der ermittelten Knoten des Erreichbarkeitsgraphen wird abgespeichert.
- **Anzahl der Deblocker:** Die Anzahl der von dem Model-Checking-Verfahren erzeugten Deblocker wird gemessen.
- **Statemaps:** Die Zahl der erzeugten Statemaps wird abgespeichert.
- **Wordmaps:** Die Anzahl der erzeugten Wordmaps wird ausgegeben.

8.4.1.1 Graphstruktur Rows

Dieser Abschnitt behandelt die Untersuchungen zur Graphstruktur Rows (vergleiche Abbildung 8.8 (a)). Diese Tests sollen insbesondere den herausragenden Nutzen des in dieser Arbeit entwickelten PARTIAL-ORDER-REDUCTION-Verfahrens bewerten helfen, da bei der Graphstruktur Rows eine besonders hohe Anzahl von gleichwertigen Pfaden im Erreichbarkeitsgraphen existiert, von denen nur einer von Interesse ist (vergleiche das Beispiel in Abbildung 6.9). Bei den nachfolgenden Untersuchungen werden Datenflußgraphen mit identischen Datenflußkomponenten, mit von den übrigen Datenflußkomponenten unterschiedlichen Quellkomponenten, internen Datenflußkomponenten und Senkenkomponenten betrachtet. Außerdem werden sowohl Datenflußkomponenten mit kompatiblen als auch inkompatiblen Kommunikationsprotokollen untersucht.

Identische Kommunikationsprotokolle: Ein wichtiger Aspekt bei der Simulation eines Datenflußgraphen und der Suche eines Zyklus in dem dabei konstruierten Erreichbarkeitsgraphen ist die benötigte Zeit. Der Zeitverbrauch ist um so wichtiger, da das Verfahren entwurfsbegleitend eingesetzt werden soll (vergleiche Abschnitt 1.4.2).

Betrachtet man die Graphstruktur Rows, so sind in Abbildung 8.9 die benötigte Zeit, bis der erste den Wurzelknoten beinhaltende Zyklus im Erreichbarkeitsgraphen gefunden wurde, in Abhängigkeit von der Anzahl der Datenflußkomponenten aufgezeigt. Die Breite des untersuchten Datenflußgraphen beträgt 2. Die Tiefe wird beginnend bei 1000 jeweils um den Wert 1000 erhöht, bis der für die Simulation notwendige Speicherplatz die vorgegebene Schranke von 1 GByte überschritten hat. Die benötigten Zeiten steigen im wesentlichen linear in Abhängigkeit von der Anzahl der Datenflußkomponenten. Interessant ist auch, daß die Anzahl der Datenflußkomponenten sehr hoch ist. In Abhängigkeit vom verwendeten Kommunikationsprotokoll sind Datenflußgraphen mit bis zu 200000 Datenflußkomponenten untersucht worden. Damit ist das vorgestellte Verfahren im Falle des Kommunikationsprotokolls \mathcal{O}^* , das dem Kommunikationsverhalten der klassischen Datenflußparadigmen entspricht, durchaus konkurrenzfähig mit dem im Stand der Technik eingesetzten Lösen von linearen Gleichungssystemen (vergleiche Abschnitt 2.2.10). Im Gegensatz zu diesem Stand der Technik können mit dem in dieser Arbeit

entwickelten Model-Checking-Verfahren auch Datenflußgraphen aus gefärbten Datenflußparadigmen (vergleiche Abschnitt 4.3.3) untersucht werden.

Abbildung 8.10 zeigt die benötigte Zeit in Abhängigkeit von der Anzahl der Datenflußkomponenten für das Kommunikationsprotokoll $(sm^*e|o)^*$. Dabei wurde die Breite der untersuchten ROWS-Graphstrukturen zwischen 2 und 5 variiert. Die Zeiten wachsen wiederum linear, wobei nur die Anzahl der Datenflußkomponenten und nicht die Breite ausschlaggebend ist. Dies unterstreicht die besondere Qualität der verwendeten PARTIAL ORDER REDUCTION in Kombination mit der ausgenutzten KOMMUTATIVITÄT beim Expandieren von Knoten des Erreichbarkeitsgraphen (vergleiche Abschnitt 6.4.5).

Der Speicherverbrauch des Model Checkers stellt eine weitere zentrale Größe bei dessen Bewertung dar. Abbildung 8.11 zeigt den Speicherverbrauch in Abhängigkeit von der Anzahl der Datenflußkomponenten eines Datenflußgraphen. Der Datenflußgraph besitzt eine ROWS-Graphstruktur mit der Breite 2. Der Speicherverbrauch wächst linear mit der Anzahl der Datenflußkomponenten. Die einzelnen Kurvenverläufe enden bei jeweils dem Datenflußgraphen, der noch vollständig innerhalb der Grenze von 1 GByte simuliert werden konnte. Es ist zu erkennen, daß für unterschiedliche Kommunikationsprotokolle der Speicherverbrauch unterschiedlich stark anwächst.

Abbildung 8.12 zeigt den Speicherverbrauch in Abhängigkeit von der Anzahl der Datenflußkomponenten für das Kommunikationsprotokoll $(sm^*e|o)^*$ bei unterschiedlicher Breite der Graphstruktur ROWS. Der Speicherverbrauch ist im wesentlichen nur von der Anzahl der Datenflußkomponenten und nicht von der Breite abhängig.

Die Anzahl der Knoten des Erreichbarkeitsgraphen wächst linear mit der Anzahl der Datenflußkomponenten. Die Abbildungen 8.13 und 8.14 veranschaulichen dieses Ergebnis. Die Anzahl der Datenflußkomponenten und die Art der verwendeten Kommunikationsprotokolle beeinflussen den Kurvenverlauf. Betrachtet man dagegen die Anzahl der Knoten in Abhängigkeit von der Anzahl der Datenflußkomponenten unter Berücksichtigung verschiedener Breiten, so beobachtet man im Falle der Graphstruktur ROWS, daß die Breite aufgrund der in dieser Arbeit entwickelten PARTIAL ORDER REDUCTION keinerlei Rolle spielt (vergleiche Abbildung 8.14).

Die Abbildung 8.15 zeigt die Anzahl der erzeugten Statemaps in Abhängigkeit von der Anzahl der Datenflußkomponenten für die Graphstruktur ROWS unter Berücksichtigung unterschiedlicher Kommunikationsprotokolle. Man beobachtet eine lineare Abhängigkeit zwischen der Anzahl der Datenflußkomponenten und der Anzahl der benötigten Statemaps.

In Abbildung 8.16 ist die Anzahl der Statemaps für das Kommunikationsprotokoll $(sm^*e|o)^*$ in Abhängigkeit von der Anzahl der Datenflußkomponenten aufgetragen. Dabei wurde die Breite der Datenflußgraphen variiert. Man erkennt im wesentlichen ein lineares Wachstum der Anzahl der Statemaps in Abhängigkeit von der Anzahl der Datenflußkomponenten.

Die Kurvenverläufe für die Wordmaps sind ähnlich wie die Kurven zu den Statemaps. Die Abbildung 8.17 zeigt die benötigten Wordmaps in Abhängigkeit von der Anzahl der Datenflußkomponenten für die Graphstruktur ROWS unter Berücksichtigung unterschiedlicher Kommunikationsprotokolle. Man beobachtet eine lineare Abhängigkeit zwischen der Anzahl der Datenflußkomponenten und der Anzahl der benötigten Wordmaps.

In Abbildung 8.18 ist die Anzahl der Wordmaps in Abhängigkeit von der Anzahl der Datenflußkomponenten und der Breite der Graphstruktur aufgetragen. Auch hier ist die Abhängigkeit linear.

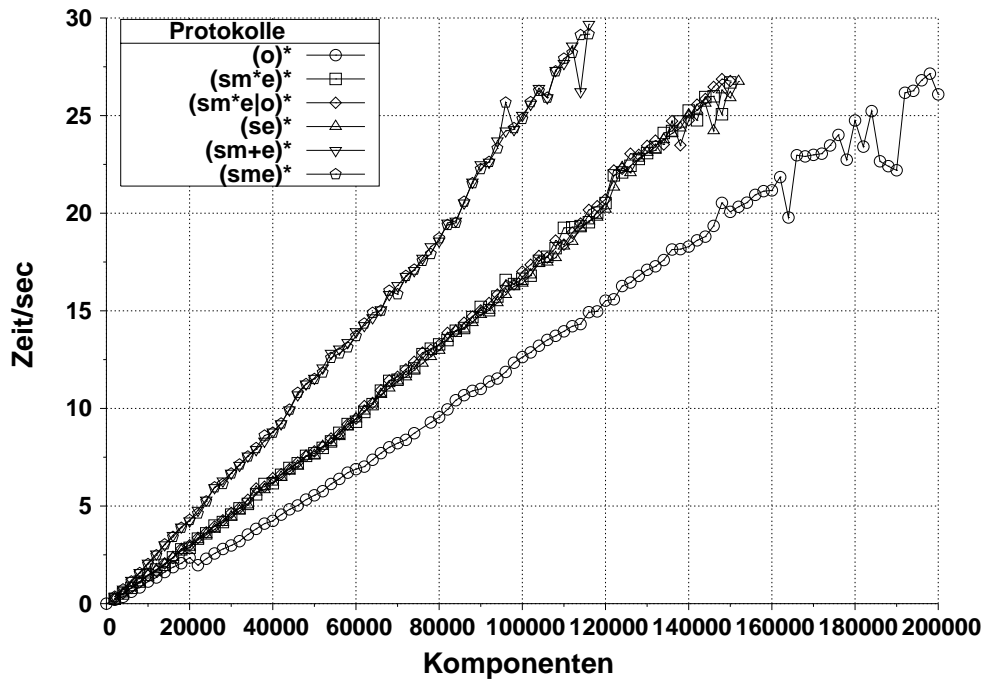


Abbildung 8.9: Zeitverbrauch=F(Komp.) (Rows, Breite = 2, ident. Prot.)

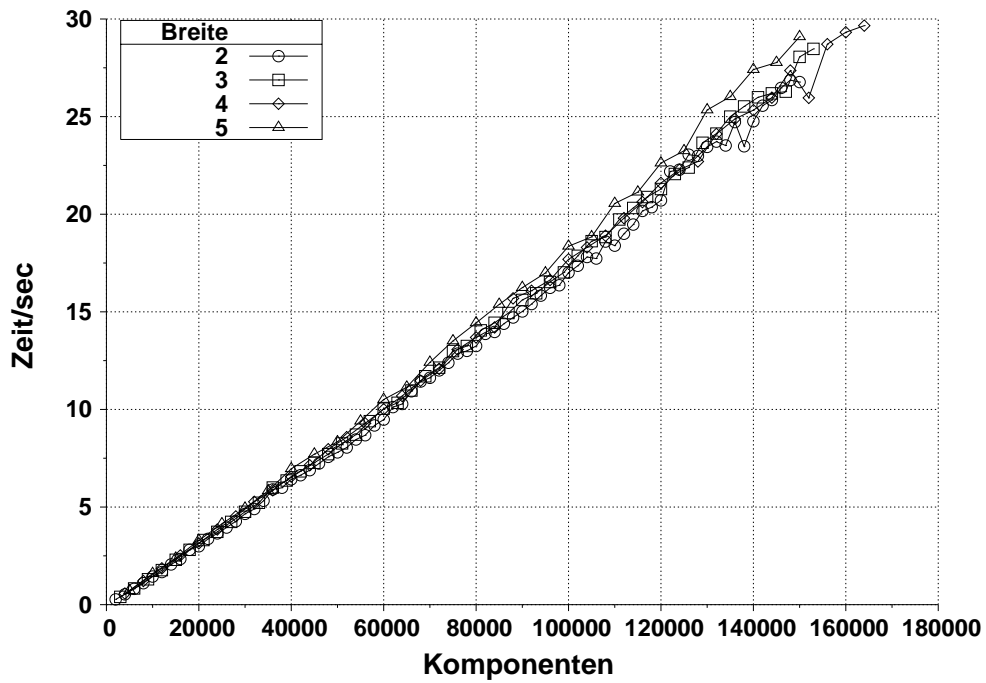


Abbildung 8.10: Zeitverbrauch=F(Komp.) (Rows, Prot.=(sm*e|o)*, ident. Prot.)

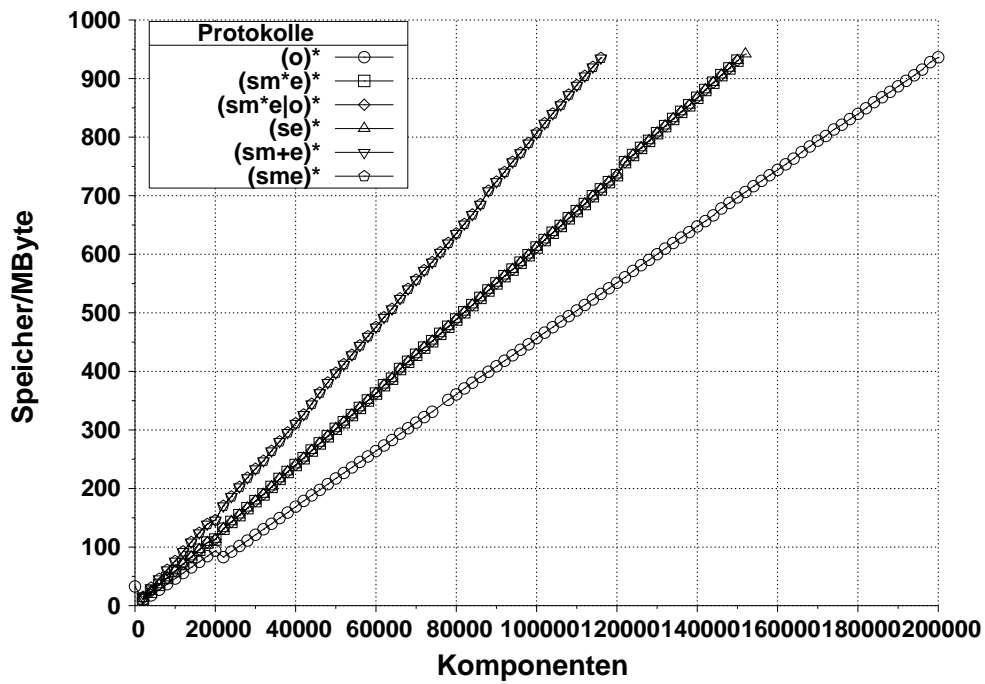


Abbildung 8.11: Speicherverbrauch=F(Komp.) (Rows, Breite = 2, ident. Prot.)

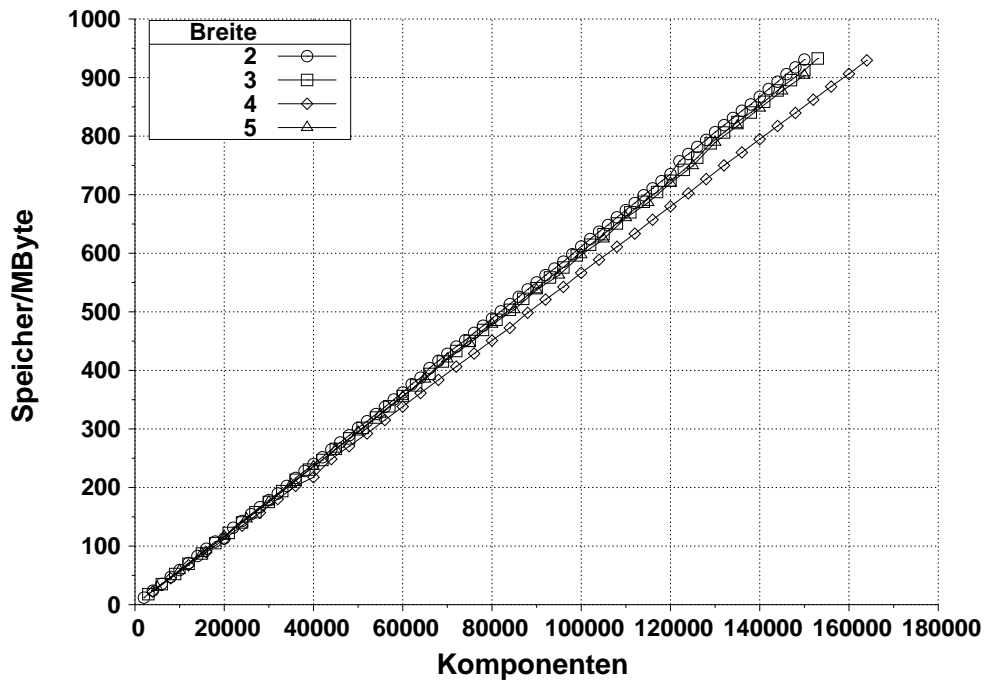


Abbildung 8.12: Speicherverbrauch=F(Komp.) (Rows, Prot.=(sm*e|o)*, ident. Prot.)

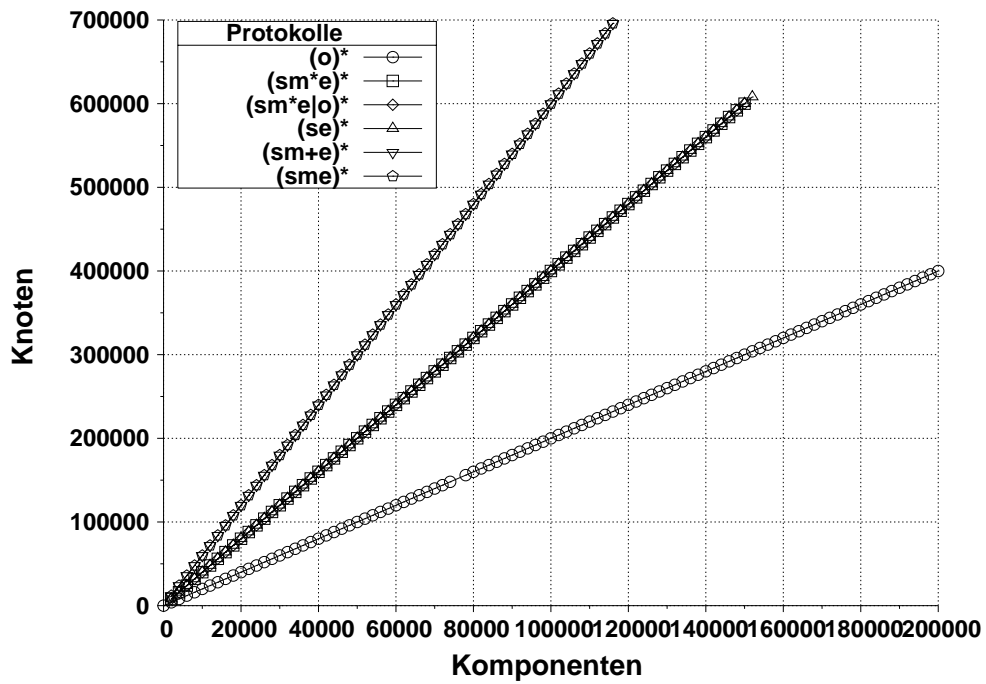


Abbildung 8.13: Knoten = F(Komp.) (Rows, Breite = 2, ident. Prot.)

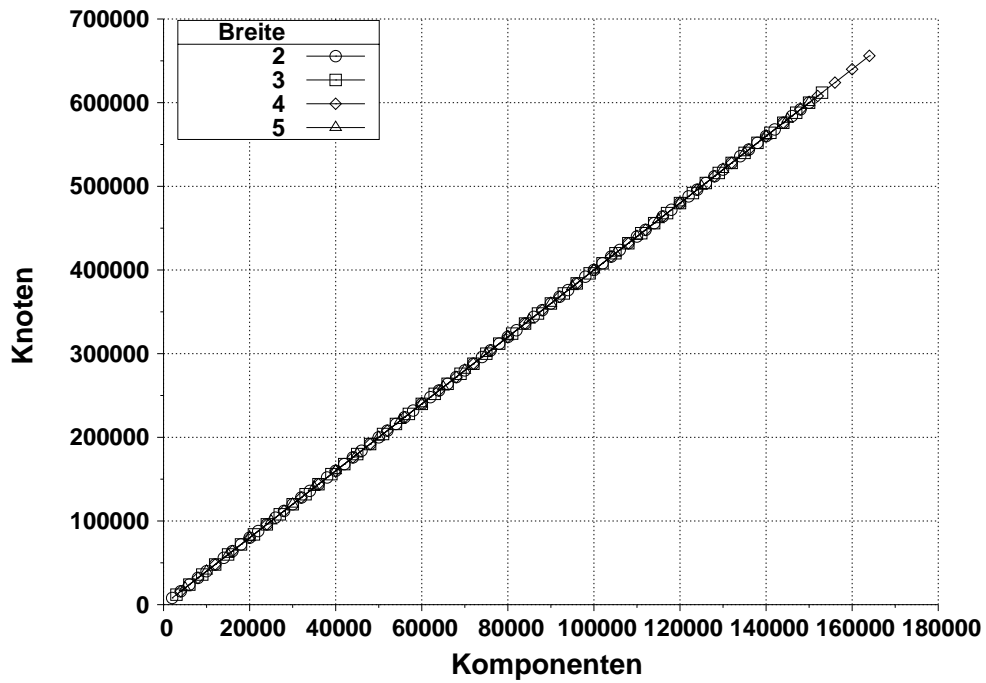


Abbildung 8.14: Knoten = F(Komp.) (Rows, Prot. = (sm*e|o)*, ident. Prot.)

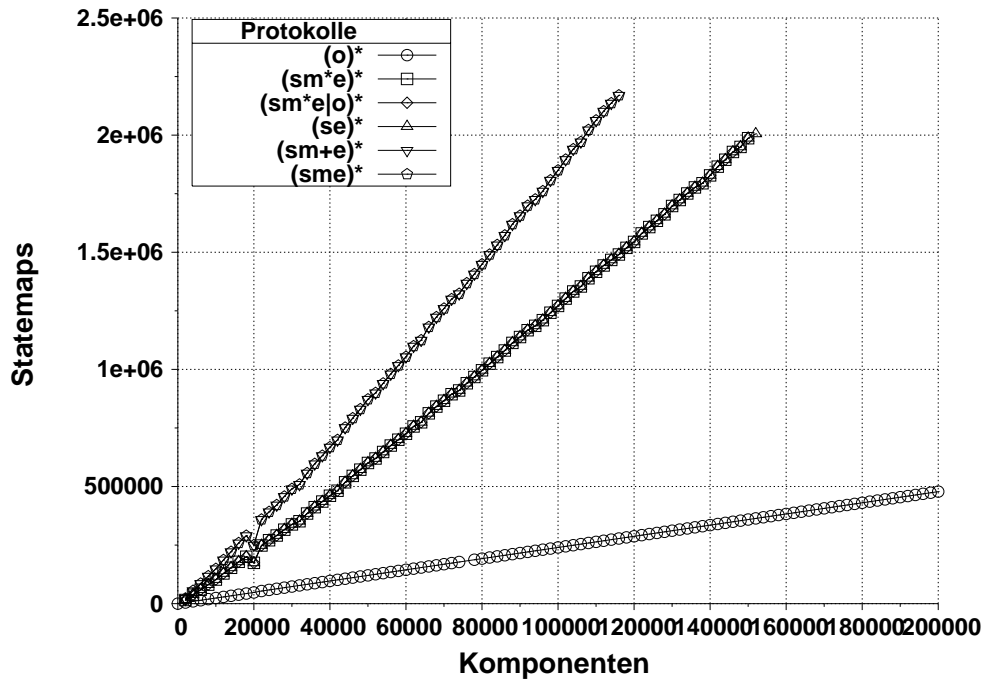


Abbildung 8.15: Statemaps = F(Komp.) (Rows, Breite = 2, ident. Prot.)

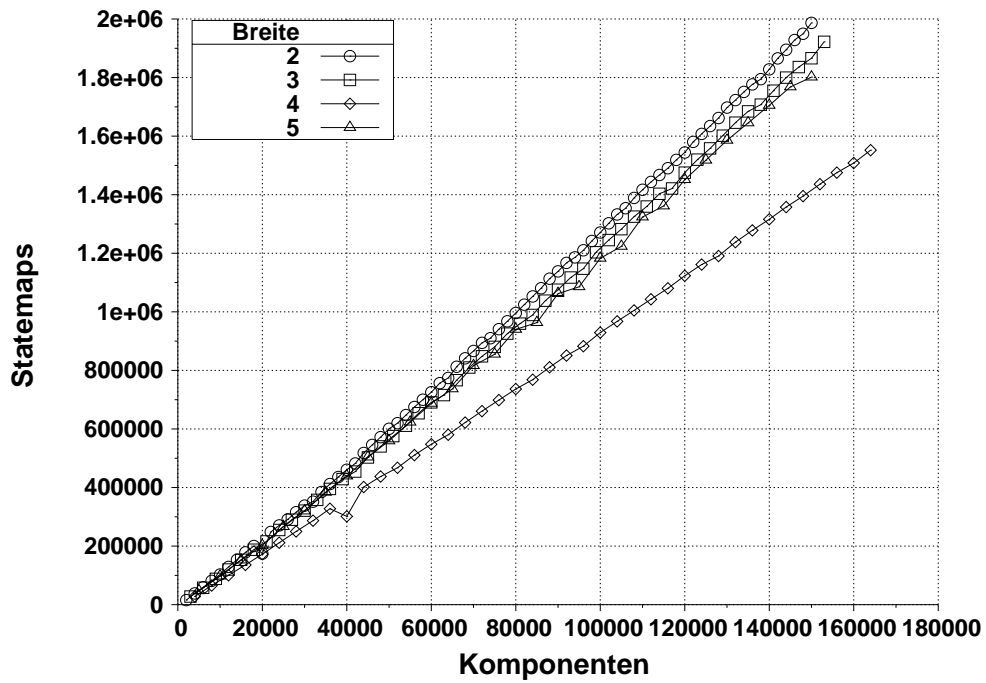


Abbildung 8.16: Statemaps = F(Komp.) (Rows, Protokoll = (sm*e|o)*, ident. Prot.)

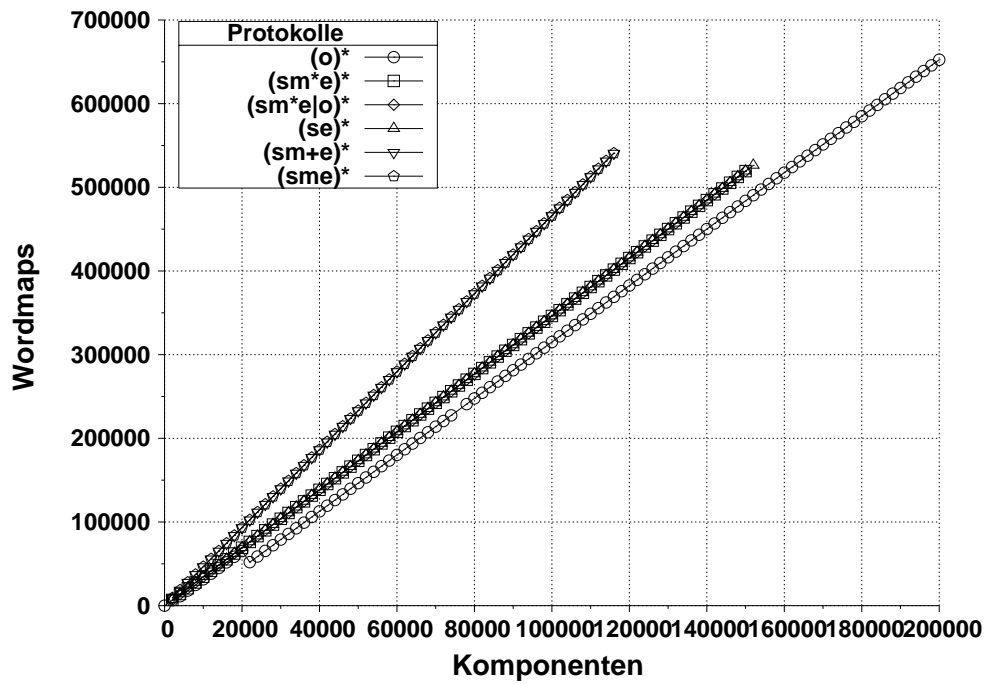


Abbildung 8.17: Wordmaps = F(Komp.) (Rows, Breite = 2, ident. Prot.)

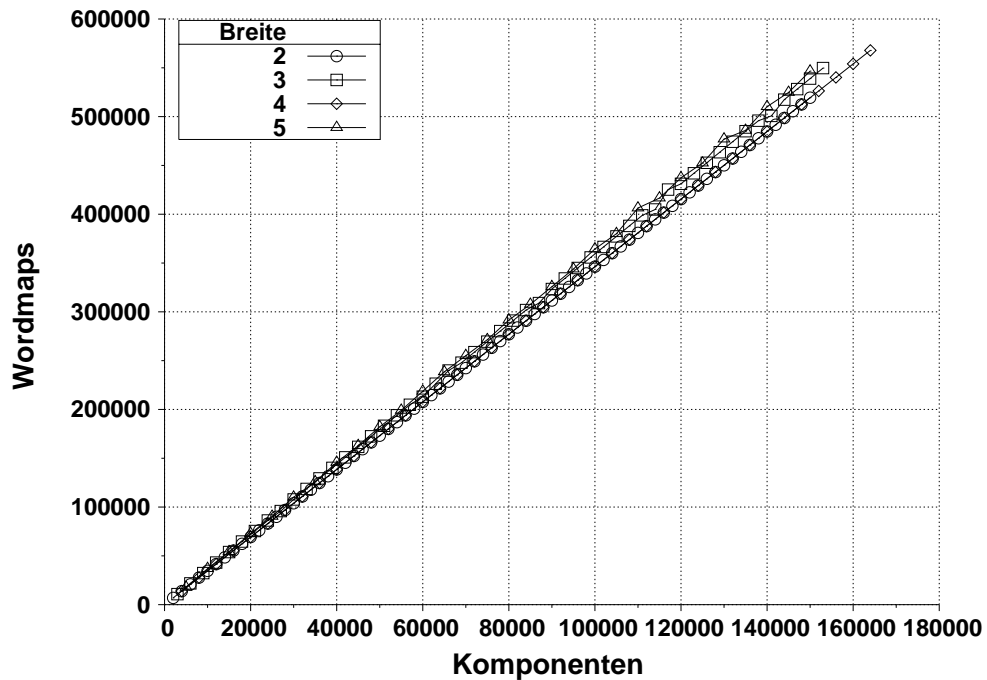


Abbildung 8.18: Wordmaps = F(Komp.) (Rows, Prot. = (sm*e|o)*, ident. Prot.)

Unterschiedliche Kommunikationsprotokolle: Es wurden eine Vielzahl von Tests mit Datenflußkomponenten durchgeführt, die voneinander unterschiedliche Kommunikationsprotokolle realisieren. Dabei kann man unterscheiden, ob KOMPATIBLE beziehungsweise INKOMPATIBLE Kommunikationsprotokolle verwendet wurden. Außerdem wurden jeweils die EINGABEPROTOKOLLE, die INTERNEN KOMMUNIKATIONSPROTOKOLLE beziehungsweise die AUSGABEPROTOKOLLE mit zum restlichen Datenflußgraphen unterschiedlichen Kommunikationsprotokollen versehen. Da sich die Ergebnisse bis auf leichte zeitliche Variationen und jeweils unterschiedlicher maximaler Anzahl von Datenflußkomponenten nicht unterscheiden, werden im folgenden nur die Kurvenverläufe für unterschiedliche interne Kommunikationsprotokolle aufgeführt.

Kompatible Kommunikationsprotokolle: Betrachtet man den Zeitverbrauch in Abhängigkeit von der Anzahl der Datenflußkomponenten mit den angegebenen Kommunikationsprotokollen für Quellen und Senken, so ergibt sich wiederum ein im wesentlichen linearer Zusammenhang (siehe Abbildung 8.19).

Auch die übrigen im Falle der identischen Kommunikationsprotokolle hergeleiteten Aussagen lassen sich hier ableiten. Da die Kurvenverläufe für Speicher, Anzahl Wordmaps, Anzahl Statemaps und Anzahl Knoten sehr ähnlich aussehen, sind diese hier nicht aufgeführt.

Inkompatible Kommunikationsprotokolle: Werden interne Datenflußkomponenten verwendet, deren Kommunikationsprotokolle inkompatibel mit den Kommunikationsprotokollen der Quellen und Senken sind, dann ergeben sich die in Abbildung 8.20 dargestellten Kurvenverläufe. Man erkennt, daß der Zeitverbrauch geringer ist als bei kompatiblen Kommunikationsprotokollen. Dies unterstreicht die Eignung des Model-Checking-Verfahrens für den interaktiven Entwurf, bei dem man vor allem erkennen möchte, ob die zuletzt eingefügte Datenflußkomponente mit den bereits vorhandenen Datenflußkomponenten kompatibel ist.

8.4.1.2 Graphstruktur Net

Dieser Abschnitt behandelt die Untersuchungen zur Graphstruktur Net (vergleiche Abbildung 8.8 (b)). Dabei werden Datenflußgraphen mit identischen Datenflußkomponenten, mit von den übrigen Datenflußkomponenten unterschiedlichen Quellkomponenten, internen Datenflußkomponenten und Senkenkomponenten betrachtet. Außerdem werden sowohl Datenflußkomponenten mit kompatiblen als auch inkompatiblen Kommunikationsprotokollen untersucht.

Identische Kommunikationsprotokolle: In Abbildung 8.21 ist der Zeitverbrauch des Model Checkers für verschiedene Kommunikationsprotokolle in Abhängigkeit von der Anzahl der Datenflußkomponenten visualisiert. Man beobachtet auch hier, daß die Zeit im wesentlichen linear mit der Anzahl der Datenflußkomponenten wächst. Der Zeitverbrauch in Abhängigkeit von der Anzahl der Datenflußkomponenten unter Berücksichtigung verschiedener Breiten der Graphstruktur Net ist für das Kommunikationsprotokoll $(sm^*e|o)^*$ in Abbildung 8.22 dargestellt.

In Abbildung 8.23 ist der Speicherverbrauch des Model Checkers für die Graphstruktur Net für verschiedene Kommunikationsprotokolle in Abhängigkeit von der Anzahl der Datenflußkomponenten aufgezeigt. Der Speicherverbrauch wächst linear mit der Anzahl der Datenflußkomponenten. Bis zu etwa 140000 Datenflußkomponenten können erfolgreich simuliert werden.

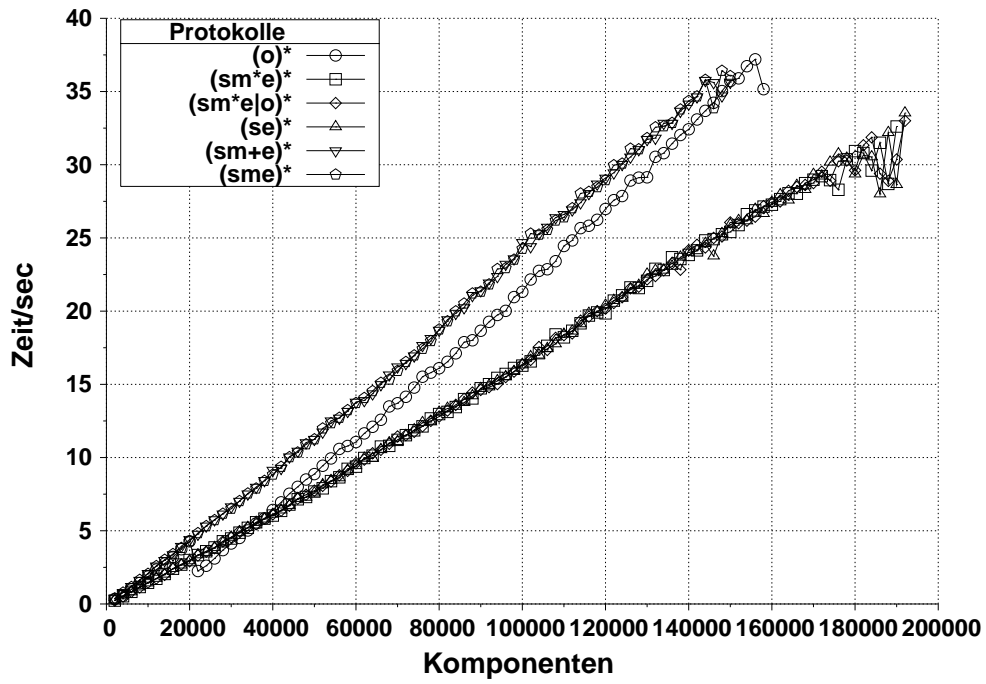


Abbildung 8.19: Zeitverbrauch=F(Komp.) (Rows, Breite = 2, komp. Prot.)

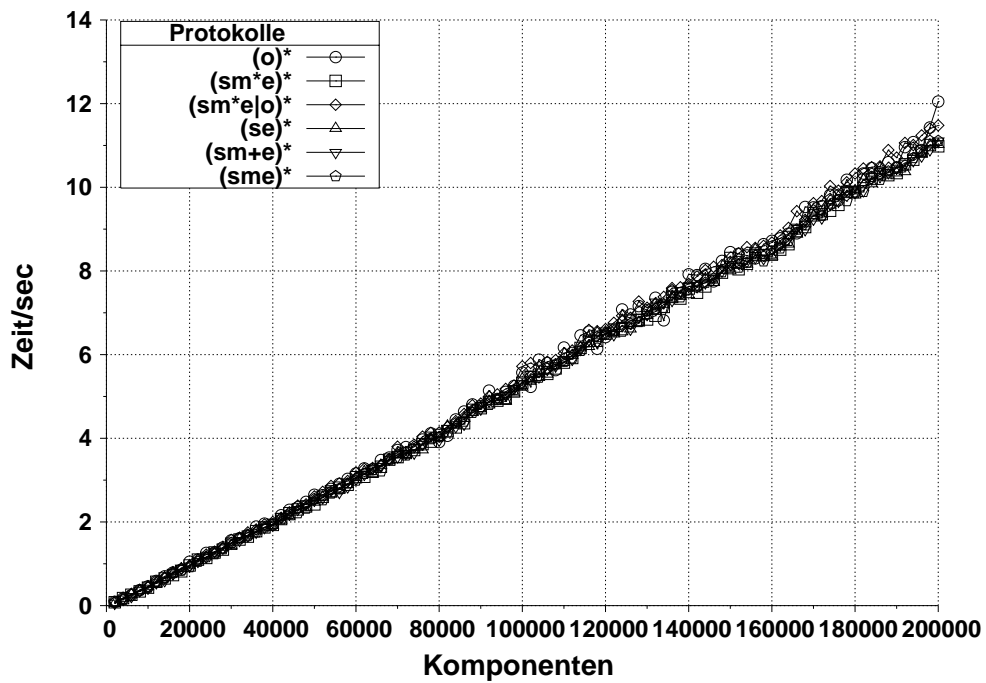


Abbildung 8.20: Zeitverbrauch=F(Komp.) (Rows, Breite = 2, inkomp. Prot.)

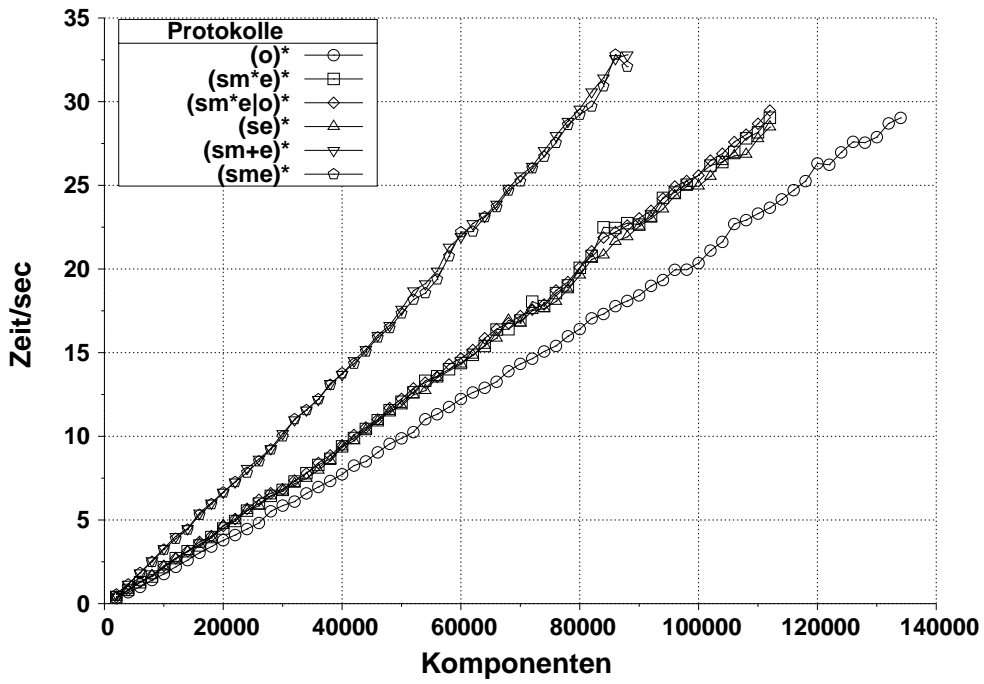


Abbildung 8.21: Zeitverbrauch = F(Komp.) (Net, Breite = 2, ident. Prot.)

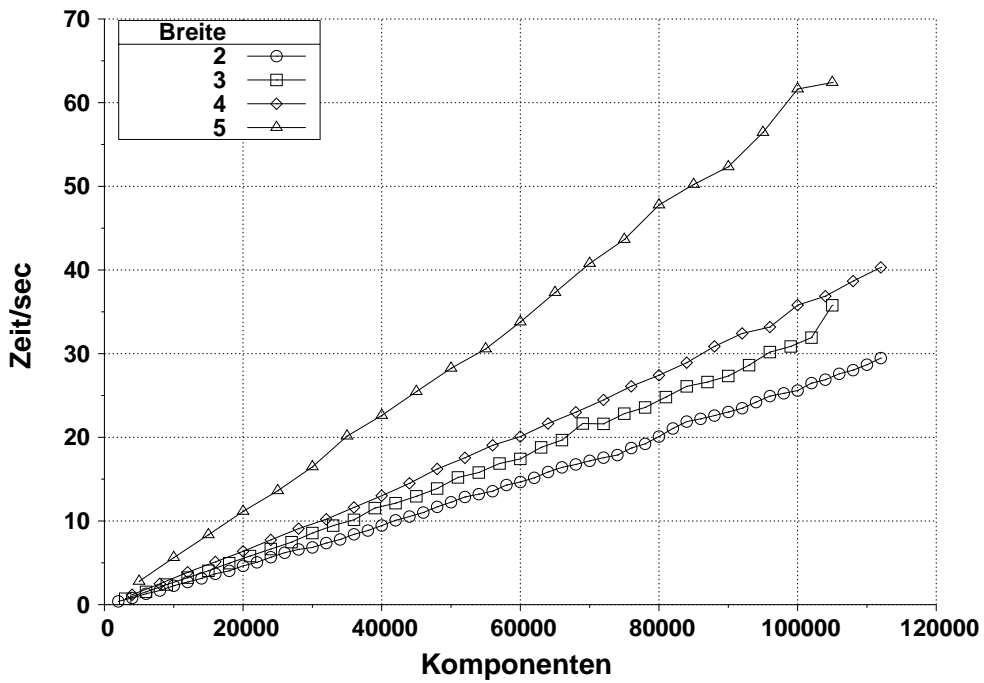


Abbildung 8.22: Zeitverbrauch = F(Komp.) (Net, Prot. = (sm*e|o)*, ident. Prot.)

Betrachtet man den Speicherverbrauch in Abhängigkeit von der Anzahl der Datenflußkomponenten bei variierter Breite und gleichbleibendem Kommunikationsprotokoll ($sm^*e|o$)*, so erkennt man auch hier, daß der Speicherverbrauch im wesentlichen von der Anzahl der Datenflußkomponenten und nicht von der Breite abhängt (vergleiche Abbildung 8.24). Letzteres resultiert aus der Effizienz der in Abschnitt 6.4 beschriebenen PARTIAL ORDER REDUCTION.

Die Anzahl der Knoten wächst linear mit der Anzahl der Datenflußkomponenten (vergleiche Abbildung 8.25). Die Breite spielt dabei eine untergeordnete Rolle (siehe Abbildung 8.26).

In den Abbildungen 8.27 und 8.28 ist jeweils die Anzahl der Statemaps in Abhängigkeit von der Anzahl der Datenflußkomponenten aufgetragen. Dabei beobachtet man, daß die Anzahl der Statemaps linear mit der Anzahl der Datenflußkomponenten wächst.

Die Kurvenverläufe für die Wordmaps sind ähnlich wie die Kurven zu den Statemaps. Die Abbildung 8.29 zeigt die benötigten Wordmaps in Abhängigkeit von der Anzahl der Datenflußkomponenten für verschiedene Kommunikationsprotokolle. Man erkennt die lineare Abhängigkeit zwischen der Anzahl der Datenflußkomponenten und der Anzahl der benötigten Wordmaps.

In Abbildung 8.30 ist die Anzahl der Wordmaps in Abhängigkeit von der Anzahl der Datenflußkomponenten für verschieden breite Graphstrukturen aufgetragen. Auch hier ist die Abhängigkeit der Wordmaps von der Anzahl der Datenflußkomponenten linear. Der Einfluß des Parameters Breite ist dabei vergleichsweise gering.

Unterschiedliche Kommunikationsprotokolle: Im folgenden wird die Verwendung von internen Datenflußkomponenten untersucht, deren Kommunikationsprotokolle sich von den Kommunikationsprotokollen der Quellen und Senken unterscheiden. Es wurden auch hier Untersuchungen mit vom Rest des Datenflußgraphen verschiedenen Quellen- und Senkenkomponenten durchgeführt. Da diese Ergebnisse sich aber nicht schwerwiegend voneinander abweichen, wurden nur die oben genannten Resultate in diese Arbeit aufgenommen.

Kompatible Kommunikationsprotokolle: Betrachtet man für diesen Fall Datenflußgraphen mit kompatiblen Kommunikationsprotokollen, so erkennt man ein lineares Wachstum (vergleiche Abbildung 8.31).

Inkompatible Kommunikationsprotokolle: Auch bei der Verwendung inkompatibler Kommunikationsprotokolle wird das Nichtvorhandensein eines Zyklus im Erreichbarkeitsgraphen in vergleichsweise kurzer Zeit erkannt (siehe Abbildung 8.32).

8.4.1.3 Graphstruktur Rows mit Rückkopplungen

Dieser Abschnitt behandelt die Untersuchungen zur Graphstruktur Rows mit Rückkopplungen (RowsF; vergleiche Abbildung 8.8 (c)). Dabei werden Datenflußgraphen mit identischen Datenflußkomponenten, mit von den übrigen Datenflußkomponenten unterschiedlichen Quellkomponenten, internen Datenflußkomponenten und Senkenkomponenten betrachtet. Außerdem werden sowohl Datenflußkomponenten mit kompatiblen als auch inkompatiblen Kommunikationsprotokollen untersucht.

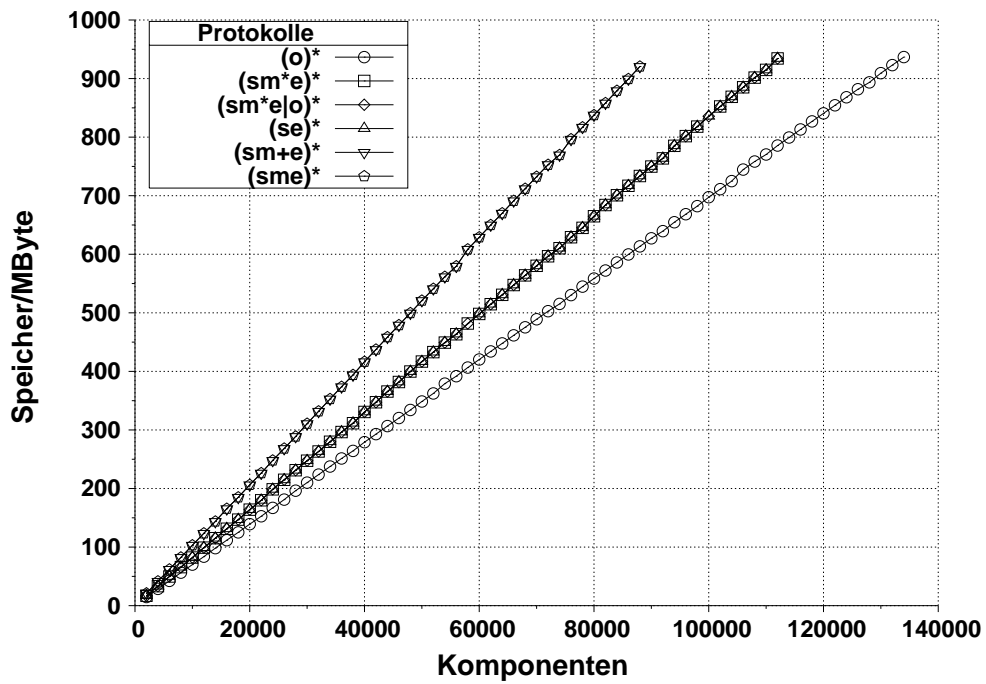


Abbildung 8.23: Speicherverbrauch=F(Komp.) (Net, Breite = 2, ident. Prot.)

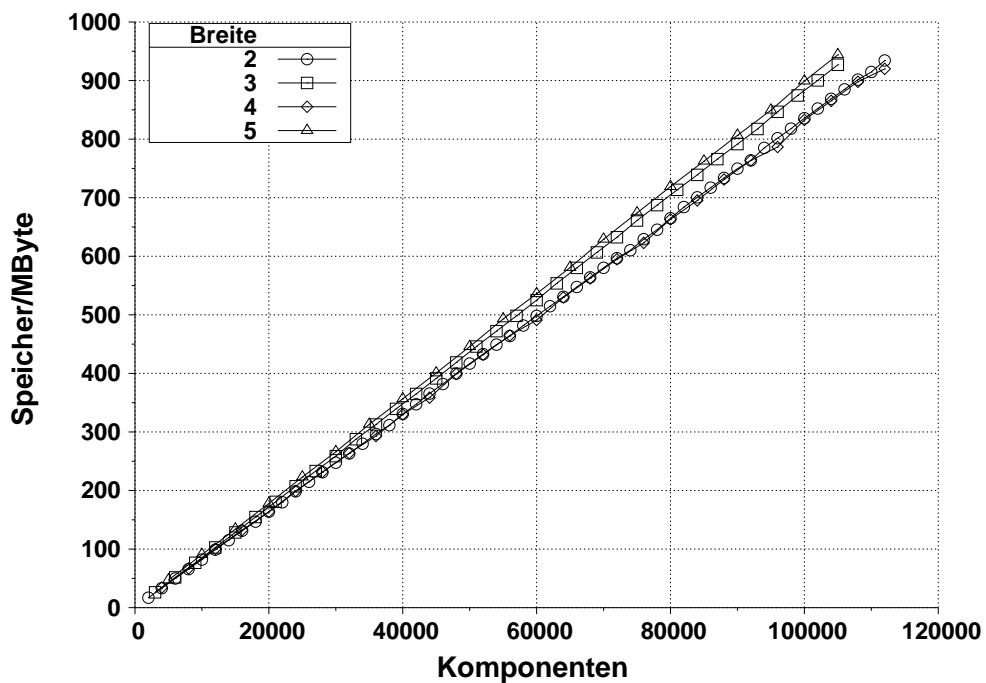


Abbildung 8.24: Speicherverbrauch=F(Komp.) (Net, Prot.=(sm*e|o)*, ident. Prot.)

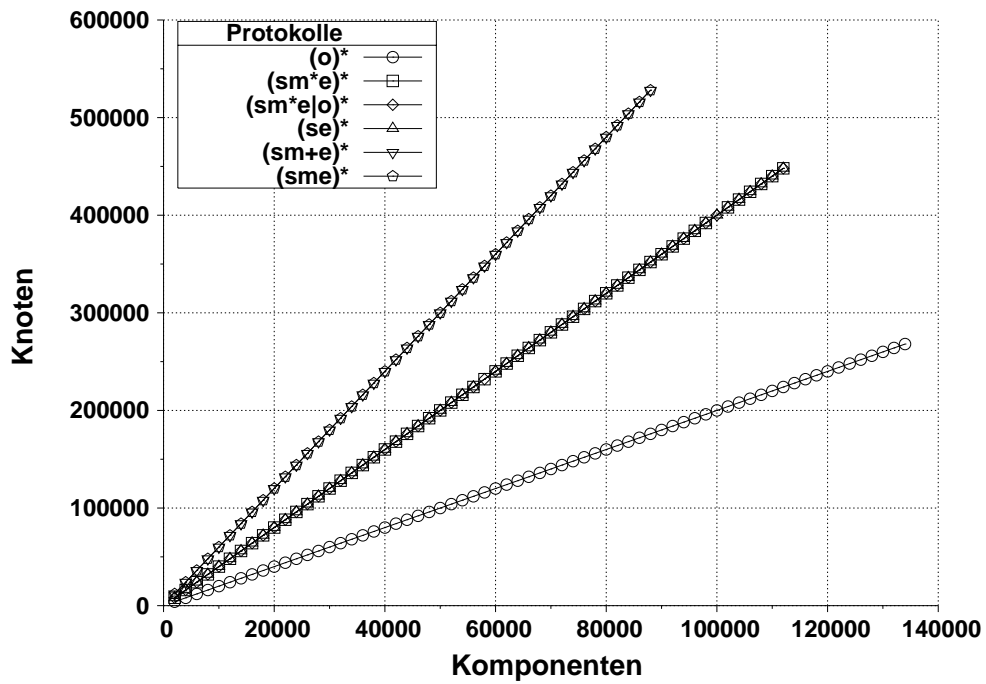


Abbildung 8.25: Knoten = F(Komp.) (Net, Breite = 2, ident. Prot.)

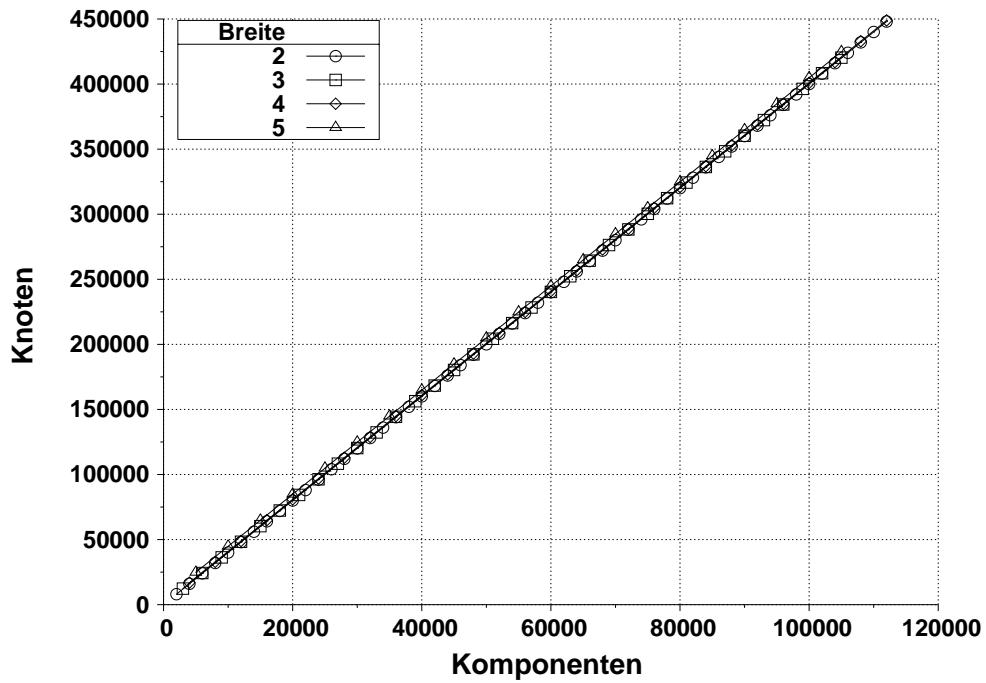


Abbildung 8.26: Knoten = F(Komp.) (Net, Prot. = (sm*e|o)*, ident. Prot.)

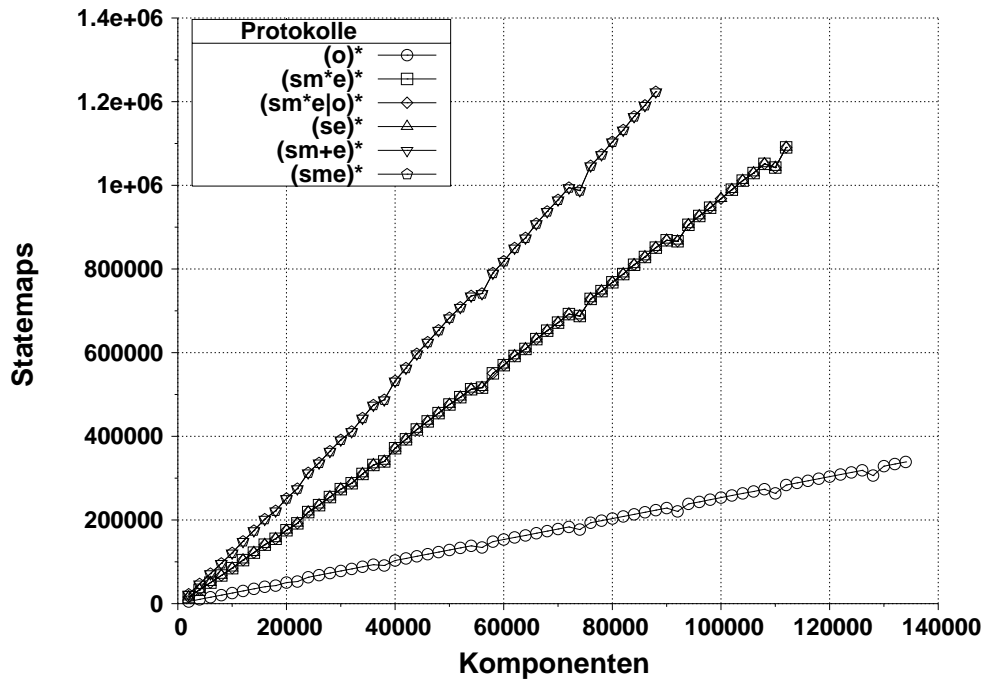


Abbildung 8.27: Statemaps = F(Komp.) (Net, Breite = 2, ident. Prot.)

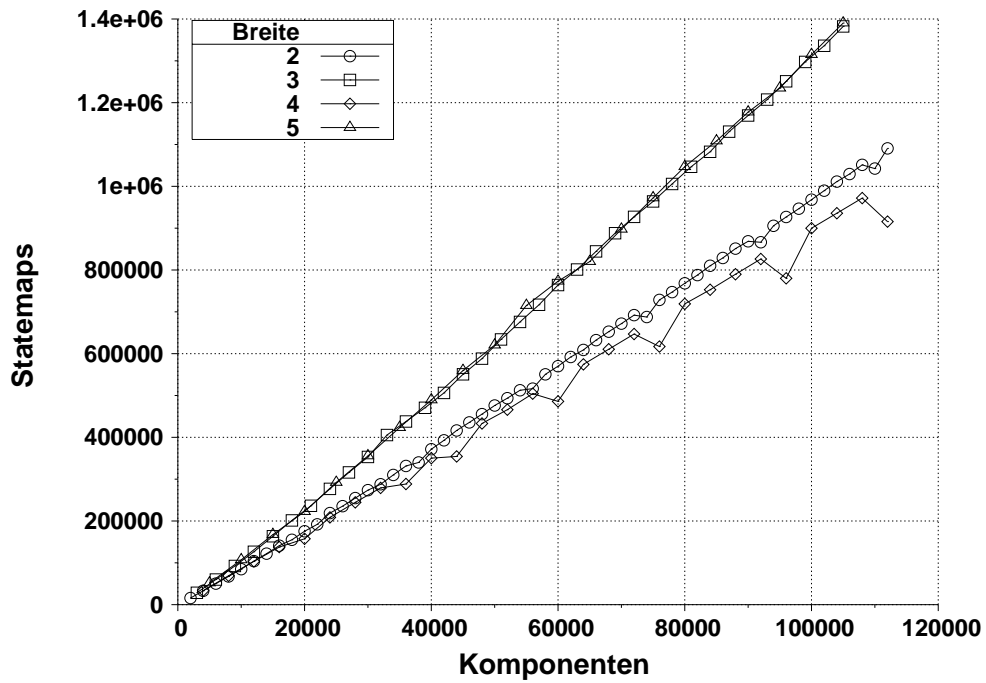


Abbildung 8.28: Statemaps = F(Komp.) (Net, Prot. = (sm*e|o)*, ident. Prot.)

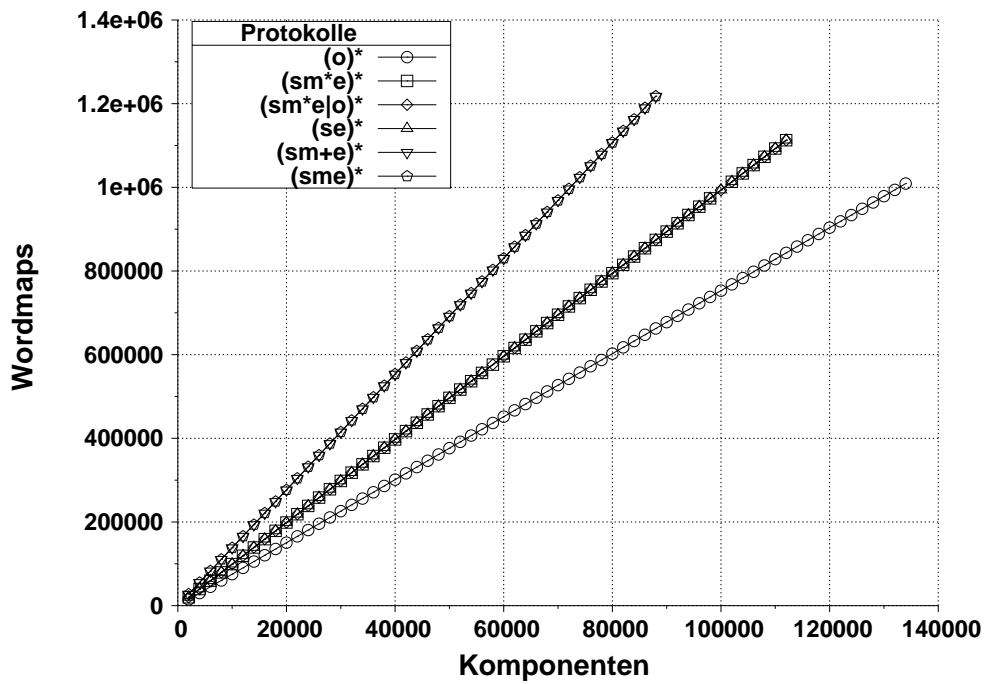


Abbildung 8.29: Wordmaps=F(Komp.) (Net, Breite = 2, ident. Prot.)

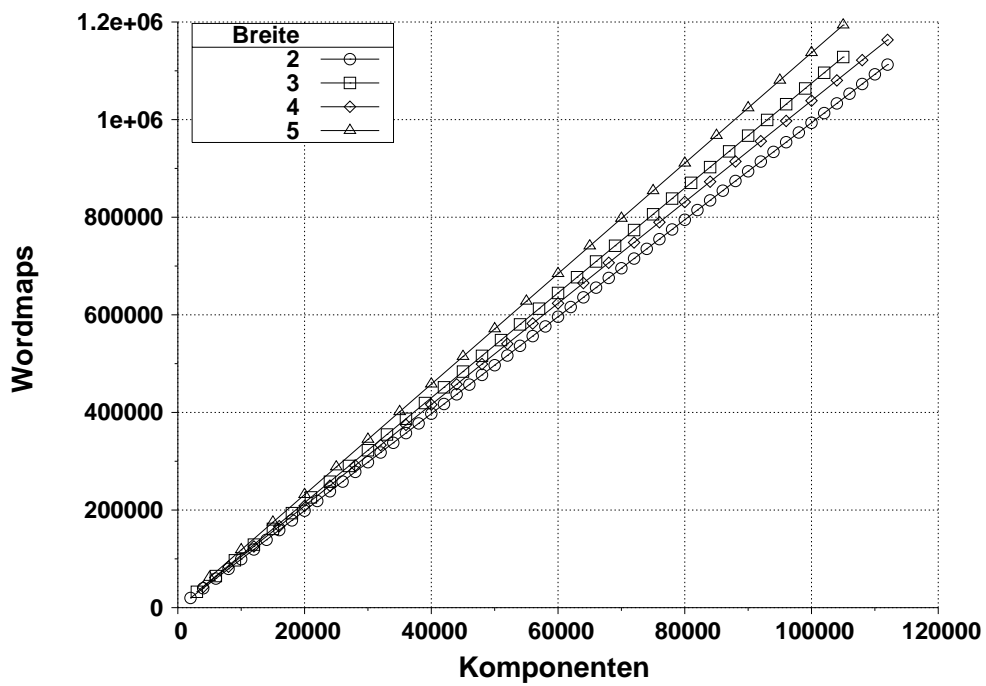


Abbildung 8.30: Wordmaps=F(Komp.) (Net, Prot.=(sm*e|o)*, ident. Prot.)

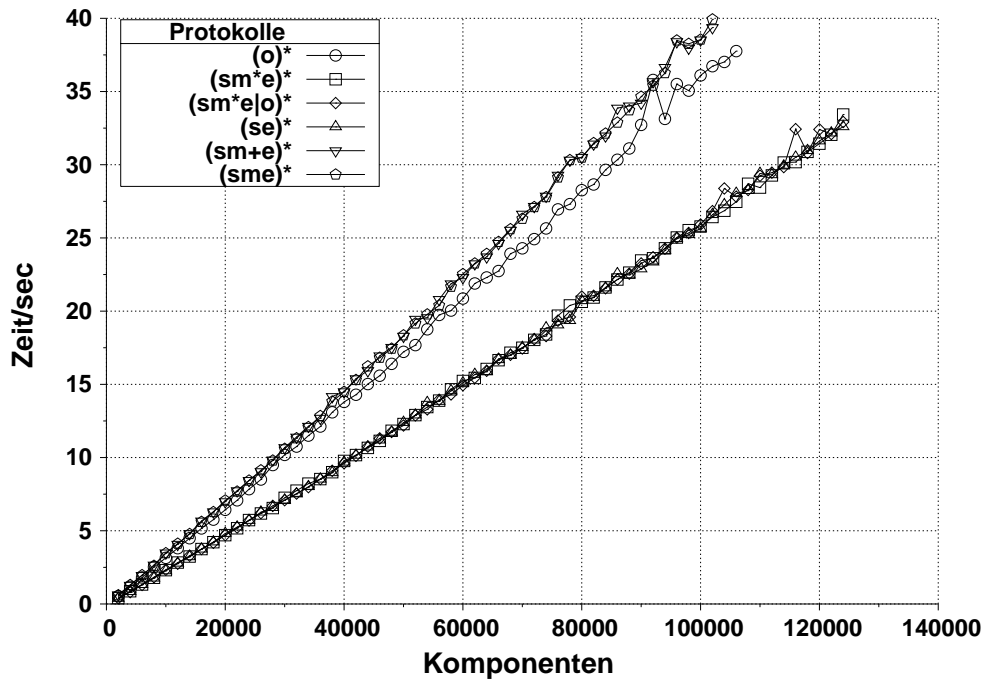


Abbildung 8.31: Zeitverbrauch=F(Komp.) (Net, Breite = 2, komp. Prot.)

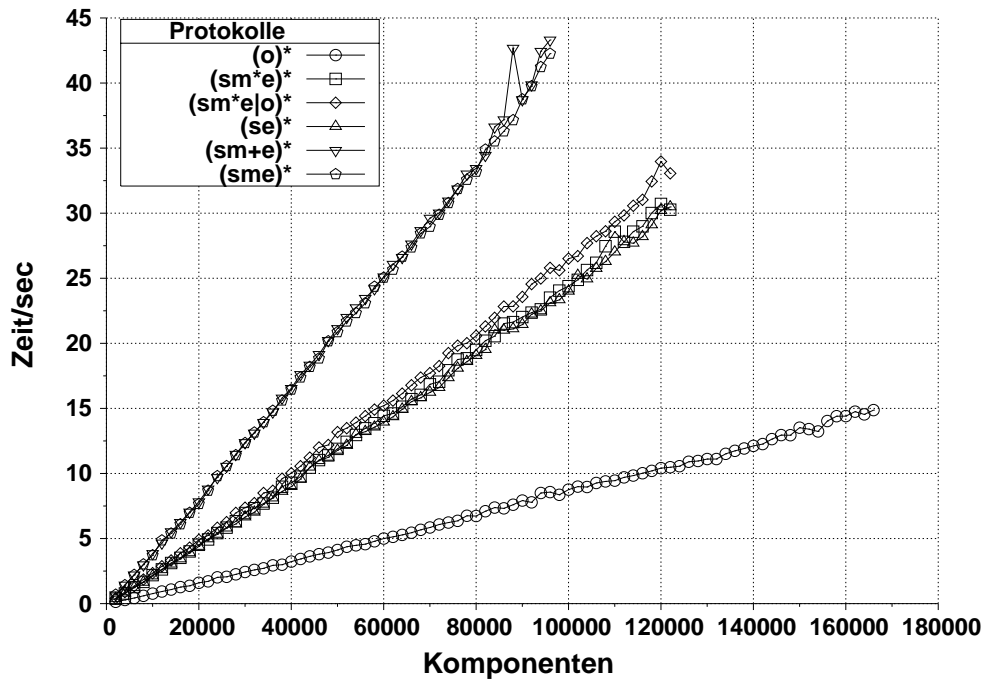


Abbildung 8.32: Zeitverbrauch=F(Komp.) (Net, Breite = 2, inkomp. Prot.)

Identische Kommunikationsprotokolle: Betrachtet man den Verlauf des Zeitverbrauchs in Abbildung 8.33 im Vergleich zu dem Fall ROWS ohne Rückkopplungen, so beobachtet man, daß der Zeitverbrauch um beinahe die Hälfte gesunken ist. Da die Kurvenverläufe bezüglich Speicherverbrauch, Anzahl der Knoten des Erreichbarkeitsgraphen und so fort ähnlich verlaufen wie bei der Graphstruktur ROWS, sind diese im folgenden nicht aufgeführt.

Unterschiedliche Kommunikationsprotokolle: Bei den zahlreichen Versuchen zu der Graphstruktur ROWSF mit Rückkopplungen wurden Untersuchungen sowohl mit vom Rest des Datenflußgraphen verschiedenen Quellenkomponenten, internen Datenflußkomponenten als auch Senkenkomponenten durchgeführt. Da sich die Ergebnisse nicht schwerwiegend unterscheiden, werden im folgenden nur die Resultate für Datenflußgraphen mit unterschiedlichen internen Kommunikationsprotokollen wiedergegeben.

Kompatible Kommunikationsprotokolle: Es ist ein beinahe linearer Zusammenhang zwischen der Anzahl der Datenflußkomponenten und der für die Zyklensuche benötigten Zeit zu erkennen (siehe Abbildung 8.34).

Inkompatible Kommunikationsprotokolle: Die zum Erkennen des Nichtvorhandenseins eines Zyklus notwendige Zeit steigt linear mit der Anzahl der Datenflußkomponenten unabhängig von den in den Quellen und Senken verwendeten Kommunikationsprotokollen (vergleiche Abbildung 8.35).

8.4.1.4 Graphstruktur Net mit Rückkopplungen

Dieser Abschnitt behandelt die Untersuchungen zur Graphstruktur Net mit Rückkopplungen (NetF; vergleiche Abbildung 8.8 (d)). Dabei werden Datenflußgraphen mit identischen Datenflußkomponenten, mit von den übrigen Datenflußkomponenten unterschiedlichen Quellkomponenten, internen Datenflußkomponenten und Senkenkomponenten betrachtet. Außerdem werden sowohl Datenflußkomponenten mit kompatiblen als auch inkompatiblen Kommunikationsprotokollen untersucht.

Identische Kommunikationsprotokolle: Bei den Untersuchungen identischer Kommunikationsprotokolle traten bei der Graphstruktur NetF ähnliche Kurvenverläufe wie bei der Graphstruktur Net zu Tage. Daher wird hier nur die beinahe lineare Abhängigkeit des Zeitverbrauchs von der Anzahl der Datenflußkomponenten (siehe Abbildung 8.36) veranschaulicht.

Unterschiedliche Kommunikationsprotokolle: Wie bei den vorausgegangenen Untersuchungen, wurde auch hier auf die Wiedergabe der Ergebnisse für vom Rest des Datenflußgraphen abweichende Eingabe- beziehungsweise Ausgabeprotokolle verzichtet. Es werden nur die Ergebnisse für sich unterscheidende interne Kommunikationsprotokolle dargestellt.

Kompatible Kommunikationsprotokolle: Abbildung 8.37 zeigt die lineare Abhängigkeit der benötigten Zeit von der Anzahl der Datenflußkomponenten.

Inkompatible Kommunikationsprotokolle: In Abbildung 8.38 erkennt man die lineare Abhängigkeit der Zeit von der Anzahl der Datenflußkomponenten.

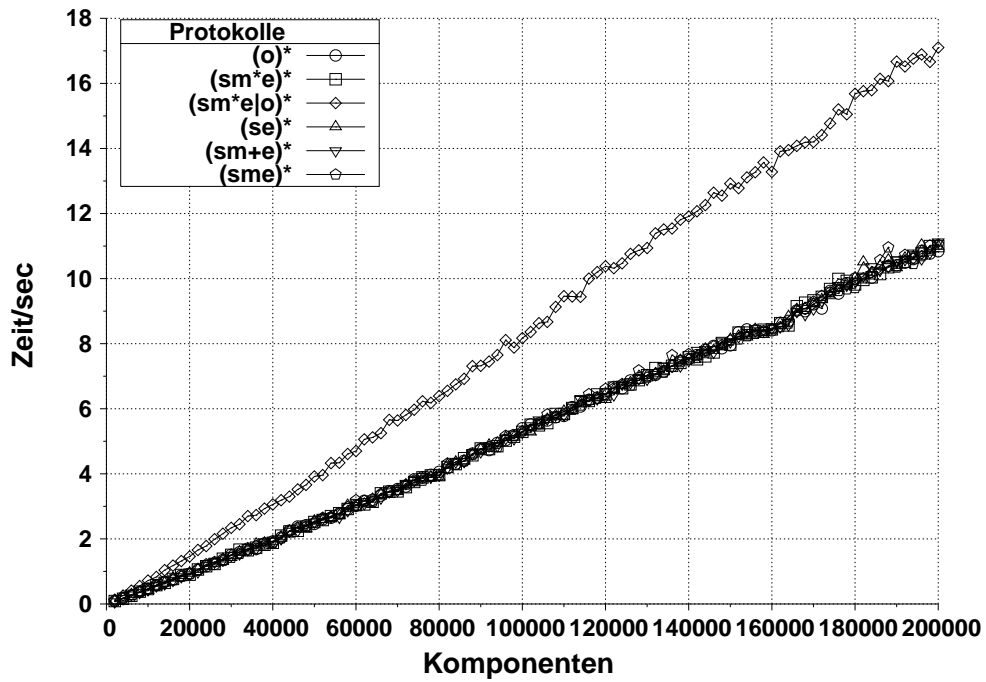


Abbildung 8.33: Zeitverbrauch=F(Komp.) (RowsF, Breite = 2, ident. Prot.)

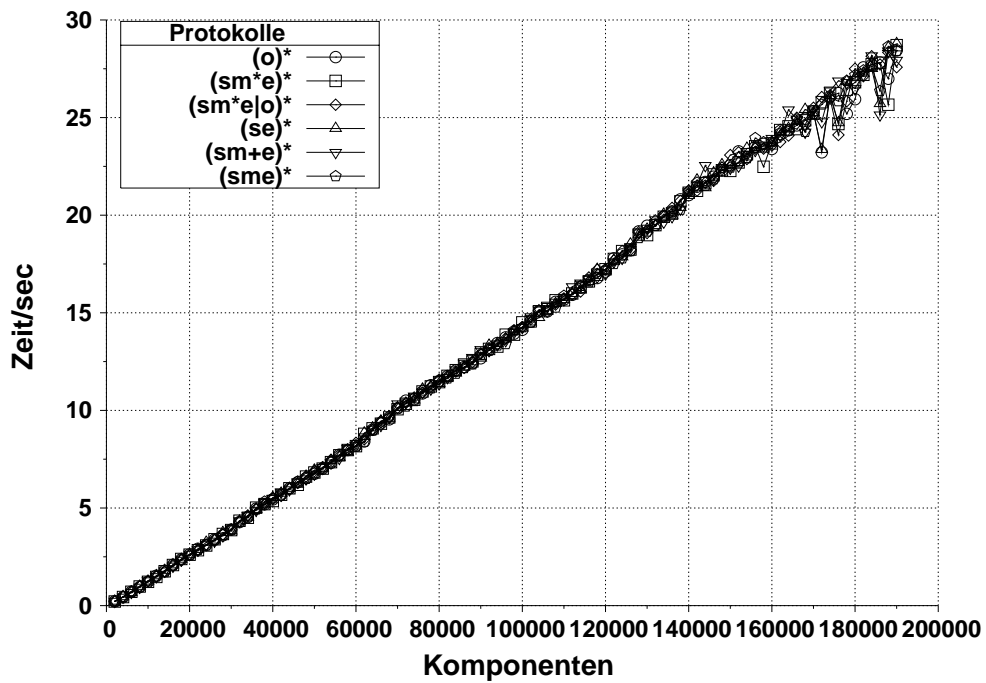


Abbildung 8.34: Zeitverbrauch=F(Komp.) (RowsF, Breite = 2, komp. Prot.)

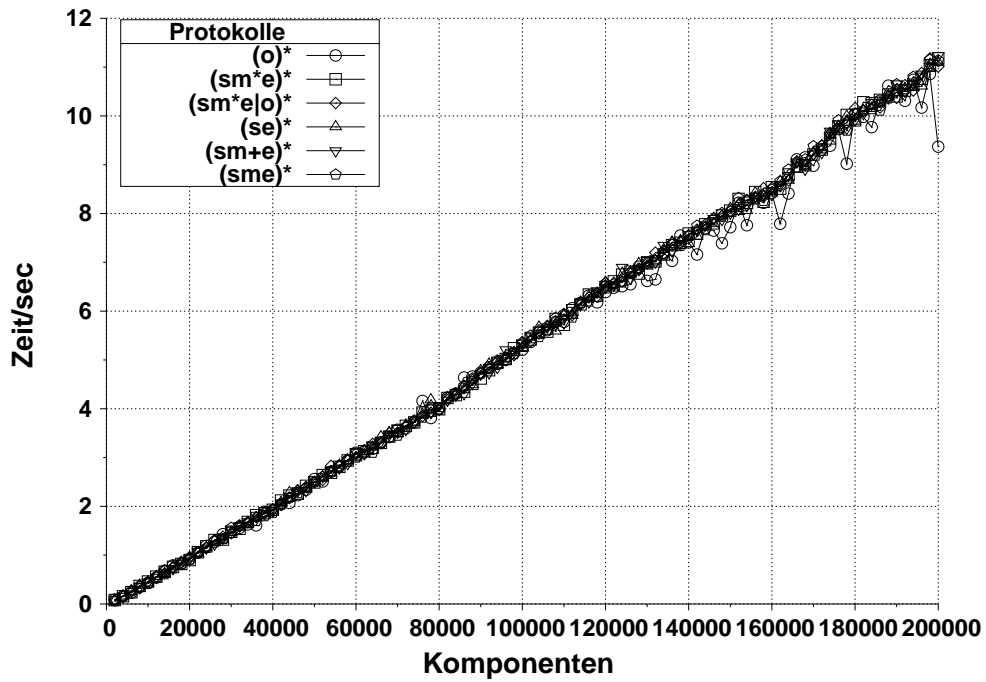


Abbildung 8.35: Zeitverbrauch=F(Komp.) (RowsF, Breite = 2, inkomp. Prot.)

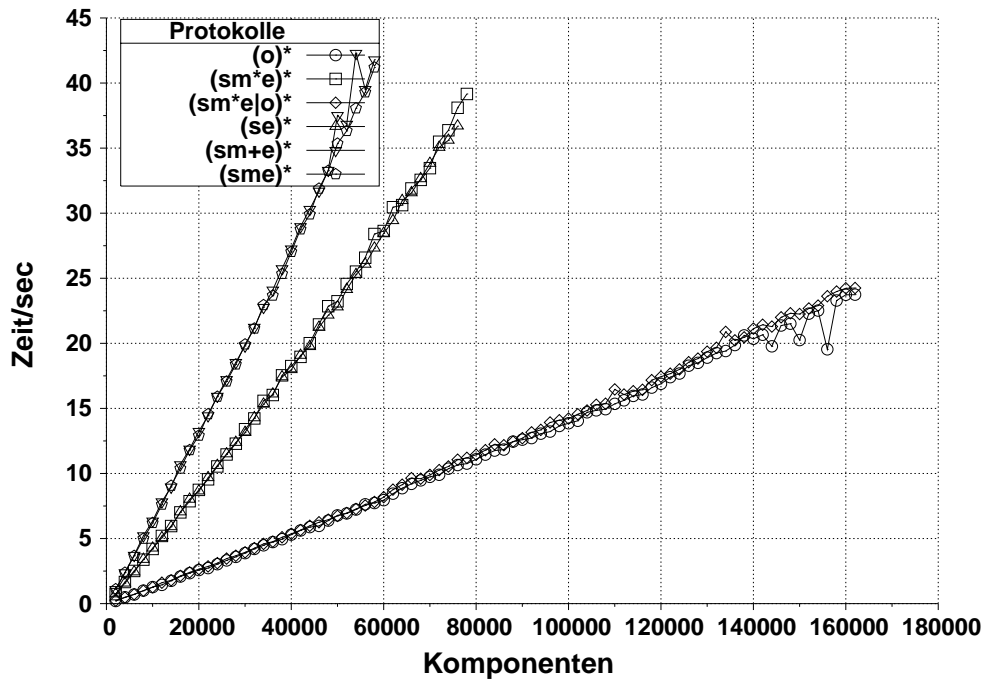


Abbildung 8.36: Zeitverbrauch=F(Komp.) (NetF, Breite = 2, ident. Prot.)

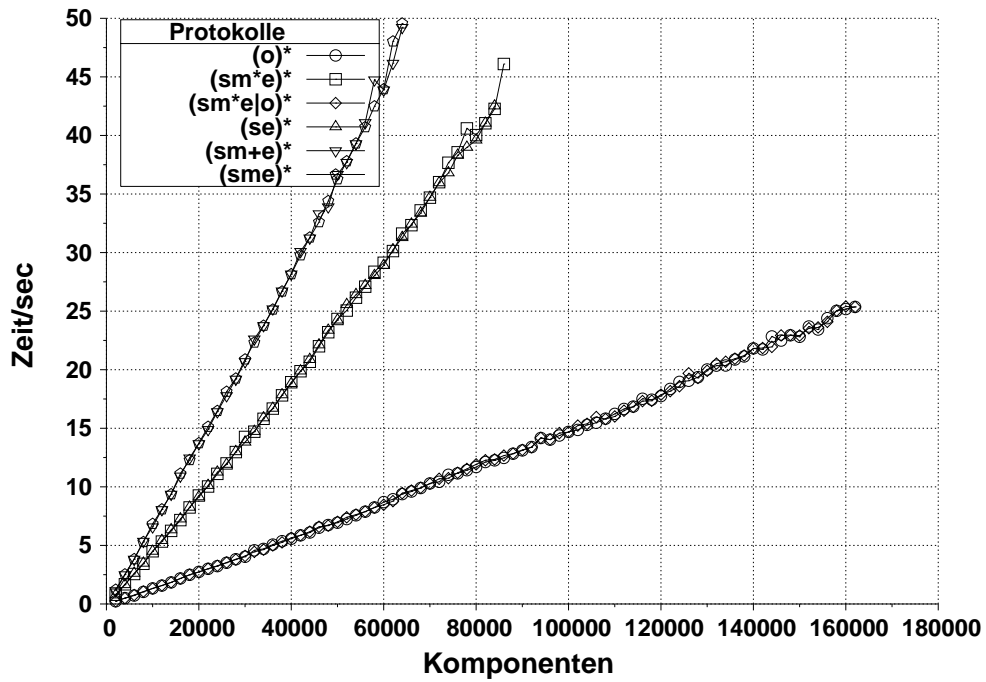


Abbildung 8.37: Zeitverbrauch=F(Komp.) (NetF, Breite = 2, komp. Prot.)

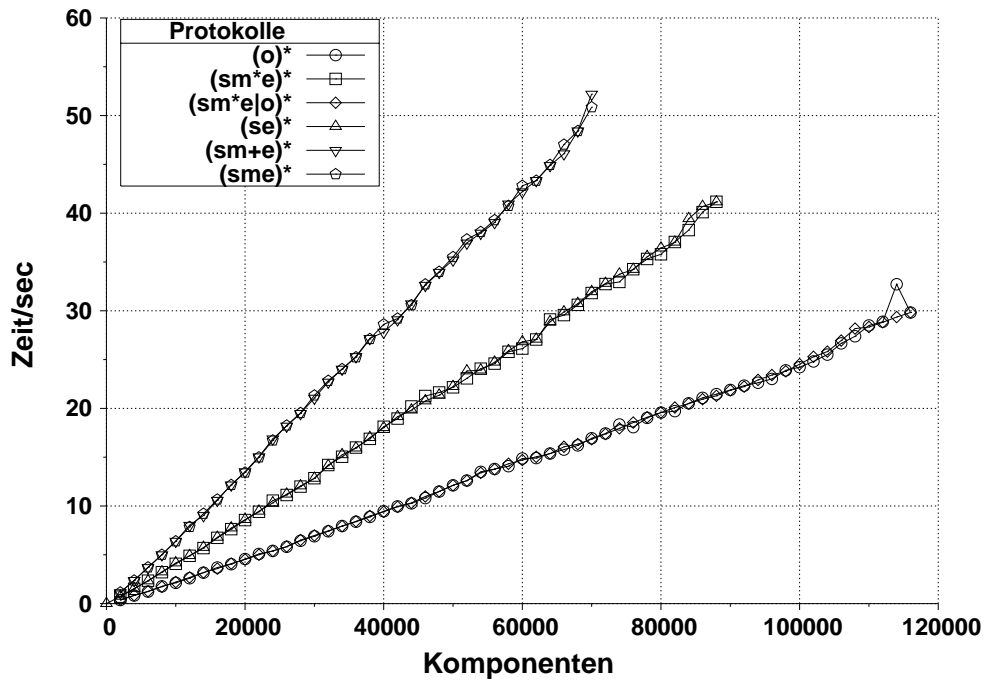


Abbildung 8.38: Zeitverbrauch=F(Komp.) (NetF, Breite = 2, inkomp. Prot.)

8.4.1.5 Graphstruktur Random

Dieser Abschnitt behandelt die Untersuchungen zur Graphstruktur Random (vergleiche Abbildung 8.8 (e)). Diese Graphstruktur soll die Struktur realer Datenflußgraphen möglichst gut nachbilden und es so ermöglichen, die Eignung des in dieser Arbeit entwickelten Model-Checking-Verfahrens für den alltäglichen Einsatz zu testen.

Abbildung 8.39 zeigt für unterschiedlich breite Datenflußgraphen mit zufällig erzeugter Struktur die für die Bestimmung eines Zyklus benötigten Zeiten. Man erkennt, daß in diesem Fall ein nichtlineares Wachstum vorliegt, da je nach Struktur unterschiedlich lange benötigt wird, einen Zyklus im Erreichbarkeitsgraphen zu finden. Dennoch ist ersichtlich, daß selbst für große Datenflußgraphen die Analyse in vergleichsweise kurzer Zeit abgeschlossen ist.

8.4.1.6 Weitere Untersuchungen

Der Einfluß einer Reihe weiterer Parameter, die beispielsweise die Verwendung unterschiedlicher Gewichte steuern oder diverse Einstellungen im Suchalgorithmus beeinflussen, sind im folgenden dargestellt. So kann man zum Beispiel den BLOCKIER/DEBLOCKIER-Mechanismus ein- beziehungsweise ausschalten. Außerdem ist es möglich, die PARTIAL ORDER REDUCTION zu aktivieren oder zu deaktivieren. Der Model Checker ist in der Lage, nach nur einem Zyklus im Erreichbarkeitsgraphen, welcher den Wurzelknoten beinhaltet, beziehungsweise nach allen solchen Zyklen zu suchen. Zudem ist eine Gegenüberstellung der beiden alternativ verwendeten Datenstrukturen Incrementalmaps beziehungsweise Ordered Maps interessant.

Kantengewichte: In den bisherigen Tests waren alle Kantengewichte auf 1 gesetzt. In diesem Abschnitt wird untersucht, wie sich eine zufällige Bestimmung dieser Gewichte auf das Laufzeitverhalten des Model Checkers auswirkt. Abbildung 8.40 zeigt den Zeitverbrauch für einen Datenflußgraphen mit Graphstruktur ROWS, Breite 2 und Tiefe = 10000. Dabei wurde die Wahrscheinlichkeit, daß ein Kantengewicht größer 1 ist, von 0 jeweils um den Wert 1 auf 100 erhöht. Dabei zeigt sich, daß abhängig von der Art der verwendeten Kommunikationsprotokolle Schwankungen im Zeitverbrauch auftreten.

Blockieren/Deblockieren: In diesem Abschnitt wird der Einfluß des in dieser Arbeit entwickelten BLOCKIER/DEBLOCKIER-Mechanismus im Falle kompatibler beziehungsweise inkompatibler Kommunikationsprotokolle untersucht.

Kompatible Kommunikationsprotokolle: Abbildung 8.41 zeigt den Zeitverbrauch in Abhängigkeit von der Anzahl der Datenflußkomponenten für kompatible Kommunikationsprotokolle. Dabei wurde der Blockier/Deblockier-Mechanismus aktiviert beziehungsweise deaktiviert. Mit eingeschalteter PARTIAL ORDER REDUCTION ist kein Unterschied festzustellen.

Inkompatible Kommunikationsprotokolle: Zur Verdeutlichung des enormen Nutzens dieses BLOCKIER/DEBLOCKIER-Mechanismus wird das in Abbildung 6.5 dargestellte Beispiel aus Kapitel 6 noch einmal aufgegriffen, mit welchem die Notwendigkeit des Blockier-/Deblockier-Mechanismus motiviert wurde. Aufgrund der gewählten Gewichtung wächst die Anzahl der Token auf der unteren Kante ins Unendliche. Tabelle 8.5 zeigt den Zeitverbrauch für diesen

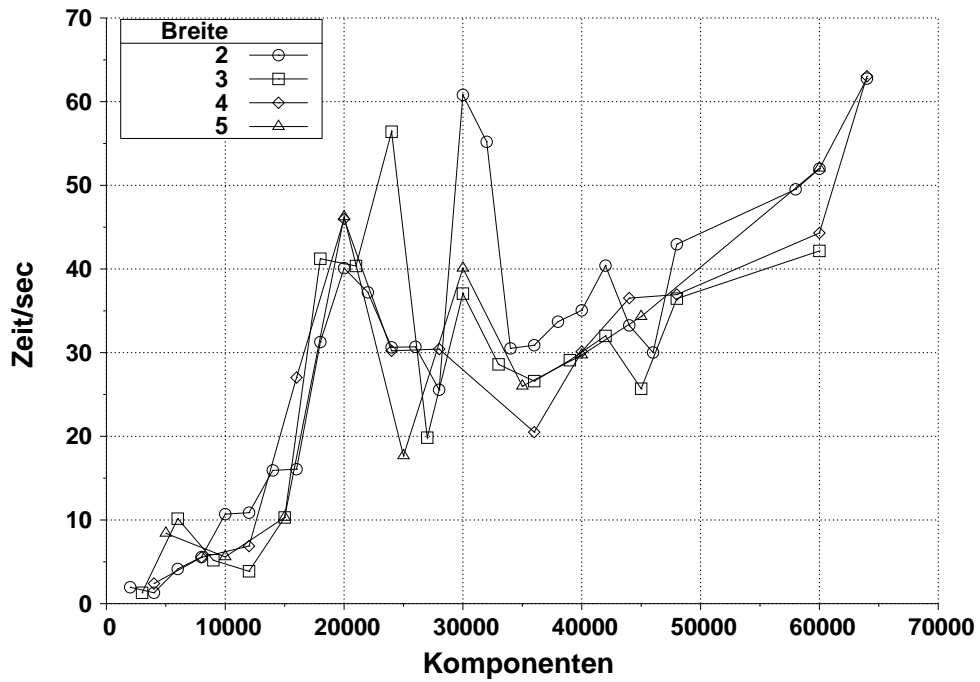


Abbildung 8.39: Zeitverbrauch = F(Komp.) (Random, Prot. = (sm*e|o)*, ident. Prot.)

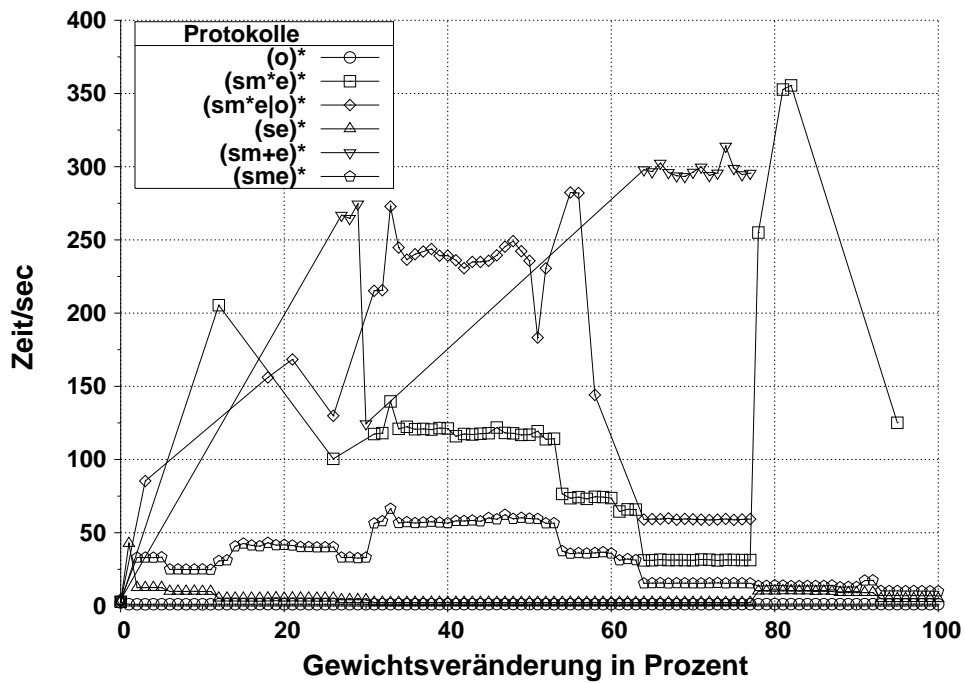


Abbildung 8.40: Zeitverbrauch = F(Gewichte) (Rows, Breite = 2, Tiefe = 10000, ident. Prot.)

Aktiviert	Deaktiviert
~ 0 sec	> 3600 sec

Tabelle 8.5: Zeitverbrauch = F(Komp.) (Rows, Breite = 1, Prot. = o*, ohne/mit Blockieren/Deblockieren, inkomp. Prot.)

Datenflußgraphen ohne beziehungsweise mit aktiviertem Blockier/Deblockier-Mechanismus. Man beobachtet – wie erwartet –, daß der Blockier/Deblockier-Mechanismus für das erfolgreiche Terminieren des Model-Checking-Verfahrens unabdingbar ist. Ohne Blockier/Deblockier-Mechanismus wird der Model-Checking-Algorithmus nur aufgrund des vorgegebenen Zeitlimits von einer Stunde abgebrochen.

Partial Order Reduction: In diesem Abschnitt wird der Einfluß der PARTIAL ORDER REDUCTION veranschaulicht. Abbildung 8.42 zeigt den Zeitverbrauch für die Graphstruktur ROWS in Abhängigkeit von der Anzahl der Datenflußkomponenten des Datenflußgraphen. Mit Partial Order Reduction benötigt das Model-Checking-Verfahren nur wenige Sekunden zur Untersuchung eines Datenflußgraphen. Ohne Partial Order Reduction wird selbst für kleine Datenflußgraphen bereits circa eine Stunde benötigt. Manche Untersuchungen wurden auch nach einer Stunde aufgrund der vorgegebenen Zeit- oder Speicherlimits ohne Ergebnis abgebrochen. Man erkennt also, daß die Partial Order Reduction von herausragender Bedeutung für die Schnelligkeit und somit auch für den Erfolg der Zyklensuche bei den gegebenen Zeit- und Speicherbeschränkungen ist.

Ordered Maps vs. Incrementalmaps: Abbildung 8.43 zeigt den Zeitverbrauch bei Verwendung der Datenstruktur Ordered Map (vergleiche Abschnitt 7.3.2.2), wobei die Anzahl der Bits, die indirekt die Anzahl der Zeiger pro Ordered Map repräsentieren (2 Bits resultieren in $2^2 = 4$ Zeigern) aufgezeigt ist. Es ergibt sich, daß unter Verwendung von 4 Bits die Zeit minimal wird.

Abbildung 8.44 zeigt den Speicherverbrauch bei Verwendung der Datenstruktur Ordered Map. Es ergibt sich, daß unter Verwendung von 3 beziehungsweise 4 Bits der Speicherverbrauch minimal wird. In allen hier vorgestellten Versuchen ist somit die Anzahl der Bits auf 4 gesetzt.

Die hervorragende Eignung von Ordered Maps ist bereits in den vorausgegangenen Abschnitten verdeutlicht worden. Betrachtet man Abbildung 8.45, so beobachtet man, daß sich die Zeit für eine erfolgreiche Simulation in Abhängigkeit von der Anzahl der Datenflußkomponenten unter Verwendung von Incrementalmaps nicht mehr linear entwickelt. Dies liegt daran, daß die Schritte, die notwendig sind, um einen Wert aus der Incrementalmap auszulesen, mit zunehmender Anzahl von Incrementalmaps wächst, wohingegen die Anzahl der Schritte bei den Ordered Maps unabhängig von der Anzahl der gespeicherten Werte konstant ist. Außerdem sind die Zeiten für eine erfolgreiche Zyklensuche im Erreichbarkeitsgraphen bei Incrementalmaps erheblich höher als bei Ordered Maps (vergleiche dazu die Abbildungen 8.45 und 8.9).

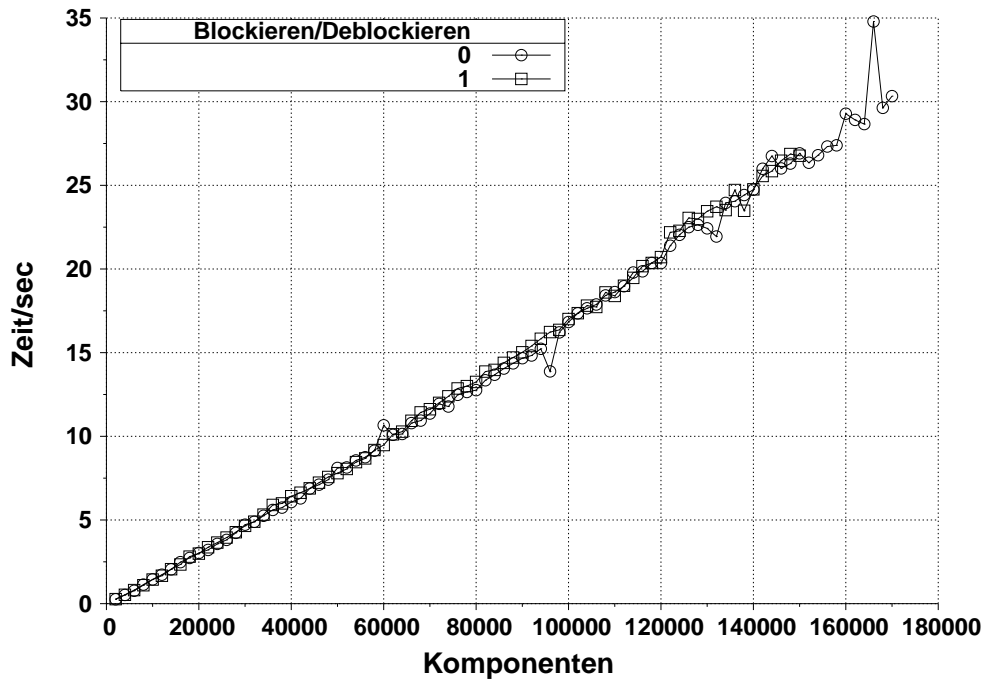


Abbildung 8.41: Zeitverbrauch = F(Komp.) (Rows, Breite = 2, Prot. = (sm * e | o)*, komp. Prot.)

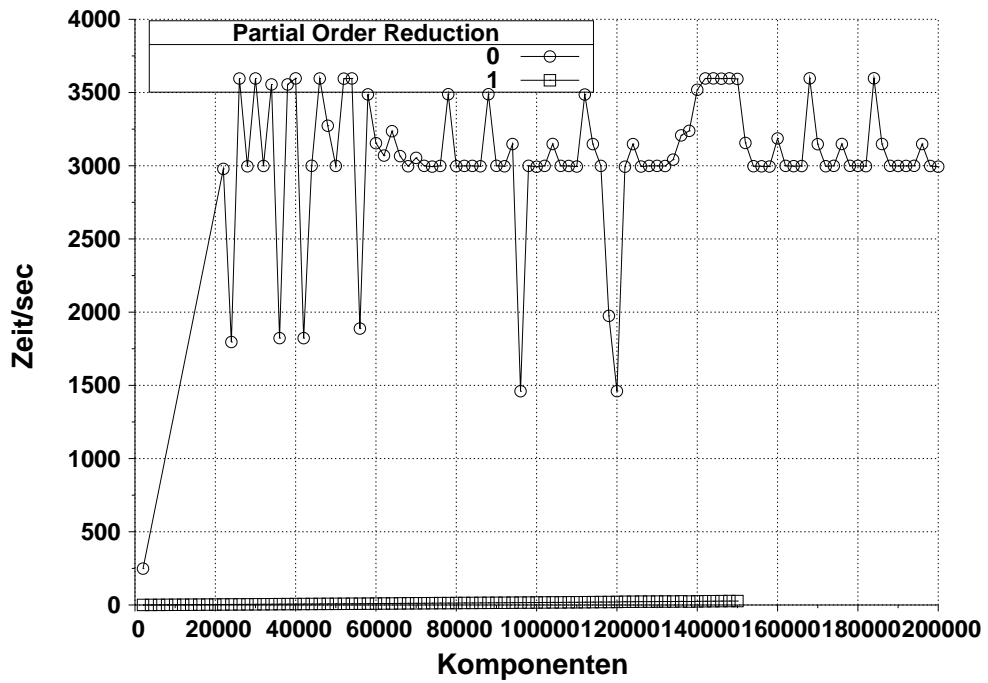


Abbildung 8.42: Zeitverbrauch = F(Komp.) (Rows, Breite = 2, Prot. = (sm * e | o)*, ident. Prot.)

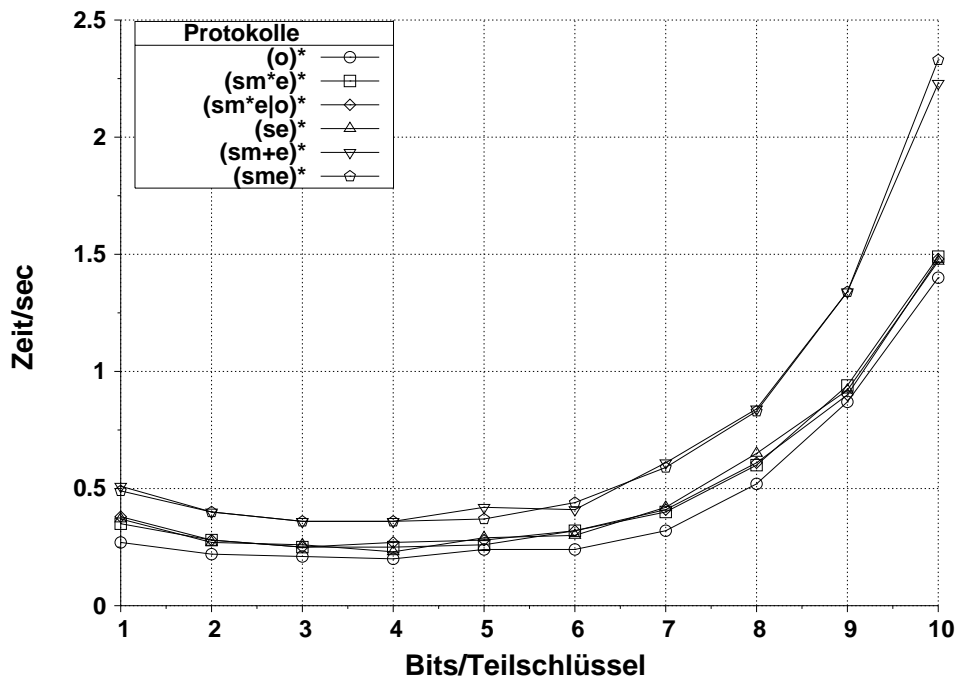


Abbildung 8.43: Zeitverbrauch=F(Bits/Teilschl.) (Rows, Breite = 2, Tiefe = 1000, ident. Prot.)

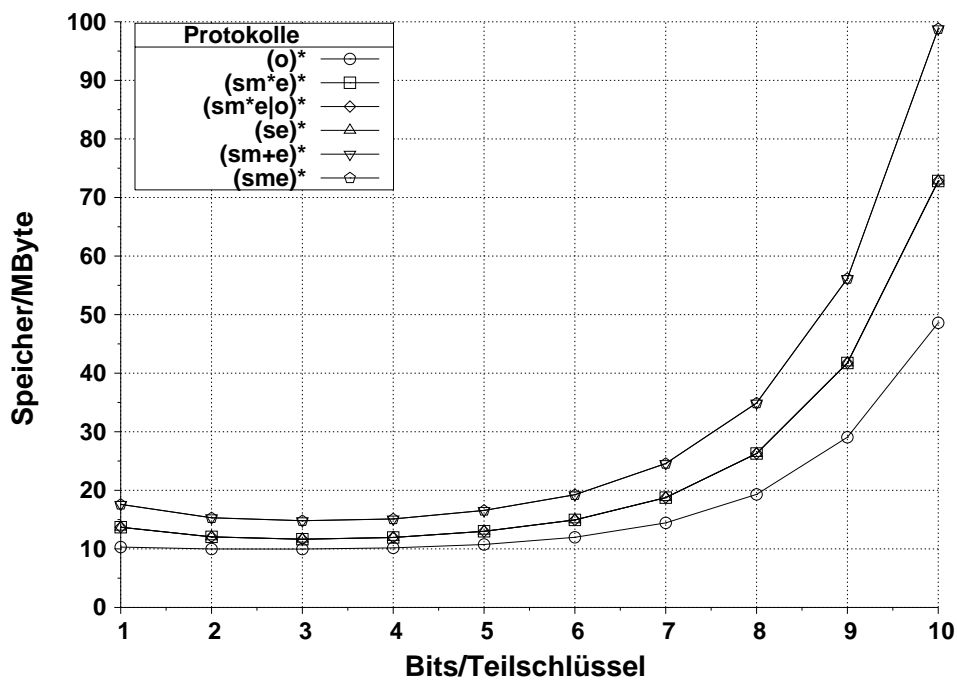


Abbildung 8.44: Speicherverbrauch=F(Bits/Teilschl.) (Rows, Breite = 2, Tiefe = 1000, ident. Prot.)

Anzahl Inittoken	Zyklus vorhanden	Zeitverbrauch
0	nein	~ 0 sec
1	nein	~ 0 sec
2	nein	~ 0 sec
3	nein	~ 0 sec
4	ja	~ 0 sec
5	ja	~ 0 sec
6	ja	~ 0 sec

Tabelle 8.6: Ergebnisse zu Distribute-Merge-Konstrukt

Colored-BDF- und Colored-DDF-Datenflußgraphen: In diesem Abschnitt werden exemplarisch verschiedene gefärbte BDF- und DDF-Datenflußgraphen (vergleiche Abschnitte 4.3.3.2 und 4.3.3.3) mit Hilfe des Model-Checking-Algorithmus untersucht. Zu diesem Zweck werden die Kontrollflußstrukturen `if-then-else`, `while` und `Distribute-Merge` (vergleiche Abschnitt 4.3.3.2 näher betrachtet).

If-Then-Else: Abbildung 8.46 zeigt den Testgraphen zur Untersuchung einer `if-then-else`-Verzweigung. Bei der Ansteuerung von `CSwitch` und `CSelect` mit der gleichen Folge von Booleschen Token wurde nach 0.01 sec ein Zyklus gefunden. Demgegenüber dauerte es gleichfalls 0.01 sec, bis der Model Checker bei einer gleichzeitigen Ansteuerung von `CSwitch` und `CSelect` mit unterschiedlichen booleschen Werten erkannte, daß kein Zyklus existiert.

While: Abbildung 8.47 zeigt den Testgraphen zur Untersuchung des `while`-Konstrukts. Wenn ein entsprechendes Initialisierungstoken, wie in der Abbildung dargestellt, verwendet wird, braucht der Model Checker < 0.01 sec Zeit, um den entsprechenden Zyklus zu finden.

Distribute-Merge: Abbildung 8.48 zeigt den Testgraphen zur Untersuchung der nichtdeterministischen `Merge`- und `Distribute`-Komponenten. Es wurden dabei Gewichte unterschiedlich von 1 verwendet. Zudem wurden nur Kommunikationsprotokolle basierend auf 0 eingesetzt. In Tabelle 8.6 ist aufgeführt, wie lange der Model Checker für verschiedene Anzahlen von Initialisierungstoken auf der Rückkopplungskante benötigt, um zu erkennen, ob ein Zyklus existiert.

Die Tabelle 8.7 stellt den gefundenen Zyklus im Erreichbarkeitsgraphen dar für den Fall von fünf Initialisierungstoken auf der Rückkopplungskante. Die unter der Bezeichnung `Id` aufgeführten Nummern kennzeichnen die einzelnen Knoten des Zyklus im Erreichbarkeitsgraphen. Im Feld `Fat` ist die Nummer des jeweiligen Vorgängerknotens aufgeführt. Die Spalte `F : T` gibt den Fifomaten und dessen zugehörige Transition an, die gerade bearbeitet wurden. Mit `St` ist die Tabellenspalte gekennzeichnet, welche den Status des betroffenen Knotens wiedergibt. Die Spalte `So` gibt die Anzahl der Söhne des jeweiligen Knotens des Erreichbarkeitsgraphen an. Da in dieser Tabelle nur Knoten, die im Zyklus enthalten sind, aufgeführt wurden, aber im Erreichbarkeitsgraphen durchaus mehr Knoten enthalten sein können, ist diese Anzahl bisweilen größer als 1. Die verbleibenden Spalten bezeichnen die einzelnen Zustände der Fifoinhalte beziehungsweise die jeweiligen Fifoinhalte.

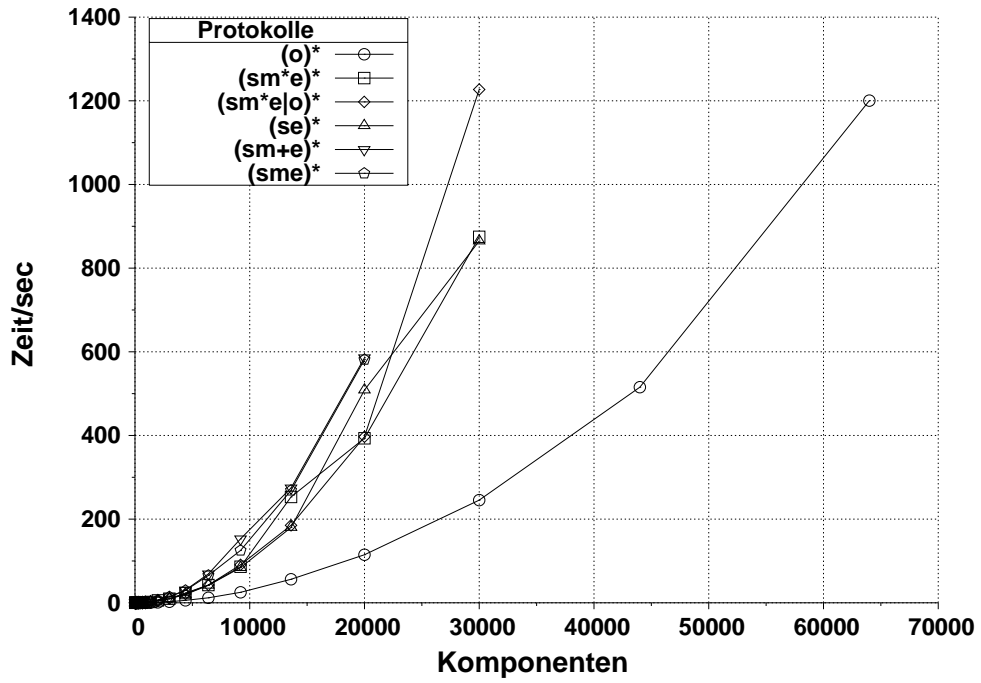


Abbildung 8.45: Zeitverbrauch=F(Komp.) (Rows, Breite = 2, Incrementalmaps, ident. Prot.)

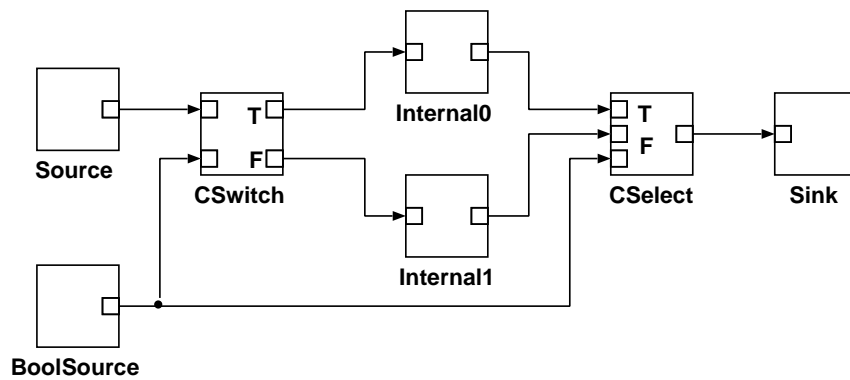


Abbildung 8.46: if-then-else-Verzweigung

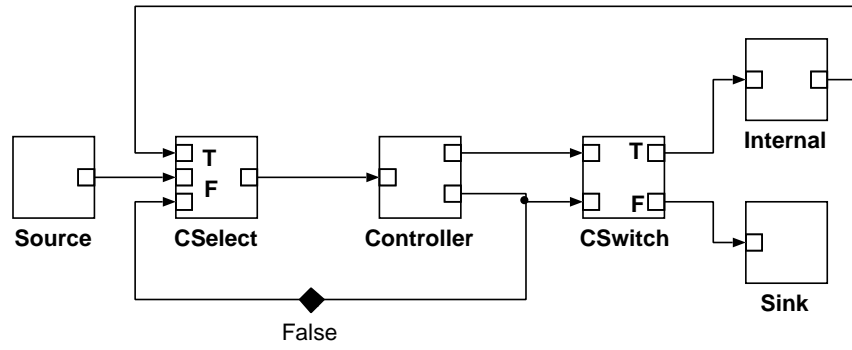


Abbildung 8.47: while-Schleife

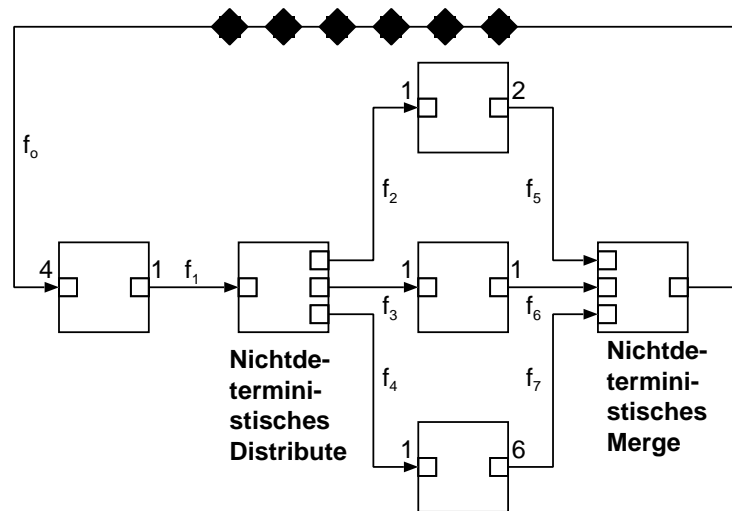


Abbildung 8.48: Distribute-Merge-Konstrukt

Tabelle 8.7: Zyklus im Erreichbarkeitsgraphen für Distribute-Merge-Konstrukt mit 5 Init tokens

Id	Fat	F : T	St	So	1To1	1To2	1To6	4To1	D	M	f ₀	f ₁	f ₂	f ₃	f ₄	f ₅	f ₆	f ₇
0			D+E	1	S0	S0	S0	S0	S0	S0	o o o o o							
1	0	4To1:S0_s0	D+E	1	S0	S0	S0	s0	S0	S0	o							
2	1	4To1:s0_S0	D+E	1	S0	S0	S0	S0	S0	S0	o	o						
3	2	D:S0_s0	D+E	3	S0	S0	S0	S0	s0	S0	o							
4	3	D:s0_S0c	D+E	1	S0	S0	S0	S0	S0	S0	o				o			
5	4	1To6:S0_s0	D+E	1	S0	S0	s0	S0	S0	S0	o							
6	5	1To6:s0_S0	B+E	1	S0	S0	S0	S0	S0	S0	o							o o o o o o
7	6	M:S0_s0c	D+E	1	S0	S0	S0	S0	S0	s0	o							o o o o o
8	7	M:s0_S0	B+E	1	S0	S0	S0	S0	S0	S0	o o							o o o o o
9	8	M:S0_s0c	D+E	1	S0	S0	S0	S0	S0	s0	o o							o o o o
10	9	M:s0_S0	B+E	1	S0	S0	S0	S0	S0	S0	o o o							o o o o
11	10	M:S0_s0c	D+E	1	S0	S0	S0	S0	S0	s0	o o o							o o o
12	11	M:s0_S0	B+E	2	S0	S0	S0	S0	S0	S0	o o o o							o o o
13	12	M:S0_s0c	D+E	1	S0	S0	S0	S0	S0	s0	o o o o							o o
14	13	M:s0_S0	D+E	2	S0	S0	S0	S0	S0	S0	o o o o o							o o
15	14	M:S0_s0c	D+E	1	S0	S0	S0	S0	S0	s0	o o o o o							o
16	15	M:s0_S0	D+E	1	S0	S0	S0	S0	S0	S0	o o o o o o							o
17	16	4To1:S0_s0	D+E	1	S0	S0	S0	s0	S0	S0	o o							o
18	17	4To1:s0_S0	D+E	1	S0	S0	S0	S0	S0	S0	o o		o					o
19	18	D:S0_s0	D+E	3	S0	S0	S0	S0	s0	S0	o o							o
20	19	D:s0_S0a	D+E	1	S0	S0	S0	S0	S0	S0	o o		o					o
21	20	1To2:S0_s0	D+E	1	S0	s0	S0	S0	S0	S0	o o							o
22	21	1To2:s0_S0	E	2	S0	S0	S0	S0	S0	S0	o o							o
23	22	M:S0_s0c	D+E	1	S0	S0	S0	S0	S0	s0	o o					o o		
24	23	M:s0_S0	E	1	S0	S0	S0	S0	S0	S0	o o o					o o		
25	24	M:S0_s0a	D+E	1	S0	S0	S0	S0	S0	s0	o o o					o		
26	25	M:s0_S0	E	2	S0	S0	S0	S0	S0	S0	o o o o					o		
27	26	M:S0_s0a	D+E	1	S0	S0	S0	S0	S0	s0	o o o o							
28	27	M:s0_S0	C+R	0	S0	S0	S0	S0	S0	S0	o o o o o							

8.4.1.7 Abschließende Bewertung

In diesem Abschnitt werden die wesentlichen Ergebnisse der mit dem Model Checker durchgeführten Tests zusammengefaßt. Tabelle 8.8 gibt dabei die Auswirkungen der untersuchten Parameter wieder.

Insgesamt kann man festhalten, daß der Zeitverbrauch für die Analyse von Datenflußgraphen im Bereich der Bild- und Signalverarbeitung, die gekennzeichnet sind durch

- eine zufällige Graphstruktur mit gelegentlichen Rückkopplungen,
- eine Graphgröße, die sich im Bereich einiger Dutzend bis hin zu einigen hundert Datenflußkomponenten bewegt,
- eine Mischung verschiedener Datenflußkomponenten mit einer vergleichsweise geringen Anzahl von Schnittstellen und einer Kombination unterschiedlicher Kommunikationsprotokolle beziehungsweise
- eine geringe Anzahl von Gewichten größer 1,

sich in dem Rahmen von einigen Millisekunden bis hin zu einigen wenigen Minuten bewegen wird und einen entwurfsbegleitenden Einsatz des vorgestellten Model-Checking-Verfahrens erlaubt. Die Eignung des Model Checkers für die Zyklensuche in Erreichbarkeitsgraphen der turingvollständigen Datenflußparadigmen Colored BDF und Colored DDF – im Rahmen der durch die Berechnungskomplexität eingeschränkten Möglichkeiten – wurde erfolgreich anhand einer Reihe von Beispielen verdeutlicht.

8.4.2 Vergleich mit dem Model Checker Spin

Mit Hilfe des Model Checkers **Spin** (siehe Abschnitt 2.4.5) wurden eine Reihe von Versuchen durchgeführt (siehe [Köc04]), welche einen Vergleich von **Skylla** mit dem aktuellen Stand der Technik ermöglichen. Einleitend sind die Aufrufparameter und diverse Einschränkungen von **Spin** aufgelistet. Im Anschluß daran sind die Ergebnisse der durchgeführten Versuche beschrieben. Ein Vergleich von **Spin** und **Skylla** schließt diesen Abschnitt ab.

Aufrufparameter: Der Model Checker **Spin** besitzt eine Reihe von Aufrufparametern, die im folgenden erläutert werden. Dabei gilt, daß **Spin** in allen Testläufen eines Versuches mit identischen Aufrufparametern ausgeführt wurde.

Um mit **Spin** ein Modell untersuchen zu können, muß das Modell zuerst in der Beschreibungssprache **Promela** spezifiziert werden. Diese Spezifikation wird dann in ein C-Programm übersetzt, compiliert und ausgeführt.

Bei der Compilierung können dabei folgende Parameter modifiziert werden [Hol04]:

- **-DCOLLAPSE:** Verwenden des verlustlos arbeitenden Kompressionsmodus **COLLAPSE**.

Graphparameter	
Graphstruktur	Die untersuchten Graphstrukturen haben einen deutlichen Einfluß auf den Zeitverbrauch.
Graphgröße	In den getätigten Untersuchungen ergab sich bei gleicher Graphstruktur häufig ein linearer Anstieg des Zeitverbrauchs in Abhängigkeit von der Graphgröße. Aufgrund der effizienten PARTIAL ORDER REDUCTION ist zum Beispiel bei der Graphstruktur ROWS die Breite ohne Einfluß.
Rückkoppungen	Je weniger Quellen in einem Datenflußgraphen enthalten sind, desto schneller ist das Model-Checking-Verfahren.
Kantenparameter	
Kapazität	Solange die Kapazität ausreichend groß ist, hat diese keinen Einfluß auf die Geschwindigkeit der Zyklensuche.
Komponentenparameter	
Protokolle	Je komplexer die in einem Datenflußgraphen enthaltenen Kommunikationsprotokolle sind, desto höher ist der Zeitverbrauch bei der Zyklensuche.
Anzahl Schnittstellen	Da die Anzahl der Schnittstellen bei Datenflußkomponenten im Bereich der Bild- und Signalverarbeitung relativ klein ist, ist auch deren Einfluß auf den Zeitverbrauch unproblematisch.
Gewichte	Werden in einem Datenflußgraphen Gewichte größer 1 eingesetzt, so steigt der Zeitbedarf für die Zyklensuche. Dabei hängt der Zeitverbrauch aber nicht direkt von der Anzahl der Gewichte, welche größer als 1 sind, ab.
Algorithmusparameter	
Timeout	Aufgrund der in dieser Arbeit entwickelten Techniken wie PARTIAL ORDER REDUCTION und BLOCKIEREN/DEBLOCKIEREN können große Datenflußgraphen mit einem Zeitverbrauch weit unter dem vorgegebenen Timeout von 1 Stunde untersucht werden.
Speicherlimit	Es können große Datenflußgraphen innerhalb des vorgegebenen Limits von 1 GByte untersucht werden.
Suchziel	Die Bestimmung eines Zyklus im Erreichbarkeitsgraphen, welcher den Wurzelknoten beinhaltet, ist in vergleichsweise kurzer Zeit möglich.
BLOCKIEREN	Werden Datenflußkomponenten dergestalt miteinander verknüpft, daß ein Speicherüberlauf auftritt, so erlaubt der BLOCKIER/DEBLOCKIER-Mechanismus deren Analyse. Ohne dieses Verfahren würde der Model-Checking-Algorithmus nicht terminieren.
PARTIAL ORDER REDUCTION	Gibt es mehrere alternative Pfade im Erreichbarkeitsgraphen, so beschränkt das hier entwickelte Partial-Order-Reduction-Verfahren die Suche auf einen dieser Pfade.
Ordered Maps	Untersuchungen haben ergeben, daß der Zeit- und Speicherverbrauch der Datenstruktur Ordered Maps bei einer Anzahl von 4 Bits für einen Teilschlüssel optimal ist.

Tabelle 8.8: Abschließende Bewertung des Model Checkers

- **-DXUSAVE:** Zeigt an, daß im Promela-Modell exklusive Schreib- und Lesezugriffe auf Ein-/Ausgabekanäle angegeben wurden und daß diese Information bei der in Spin implementierten Variante von PARTIAL ORDER REDUCTION genutzt werden kann.
- **-DVECTORSZ=32768:** Legt die maximale Größe des Statevectors, der zur Kodierung eines Zustandes verwendet wird, auf 32768 Bits fest. Dieser Wert wurde durch empirische Beobachtung ermittelt [Köc04].
- **-DMEMLIM:** Gibt den maximal zu verwendenden Speicher an und veranlaßt Spin, Informationen über den tatsächlichen Speicherbedarf auszugeben.

Das compilierte Modell kann dann anschließend ausgeführt werden. Dabei sind folgende Parameter einstellbar:

- **-a:** Dient zur Angabe eines Never-Claims [Hol04]. Auf diese Weise kann zum Beispiel die Suchanfrage nach einem Zyklus, welcher den Wurzelknoten des Erreichbarkeitsgraphen beinhaltet, formuliert werden.
- **-m:** Mit Hilfe dieses Parameters wird die Tiefensuchgrenze festgelegt. In den vorliegenden Untersuchungen war dieser Wert auf 2600000 gesetzt.
- **-w:** Spezifiziert die Größe der zu verwendenden Hash-Tabelle. In dieser Arbeit wurde der Wert auf 23 gesetzt. Damit erhält man 2^{23} Bit.
- **-n:** Dieser Parameter schaltet die Anzeige des bei einer Simulation nicht erreichten Codes ab.
- **-c:** Mit diesem Parameter wird die Anzahl der Fehler $n \in \mathbb{N}$ übergeben, bei deren Überschreiten das Programm abbrechen soll.
- **-v:** Dient zur Anzeige zusätzlicher Informationen [Hol91, Köc04].

Einschränkungen von Spin: Spin beinhaltet mehrere in der Implementation fest einkodierte Einschränkungen, die den Vergleich auf sehr kleine Datenflußgraphen beschränken. So gilt beispielsweise:

- Die Anzahl der Prozesse und damit die Anzahl der Datenflußkomponenten ist auf 256 beschränkt. Diese Zahl wird aber weiter eingeschränkt:
 - Aufgrund eines Bugs in der verwendeten Spin-Version 4.20, wo zur Speicherung der Zahl der Datenflußkomponenten eine Variable vom Typ `signed char` anstelle einer `unsigned char`-Variable verwendet wurde, können nur Datenflußgraphen mit 128 Datenflußkomponenten untersucht werden.

- Von diesen 128 Prozessen wird ein INIT-Prozeß beispielsweise zur Initialisierung von Rückkopplungskanten mit Initialisierungstoken benötigt¹. Somit verbleiben noch 127 Prozesse für 127 Datenflußkomponenten.
- Die Anzahl der Fifos ist auf 256 beschränkt.
- Die Kapazitäten der Fifos müssen sehr klein gewählt werden, damit Spin in endlicher Zeit terminiert. Daher werden in der Regel die Kapazitäten auf 1 gesetzt.
- Spin nutzt nur den physikalisch freien Speicherplatz. Steht also unter Linux ein physikalischer Speicher von 1 GByte zur Verfügung und wird davon beispielsweise durch andere Benutzer bereits 600 MByte belegt, so bricht Spin bei einem größeren Bedarf als 400 MByte mit einer knappen Fehlermeldung ab. Dabei ist für den Benutzer unklar, ob dieser Abbruch aus einem Fehler im Modell oder aus zu geringem freien physikalischem Speicher resultierte.
- Die Bedienung von Spin ist sehr umständlich. So existiert keine dynamische Speicherverwaltung. Der Benutzer muß selber vor Ausführung des Modells eine Abschätzung folgender Werte vornehmen:
 - Größe der Hashtabelle
 - Größe des Tiefensuchstacks
 - Größe des freien physikalischen Speichers

Jegliche Fehleinschätzung führt zum vorzeitigen Abbruch. Selbst für einen erfahrenen Benutzer ist die korrekte Vorabangabe dieser Größen für ein zu untersuchendes Modell in der Regel nicht möglich. Dies hat zur Folge, daß man sich schrittweise an die richtige Einstellung herantasten muß, wobei – wie im vorigen Punkt geschildert – nicht notwendigerweise ersichtlich ist, daß ein vorzeitiger Abbruch aus einer nicht ausreichenden Speicheranforderung resultiert.

- Mittels des Parameters `-c` kann man steuern, wie viele „Fehler“-meldungen Spin ausgeben soll. Dies ist insbesondere in Kombination mit dem Parameter `-a`, mit dessen Hilfe Never-Claims zur Erkennung von Zyklen in den Wurzelknoten des Erreichbarkeitsgraphen definiert werden, sehr umständlich. Denn neben diesen Verletzungen der Bedingung für Wurzelzyklen werden auch Akzeptanzzyklen angezeigt. Akzeptanzzyklen resultieren beispielsweise aus Zustandstransitionen von Automaten, welche den gleichen Start- und Endzustand besitzen. Man hat also die Wahl, sich zuwenige Meldungen ausgeben zu lassen und den Zyklus in den Wurzelknoten nicht zu erkennen, beziehungsweise mit einer Flut von Ausgaben konfrontiert zu werden. Nachteilig dabei ist zudem, daß auch der Zeitverbrauch bei hoher Anzahl von Meldungen erheblich ansteigt. Der Benutzer muß sich also durch eine schrittweise Anpassung der Anzahl der Meldungen an das richtige Ergebnis herantasten.

¹In Spin sind Kommunikationskanäle globale Variablen, denen nur durch einen Initialisierungsprozeß Initialisierungstoken zugewiesen werden können.

- Dem Benutzer bleibt aufgrund einer fehlenden entsprechenden Ausgabe größtenteils verborgen, welche Suchoptimierungen **Spin** anwendet.
- Es existiert zwar ein syntaktisches Konstrukt, um in **Spin** atomare Operationen zu kennzeichnen. Allerdings ignoriert **Spin** bei der Ausführung gegebenenfalls diese vom Benutzer spezifizierte Atomarität. Ist eine der Teiloperationen nicht ausführbar, wird die Ausführung der atomaren Operation unterbrochen und ein anderer Automat ausgeführt. Dies hat mehrere negative Konsequenzen:
 - Es findet eine Verletzung der vom Benutzer erwarteten Ausführungssemantik statt. Das Ergebnis kann sich erheblich von einem prognostizierten Resultat unterscheiden.
 - Es findet kein STATEMENT-MERGING (vergleiche Abschnitt 2.4.5) für atomare Operationen statt. Das bedeutet, daß die einzelnen Operationen nicht zu einer zusammengefaßt werden. Denn es muß ja eine Unterbrechung möglich sein, und diese kann nur in einem Zwischenzustand des Automatenmodells erfolgen.

Man kann in **Spin** auch für eine atomare Operation keine logischen Bedingungen (Guards) definieren, die im Voraus die Ausführbarkeit aller Teiloperationen sicherstellen würden.

- Durch einzelne syntaktische Konstrukte wie beispielsweise Progresslabels [Hol04] werden vom Benutzer unbemerkt weitere Zustände in das Automatenmodell eingefügt, welche die Größe des Erreichbarkeitsgraphen wiederum erhöhen. Dies ist aus der Dokumentation von **Spin** nicht ersichtlich. Daraus folgt, daß der Benutzer mit syntaktischen Konstrukten sehr vorsichtig umgehen muß, wenn er nicht eine VERSTECKTE VERKOMPLIZIERUNG DES MODELLS riskieren will.

Diese unterschiedlichen Einschränkungen von **Spin** verhinderten eine so extensive Untersuchung wie bei **Skylla**. Im folgenden werden die durchgeführten Untersuchungen und die daraus gezogenen Rückschlüsse dargestellt.

8.4.2.1 Graphstruktur Rows

In diesem Versuch wird die Graphstruktur **Rows** (vergleiche Abbildung 8.8 (a)) untersucht, wobei die Ausführungszeit des eigentlichen Algorithmus in Abhängigkeit der Datenflußkomponenten betrachtet wird. Die gesamte Ausführungszeit, welche auch das Compilieren beinhaltet und die erheblich größer ist, wird hier nicht berücksichtigt.

Identische Kommunikationsprotokolle: Abbildung 8.49 zeigt die Ausführungszeit in Abhängigkeit von der Anzahl der Datenflußkomponenten. Alle Datenflußkomponenten führen identische Kommunikationsprotokolle aus. Dabei erkennt man, daß außer für das Kommunikationsprotokoll $(sm^*e|o)^*$ die gemessenen Zeiten sehr klein sind. Im Falle von $(sm^*e|o)^*$ wurde in einem ersten Durchlauf kein Zyklus entdeckt. Erst als in der **Promela**-Beschreibung der Kommunikationsprotokolle per Hand die Reihenfolge der von demselben Zustand des zugehörigen Automatenmodells ausgehenden Transitionen modifiziert wurde, konnte ein Zyklus gefunden

werden. Aufgrund des fehlenden Einblicks in die Vorgehensweise des Algorithmus kann nur eine Vermutung über die möglichen Gründe angestellt werden:

- An der eingestellten Tiefensuchgrenze des Algorithmus liegt es nicht, da diese nicht erreicht wird. Dies zeigt aber auch, daß keine reine Tiefensuche verwendet wird.
- Eventuell werden bei Kanten im Automatenmodell, die denselben Start- und Endknoten haben, diese so oft wie möglich ausgeführt, bevor andere Kanten behandelt werden.
- Diese Kanten mit gleichem Start- und Endzustand wirken sich insbesondere dann schädlich für Spin aus, wenn der betrachtete Zustand kein initialer Zustand des jeweiligen Automaten ist. Handelt es sich um einen initialen Zustand befindet sich der Automat ja nach einem Durchlauf wieder in der Ausgangssituation.

Die aufgrund der durchgeführten Veränderungen ermittelten Zeiten für das Kommunikationsprotokoll $(sm^*e|o)^*$ sind dennoch deutlich höher als bei den anderen Kommunikationsprotokollen. Zudem weist die entsprechende Kurve ein nichtlineares Wachstum auf.

Abbildung 8.50 zeigt das Ergebnis einer Untersuchung, die sich von der vorausgehenden nur in der Anzahl der Ports, die hier auf 2 gesetzt wurde, unterscheidet. Die gemessenen Zeiten zeigen ein nichtlineares Wachstum in Abhängigkeit von der Anzahl der Datenflußkomponenten. In diesem Versuch konnte das oben geschilderte von der Erwartung abweichende Verhalten bei Verwendung des Kommunikationsprotokolls $(sm^*e|o)^*$ nicht beobachtet werden. Allerdings ist zu erkennen, daß sich die gemessenen Zeiten trotz der geringen Graphgröße zum Teil bereits im Sekundenbereich bewegen.

Unterschiedliche Eingabeprotokolle: Variiert man die Eingabeprotokolle, so ergeben sich für den Zeitverbrauch zur Bestimmung eines Zyklus in Abhängigkeit von der Anzahl der Datenflußkomponenten die in Abbildung 8.51 dargestellten Kurven. Die gemessenen Zeiten sind sehr klein und daher kommt es aufgrund der Meßgenauigkeiten zu den schwankenden Kurvenverläufen.

Abbildung 8.52 zeigt einen ähnlichen Versuch, der sich nur durch die Breite von 2 vom vorausgegangenen unterscheidet. Hier beobachtet man eine durchschnittliche Zunahme der benötigten Zeit um den Faktor 10 im Vergleich zu Abbildung 8.51. Die Kurvenverläufe, die aufgrund der in Spin gegebenen Limitierung der Datenflußkomponenten und Kanten bei etwa 128 Datenflußkomponenten enden, lassen ein nichtlineares Wachstum erkennen. Für das Kommunikationsprotokoll $(sm^*e|o)^*$ konnten keine Zyklen ermittelt werden. Dies liegt daran, daß der zugehörige Fifomat zwei Zustände besitzt, von denen mehrere Transitionen ausgehen. Dies bereitet Spin Probleme, da das Programm sehr oft dieselbe Transition wählt. So wird gegebenenfalls nur die Transition, die mit o beschriftet ist, selektiert.

8.4.2.2 Graphstruktur Net

In diesem Versuch wird die Graphstruktur Net (vergleiche Abbildung 8.8 (b)) untersucht, wobei die Ausführungszeit des eigentlichen Algorithmus in Abhängigkeit der Datenflußkomponenten betrachtet wird. Die gesamte Ausführungszeit, welche auch das Compilieren beinhaltet und die erheblich größer ist, wird hier nicht berücksichtigt.

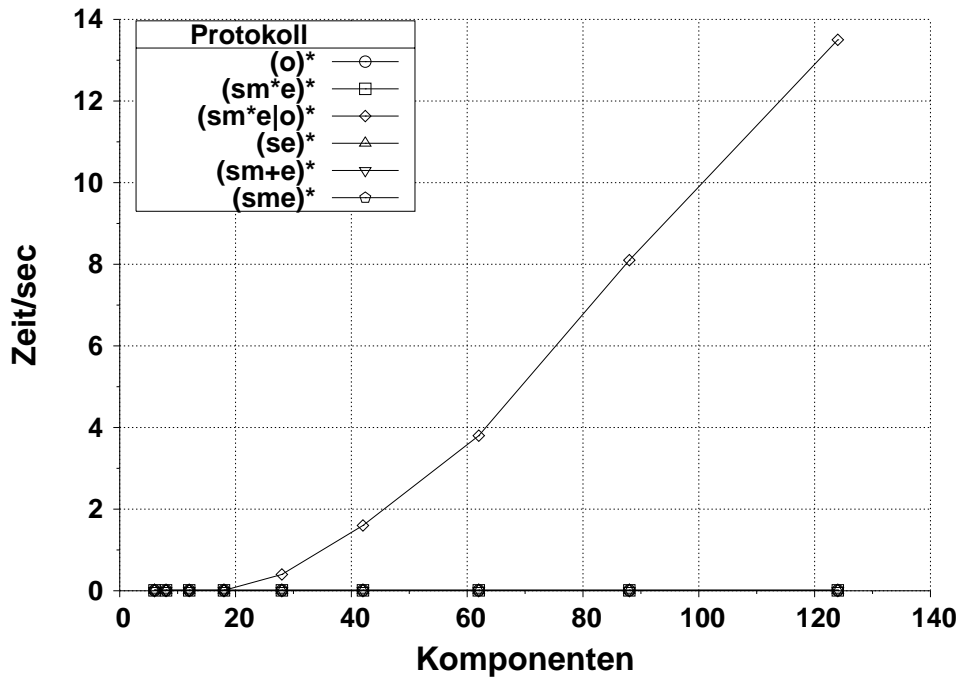


Abbildung 8.49: Zeitverbrauch = F(Komp.) (Rows, Breite = 2, Ports = 1, Kap. = 1, ident. Prot.)

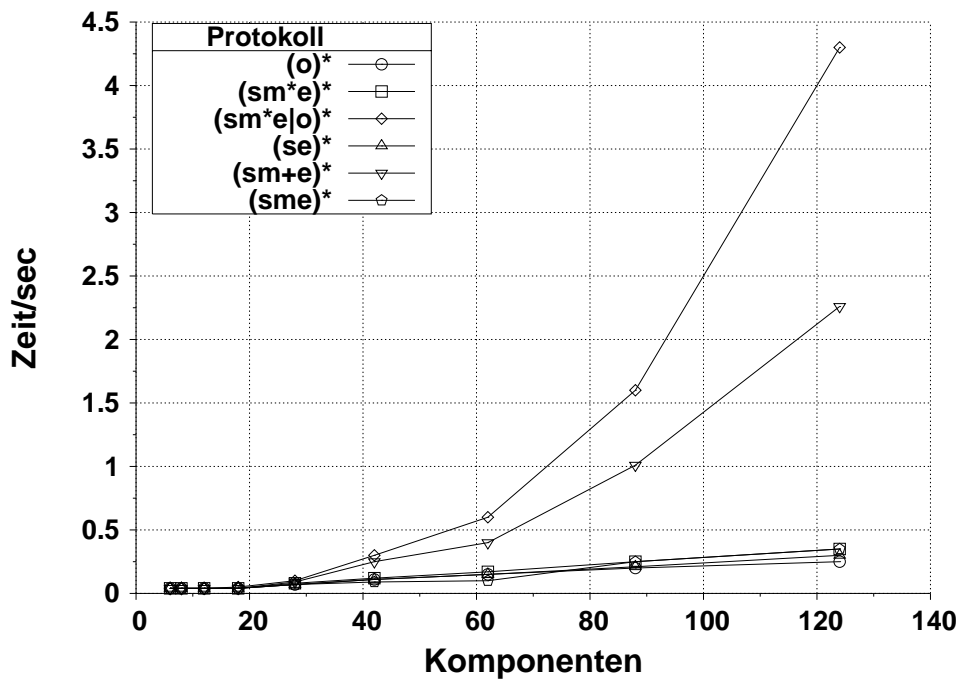


Abbildung 8.50: Zeitverbrauch = F(Komp.) (Rows, Breite = 2, Ports = 2, Kap. = 1, ident. Prot.)

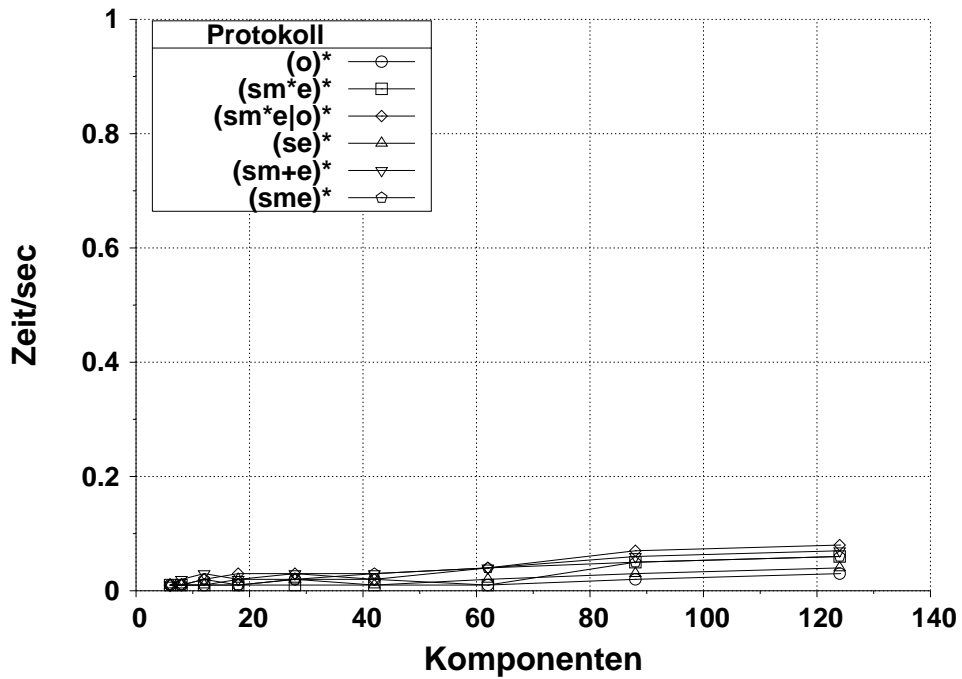


Abbildung 8.51: Zeitverbrauch=F(Komp.) (Rows, Breite= 2, Ports= 1, Kap.= 1, komp. Prot.)

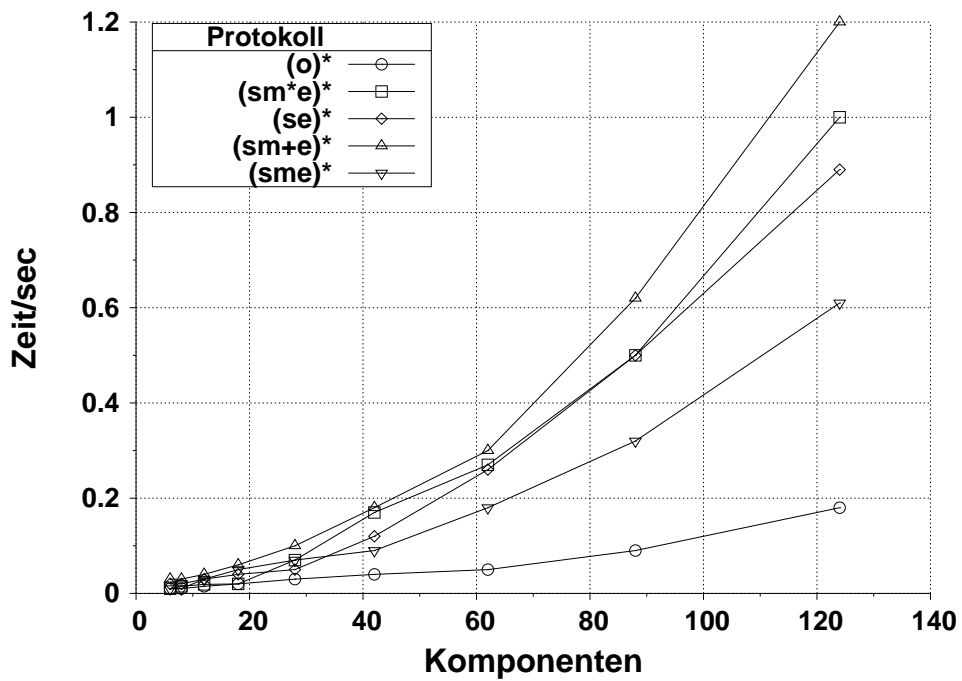


Abbildung 8.52: Zeitverbrauch=F(Komp.) (Rows, Breite= 2, Ports= 2, Kap.= 1, komp. Prot.)

Identische Kommunikationsprotokolle: Bei der zu untersuchenden Graphstruktur **Net** ist die Breite auf 4 eingestellt. Die Kapazitäten sind auf 1 gesetzt. Die Tiefe wird variiert.

Betrachtet man Abbildung 8.53, so erkennt man, daß der Zeitverbrauch für die geringe Graphgröße vergleichsweise hoch ist. Dabei tritt das nichtlineare Wachstum des Zeitverbrauchs in Abhängigkeit von der Zahl der Datenflußkomponenten wieder klar hervor.

Tabelle 8.9 zeigt das Ergebnis einer Untersuchung, die sich von der vorausgehenden nur in der Anzahl der Ports, die hier auf 2 gesetzt wurde, unterscheidet. Die Kommunikationsprotokolle $(sm^*e)^*$, $(sm^+e)^*$ und $(sm^*e|o)^*$ lieferten keine Ergebnisse aufgrund eines zu großen Zeitverbrauchs selbst bei kleiner Anzahl von Datenflußkomponenten. Für die Kommunikationsprotokolle $(se)^*$ und $(sme)^*$ konnten jedoch Zeiten ermittelt werden. Die gemessenen Zeiten zeigen ein nichtlineares Wachstum in Abhängigkeit von der Anzahl der Datenflußkomponenten. Dabei ist allerdings zu beachten, daß die angegebenen Werte erst durch sukzessives Herantasten, bei dem die Anzahl der zu produzierenden Meldungen mittels des Parameters **-c** (siehe oben) schrittweise angepaßt wurde, ermittelt wurden. Das bedeutet, daß die Gesamtzeit, die zur Bestimmung dieser Ergebnisse jeweils aufgewandt wurde, deutlich höher ist². Dieser deutlich erhöhte Zeitverbrauch im Vergleich zur Versuchsreihe mit je 1 Eingangs- und Ausgangsport zeigt die Grenzen der Anwendbarkeit von **Spin** für die vorgestellte Problemstellung deutlich auf.

Unterschiedliche Eingabeprotokolle: Werden in den Quellkomponenten andere Kommunikationsprotokolle verwendet als in den internen Datenflußkomponenten, so ergeben sich die in Abbildung 8.54 dargestellten Kurvenverläufe. Aufgrund der kleinen Zeiten schlagen Meßungenauigkeiten voll zu Buche. Daher sind die aus den Diagrammen ablesbaren Merkmale nur folgende:

- Die Zeiten, einen Zyklus im Erreichbarkeitsgraphen, der den Wurzelknoten beinhaltet, zu finden, sind klein.
- Es wurde in jedem Fall ein Zyklus gefunden.

Tabelle 8.10 zeigt einen ähnlichen Versuch, der sich nur in der Breite, welche auf den Wert 2 gesetzt ist, vom vorausgegangenen Versuch unterscheidet. Für die Kommunikationsprotokolle $(sm^*e)^*$, $(sm^+e)^*$, $(sm^*e|o)^*$ sind die Zeiten ab 12 Datenflußkomponenten bereits so hoch, daß keine Ergebnisse mehr produziert werden konnten. Die Zahlenwerte, die bereits bei 62 Datenflußkomponenten enden, da für größere Datenflußgraphen der Zeitverbrauch bereits zu hoch ist, lassen ein stark nichtlineares Wachstum des Zeitverbrauchs in Abhängigkeit von der Anzahl der Datenflußkomponenten erkennen. Dabei ist wiederum zu beachten, daß diese Ergebnisse erst durch sukzessive Anpassung der Anzahl der zu produzierenden Fehlermeldungen ermittelt wurden. Damit ist der Zeitverbrauch des Benutzers aufgrund dieser iterativen Herantastung erheblich höher. Auch hier ist die Schlußfolgerung zu ziehen, daß **Spin** für die vorgegebene Problemstellung ungeeignet ist. Denn für den Benutzer kleine Änderungen im Modell beziehungsweise Datenflußgraphen wie zum Beispiel die Änderung der Anzahl der Ports lassen bereits die Analyse scheitern.

²Da der Wert von **-c** von Hand angepaßt wurde, war eine exakte Messung nicht möglich.

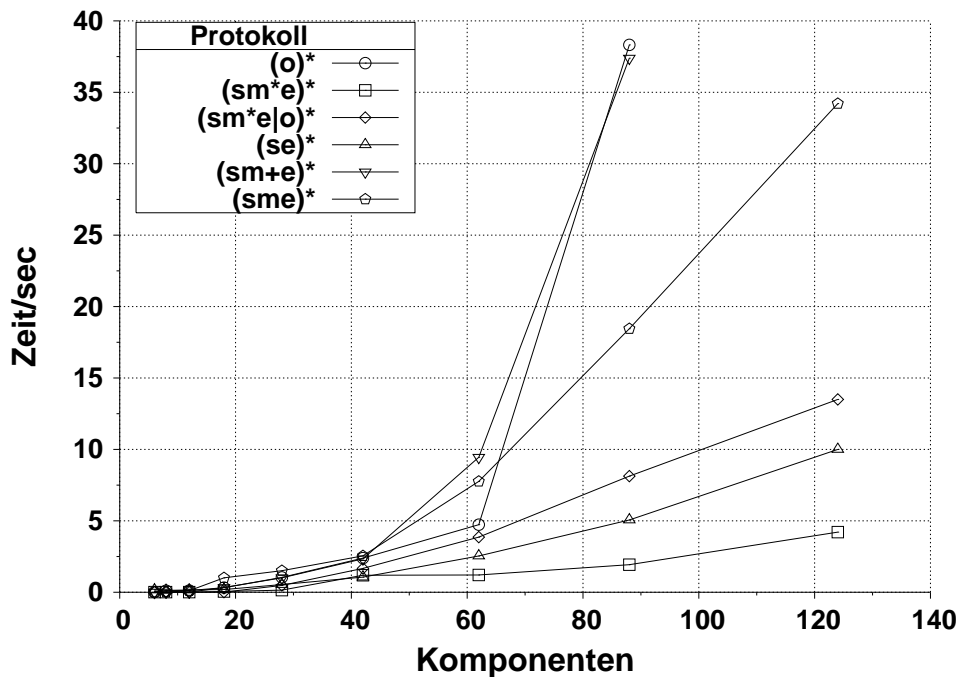


Abbildung 8.53: Zeitverbrauch = F(Komp.) (Net, Breite = 4, Ports = 1, Kap. = 1, ident. Prot.)

(se)*							
Komponenten	12	16	28	42	62	88	124
Zeit/sec	0,01	0,16	0,22	0,37	0,76	1,10	2,76
Parameter -c	18	184	155	155	155	46	155
(sme)*							
Komponenten	12	16	28	42	62	88	124
Zeit/sec	0,02	0,11	0,24	0,36	0,83	1,22	2,93
Parameter -c	13	120	139	139	139	57	139

Tabelle 8.9: Zeitverbrauch = F(Komp.) (Net, Breite = 4, Ports = 2, Kap. = 1, ident. Prot.)

(sme)*							
Komponenten	12	16	28	42	62	88	124
Zeit/sec	0,60	1,39	5,51	13,24	37,95	—	—
Parameter -c	63	35	56	73	69	—	—

Tabelle 8.10: Zeitverbrauch = F(Komp.) (Net, Breite = 4, Ports = 2, Kap. = 1, komp. Prot.)

8.4.2.3 Graphstruktur Rows mit Rückkopplungen

In diesem Abschnitt wird die Graphstruktur Rows mit Rückkopplungskanten (RowsF; vergleiche Abbildung 8.8 (c)) untersucht. Aufgrund dieser Rückkopplungen gibt es nur interne Datenflußkomponenten. Deshalb können im folgenden auch die Kommunikationsprotokolle der Quellkomponenten nicht variiert werden.

Die Abbildungen 8.55 und 8.56 zeigen auf, daß bei Vorhandensein von Rückkopplungskanten die Zeiten zur Bestimmung eines Zyklus in den Wurzelknoten des Erreichbarkeitsgraphen kleiner sind als ohne Rückkopplungen. Dies resultiert aus der geringeren Anzahl von Knoten des Erreichbarkeitsgraphen.

8.4.2.4 Graphstruktur Net mit Rückkopplungen

Für die in den Abbildungen 8.57 und 8.58 dargestellten Versuchsreihen mit der Graphstruktur Net in Kombination mit Rückkopplungskanten (NetF; vergleiche Abbildung 8.8 (d)) führen zu den gleichen Schlußfolgerungen wie bei der Graphstruktur Rows mit Rückkopplungen (RowsF).

8.4.2.5 Weitere Untersuchungen

In diesem Abschnitt werden weitere Untersuchungen vorgestellt, die beispielsweise den Einfluß der Fifokapazitäten auf die zur Bestimmung eines Zyklus aufzuwendende Zeit betrachten.

Kapazitäten: Dieser Versuch untersucht unterschiedliche Kapazitäten der Fifos bei einer Rows-Graphstruktur (vergleiche Abbildung 8.8). In Abbildung 8.59 erkennt man, wie zunehmende Kapazitätswerte den Zeitverbrauch bei der Zyklensuche ansteigen lassen. Setzt man die Kapazitätswerte auf unendlich beziehungsweise einen sehr hohen Zahlenwert, so ist Spin mit den vorgegebenen Hardware-Ressourcen nicht in der Lage, einen Zyklus zu finden. Zu erwähnen bleibt noch, daß bei dieser Versuchsreihe das einfachste Kommunikationsprotokoll betrachtet wurde. Komplexere Kommunikationsprotokolle ziehen auch einen höheren Zeitverbrauch bei der Zyklensuche bei ansteigenden Kapazitätswerten nach sich.

8.4.2.6 Abschließende Gegenüberstellung

In Tabelle 8.11 sind die wesentlichen Bewertungskriterien für die in Spin beziehungsweise Skylla umgesetzten Model Checker einander gegenübergestellt. Die Bewertungskriterien lassen sich dabei in MODELSPEZIFISCHE, die HANDHABUNG BETREFFENDE und DEN ALGORITHMUS BESCHREIBENDE Kriterien unterteilen.

Skylla unterscheidet sich von Spin bezüglich der Modellierungsmächtigkeit erheblich. Zum einen gibt es keine durch die Verwendung von Datenstrukturen wie char fest eingebaute Beschränkungen der Modellgröße. Zum anderen bedingen die syntaktischen Konstrukte des in dieser Arbeit entwickelten Model Checkers keine heimliche Zunahme der Zustandszahl der beinhalteten Automaten. Außerdem verhalten sich atomare Operationen in Skylla wirklich als atomare

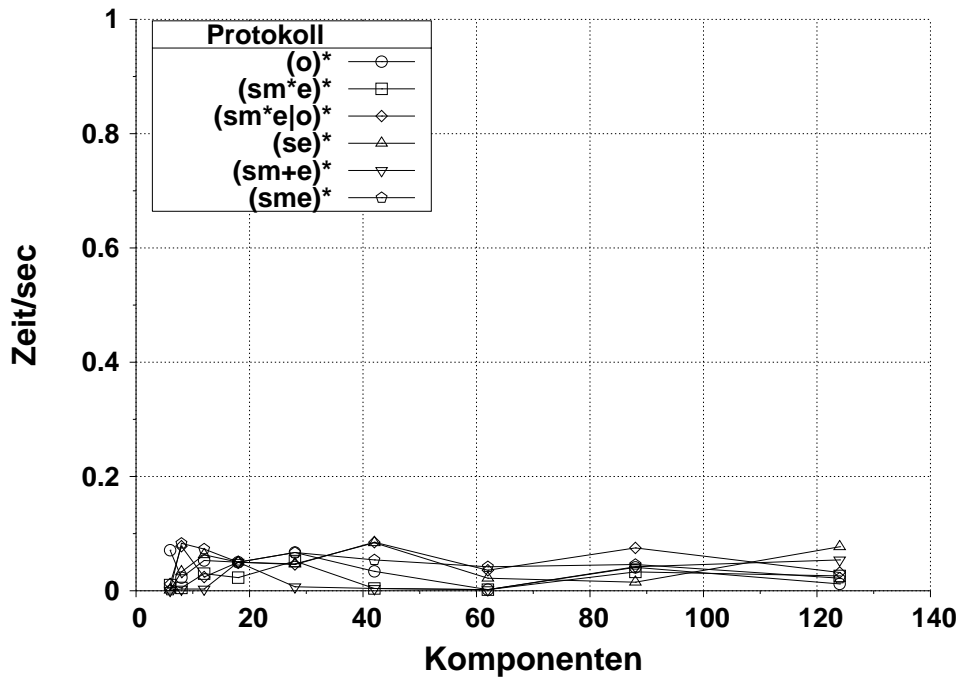


Abbildung 8.54: Zeitverbrauch=F(Komp.) (Net, Breite = 4, Ports = 1, komp. Prot.)

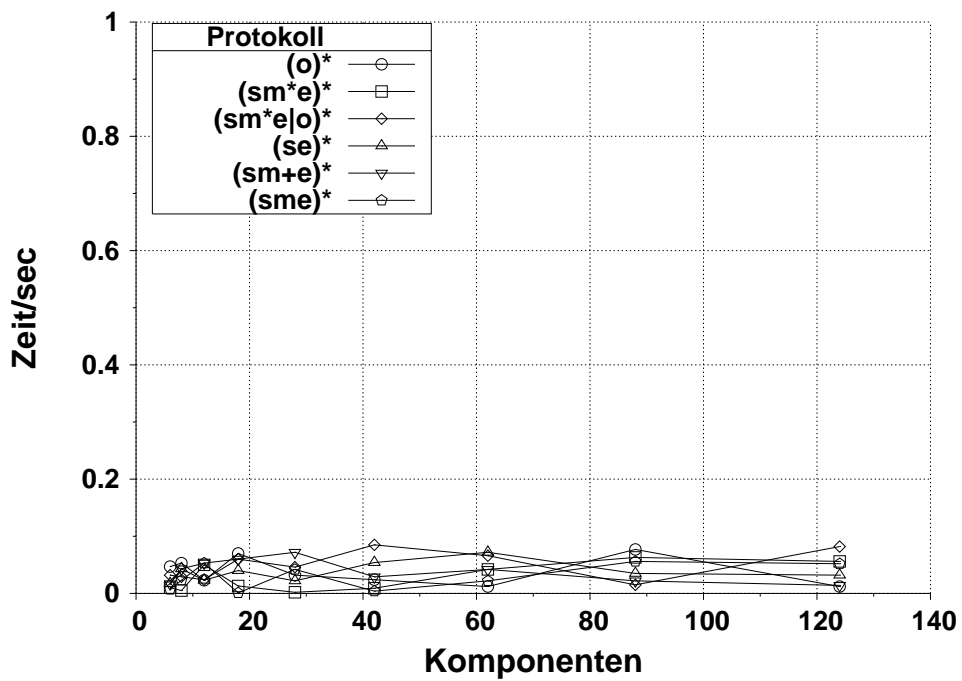


Abbildung 8.55: Zeitverbrauch=F(Komp.) (RowsF, Breite = 2, Ports = 1, ident. Prot.)

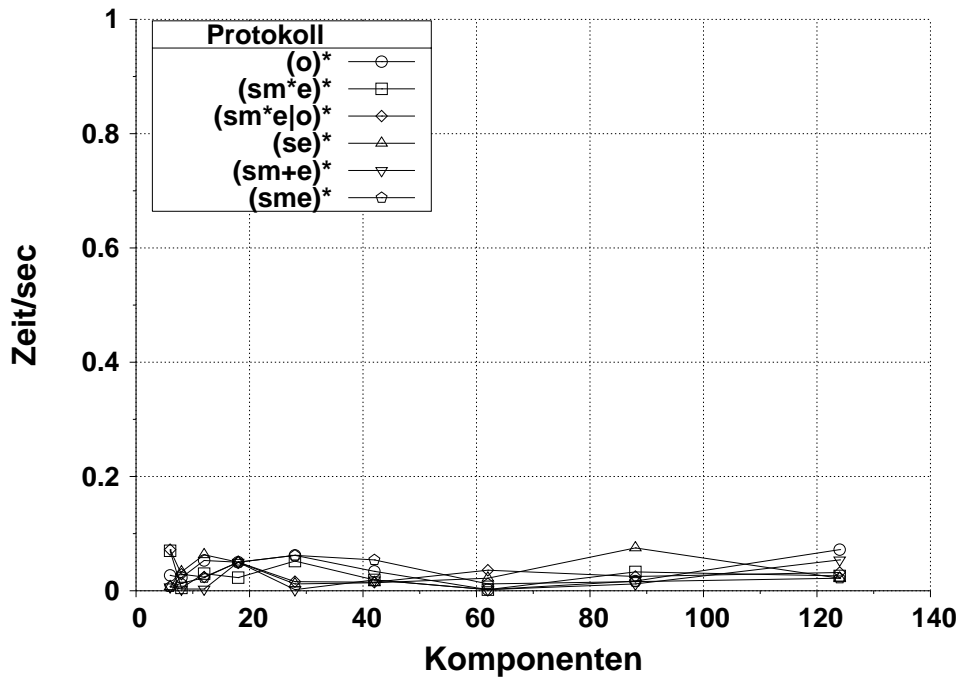


Abbildung 8.56: Zeitverbrauch=F(Komp.) (RowsF, Breite = 2, Ports = 2, ident. Prot.)

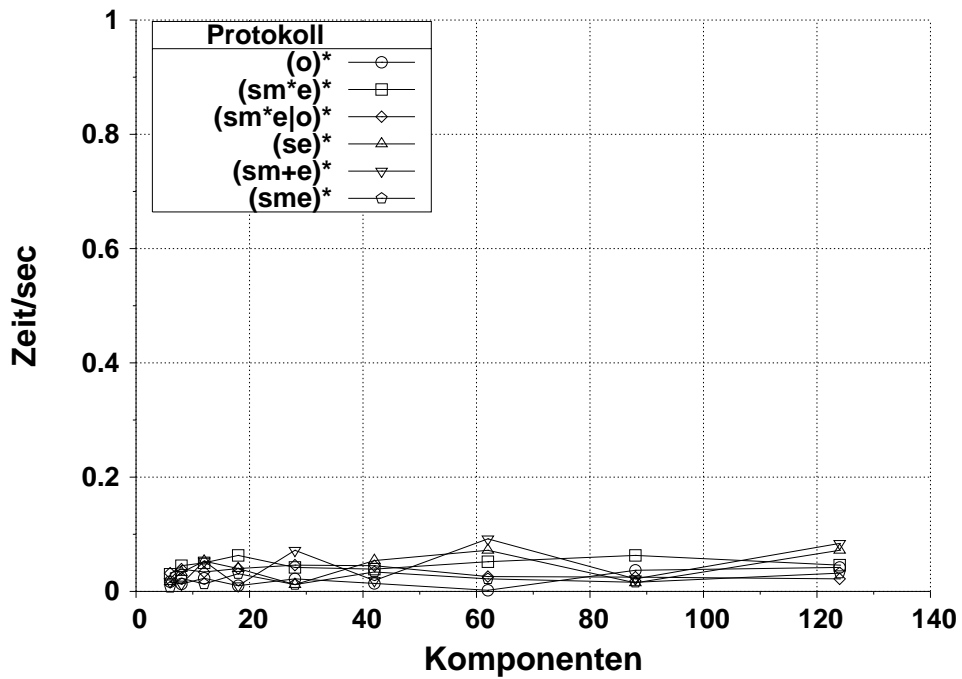


Abbildung 8.57: Zeitverbrauch=F(Komp.) (NetF, Breite = 4, Ports = 1, ident. Prot.)

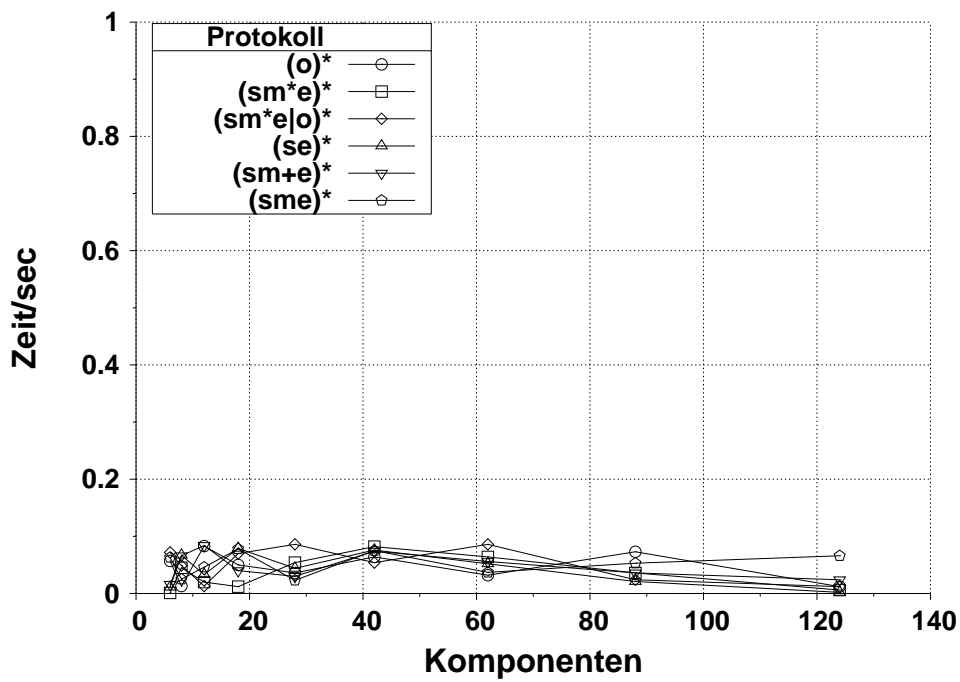


Abbildung 8.58: Zeitverbrauch=F(Komp.) (NetF, Breite = 4, Ports = 2, ident. Prot.)

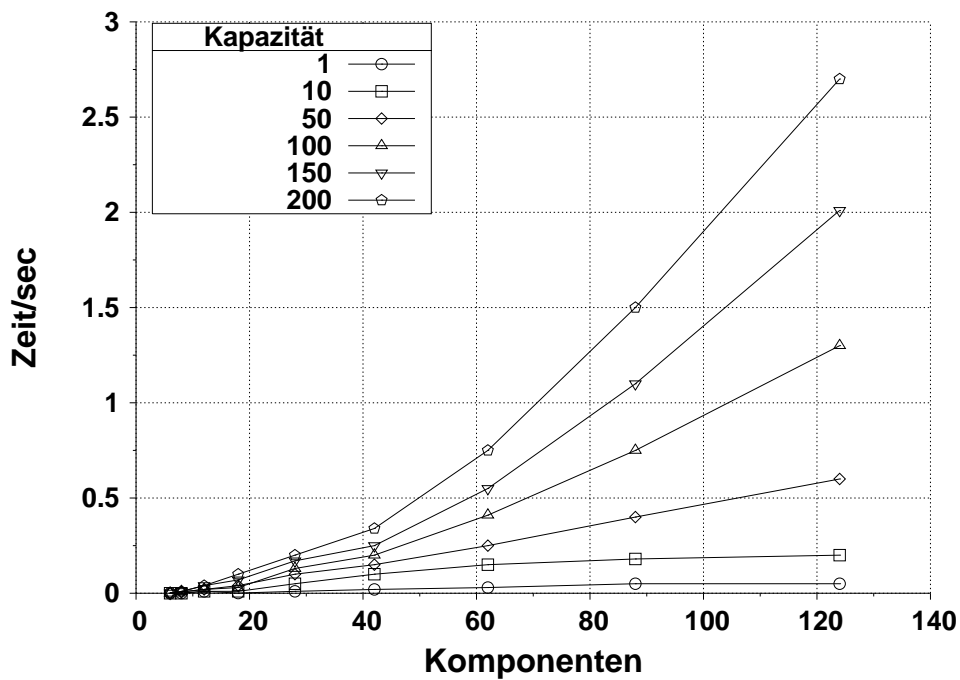


Abbildung 8.59: Zeitverbrauch=F(Komp.) (Rows, Prot. = o*, Breite = 2, Ports = 1, Kanalkapazität, ident. Prot.)

	Skylla	Spin
Modellspezifische Kriterien		
Anzahl Komponenten	Unbegrenzt	127
Anzahl Kanten	Unbegrenzt	256
Kapazitäten	Unbegrenzt	Vom Benutzer möglichst klein einzustellen
Protokolle	Keine Probleme	Schwierigkeiten mit Fifomaten mit mehr als einer ausgehenden Kante pro Zustand
Testmuster	Keine Probleme	Schwierigkeiten mit Fifomaten mit mehr als einer ausgehenden Kante pro Zustand
Atomare Operationen	Möglich	Syntaktisches Konstrukt vorhanden, Ausführungssemantik widerspricht Erwartung
Anzahl Automatenzustände	Entsprechend dem Modell	Unerwartete zusätzliche Zustände durch einige syntaktische Konstrukte
Handhabung		
Benutzerintervention	Keine	In erheblichem Umfang: <ul style="list-style-type: none"> • Ausführen des Programms in drei Schritten <ul style="list-style-type: none"> – Generieren des Modells – Compilieren – Ausführen • Speicheraufteilung per Hand (setzt Kenntnis des Ergebnisses voraus) • Herantasten an Ausgabe durch Anpassen der Zahl der auszugebenden Meldungen
Suchalgorithmus		
Suchstrategie	Guided Search	Tiefensuche mit manueller Beschränkung der Suchtiefe plus weitere nicht dokumentierte heuristische Ansätze
PARTIAL ORDER REDUCTION	Pfadreduktionsmechanismus (vergleiche Kapitel 6)	STATEMENT MERGING plus ein sich von dem in dieser Arbeit entwickelten Verfahren unterscheidender Algorithmus zur Pfadreduktion
Allgemeingültigkeit	Spezialisiert auf Suche nach Zyklen in Wurzelknoten eines Erreichbarkeitsgraphen	Mit Hilfe von Never-Claims beziehungsweise LTL-Formeln Spezifikation komplexer Suchanfragen möglich

Tabelle 8.11: Abschließende Gegenüberstellung von Skylla und Spin

Operationen. Betrachtet man die Handhabung, so ist bei dem in dieser Arbeit entwickelten Model Checker kein umständlicher Übersetzungsschritt notwendig. Die Speicherverwaltung erfolgt dynamisch durch das Programm und muß nicht – wie in Spin üblich – aufwendig durch den Benutzer unter Vorwegnahme des Model-Checking-Ergebnisses durchgeführt werden.

Der in dieser Arbeit konzipierte Suchalgorithmus `guidedSearch()` ist auf die Analyse von Datenflußgraphen zugeschnitten und damit wesentlich erfolgreicher bei der Bestimmung von den Wurzelknoten beinhaltenden Zyklen in dem Erreichbarkeitsgraphen als Spin. In Spin kann man dagegen unterschiedlichste Suchkriterien mittels Never-Claims beziehungsweise LTL-Formeln definieren. Der in Spin realisierte Suchalgorithmus ist als Tiefensuche mit manuell einstellbarer Tiefensuchgrenze beschrieben [Hol04]. Allerdings weicht das beobachtete Verhalten gelegentlich von dieser Beschreibung ab (vergleiche Abschnitt 8.4.2.1).

8.5 Ablaufsteuerung

Im folgenden wird anhand mehrerer Fallstudien der Einsatz des Bild- und Signalverarbeitungswerkzeuges *Skylla* untersucht. Dabei wurden die einzelnen Datenflußkomponenten durch Komponentenprogrammierer implementiert und der Gesamtalgorithmus in Form eines Datenflußgraphen durch einen Anwendungsprogrammierer mittels visueller Programmierung zusammengefügt. Es kamen an mehreren Stellen von Forwiss Passau entwickelte Bildverarbeitungsfunktionen in Form von Shared Libraries zum Einsatz. Außerdem dienten bei einigen Datenflußgraphen entsprechende lconnect-Datenflußgraphen als Vorbild. Bei den Testläufen wurden bereits vorhandene Bild- und Signaldaten verwendet. Für die Zurverfügungstellung all dieser Materialien sei Herrn Dipl.-Inform. Reiner Kickingeder und Herrn Prof. Dr. Donner beziehungsweise Micro-Epsilon Meßtechnik GmbH & Co. KG gedankt.

8.5.1 Fallstudie: Projektive Rekonstruktion in der Stereobildverarbeitung

In dieser ersten Fallstudie wurde ein im Rahmen der High Tech Offensive Bayern (Projekt Nummer 13: „Graphisch programmierte Softwarebausteine“) entwickelter Algorithmus zur projektiven Rekonstruktion in der Stereobildverarbeitung [Fau93, GHW04] in *Skylla* implementiert [FK04]. Das Ziel dieses Verfahrens ist die projektive Rekonstruktion einer räumlichen Anordnung mit Hilfe zweier Kameras. Projektiv bedeutet dabei, daß die Verhältnisse der Punkte im Raum zueinander bis auf ein skalares Vielfaches der Realität bestmöglich entsprechen. Der Algorithmus soll es beispielsweise einem beweglichen Roboter erlauben, sich in seiner Umgebung zu orientieren und zu navigieren. Abbildung 8.60 zeigt die Anordnung der beiden Kameras zur Vermessung des Raumes. Beide Kameras erfassen einen gemeinsamen Bildbereich. Zu einem Zeitpunkt wird dabei jeweils ein Bild digitalisiert und an den Datenflußgraphen übergeben.

Der in *Skylla* entwickelte Datenflußgraph ist in Abbildung 8.61 dargestellt. Beide Bilder werden in den Datenflußgraphen übernommen und mittels Datenflußkomponenten vom Typ `FilterImage` gefiltert, um störendes Rauschen zu eliminieren. Anschließend extrahiert die Datenflußkomponente `PointExtract` markante Punkte. Dazu wird das Bild in 3×3 -Felder zerlegt und jeweils ein Punkt nach dem Harris-Kriterium ermittelt. Die Datenflußkomponente

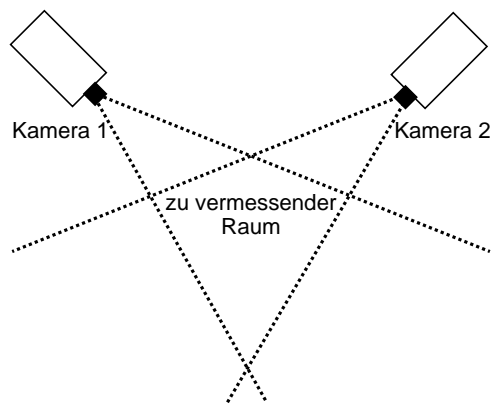


Abbildung 8.60: Meßanordnung (Projektive Rekonstruktion in der Stereobildverarbeitung)

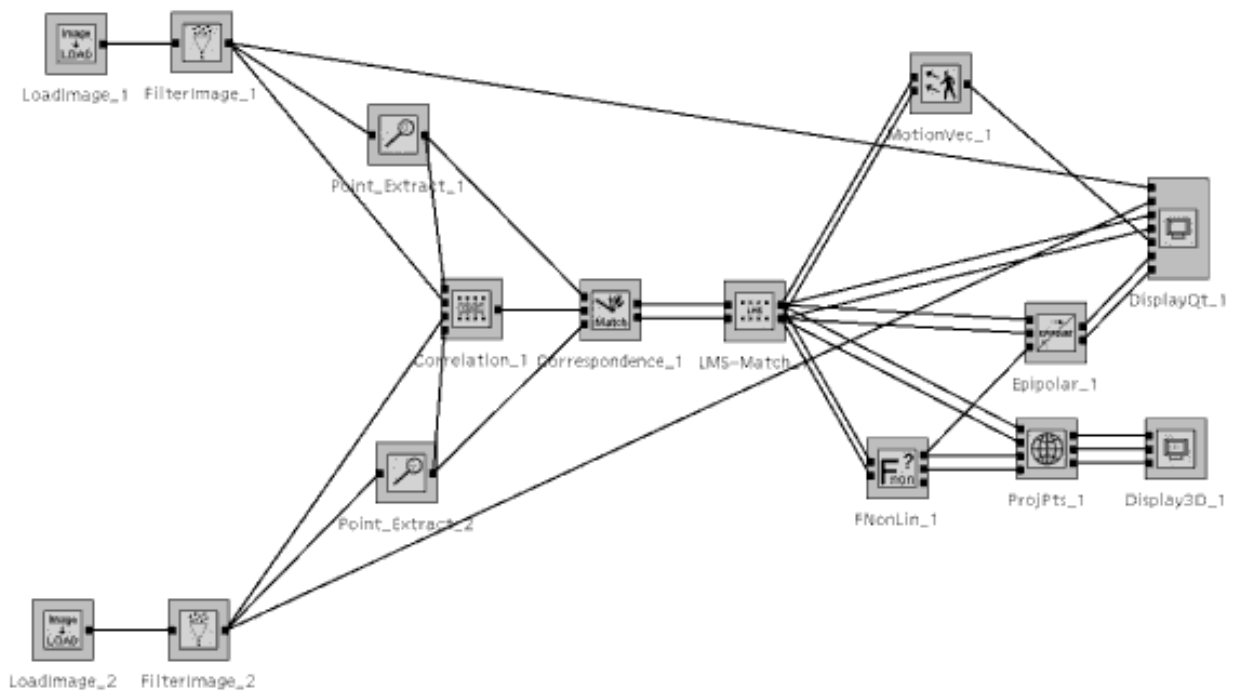


Abbildung 8.61: Datenflußgraph (Projektive Rekonstruktion in der Stereobildverarbeitung)

Correlation setzt die markanten Punkte der beiden Bilder zueinander in Beziehung und gibt eine Kreuzkorrelationsmatrix an seine Nachfolgerkomponenten weiter. Die Datenflußkomponente Correspondence eliminiert alle markanten Punkte, deren Korrelationswert unterhalb eines vom Benutzer vorgegebenen Schwellwertes liegt. Dabei werden aber auch zusätzliche Bedingungen, die zum Beispiel die Nachbarschaftsbeziehung der Punkte betreffen, überprüft. Mit Hilfe der Datenflußkomponente LMS-Match sucht man weitere fehlerhafte Zuordnungen von Punkten zu eliminieren. Es werden korrespondierende Punktepaare, deren Abstände zu den epipolaren Li-



Abbildung 8.62: Überlagerte Bilder zweier Kameras mit Verschiebungsvektoren

nien (siehe unten) signifikant hoch sind, entfernt. Dazu wird iterativ eine Fundamentalmatrix geschätzt. LMS steht für Least Median of Squares. Die Datenflußkomponente **MotionVec** berechnet die Verschiebungsvektoren zwischen den verbleibenden korrespondierenden markanten Punkten. In der Komponente **FNonLin** wird eine 3×3 -Fundamentalmatrix [Fau93] berechnet. Dabei spielen Daten der Kamerakalibrierung wie

- extrinsische Parameter: Position der Kameras im Raum wie zum Beispiel Verschiebung und Rotation zu einem Bezugskordinatensystem
- intrinsische Parameter: Aussagen über die Optik der Kamera wie beispielsweise radiale Verzeichnung, kissenartige oder trapezförmige Verzerrungen

eine zentrale Rolle. Die Datenflußkomponente **Epipolar** berechnet zu jedem markanten Punkt dessen epipolare Linie. Auf dieser Linie liegt der zugehörige korrespondierende Punkt, falls die verwendete Kamera sich annähernd wie eine Lochkamera verhält. Für diese Berechnung wird die Fundamentalmatrix ermittelt. Alle epipolaren Linien in einem Bild schneiden sich dabei in einem Punkt, der als **EPIPOL** bezeichnet wird. Er ist das Bild des einen Kamerazentrums bezüglich

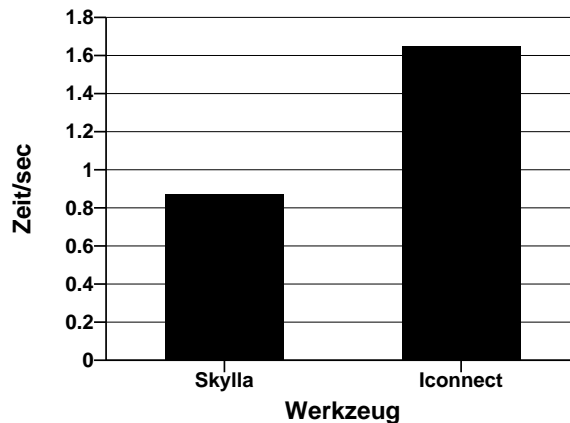


Abbildung 8.63: Vergleich Skylla und Iconnect (Projektive Rekonstruktion in der Stereobildverarbeitung)

der anderen Kamera. Die Datenflußkomponente ProjPts rekonstruiert 3D-Punkte aus den korrespondierenden Punktepaaren, welche mit Hilfe von Display3D dreidimensional dargestellt werden. Die Datenflußkomponente DisplayQt visualisiert Bilder, Punkte, Vektoren und Linien im zweidimensionalen Raum. Abbildung 8.62 zeigt die überlagerten Bilder zweier Kameras und die berechneten Verschiebungsvektoren zwischen den Bildern.

In Abbildung 8.63 sind die Gesamtlaufzeiten des Datenflußgraphen unter Skylla und unter Iconnect [MRSS01] einander gegenübergestellt. Es wurde dabei ein Rechner vom Typ C (siehe Tabelle 8.1) verwendet. Es ist zu beachten, daß Iconnect unter Windows 2000 läuft, während Skylla ein Unix-basiertes Werkzeug ist. Unter Skylla benötigt der Datenflußgraph ungefähr die Hälfte der Zeit wie unter Iconnect.

Da Skylla auch die verteilte Ausführung von Datenflußgraphen unterstützt, wird in Abbildung 8.64 die Ausführungszeit auf einem Rechner vom Typ C (siehe Tabelle 8.1) der Ausführungszeit auf zwei Rechnern, wobei einer vom Typ C und einer vom Typ D ist, einander gegenübergestellt. Die Graphik zeigt die weitere Beschleunigung der Ausführung, welche aber aufgrund der geringen Parallelisierbarkeit des Datenflußgraphen nur bei circa 19 Prozent liegt.

Abbildung 8.65 zeigt die Ausführung des Datenflußgraphen auf einer Menge von Rechnern vom Typ B (vergleiche Tabelle 8.1). Die gesamte Eingabe des Datenflußgraphen umfaßt dabei 50 Bilder, die mit einer Abtastperiode von 0.1, 0.5 beziehungsweise 1 Sekunde erfaßt wurden. Obwohl in diesem Fall eine größere Anzahl von Rechnern zur Verfügung stand, ergab sich aufgrund der eingeschränkten Parallelisierbarkeit durch die Verwendung von mehr als drei Rechnern keine nennenswerte Verbesserung mehr.

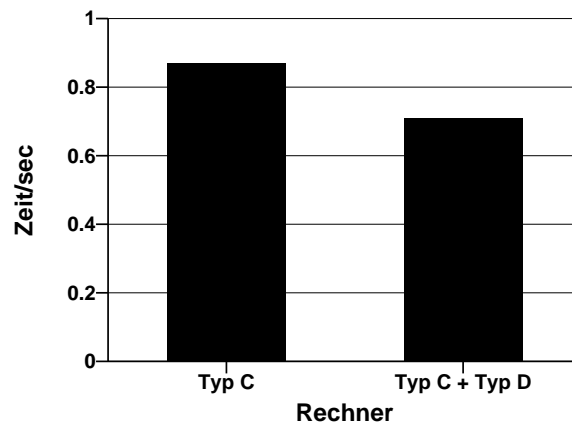


Abbildung 8.64: Verwendung mehrerer Rechner (Projektive Rekonstruktion in der Stereobildverarbeitung)

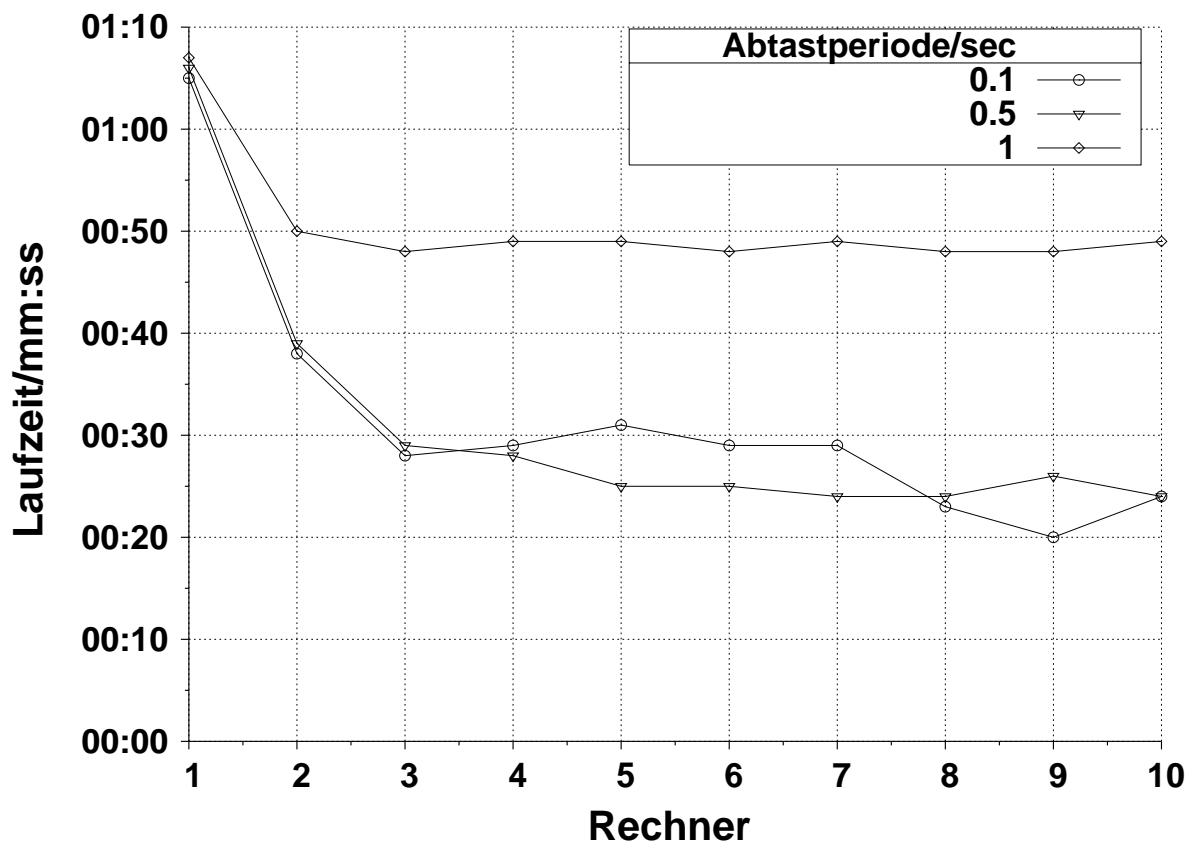


Abbildung 8.65: Ausführungszeit = F(Rechner vom Typ B) (Bilder = 50, Abtastperiode) (Projektive Rekonstruktion in der Stereobildverarbeitung)

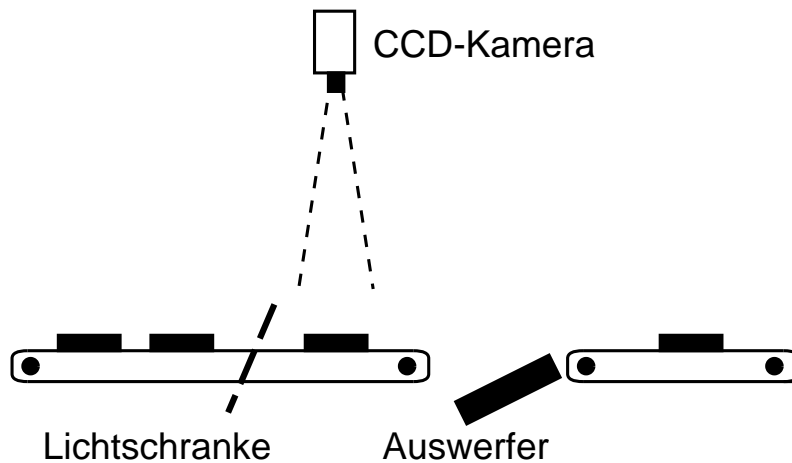


Abbildung 8.66: Meßanordnung (Qualitätskontrolle von Beilagscheiben)

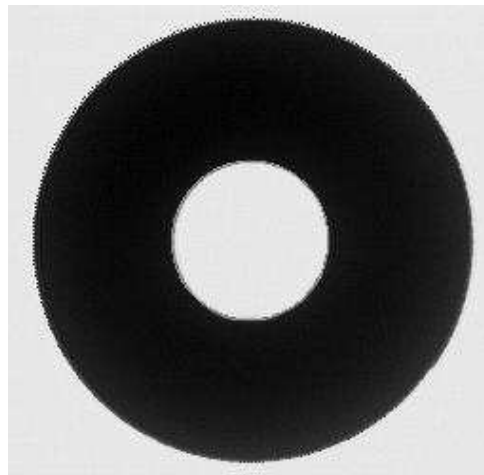


Abbildung 8.67: Eingabebild (Qualitätskontrolle von Beilagscheiben)

8.5.2 Fallstudie: Qualitätskontrolle von Beilagscheiben

In dieser Fallstudie wurde ein Algorithmus zur Qualitätskontrolle von Beilagscheiben implementiert [SH04]. Das Ziel dieses Datenflußgraphen ist das Erkennen und Entfernen von Beilagscheiben minderer Qualität aus dem Produktionsprozess (siehe Abbildung 8.66). In diesem Zusammenhang gilt mein Dank neben den in der Einführung dieses Abschnittes genannten Personen der Firma Micro-Epsilon Meßtechnik GmbH & Co. KG. für die Zurverfügungstellung des entsprechenden Iconnect-Datenflußgraphen.

In dieser Fallstudie werden Bilder der Beilagscheiben (vergleiche Abbildung 8.67) aufgenommen und mittels eines Datenflußgraphen analysiert (vergleiche Abbildung 8.68). Die Datenflußkomponente LoadImage lädt diese Bilder in den Datenflußgraphen. Außerdem wer-

den die Kalibrierungsdaten der Kamera aus einer Datei mit Hilfe der Datenflußkomponente `LoadCalibration` eingelesen. Dabei werden die intrinsischen und extrinsischen Parameter (vergleiche Abschnitt 8.5.1) getrennt weitergegeben. Mit Hilfe der Datenflußkomponente `SubSample` wird das für die Berechnung verwendete Bild verkleinert. Dadurch wird die Analyse beschleunigt. Diese Verkleinerung geschieht durch Unterabtastung des Bildes, was zu einer Reduzierung der Auflösung führt. Die Datenflußkomponente `ImageOp` errechnet aus einem Grauwertbild ein Schwarz/Weiß-Bild. Eine Koordinatenliste aller Pixel, die von 0 verschieden sind, liefert die Datenflußkomponente `ContourList`. Der aufgrund der Unterabtastung des Bildes verfälschte Schwerpunkt wird in dieser Datenflußkomponente auf die exakten Koordinaten verschoben. `FindCenter` errechnet den Schwerpunkt einer Menge von Punkten. Die Datenflußkomponente `Fan` ermittelt in einem Bild entlang von Strahlen Grauwertübergänge, die in einem Punktearray weitergegeben werden. Die Strahlen sind radial um einen Startpunkt angeordnet. Durch diese strahlenförmige Abtastung des Bildes werden die inneren und äußeren Umrisse der Beilagscheibe ermittelt. Die Datenflußkomponente `CalibFeat` rechnet Punkte zwischen dem Bild- und dem Weltkoordinatensystem um. Die zweidimensionalen Bildpunkte werden somit in reale Objektkoordinaten transferiert. Die Datenflußkomponente `FitCircle` approximiert einen Kreis durch eine gegebene Punktemenge. Durch die Verwendung realer Objektkoordinaten ist es möglich, die wirkliche Größe des Kreises zu ermitteln. Als Fehlermaße können vom Benutzer der euklidische Abstand zwischen Punkt und Kreis oder eine kombinierte Methode aus algebraischem und euklidischem Abstand verwendet werden. Mit Hilfe der Datenflußkomponente `Distances` werden die Abstände einer Punktemenge von einem Kreis ermittelt und als Array ausgegeben. `CreatImage` erzeugt aus einer oder mehreren Punktelisten ein Bild. Für diese Visualisierung werden aber Bildkoordinaten und keine realen Objektkoordinaten verwendet. Die Komponente `CheckRadius` überprüft den Radius von Kreisen anhand eines Richtwertes und erlaubter Toleranz. Dabei wird `true` ausgegeben, falls die Toleranz nicht überschritten wurde. Im anderen Fall wird `false` als Ergebnis weitergegeben. Die Datenflußkomponente `DigitalDisplay` stellt Boolesche Werte graphisch dar.

Abbildung 8.69 zeigt die Ausführung des Datenflußgraphen zur Qualitätskontrolle von Beilagscheiben auf einer Menge von Rechnern vom Typ B (vergleiche Tabelle 8.1).

Die gesamte Eingabe des Datenflußgraphen umfaßt dabei 50 Bilder, die mit einer Abtastperiode von 0.1, 0.2, 0.5 beziehungsweise 1 Sekunde erfaßt werden. Man kann erkennen, daß eine Verringerung der Abtastperiode bei gleichzeitiger Verwendung mehrerer Rechner eine Beschleunigung der Abarbeitung zur Folge hat. Dies liegt in einer besseren Ausnutzung des PIPELINEEFFEKTES. Bei hohen Abtastperioden warten die Rechner einen großen Teil der Zeit untätig auf neue Eingaben. Bei kleinen Abtastperioden wird die zur Verfügung stehende Rechenleistung optimal ausgenutzt.

8.5.3 Fallstudie: Ermittlung des gemeinsamen Sichtbarkeitsbereichs zweier Kameras

In dieser Fallstudie wurde ein Algorithmus zur Bestimmung des gemeinsamen Sichtbarkeitsbereichs zweier Kameras implementiert [Pra05]. Dieser Algorithmus ist bisher nicht in `lconnect`

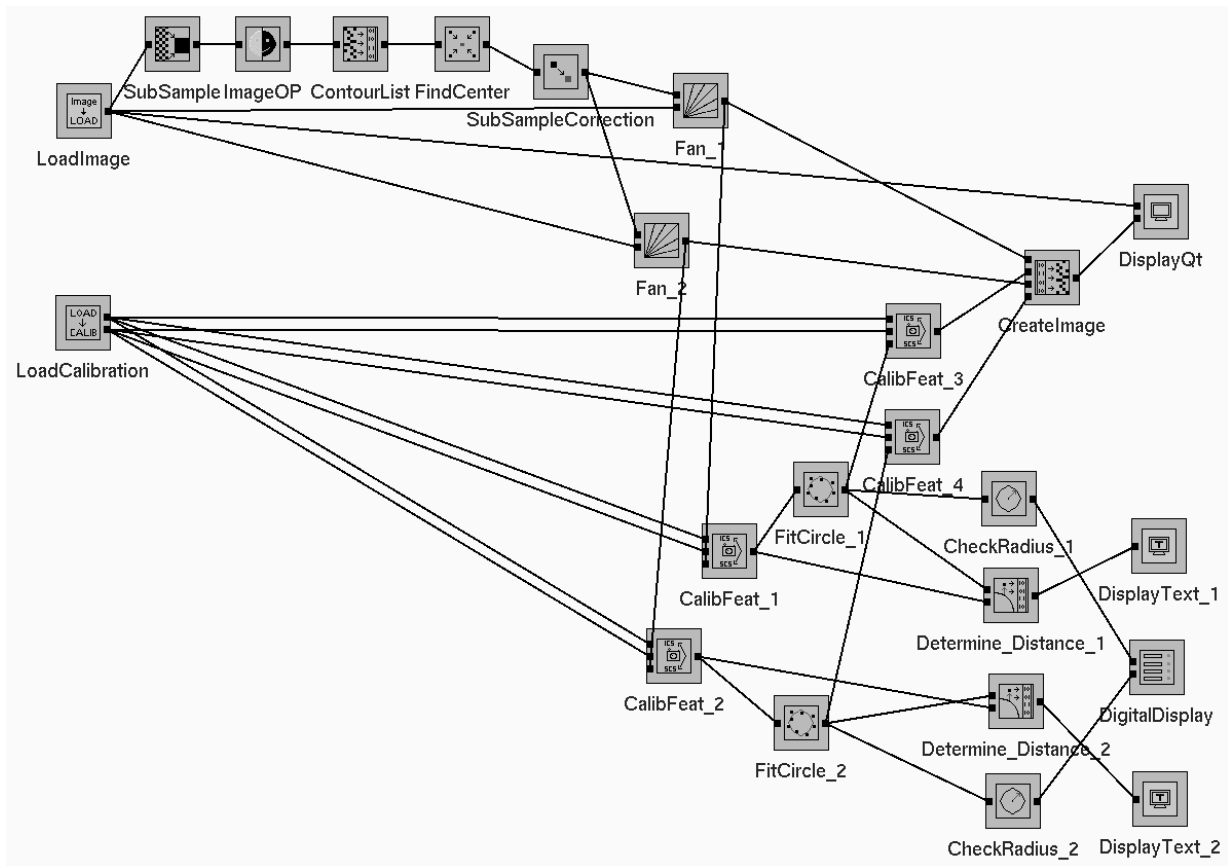


Abbildung 8.68: Datenflußgraph (Qualitätskontrolle von Beilagscheiben)

realisiert. Ziel des Verfahrens ist die Ermittlung einer Ebene mit kleinstmöglichem Abstand zu den Kameras, in der sich die Sichtbarkeitsbereiche zweier Kameras an ihren Rändern berühren.

Abbildung 8.70 zeigt den Versuchsaufbau, wobei die genauen Positionen der Kameras bekannt sind. Dabei werden Sichtstrahlen mit einer Ebene geschnitten, welche iterativ in der Höhe verschoben wird. Für beide Kameras wird jeweils eine konvexe Hülle über die Menge der erhaltenen Schnittpunkte berechnet. Anschließend werden die beiden konvexen Hüllen geschnitten. Die Höhe, in welcher sich die konvexen Hüllen an ihren Rändern berühren, stellt das gesuchte Ergebnis dar.

Abbildung 8.71 zeigt den Datenflußgraphen zur Ermittlung des Sichtbarkeitsbereichs der beiden Kameras. Als Eingabebilder wurden ein Kreis, ein Dreieck und eine Wolke von Punkten verwendet (siehe Abbildung 8.72).

- **VORVERARBEITUNG:** Die Datenflußkomponenten vom Typ **LoadPGM** laden die Bilddateien. Zu den geladenen Bildern werden jeweils in einer Datenflußkomponente vom Typ **ViewingRays** Sichtstrahlen ermittelt. Bei allen Berechnungen in dem Datenflußgraphen wird von einem idealen Kameramodell ausgegangen. Um eine reale Kamera verwenden zu können, muß man in der Datenflußkomponente **ViewingRays** die Daten zur Kamera-

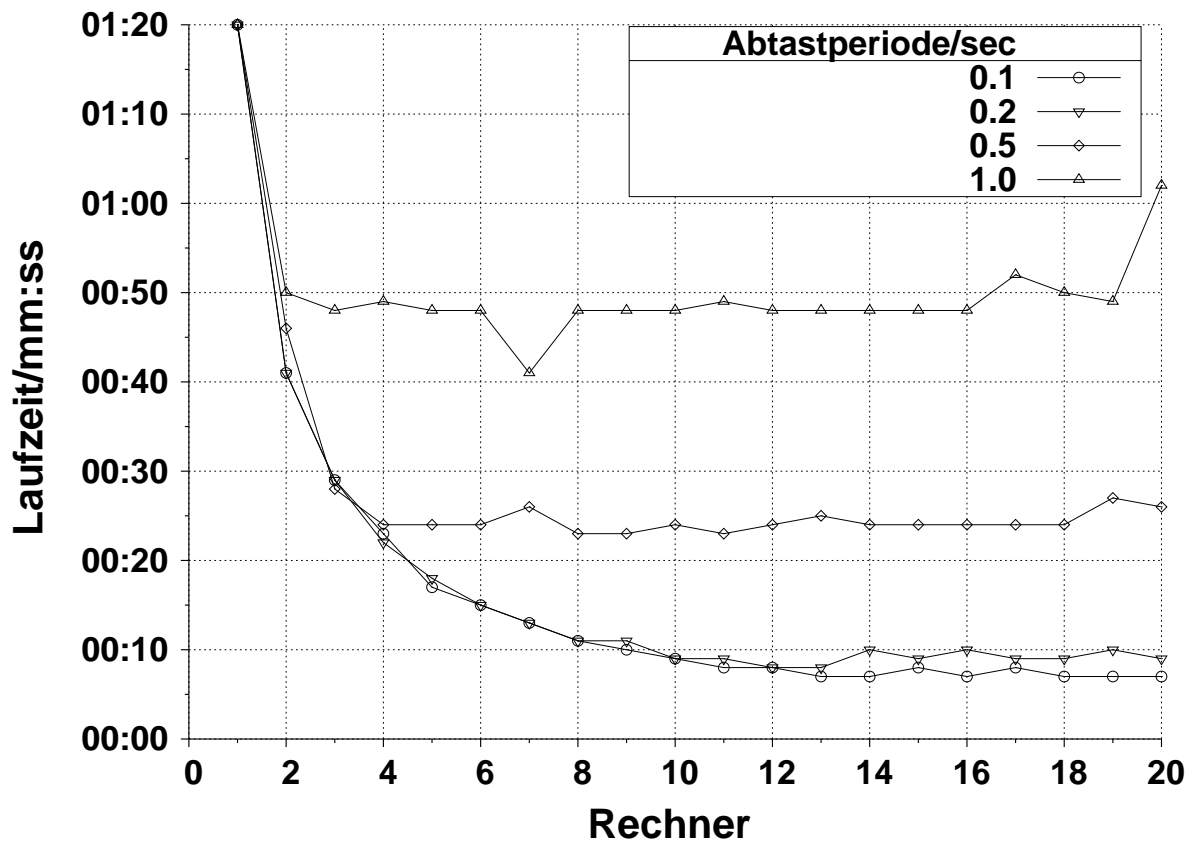


Abbildung 8.69: Ausführungszeit = F(Rechner vom Typ B) (Bilder = 50, Abtastperiode) (Qualitätskontrolle von Beilagscheiben)

kalibrierung einstellen, damit alle Abweichungen vom idealen Modell einer Kamera wie beispielsweise Verdrehungen und Verzerrungen herausgerechnet werden können.

- **BINÄRE SUCHE:** In einer binären Suche, die mit Hilfe von `while`- und `for-next`-Schleifen realisiert ist, wird nun der Abstand z der Ebene ermittelt, in welchem sich die konvexen Hüllen der Schnittpunkte der Sichtstrahlen mit eben dieser Ebene berühren.
 - **WHILE-SCHLEIFEN ZUR ERMITTLUNG DER ERSTEN SCHNITTMENGE:** Diese aus den beiden Bildern ermittelten Sichtstrahlen bilden die Eingabe für jeweils eine `while`-Schleife im Datenflußgraphen. Im folgenden wird nur eine der beiden Schleifen beschrieben, da die andere analog arbeitet. Die Datenflußkomponenten `Adapter_1_1` und `GenOnDemand_1` bewirken, daß Bilder in die `while`-Schleife gelangen. Solange die `while`-Schleife ausgeführt wird, sorgen die Datenflußkomponenten `ArithmeticModule_1`, `Switch_3` und `Select_3` dafür, daß an `PlaneIntersect_1` die Abstände der Ebenen in z -Richtung übergeben werden. Diese werden in jedem Durchlauf verdoppelt und betragen beispielsweise 1, 2, 4, 8. Die Datenflußkompo-

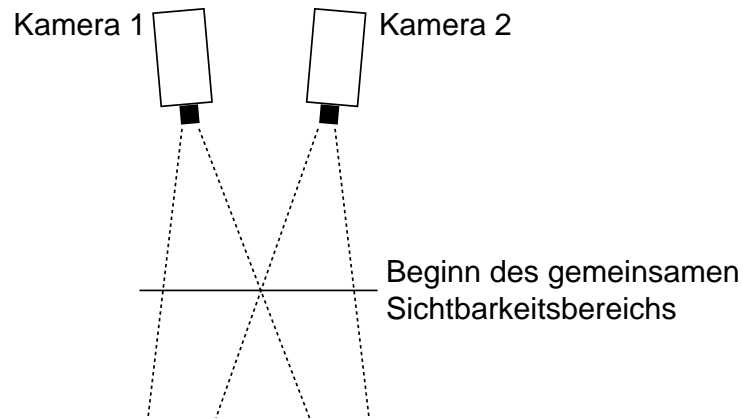


Abbildung 8.70: Meßanordnung (Ermittlung des gemeinsamen Sichtbarkeitsbereichs zweier Kameras)

nente `PlaneIntersect` schneidet ein Bündel von Sichtstrahlen mit dieser sich im Abstand z befindlichen Ebene. Die konvexe Hülle der resultierenden Schnittpunkte wird mit der Datenflußkomponente `ConvexHull` ermittelt. In dieser Datenflußkomponente ist der Algorithmus Graham-Scan implementiert. `ConvexHullIntersect` schneidet die in den beiden `while`-Schleifen ermittelten konvexen Hüllen (vergleiche Abbildung 8.73). `HullArea_1`, `GenOnDemand_2` und `CompareModule_1` überprüfen, ob die Fläche der resultierenden konvexen Hülle größer 0 ist.

- FOR-NEXT-SCHLEIFEN ZUR ITERATIVEN VERKLEINERUNG DER SCHNITTMENGE: Wenn der Schnitt der beiden konvexen Hüllen eine Fläche größer 0 ergibt, wird der zweite Teil des Algorithmus gestartet. Der ermittelte Wert für den Abstand z und die Sichtstrahlen werden in die `for-next`-Schleife übernommen. Es wird eine Kopie z' von z erzeugt. Durch `GenOnDemand_3` und `ArithmeticModule_3` wird die Kopie z' des Abstandswertes jeweils halbiert. Der Wert z' wird rückgekoppelt. Ist die Schnittmenge der beiden konvexen Hüllen leer, wird der halbierte Wert zu z hinzuaddiert. Ansonsten wird z' von z subtrahiert. Dazu werden die Datenflußkomponenten `ArithmeticModule_3` und `ArithmeticModule_4`, `Switch_3` und `Switch_4` und `Select_4` eingesetzt.
- NACHBEARBEITUNG: Sind die `for-next`-Schleifen abgearbeitet, dann wird der ermittelte Wert am oberen Ausgang der Datenflußkomponente `UnZipModule_2` weitergegeben. Am unteren Ausgang der Datenflußkomponente wird der maximale relative Fehler zum gesuchten Wert ausgegeben. Am Ausgang der `for-next`-Schleifen liegen die Sichtstrahlen an, welche mittels einer weiteren `PlaneIntersect`-Komponente mit einer Ebene, deren z -Abstand am Ausgang der Datenflußkomponente `UnZipModule_2` anliegt, geschnitten werden.

Abbildung 8.74 zeigt die benötigte Laufzeit des Datenflußgraphen in Abhängigkeit von der

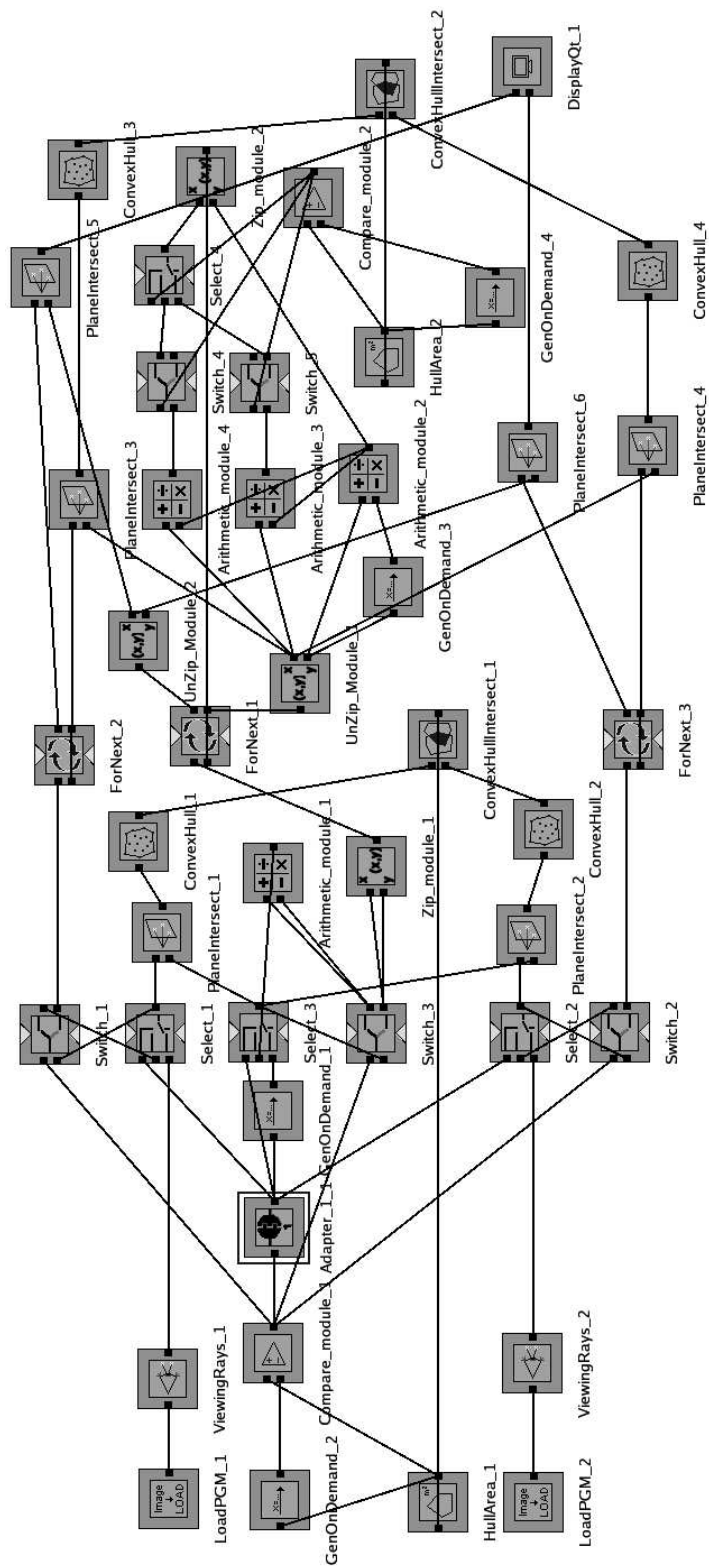


Abbildung 8.71: Datenflußgraph (Ermittlung des gemeinsamen Sichtbarkeitsbereichs zweier Kameras)

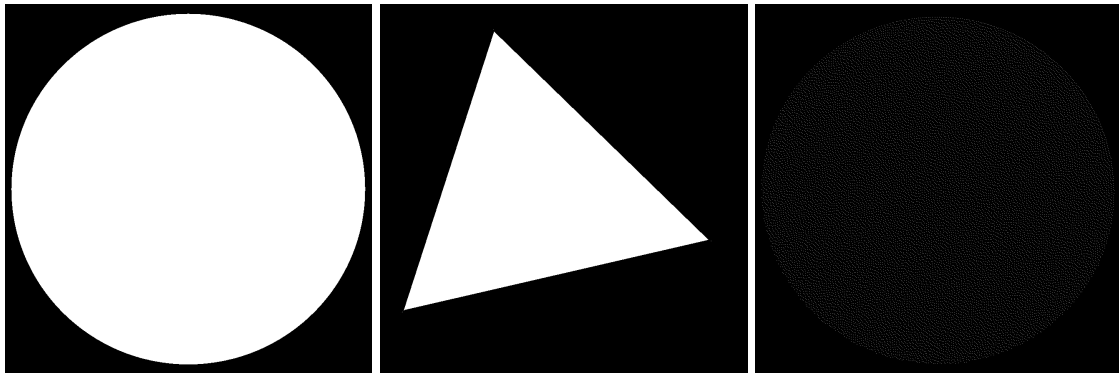


Abbildung 8.72: Eingabedaten (Kreis, Dreieck, Wolke) (Ermittlung des gemeinsamen Sichtbarkeitsbereichs zweier Kameras)

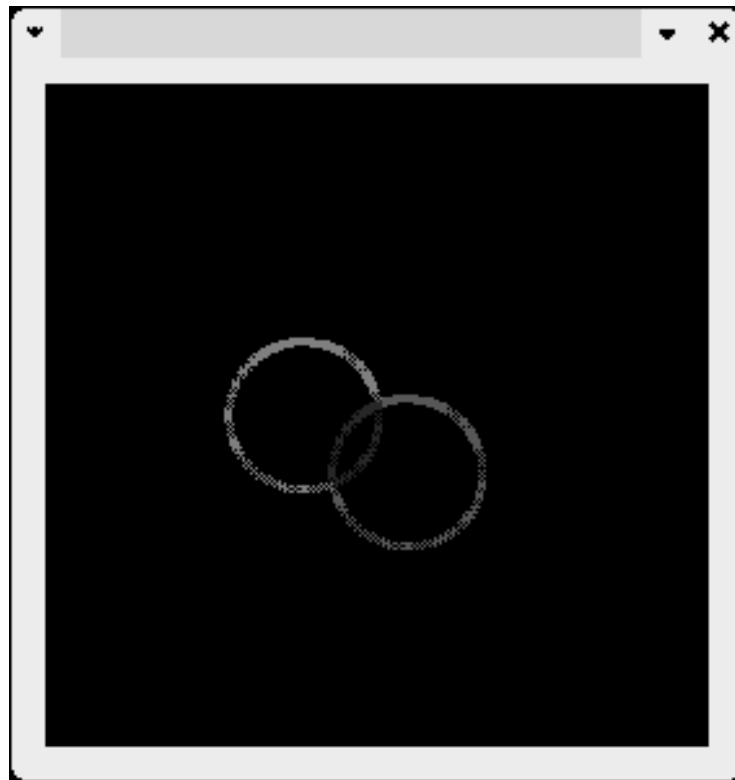


Abbildung 8.73: Schnitt der beiden Sichtbarkeitsbereiche

Anzahl der verwendeten Rechner für die drei Arten von Eingabebildern. Es wurden jeweils 10 Bilder verarbeitet, die mit einer Abtastperiode von 5 Sekunden in den Rechner übernommen wurden. Dabei zeigt sich zum einen, daß die Bestimmung der konvexen Hülle für Dreiecke – wie erwartet – am schnellsten erfolgte und Kreise den größten Rechenaufwand verursachten. Bei

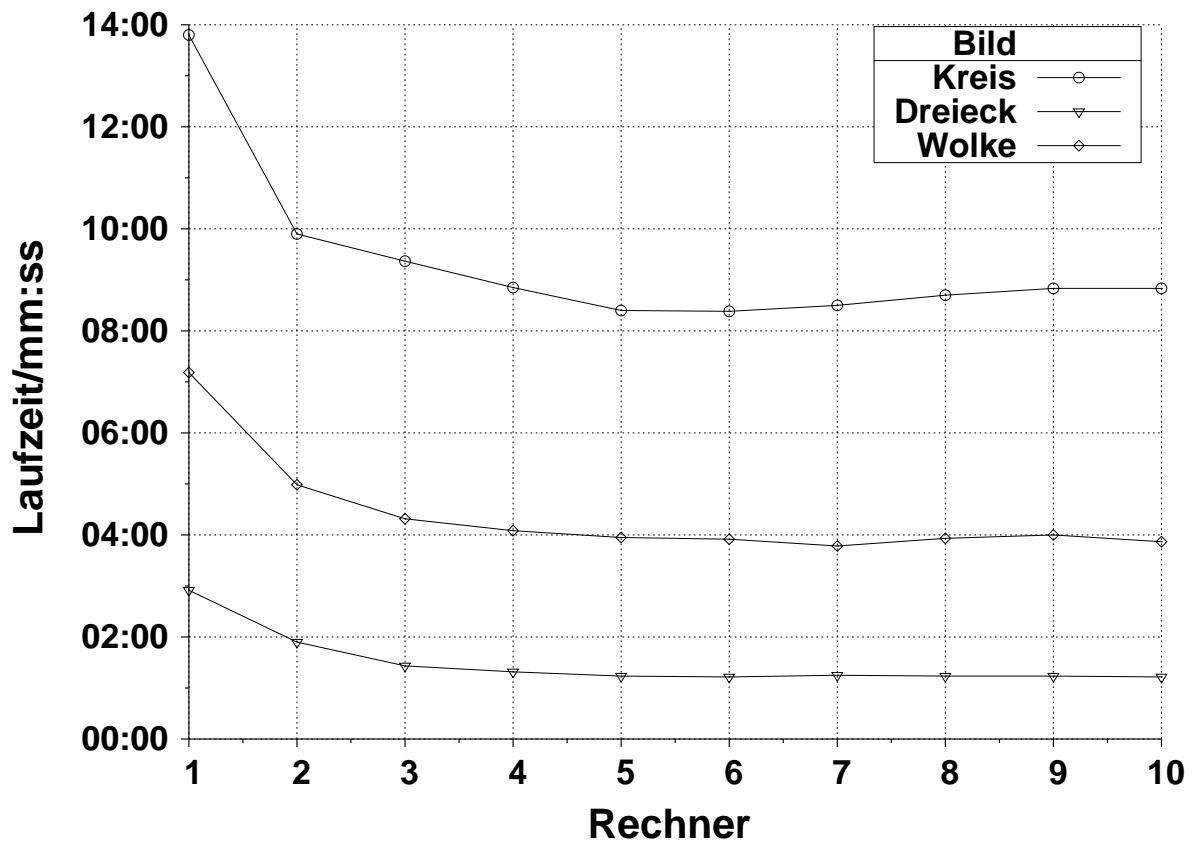


Abbildung 8.74: Ausführungszeit = $F(\text{Rechner vom Typ B})$ (Bilder = 10, Abtastperiode = 5 sec, Eingabedaten) (Ermittlung des gemeinsamen Sichtbarkeitsbereichs zweier Kameras)

der Verwendung von vier oder mehr Rechnern ergibt sich keine Beschleunigung der benötigten Rechenzeit mehr.

In Abbildung 8.76 ist der Speicherverbrauch in Abhängigkeit von der Laufzeit aufgetragen. Dabei wurde eine Abtastperiode von 1 Sekunde verwendet. Insgesamt standen für die Berechnung 10 Rechner vom Typ B (vergleiche Tabelle 8.1) zur Verfügung. Man erkennt, daß nach einem beinahe linearen Anstieg des Speicherverbrauchs dieser in demselben Maße schrumpft. Dies liegt daran, daß die Bilder zuerst eingelesen werden, was man an den kleinen Zacken die der Kurve überlagert sind, erkennen kann, und anschließend eine Informationsverdichtung stattfindet. Das heißt, es werden Punkte aus den Bildern extrahiert und anschließend die Bilder freigegeben. Außerdem beobachtet man wieder den unterschiedlichen Speicherbedarf der verschiedenen Testbilder.

In Abbildung 8.75 ist der Speicherbedarf in Abhängigkeit von der Laufzeit aufgetragen. Dabei wurde die Anzahl der Rechner variiert und die Abtastperiode auf 1 eingestellt. Die Testbilder waren Kreise. Man beobachtet, daß die Verwendung von 5 Rechnern die Rechenzeit im Vergleich zu 1 Rechner beinahe halbiert. Die Verwendung von 10 Rechnern brachte im Vergleich

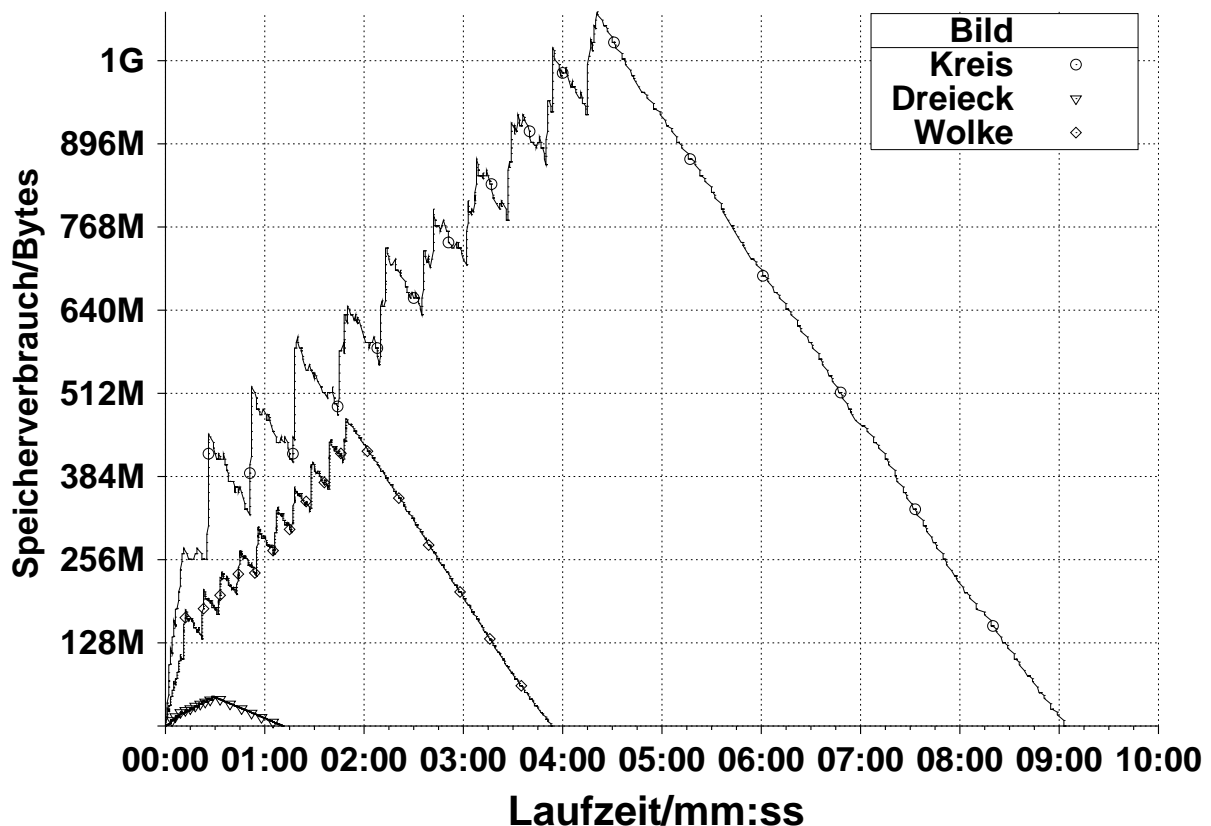


Abbildung 8.75: Speicherverbrauch = F(Laufzeit) (Bilder = 10, Abtastperiode = 1 sec, Eingabedaten) (Ermittlung des gemeinsamen Sichtbarkeitsbereichs zweier Kameras)

dazu keine Verbesserung mehr. Wieder kann man an den kleinen Zacken erkennen, wann die Bilder geladen wurden. Außerdem ist der Gesamtspeicherverbrauch bei 5 oder 10 Rechnern deutlich höher als bei 1 Rechner. Dies liegt daran, daß bei der verteilten Ausführung des Datenflußgraphen Daten redundant abgespeichert werden. Eine Ursache dafür ist die Verwendung von Mehrfachkanten im Datenflußgraphen. Von dem Ausgabeinterface einer Datenflußkomponente gehen mehrere Kanten zu verschiedenen anderen Datenflußkomponenten, welche unter Umständen auf verschiedenen Rechnern ausgeführt werden.

8.6 Zusammenfassung der Ergebnisse

Die in diesem Abschnitt vorgestellten Resultate erfüllen die in Abschnitt 8.1 gestellten Anforderungen:

- **Überprüfen der Effizienz:** Eine wichtige Voraussetzung für den entwurfsbegleitenden Einsatz der vorgestellten Verfahren ist die Geschwindigkeit, mit welcher große Datenflußgraphen untersucht werden können.

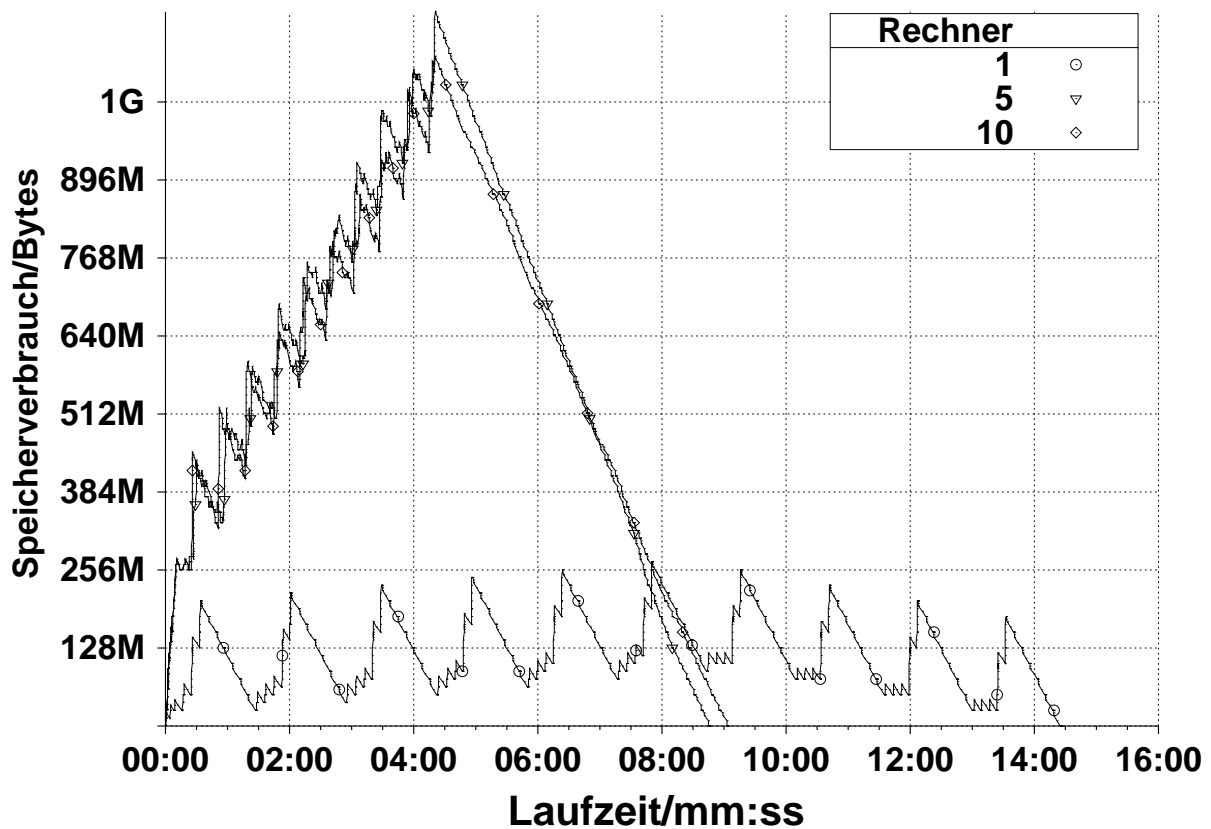


Abbildung 8.76: Speicherverbrauch=F(Laufzeit) (Bilder = 10, Eingabedaten=Kreise, Rechner vom Typ B) (Ermittlung des gemeinsamen Sichtbarkeitsbereichs zweier Kameras)

- INTERFACE-TYPSYSTEM: Es wurden verschiedene Kriterien wie Einfügereihenfolge der Kanten, Komplexität der Typconstraints, die Größe der betrachteten Datenflußgraphen und das Verhalten beim Löschen einer Kante untersucht. Das Fazit dieser Tests ist, daß sich das Interface-Typsystme aufgrund der gemessenen Zeiten hervorragend für den entwurfsbegleitenden Einsatz eignet.
- MODEL CHECKING: In über 160000 Tests wurde der Einfluß diverser Faktoren wie
 1. Graphparameter: Graphstruktur, Graphgröße, Rückkopplungen
 2. Kantenparameter: Kapazität
 3. Komponentenparameter: Kommunikationsprotokolle, Anzahl der Schnittstellen, Gewichte
 4. Algorithmusparameter: Aktivierung des BLOCKIER/DEBLOCKIER-Mechanismus, Aktivierung der PARTIAL ORDER REDUCTION, Anzahl der Bits für Teilschlüssel der Datenstruktur Ordered Maps
 untersucht. Es zeigte sich, daß sich mit Hilfe dieses Model-Checking-Verfahrens –

vor allem auch wegen der Wirksamkeit der in dieser Arbeit entwickelten Verfahren der PARTIAL-ORDER-REDUCTION und des BLOCKIERENS/DEBLOCKIERENS – große Datenflußgraphen in ausreichend kleinen Zeiten nach Zyklen im Erreichbarkeitsgraphen analysieren lassen. Damit eignet sich dieses Verfahren in besonderer Weise für den entwurfsbegleitenden Einsatz.

- **Herstellen der Relation zum Stand der Technik:** Um festzustellen, inwieweit durch die in dieser Arbeit entwickelten Verfahren eine Verbesserung zum aktuellen Stand der Technik gegeben ist, wurden etliche FALLSTUDIEN und ein TOOLVERGLEICH durchgeführt.

- INTERFACE-TYPSYSTEM: Es gibt zur Zeit kein wirklich vergleichbares Interface-Typsystem für datenflußorientierte komponentenbasierte eingebettete Systeme (vergleiche Abschnitt 2.3). Daher konnte auch keine Gegenüberstellung zum Beispiel hinsichtlich zeitlicher Effizienz durchgeführt werden.
- MODEL CHECKING: Der in dieser Arbeit entwickelte Model Checker wurde dem Model Checker Spin gegenübergestellt. Umfassende Untersuchungen haben ergeben, daß das hier umgesetzte Verfahren im genannten Anwendungsgebiet der Bild- und Signalverarbeitung für datenflußorientierte eingebettete Systeme Spin in weiten Teilen überlegen ist:
 - * So können größere Datenflußgraphen als in Spin untersucht werden.
 - * Im Gegensatz zu Spin müssen die Kapazitäten der Fifos nicht vom Benutzer auf kleine Werte eingestellt werden.
 - * Es können zudem Datenflußkomponenten mit komplexeren Kommunikationsprotokollen untersucht werden.

Weitere Punkte finden sich in Abschnitt 8.4.2.

- ABLAUFSTEUERUNG: Die Ablaufsteuerung von Skylla wurde mit der von lconnect verglichen. Dabei zeigte sich, daß Skylla aufgrund der automatischen verteilten Ausführung und der optimierten Kommunikation via Shared Memory beziehungsweise TCP/IP bei der Ausführung von Datenflußgraphen effizienter war (vergleiche Abschnitt 7.3.3.3).

- **Veranschaulichen des Verhaltens an Fallstudien:**

- INTERFACE-TYPSYSTEM: Es wurde ein Datenflußgraph zur Qualitätskontrolle von Getränkedosen zum Testen des Interface-Typsystms realisiert. Die erzielten Ergebnisse unterstreichen die Resultate der systematischen Tests.
- ABLAUFSTEUERUNG: Es wurden drei Fallstudien, nämlich zur projektiven Rekonstruktion in der Stereobildverarbeitung, zur Qualitätskontrolle von Beilagscheiben und zur Ermittlung des gemeinsamen Sichtbarkeitsbereichs zweier Kameras, durchgeführt. Die erzielten Ergebnisse unterstreichen die Effizienz der verteilten Ausführung.

Damit kann insgesamt festgestellt werden, daß die in dieser Arbeit vorgestellten Modelle und Verfahren insbesondere einen wichtigen Beitrag zur ENTWURFSBEGLEITENDEN QUALITÄTSSICHERUNG im Softwareentwurf für eingebettete Systeme darstellen.

Kapitel 9

Zusammenfassung und Ausblick

Man hat sich bemüht. [Willy Brandt]

Ausgehend von der dieser Arbeit zugrundeliegenden Problemstellung stellt dieses Kapitel deren Lösung zusammenfassend dar. Der Schwerpunkt liegt auf den verwendeten Modellen und Verfahren, wobei deren Grenzen ausgelotet werden und deren Überprüfung kurz beschrieben wird. Die Bedeutung dieser Arbeit für das Forschungsgebiet und die Praxis steht nach einer überblicksartigen Darstellung der innovativen Aspekte im Zentrum. Ein kurzer Abriß über den Einsatz der vorgestellten Modelle und Verfahren in anderen Anwendungsgebieten rundet dieses Kapitel und diese Dissertation ab.

9.1 Wiederaufgreifen der Problemstellung

Das Thema dieser Arbeit lautet „Komponentenbasierte Softwareentwicklung für datenflußorientierte eingebettete Systeme“ (vergleiche Kapitel 1). Dabei sollen Probleme, die durch das Erkennen von Fehlern in späten Softwareentwicklungsphasen entstehen und die unter Umständen nicht nur erhebliche Kosten zur Folge haben sondern auch Unfälle verursachen können, zum frühestmöglichen Zeitpunkt während des Entwurfs vermieden werden. Zusätzlich zu dieser entwurfsbegleitenden frühen Qualitätssicherung war ein weiterer Schwerpunkt die effiziente Kommunikation zwischen den Datenflußkomponenten.

9.2 Problemlösung

Die Problemlösung basiert auf einer Menge aufeinander aufbauender Modelle und dazu passender Verfahren. Dieser Abschnitt soll meinen eigenen Beitrag und dessen Neuartigkeit hervorheben.

9.2.1 Modelle

Zur Lösung der genannten Problematik wurden mehrere aufeinander aufbauende Modelle entwickelt:

- **MODELL PHYSIKALISCHER SIGNALE:** Bei dem in Kapitel 3 vorgestellten Signalmodell handelt es sich um ein PRAXISNAHES MODELL das explizit zur Unterstützung entwurfsbegleitender Fehlererkennung konzipiert ist, indem es die Definition von Typconstraints und Kommunikationsprotokollen erlaubt. Die Beschreibungsmöglichkeiten von physikalischen Signalen in diesem Modell sind SEHR FLEXIBEL, weil Zeitsignale mit äquidistanter, nichtäquidistanter und auf Signalsegmente bezogener äquidistanter Abtastung dargestellt werden können. Zudem ist es möglich, physikalische Signale mit unterschiedlichen Definitionsbereichen wie zum Beispiel Zeitbereich und Frequenzbereich zu erfassen. Der TRANSPORT-OVERHEAD VERMINDERT sich erheblich, indem zwischen relevanten und irrelevanten Signaldaten unterschieden wird und nur relevante Signaldaten übertragen werden. Eine weitere Maßnahme zur Datenreduktion ist die Rekonstruktion des Zeitbereichs jedes Signalsegments aus dessen Startzeitpunkt und Abtastperiode. Dieses Signalmodell beschreibt physikalische Signale als Ströme gefärbter Token. Das Signalmodell unterstützt UNTERSCHIEDLICHE ABSTRAKTIONSEBENEN. Eine detailliertere Sicht der Token umfaßt sechs Signalmerkmale, die den Definitionsbereich, den Wertebereich und Aspekte des Datentransports physikalischer Signale beschreiben. Diese Sicht ist Grundlage des Modells der Interfacetypen und des dazugehörigen Typbestimmungsverfahrens. Eine abstraktere Sicht kennzeichnet Token allein durch ihre Farben. Diese Sichtweise ist die Basis des Modells der Kommunikationsprotokolle und des darauf aufbauenden Model-Checking-Verfahrens.
- **MODELL DER DATENFLUSSKOMPONENTEN:** Die klassischen Datenflußparadigmen SDF, BDF und DDF (siehe Abschnitt 2.2) sind in Kapitel 4 zu den GEFÄRBTEN DATENFLUSS-PARADIGMEN Colored SDF, Colored BDF und Colored DDF erweitert worden, um das neue Signalmodell gewinnbringend bei der Definition von Typconstraints und Kommunikationsprotokollen einsetzen zu können. Durch die abgeleiteten Typconstraints und Kommunikationsprotokolle, welche die WOHLDEFINIERTHEIT der jeweiligen Datenflußkomponenten garantieren, ist eine EFFEKTIVE ENTWURFSBEGLEITENDE FEHLERERKENNUNG möglich. Die explizite Modellierung der Tokenmaschinen der einzelnen Datenflußkomponenten (vergleiche Abschnitt 4.3.2) – wobei alle Signalverarbeitungskomponenten nicht turingäquivalent sind – erlaubt dem Entwickler die ENTSCHEIDUNG ZWISCHEN MODELLIERUNGSMÄCHTIGKEIT UND ANALYSIERBARBEIT bei der Komponentenauswahl. Die vorgestellte funktionale Beschreibung entspricht der denotationellen Semantik der deterministischen Datenflußkomponenten.
- **MODELL DER INTERFACETYPEN:** Durch Betrachtung von sechs Signalmerkmalen, die den Definitionsbereich, den Wertebereich und Aspekte des Datentransports von physikalischen Signalen beschreiben, wird die Zahl der durch ein Typsystem ÜBERPRÜFBAREN EIGENSCHAFTEN erheblich ERWEITERT (vergleiche Kapitel 5). Die Typdomäne ist dabei als

Kreuzprodukt von sechs Typtraits, die als vollständige Verbände modelliert werden, gegeben. Die Ordnung eines Verbandes beschreibt die Beziehung zwischen weniger und mehr definierten Typen. Es werden Konzepte wie PARAMETRISCHER POLYMORPHISMUS und OVERLOADING unterstützt. Gleichzeitig ist die Komplexität der erlaubten Typconstraints erheblich größer als bei vergleichbaren Arbeiten aus dem Stand der Technik (vergleiche Abschnitt 2.3.3). Man kann beispielsweise Typconstraints in Form von mehrwertigen Polynomen definieren.

- **MODELL DER KOMMUNIKATIONSPROTOKOLLE:** Mittels eines neuartigen Modells für Kommunikationsprotokolle, den FIFOMATEN (vergleiche Abschnitt 6.3.2), kann das DYNAMISCHE KOMMUNIKATIONSVERHALTEN von gefärbten Datenflußkomponenten unabhängig von der Berechnungskomplexität ihrer TOKENMASCHINEN – im Gegensatz zur bisherigen Modellierung mittels linearer Gleichungssysteme (vergleiche Abschnitt 2.2.10) – IN DEMSELBEN FORMALISMUS modelliert werden. Dabei beschreibt ein endlicher Automat das Ein-/Ausgabeverhalten einer Datenflußkomponente, während in Fifos mit unendlicher Kapazität die gesendeten und empfangenen gefärbten Token zwischengespeichert werden. Durch Lese- und Schreiboperationen kann ein Fifomat gefärbte Token aus den Fifos lesen beziehungsweise in diese schreiben. Das Modell beinhaltet NICHTDETERMINISMUS, da Zustandsübergänge ausgehend von einem Zustand zu verschiedenen Nachfolgerzuständen mit denselben Aktionen zugelassen sind. Auf der Ebene der Tokenmaschinen beschreibt das Fifomatenmodell eine operationelle Semantik von Datenflußkomponenten, die zu der durch die funktionale Beschreibung gegebenen denotationellen Semantik von Datenflußkomponenten paßt.

9.2.2 Verfahren

Ziel war es, einen Datenflußgraphen entwurfsbegleitend zu analysieren, indem jeweils ein geeignetes Modell der Interfacetypen und des Kommunikationsverhaltens aufgebaut und überprüft wird.

- **TYPBESTIMMUNGSVERFAHREN:** Die Interfacetypprüfung arbeitet ENTWURFSBEGLEITEND. Aus gegebenen Typconstraints und aktuellen Typbelegungen der Interfaces wird die neue Typbelegung durch Auswertung von Constraint-Propagationsregeln bestimmt. Der Typbestimmungsalgorithmus terminiert und besitzt eine Zeitkomplexität von $O(3 \cdot N \cdot |C|)$ (vergleiche Abschnitt 5.4.5). Das Verfahren ist INKREMENTELL. Das bedeutet, daß die Ergebnisse vorheriger Typprüfungen wiederverwendet werden. Außerdem ist das vorgestellte Verfahren LEICHT auf zusätzliche Signalmerkmale ERWEITERBAR.
- **MODEL-CHECKING-VERFAHREN:** Ein dediziertes Model-Checking-Verfahren, das auf der NEUARTIGEN SUCHSTRATEGIE `guidedSearch()` basiert, erlaubt sowohl die Komposition als auch die Simulation von Fifomatenmodellen. Das vorgestellte Model-Checking-Verfahren beantwortet ENTWURFSBEGLEITEND – in Abhängigkeit von der Berechnungskomplexität der zugehörigen Tokenmaschinen – Fragen nach Deadlockfreiheit, Speicherverbrauch und zyklischen Schedules von (großen) Datenflußgraphen. Dabei kommen meh-

rere innovative Methoden für die effiziente Analyse des Erreichbarkeitsgraphen basierend auf den in dieser Arbeit entwickelten neuartigen PARTIAL-ORDER-REDUCTION- und BLOCKIER/DEBLOCKIER-Mechanismen zum Einsatz.

9.2.3 Grenzen der Modelle und Verfahren

Die Grenzen der einzelnen Modelle und Verfahren sind in den jeweiligen Kapiteln detailliert beschrieben. Dort sind auch praktische Lösungen aufgezeigt, die für einen großen Teil der Anwendungen greifen.

- **MODELL PHYSIKALISCHER SIGNALE:** Bei der Erfassung von physikalischen Signalen muß der Benutzer festlegen, durch welche Triggermechanismen ein von einem Sensor erfaßtes physikalisches Signal geeignet in Signalsegmente zerlegt werden kann. Außerdem ist zu spezifizieren, wie mit teilweise gefüllten Endblöcken eines Signalsegmentes verfahren werden soll.
- **MODELL DER DATENFLUSSKOMPONENTEN:** Das vorgestellte funktionale Modell ist nicht geeignet, nichtdeterministische Datenflußkomponenten zu modellieren. Da aber nichtdeterministisches Verhalten bei dem dieser Arbeit zugrundeliegenden Entwurfsproblem nicht erwünscht ist, stellt dies keine erhebliche Einschränkung dar¹.
- **MODELL DER INTERFACETYPEN und TYPBESTIMMUNGSVERFAHREN:** Das Problem der Terminierung des Typbestimmungsalgorithmus wird durch die Verwendung vollständiger Verbände, die nur endliche Ketten beinhalten, gelöst. So wird beispielsweise die Verschachtelungstiefe von Aggregationstypen auf einen natürlichzahligen Wert eingeschränkt. Aufgrund der Unentscheidbarkeit von Hilberts 10. Problem kann der Typbestimmungsalgorithmus nicht in jedem Fall eine Lösung der Typconstraints liefern. Dies wird dem Benutzer gegebenenfalls mitgeteilt, so daß dieser geeignete Maßnahmen ergreifen kann.
- **MODELL DER KOMMUNIKATIONSPROTOKOLLE und MODEL-CHECKING-VERFAHREN:** Das Modell der Kommunikationsprotokolle (Fifomatenmodell) basierend auf unendlichen Fifos ist turingvollständig. Dies hat zur Folge, daß beispielsweise die Bestimmung von zyklischen Schedules nicht mehr entscheidbar ist. Als Lösung wurde zum einen die Einschränkung der Fifos auf endliche Kapazitäten vorgeschlagen, wodurch die Entscheidbarkeit gegeben ist. Zum anderen wurde eine Heuristik für unendliche Fifos vorgestellt, die zwar terminiert aber nicht immer eine Lösung liefert. Selbst im Falle beschränkter Fifokapazitäten bleibt allerdings der Erreichbarkeitsgraph sehr groß. Diese Problematik wird aber mittels eines PARTIAL-ORDER-REDUCTION-Verfahrens und einem BLOCKIER/DEBLOCKIER-Mechanismus erheblich gemildert.

¹Nichtdeterminismus wird aber beispielsweise bei Spezifikationsverfahren, die auf SCHRITTWEISER VERFEINERUNG [Bro99] beruhen, eingesetzt.

9.2.4 Überprüfung der Modelle und Verfahren

Durch BEWEISFÜHRUNG wurde gezeigt, daß – im Rahmen der vorgegebenen Grenzen der Entscheidbarkeit – die Verfahren terminieren und ein korrektes Ergebnis liefern. Die einzige Ausnahme bildet der Suchalgorithmus `guidedSearch()`, bei dem die Korrektheit in Form einer NICHTBEWIESENEN BEHAUPTUNG vermutet aber nicht nachgewiesen wurde. Die gegebenen Modelle und dazugehörigen Verfahren sind implementiert. Mittels einer GROSSEN ANZAHL VON TESTS in Form von Simulationen und Fallstudien wurde gezeigt, daß die implementierten Verfahren und zugehörigen Modelle sich für den entwurfsbegleitenden Einsatz hervorragend eignen. Das dabei entstandene WERKZEUG ZUR BILD- UND SIGNALVERARBEITUNG *Skylla* erlaubt für die Benutzergruppen der Anwender, Anwendungsprogrammierer, Komponentenprogrammierer und Systemprogrammierer eine effiziente Gestaltung ihrer jeweiligen Arbeitsabläufe (vergleiche Abbildung 9.1).

9.3 Innovative Aspekte und Bedeutung der Arbeit

In Abbildung 9.1 sind die innovativen Aspekte dieser Arbeit zusammenfassend dargestellt. Die Bedeutung dieser Arbeit für das Gebiet der komponentenbasierten Softwareentwicklung für datenflußorientierte eingebettete Systeme liegt dabei in der Entwicklung folgender Modelle und Verfahren:

- NEUARTIGES MODELL PHYSIKALISCHER SIGNALE
- Erweiterung bestehender Komponentenmodelle zu GEFÄRBTEN DATENFLUSSPARADIGMEN
- ERWEITERUNG VON VERBANDSMODELLEN FÜR INTERFACETYPEN und Entwicklung eines dazu passenden NEUARTIGEN TYPBESTIMMUNGSVERFAHRENS
- Entwicklung eines NEUEN MODELLS ZUR BESCHREIBUNG DES KOMMUNIKATIONSVERHALTENS VON DATENFLUSSKOMPONENTEN und eines darauf abgestimmten MODEL-CHECKING-VERFAHRENS

Die Bedeutung für die Praxis liegt in der ENTWURFSBEGLEITENDEN QUALITÄTSSICHERUNG, wobei der Programmentwickler keine fundierten Kenntnisse in Programmanalysemethodiken benötigt, sondern sich ganz auf die Probleme seiner Anwendungsdomäne konzentrieren kann. Damit trägt diese Arbeit zur ERHÖHUNG DER AKZEPTANZ FORMALER METHODEN in dieser Anwendergruppe bei.

Insgesamt kann man festhalten, daß die in dieser Arbeit enthaltenen innovativen Aspekte einen wichtigen Schritt in Richtung früher entwurfsbegleitender Qualitätssicherung darstellen (vergleiche Abschnitt 1.2) und somit einen WESENTLICHEN UND GEWINNBRINGENDEN BEITRAG zur Forschung auf diesem Gebiet bilden.

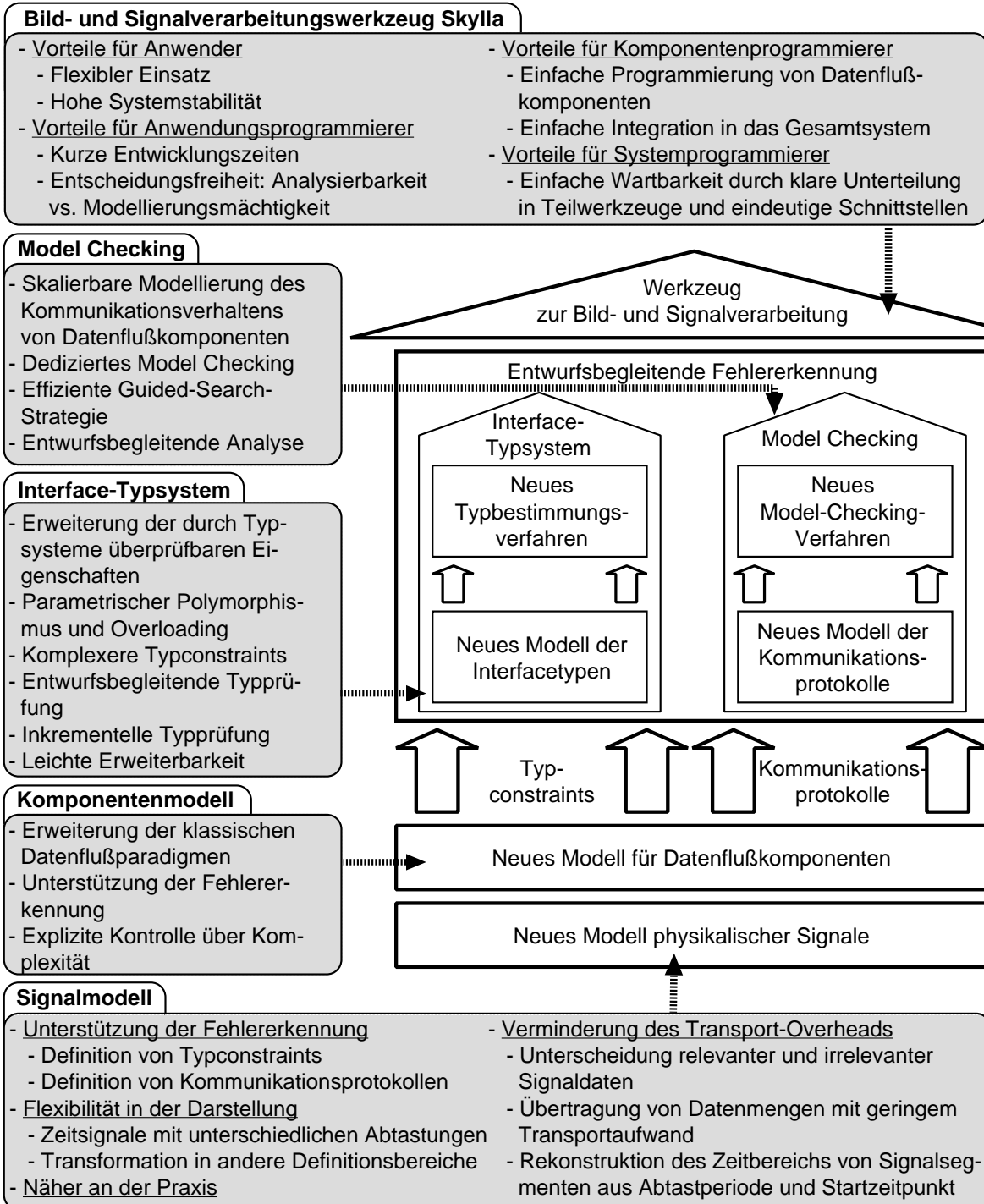


Abbildung 9.1: Lösungsgebäude und innovative Aspekte

9.4 Zusätzliche Einsatzgebiete der zentralen Ideen

Das Gesamtsystem **Skylla** ist für den Einsatz im Bereich Bild- und Signalverarbeitung in eingebetteten Systemen vorgesehen. Darüberhinaus bieten sich aber für die einzelnen Modelle und Verfahren weitergehende Verwendungsmöglichkeiten an.

- **SIGNALMODELL:** Das in Kapitel 3 vorgestellte Modell physikalischer Signale und das darauf aufbauende Komponentenmodell eignen sich – wie in Abbildung 3.3 dargestellt – für den Einsatz in einer großen Zahl von Anwendungen wie beispielsweise
 - Flugüberwachung mittels Radar
 - Unterwasserortung mittels Sonar
 - Positionsermittlung mittels GPS
 - Medizinische Anwendungen wie die Überwachung von Patienten
 - Regelung von Fließbandanlagen

Dabei sind vor allem die Unterstützung der Anwendungsprogrammierer (vergleiche Abschnitt 7.1) durch fehlervermeidende Analysemethodiken wie Interface-Typsystem und Model Checking bedeutsam. Denn in all den genannten Einsatzgebieten können Softwarefehler fatale Folgen haben.

- **KOMPONENTENMODELL:** Eine grundlegende Erkenntnis ist, daß ohne ein **WOHLDEFINIERTES MODELL** einer Komponente und damit einer **FORMALISIERUNG DES BEGRIFFES** keine vernünftige Analyse möglich ist. Daß es aber kein für alle Bereiche gültiges formales Komponentenmodell geben wird, lassen die zahlreichen Versuche, eine passende Definition zu finden [Sam97, BW98, BRS⁺00, DGOS02, Szy02], vermuten. Den Ausweg bilden formale Modelle, die auf Teilbereiche wie
 - Bild- und Signalverarbeitung
 - Hardware/Software-Codesign
 - Officeanwendungen
 - Internetapplikationen

zugeschnitten sind. Eine mögliche Art der Formalisierung bildet dabei die Beschreibung des Komponentenverhaltens mittels Funktionen. Aus dieser Beschreibung lassen sich dann – wie in dieser Arbeit – **STATISCHE CONSTRAINTS** in Form von Typregeln und **DYNAMISCHE CONSTRAINTS** in Form von Kommunikationsprotokollen herleiten.

- **INTERFACE-TYPSYSTEM:** Aufgrund der leichten Erweiterbarkeit des Interface-Typsystems ist auch ein Einsatz
 - in synchronen Sprachen wie Esterel [Ber98] oder

- in einer heterogenen Entwicklungsumgebung mit synchronen und asynchronen Bestandteilen wie Ptolemy II [LX04]

möglich. Insbesondere wird die Idee, entwurfsbegleitend Typconstraints zu überprüfen und zu einem frühestmöglichen Zeitpunkt dem Benutzer Fehler anzuzeigen, ein wichtiger Bestandteil komponentenbasierter Softwareentwicklung werden, da ansonsten keine Aussicht besteht, größere Softwaresysteme nach dem Plug&Play-Verfahren interaktiv zusammenzubauen (vergleiche Abschnitt 1.1).

- **MODEL-CHECKER:** Fifomaten sind zur Modellierung beliebiger asynchron kommunizierender Einheiten einsetzbar. So können Netzwerkkommunikationsprotokolle beziehungsweise das Kommunikationsverhalten verteilter Algorithmen beschrieben und – im Rahmen der bekannten Grenzen der Analysierbarkeit – untersucht werden. Das vorgestellte Model-Checking-Verfahren beinhaltet dabei eine Reihe innovativer Aspekte wie zum Beispiel den BLOCKIER/DEBLOCKIER-Mechanismus oder die verwendete Datenstruktur der ORDERED MAPS, welche einen lohnenden Einsatz in diesen Anwendungsfeldern erwarten lassen. Auch hier gilt, daß die Akzeptanz solcher Model-Checking-Verfahren wesentlich von ihrer Eignung für einen interaktiven Einsatz abhängen wird.

Somit kann abschließend festgehalten werden, daß über das geschilderte Anwendungsfeld hinaus die in dieser Arbeit gewonnenen Erkenntnisse, die hergeleiteten Modellierungsmethodiken und Analysealgorithmen gewinnbringend eingesetzt werden können.

Abbildungsverzeichnis

1.1	Eingebettetes System (Schematische Darstellung)	12
1.2	Beispielhafte Abarbeitung eines Datenflußgraphen	14
1.3	Deadlock (blockiertes Lesen)	16
1.4	Deadlock (blockiertes Schreiben)	17
1.5	Datenflußkomponente mit digitalen Signalen	17
1.6	Unübersichtlicher Datenflußgraph mit Laufzeitfehler (kommerziell eingesetztes Tool)	19
1.7	Fehlerverteilung und Fehlerkosten in unterschiedlichen Phasen eines Softwareprojekts	20
1.8	Qualitätssicherungsaufwand in unterschiedlichen Phasen eines Softwareprojekts .	20
1.9	Eigener Lösungsansatz	22
2.1	Signalmodelle	28
2.2	SDF-Komponenten (a) ohne und (b) mit eigenem Speicher	31
2.3	BDF-Komponenten Switch und Select	31
2.4	DDF-Komponente Merge	32
2.5	Petrinetzmodell einer SDF-Komponente	32
2.6	Petrinetzmodell einer Switch -Komponente	33
2.7	Petrinetzmodell einer Select -Komponente	33
2.8	Petrinetzmodell einer Merge -Komponente	34
2.9	Haskell-Modelle grundlegender SDF-Komponenten	34
2.10	Verband zur Typbestimmung in Ptolemy II	39
2.11	Beispiel zur Typbestimmung in Ptolemy II	40
2.12	Beispiel zweier kommunizierender Interface-Automaten	43
2.13	Beispiel einer Extended Finite State Machine	44
2.14	Model-Checking-Strategie von VeriSoft	48
2.15	Beispiel eines künstlichen Deadlocks	49
3.1	Probleme bei Vernachlässigung des Zeitbereichs physikalischer Signale	52
3.2	Problem bei stückweise konstanten physikalischen Signalen	53
3.3	Beispiele für physikalische Signale	54
3.4	Physikalische Signale in einem eingebetteten System	57
3.5	Schrittweise Einführung des neuen Signalmodells	60

4.1	Probleme bei der Kombination von traditionellem Komponentenmodell mit neuem Signalmodell	71
4.2	Nichtdeterminismus in Datenflußgraphen (klassisches Komponentenmodell) . . .	72
4.3	Nichtdeterminismus in Datenflußgraphen (Beispiele für Schedules)	73
4.4	Horizontale und vertikale Berechnungskomplexität	74
4.5	Komplexere Datenflußstrukturen	82
4.6	Datenflußgraph zur Klassifizierung von Bildschirmröhren	87
4.7	Unterschiedliche Kombinationen von Signalsegmenten (Zeitbereich)	90
5.1	Constraint-Pattern	95
5.2	Interfacetypen und Typconstraints einer Datenflußkomponente	96
5.3	Hasse-Diagramme der Typtraits	100
5.4	Typbestimmungsalgorithmus	110
5.5	Beispiel zur Interface-Typbestimmung	114
5.6	Probleme bei der Typbestimmung (Terminierung)	119
5.7	Probleme bei der Typbestimmung (Zu schwache Constraint-Propagationsregeln) .	122
5.8	Beispiel für Additionsconstraints	123
6.1	Beispielhafte Abarbeitung zweier Eingabeströme durch CMerge	129
6.2	Fifomatenmodelle einiger Datenflußkomponenten	133
6.3	Model-Checking-Algorithmus	135
6.4	Beispiel zur Komposition	138
6.5	Datenflußgraph für Simulation	139
6.6	Simulation (Brute Force)	140
6.7	Simulation (Blockieren/Deblockieren) (1)	142
6.8	Simulation (Blockieren/Deblockieren) (2)	143
6.9	Simulation (Partial Order Reduction)	147
6.10	Beispiel zur Kommutativität von Fifomaten	157
6.11	Beispiel zur Bestimmung der notwendigerweise auszuführenden Fifomaten . . .	159
6.12	Beispiel zum Blockier/Deblockier-Mechanismus	165
6.13	Simulation eines Stacks mittels zweier Fifos	173
7.1	Benutzergruppen des Bild- und Signalverarbeitungswerkzeuges Skylla	182
7.2	Sicht des Anwendungsprogrammierers auf Skylla	183
7.3	Sicht des Komponentenprogrammierers auf Skylla	185
7.4	Meet-in-the-Middle-Strategie	186
7.5	Überblick über das Gesamtsystem Skylla	186
7.6	Editor von Skylla (Screenshot)	188
7.7	Incrementalmap	189
7.8	Ordered Map	190
7.9	Überblick über die Ablaufsteuerung von Skylla	191
7.10	Kommunikationsverbindung zwischen zwei Clients	193
7.11	Adaptive Communication Environment (ACE)	194

7.12	Komponentenmodelle in Design/CPN	196
7.13	Kontextgesteuerte Dialogfenster einer Datenflußkomponente in Skylia	197
8.1	Beispiel eines Datenflußgraphen (systematische Tests)	204
8.2	Einfügezeit = F(Kanten) (Datenflußgraph mit 200 Kanten; Constraintkomplexität: difficult)	205
8.3	Mittlere Einfügezeit = F(Constraintkomplexität)	206
8.4	Mittlere Einfügezeit = F(Graphgröße)	207
8.5	Mittlere Neuberechnungszeit = F(Graphgröße, Constraintkomplexität)	208
8.6	Grauwertbild und Seitenriß einer Getränkedose	209
8.7	Einfügezeit = F(Kanten) (Qualitätskontrolle von Getränkedosen)	210
8.8	Graphstrukturen Rows , Net und Random	212
8.9	Zeitverbrauch = F(Komp.) (Rows , Breite = 2, ident. Prot.)	217
8.10	Zeitverbrauch = F(Komp.) (Rows , Prot. = $(sm^*e o)^*$, ident. Prot.)	217
8.11	Speicherverbrauch = F(Komp.) (Rows , Breite = 2, ident. Prot.)	218
8.12	Speicherverbrauch = F(Komp.) (Rows , Prot. = $(sm^*e o)^*$, ident. Prot.)	218
8.13	Knoten = F(Komp.) (Rows , Breite = 2, ident. Prot.)	219
8.14	Knoten = F(Komp.) (Rows , Prot. = $(sm^*e o)^*$, ident. Prot.)	219
8.15	Statemaps = F(Komp.) (Rows , Breite = 2, ident. Prot.)	220
8.16	Statemaps = F(Komp.) (Rows , Protokoll = $(sm^*e o)^*$, ident. Prot.)	220
8.17	Wordmaps = F(Komp.) (Rows , Breite = 2, ident. Prot.)	221
8.18	Wordmaps = F(Komp.) (Rows , Prot. = $(sm^*e o)^*$, ident. Prot.)	221
8.19	Zeitverbrauch = F(Komp.) (Rows , Breite = 2, komp. Prot.)	223
8.20	Zeitverbrauch = F(Komp.) (Rows , Breite = 2, inkomp. Prot.)	223
8.21	Zeitverbrauch = F(Komp.) (Net , Breite = 2, ident. Prot.)	224
8.22	Zeitverbrauch = F(Komp.) (Net , Prot. = $(sm^*e o)^*$, ident. Prot.)	224
8.23	Speicherverbrauch = F(Komp.) (Net , Breite = 2, ident. Prot.)	226
8.24	Speicherverbrauch = F(Komp.) (Net , Prot. = $(sm^*e o)^*$, ident. Prot.)	226
8.25	Knoten = F(Komp.) (Net , Breite = 2, ident. Prot.)	227
8.26	Knoten = F(Komp.) (Net , Prot. = $(sm^*e o)^*$, ident. Prot.)	227
8.27	Statemaps = F(Komp.) (Net , Breite = 2, ident. Prot.)	228
8.28	Statemaps = F(Komp.) (Net , Prot. = $(sm^*e o)^*$, ident. Prot.)	228
8.29	Wordmaps = F(Komp.) (Net , Breite = 2, ident. Prot.)	229
8.30	Wordmaps = F(Komp.) (Net , Prot. = $(sm^*e o)^*$, ident. Prot.)	229
8.31	Zeitverbrauch = F(Komp.) (Net , Breite = 2, komp. Prot.)	230
8.32	Zeitverbrauch = F(Komp.) (Net , Breite = 2, inkomp. Prot.)	230
8.33	Zeitverbrauch = F(Komp.) (RowsF , Breite = 2, ident. Prot.)	232
8.34	Zeitverbrauch = F(Komp.) (RowsF , Breite = 2, komp. Prot.)	232
8.35	Zeitverbrauch = F(Komp.) (RowsF , Breite = 2, inkomp. Prot.)	233
8.36	Zeitverbrauch = F(Komp.) (NetF , Breite = 2, ident. Prot.)	233
8.37	Zeitverbrauch = F(Komp.) (NetF , Breite = 2, komp. Prot.)	234
8.38	Zeitverbrauch = F(Komp.) (NetF , Breite = 2, inkomp. Prot.)	234
8.39	Zeitverbrauch = F(Komp.) (Random , Prot. = $(sm^*e o)^*$, ident. Prot.)	236

8.40	Zeitverbrauch=F(Gewichte) (Rows, Breite = 2, Tiefe = 10000, ident. Prot.) . .	236
8.41	Zeitverbrauch=F(Komp.) (Rows, Breite = 2, Prot.= $(sm^*e o)^*$, komp. Prot.) . .	238
8.42	Zeitverbrauch=F(Komp.) (Rows, Breite = 2, Prot.= $(sm^*e o)^*$, ident. Prot.) . .	238
8.43	Zeitverbrauch=F(Bits/Teilschl.) (Rows, Breite = 2, Tiefe = 1000, ident. Prot.) .	239
8.44	Speicherverbrauch = F(Bits/Teilschl.) (Rows, Breite = 2, Tiefe = 1000, ident. Prot.)	239
8.45	Zeitverbrauch=F(Komp.) (Rows, Breite = 2, Incrementalmaps, ident. Prot.) . .	241
8.46	if-then-else-Verzweigung	241
8.47	while-Schleife	242
8.48	Distribute-Merge-Konstrukt	242
8.49	Zeitverbrauch=F(Komp.) (Rows, Breite= 2, Ports = 1, Kap. = 1, ident. Prot.) .	250
8.50	Zeitverbrauch=F(Komp.) (Rows, Breite= 2, Ports = 2, Kap. = 1, ident. Prot.) .	250
8.51	Zeitverbrauch=F(Komp.) (Rows, Breite= 2, Ports = 1, Kap.= 1, komp. Prot.) .	251
8.52	Zeitverbrauch=F(Komp.) (Rows, Breite= 2, Ports = 2, Kap. = 1, komp. Prot.) .	251
8.53	Zeitverbrauch=F(Komp.) (Net, Breite= 4, Ports = 1, Kap. = 1, ident. Prot.) . . .	253
8.54	Zeitverbrauch=F(Komp.) (Net, Breite = 4, Ports = 1, komp. Prot.)	255
8.55	Zeitverbrauch=F(Komp.) (RowsF, Breite = 2, Ports = 1, ident. Prot.)	255
8.56	Zeitverbrauch=F(Komp.) (RowsF, Breite = 2, Ports = 2, ident. Prot.)	256
8.57	Zeitverbrauch=F(Komp.) (NetF, Breite = 4, Ports = 1, ident. Prot.)	256
8.58	Zeitverbrauch=F(Komp.) (NetF, Breite = 4, Ports = 2, ident. Prot.)	257
8.59	Zeitverbrauch = F(Komp.) (Rows, Prot. = o^* , Breite = 2, Ports = 1, Kanalkapazität, ident. Prot.)	257
8.60	Meßanordnung (Projektive Rekonstruktion in der Stereobildverarbeitung)	260
8.61	Datenflußgraph (Projektive Rekonstruktion in der Stereobildverarbeitung)	260
8.62	Überlagerte Bilder zweier Kameras mit Verschiebungsvektoren	261
8.63	Vergleich Skylla und Iconnect (Projektive Rekonstruktion in der Stereobildverarbeitung)	262
8.64	Verwendung mehrerer Rechner (Projektive Rekonstruktion in der Stereobildverarbeitung)	263
8.65	Ausführungszeit=F(Rechner vom Typ B) (Bilder = 50, Abtastperiode) (Projektive Rekonstruktion in der Stereobildverarbeitung)	263
8.66	Meßanordnung (Qualitätskontrolle von Beilagscheiben)	264
8.67	Eingabebild (Qualitätskontrolle von Beilagscheiben)	264
8.68	Datenflußgraph (Qualitätskontrolle von Beilagscheiben)	266
8.69	Ausführungszeit = F(Rechner vom Typ B) (Bilder = 50, Abtastperiode) (Qualitätskontrolle von Beilagscheiben)	267
8.70	Meßanordnung (Ermittlung des gemeinsamen Sichtbarkeitsbereichs zweier Kameras)	268
8.71	Datenflußgraph (Ermittlung des gemeinsamen Sichtbarkeitsbereichs zweier Kameras)	269
8.72	Eingabedaten (Kreis, Dreieck, Wolke) (Ermittlung des gemeinsamen Sichtbarkeitsbereichs zweier Kameras)	270
8.73	Schnitt der beiden Sichtbarkeitsbereiche	270

8.74	Ausführungszeit = F(Rechner vom Typ B) (Bilder = 10, Abtastperiode = 5 sec, Eingabedaten) (Ermittlung des gemeinsamen Sichtbarkeitsbereichs zweier Kameras)	271
8.75	Speicherverbrauch = F(Laufzeit) (Bilder = 10, Abtastperiode = 1 sec, Eingabedaten) (Ermittlung des gemeinsamen Sichtbarkeitsbereichs zweier Kameras) . . .	272
8.76	Speicherverbrauch = F(Laufzeit) (Bilder = 10, Eingabedaten = Kreise, Rechner vom Typ B) (Ermittlung des gemeinsamen Sichtbarkeitsbereichs zweier Kameras)	273
9.1	Lösungsgebäude und innovative Aspekte	282

Tabellenverzeichnis

4.1	Klassifikation von Datenflußkomponenten	78
5.1	Constraint-Propagationsregeln	112
6.1	Komposition vs. Simulation	177
8.1	Testplattformen	202
8.2	Abkürzungen in Tabellen- und Abbildungsbeschriftungen	202
8.3	Systematische Tests (Interface-Typsystem)	204
8.4	Systematische Tests (Model Checker)	214
8.5	Zeitverbrauch = F(Komp.) (Rows, Breite = 1, Prot. = σ^* , ohne/mit Blockieren/Deblockieren, inkomp. Prot.)	237
8.6	Ergebnisse zu Distribute-Merge-Konstrukt	240
8.7	Zyklus im Erreichbarkeitsgraphen für Distribute-Merge-Konstrukt mit 5 Inittoken	243
8.8	Abschließende Bewertung des Model Checkers	245
8.9	Zeitverbrauch = F(Komp.) (Net, Breite = 4, Ports = 2, Kap. = 1, ident. Prot.) . .	253
8.10	Zeitverbrauch = F(Komp.) (Net, Breite = 4, Ports = 2, Kap. = 1, komp. Prot.) . .	253
8.11	Abschließende Gegenüberstellung von Skylla und Spin	258

Literaturverzeichnis

- [AABJ04] ABDULLA, P. A.; ANNICHINI, A.; BOUAJJANI, A.; JONSSON, B.: *Using Forward Reachability Analysis for Verification of Lossy Channel Systems*; The Journal of Formal Methods in System Design, 2004
- [Abs04] ABSMEIER, S.: *Design und Implementierung einer prototypischen Ablaufsteuerung in CORBA für die verteilte Ausführung von Datenflußgraphen*; Diplomarbeit; Universität Passau, November 2004
- [AES98] ALLEN, G. E.; EVANS, B. L.; SCHANBACHER, D. C.: *Real-time Sonar Beamforming on a Unix Workstation Using Process Networks and POSIX Threads*; in: *32 Asilomar Conference on Signals, Systems and Computers*, November 1998
- [AH01] DE ALFARO, L.; HENZINGER, T. A.: *Interface Automata*; in: *Proceedings of the 9th Annual ACM Symposium on Foundations of Software Engineering (FSE)*; S. 109–120; ACM Press, 2001
- [Ame02] AMEY, P.: *Correctness By Construction: Better Can also Be Cheaper*; CrossTalk Magazine, The Journal of Defense Software, 2002
- [AMG02] ABOY, M.; MCNAMES, J.; GOLDSTEIN, B.: *Automatic Detection Algorithm of Intracranial Pressure Waveform Components*; in: *Proceedings of the 23rd Annual International Conference of the IEEE Engineering in Medicine and Biology Society*, 2002
- [BA81] BROCK, D.; ACKERMAN, W.: *Scenarios: A Model of Non-determinate Computation*; in: *Formalization of Programming Concepts* (Hg. Diaz, J.; Ramos, I.); LNCS 107; S. 252–259; Springer-Verlag, 1981
- [Bal96] BALZERT, H.: *Lehrbuch der Software-Technik*; Spektrum Akademischer Verlag GmbH, 1996
- [BB00] BHATTACHARYA, B.; BHATTACHARYYA, S. S.: *Parameterized Dataflow Modeling of DSP Systems*; in: *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, Juni 2000

- [BBD⁺00] BALLUCHI, A.; BENVENUTI, L.; DI BENEDETTO, M. D.; PINELLO, C.; SANGIOVANNI-VINCENTELLI, A. L.: *Automotive Engine and Power-Train Control: A Comprehensive Hybrid Model*; in: *Proceedings of the 8th IEEE Mediterranean Conference on Control and Automation (MED)*, Juli 2000
- [BD02] BABICH, F.; DEOTTO, L.: *Formal Methods for Specification and Analysis of Communication Protocols*; IEEE Communication Surveys; Bd. 4 (1), Dezember 2002
- [Ber98] BERRY, G.: *The Foundation of Esterel*; in: *Proof, Language and Interaction: Essays in Honour of Robin Milner* (Hg. Plotkin, G.; Stirling, C.; Tofte, M.); MIT Press, 1998
- [BFH⁺03] BAUHOFFER, A.; FARR, H.; HESSE, R.; LIEBL, A.; SICK, B.; SONNTAG, A.; WIMMER, M.: *Messen, Steuern und Regeln mit ICONNECT*; Kap. Anwendungsbeispiele; Vieweg-Verlag, 2003
- [BH99] BOUAJJANI, A.; HABERMEHL, P.: *Symbolic Reachability Analysis of Fifo-Channel Systems with Nonregular Sets of Configurations*; Theoretical Computer Science; Bd. 221 (1–2), S. 211–250, Juni 1999
- [BH01] BASTEN, T.; HOOGERBRUGGE, J.: *Efficient Execution of Process Networks*; in: *Communicating Process Architectures*; IOS Press, 2001
- [BHH00] BARROCA, L.; HALL, J.; HALL, P. (Hg.): *Software Architectures*; Springer-Verlag, 2000
- [Bir84] BIRKHOFF, G.: *Lattice Theory*; 3. Aufl.; American Mathematical Society, 1984
- [Bir98] BIRD, R.: *Introduction to Functional Programming Using Haskell*; Prentice Hall Series in Computer Science; 2. Aufl.; Prentice Hall, 1998
- [BJJ00] BASTEN, T.; JANSSEN, G.; JESS, J.: *Models of Digital Systems*; Techn. Ber.; Technische Universität Eindhoven, 2000
- [BLL⁺04] BROOKS, C.; LEE, E. A.; LIU, X.; NEUENDORFFER, S.; ZHAO, Y.; ZHENG, H.: *Heterogeneous Concurrent Modeling and Design in Java*; Techn. Ber.; University of California at Berkeley, Juni 2004
- [BML96] BHATTACHARYYA, S. S.; MURTHY, P. K.; LEE, E. A.: *Software Synthesis from Dataflow Graphs*; The Kluwer International Series in Engineering and Computer Science; Kluwer Academic Publishers, 1996
- [BML⁺97] BOLSENS, I.; MAN, H. J. D.; LIN, B.; VAN ROMPAEY, K.; VERCAUTEREN, S.; VERKEST, D.: *Hardware/Software Co-Design of Digital Telecommunication Systems*; in: *Proceedings of the IEEE*; Bd. 85, März 1997

- [BP03] BROY, M.; PREE, W.: *Ein Wegweiser für Forschung und Lehre im Software-Engineering eingebetteter Systeme*; Informatik Spektrum, 2003
- [Bro97] BROY, M.: *Refinement of Time*; in: *Proceedings of the 4th International AMAST Workshop on Real-Time Systems and Concurrent and Distributed Software (ARTS)*; LNCS 1231; S. 44–63, 1997
- [Bro99] BROY, M.: *A Logical Basis for Component-Based Systems Engineering*; Calculational System Design, 1999
- [Bro03a] BROY, M.: *Modular Hierarchies of Models for Embedded Systems*; in: *First ACM and IEEE Conference on Formal Methods and Models for Co-Design*; S. 183–190; IEEE Computer Society, 2003
- [Bro03b] BROY, M.: *Multi-View Modelling of Software Systems*; in: *Proceedings of the Workshop on Formal Aspects of Component Software (FACS'03)*; S. 3–10, September 2003
- [BRS⁺00] BERGNER, K.; RAUSCH, A.; SIHLING, M.; VILBING, A.; BROY, M.: *A Formal Model for Componentware*; in: *Foundations of Component-Based Systems*; S. 17–26; Cambridge University Press, 2000
- [BS01] BROY, M.; STØLEN, K.: *Specification and Development of Interactive Systems: FOCUS on Streams, Interfaces, and Refinement*; Springer-Verlag, 2001
- [Buc93] BUCK, J. T.: *Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model*; Dissertation; University of California at Berkeley, 1993
- [BW98] BROWN, A. W.; WALLNAU, K. C.: *The Current State of Component Based Software Engineering*; IEEE Software; Bd. 15 (5), S. 37–46, 1998
- [BW00] BROY, M.; WIRSING, M.: *Algebraic State Machines*; in: *Proceedings of the 8th International Conference on Algebraic Methodology and Software Technology (AMAST)*; Springer-Verlag, 2000
- [BZ83] BRAND, D.; ZAFIROPULO, P.: *On Communicating Finite-State Machines*; Journal of the Association for Computing Machinery; Bd. 30 (2), S. 323–342, 1983
- [Bör00] BÖRGER, E. (Hg.): *Architecture Design and Validation Methods*; Springer-Verlag, Januar 2000
- [Car97] CARDELLI, L.: *Handbook of Computer Science and Engineering*; Kap. Type Systems; CRC Press, 1997
- [CGP99] CLARKE, E. M.; GRUMBERG, O.; PELED, D. A.: *Model Checking*; MIT Press, 1999

- [Cro98] CROMME, L.: *Echtzeit-Farbbildverarbeitung zur visuellen Qualitätssicherung im industriellen Umfeld*; Brandenburgische Technische Universität Cottbus, 1998
- [CW85] CARDELLI, L.; WEGNER, P.: *On Understanding Types, Data Abstractions, and Polymorphism*; ACM Computing Surveys; Bd. 17 (4), S. 471–522, 1985
- [CW96] CLARKE, E. M.; WING, J. M.: *Formal Methods: State of the Art and Future Directions*; ACM Computing Surveys; Bd. 28 (4), S. 626–643, Dezember 1996
- [Czi91] CZICHOS, H. (Hg.): *Die Grundlagen der Ingenieurwissenschaften*; 29. Aufl.; Springer-Verlag, 1991
- [Den74] DENNIS, J. B.: *First Version of a Dataflow Procedure Language*; in: *Proceedings of the Programming Symposium, Paris, April 1974*; Bd. 19; S. 362–376, 1974
- [Den95] DENNIS, J. B.: *Advanced Topics in Dataflow Computing and Multithreading*; Kap. Stream Data Types for Signal Processing; IEEE Computer Society Press, April 1995
- [DGOS02] DOUCET, F.; GUPTA, R.; OTSUKA, M.; SHUKLA, S.: *An Environment for Dynamic Component Composition for Efficient Co-Design*; in: *Design Automation and Test Conference (DATE 2002)*, März 2002
- [DL04] DURAND-LOSE, J.: *A Kleene Theorem for Splittable Signals*; Information Processing Letters; Bd. 89, S. 237–245, 2004
- [DP02] DAVEY, B. A.; PRIESTLEY, H. A.: *Introduction to Lattices and Order*; 2. Aufl.; Cambridge University Press, 2002
- [Egg02] EGGERMONT, L. D. J. (Hg.): *Embedded Systems Roadmap 2002*; PROGRESS/STW, März 2002
- [ELLSV97] EDWARDS, S.; LAVAGNO, L.; LEE, E. A.; SANGIOVANNI-VINCENTELLI, A.: *Design of Embedded Systems: Formal Models, Validation, and Synthesis*; Proceedings of the IEEE; Bd. 85 (3), S. 366–390, März 1997
- [EN98] EMERSON, E. A.; NAJOSHI, K. S.: *On Model Checking for Non-Deterministic Infinite-State Systems*; in: *Proceedings of the 13th Annual IEEE Symposium on Logic in Computer Science*; S. 70–80, 1998
- [Fau93] FAUGERAS, O.: *Three-Dimensional Computer Vision, A Geometric Viewpoint*; MIT Press, 1993
- [FIS00] FINKEL, A.; IYER, S. P.; SUTRE, G.: *Well-Abstracted Transition Systems: Application to FIFO Automata*; in: *Proceedings of the 11th International Conference on Concurrency Theory*; S. 566–580; Springer-Verlag, 2000

- [FK04] FISCHL, T.; KLEPPER, M.: *Design und Implementierung eines Algorithmus zur 3D-Rekonstruktion in dem Signalverarbeitungswerkzeug Skylla*, Juli 2004; Interner technischer Bericht (Univerität Passau)
- [FM97] FINKEL, A.; MCKENZIE, P.: *Verifying Identical Communication Processes is Undecidable*; Theoretical Computer Science; Bd. 174 (1–2), S. 271–230, August 1997
- [FS01] FINKEL, A.; SCHNOEBELEN, P.: *Well-Structured Transition Systems Everywhere!*; Theoretical Computer Science; Bd. 256 (1–2), S. 63–92, 2001
- [GBG95] GAO, G. R.; BIC, L.; GAUDIOT, J.-L. (Hg.): *Advanced Topics in Dataflow Computing and Multithreading*; IEEE Computer Society Press, April 1995
- [GHJ98] GODEFROID, P.; HANMER, R. S.; JAGADEESAN, L. J.: *Model Checking Without a Model: An Analysis of the Heart-Beat Monitor of a Telephone Switch Using VeriSoft*; in: *Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'98)*; S. 124–133, März 1998
- [GHS⁺03] GARSCHHAMMER, M.; HEGERING, H.; SCHIFFERS, M.; BROY, M.; PICOT, A.: *Kommunikations- und Informationstechnik 2010+3*; Bundesamt für Sicherheit in der Informationstechnik, Mai 2003
- [GHW04] GRAF, S.; HANNING, T.; WAGNER, R.: *Projektive Rekonstruktion – Stereobildverarbeitung ohne Nebenwissen*; Techn. Ber. MIP-0405; Universität Passau, April 2004
- [God97] GODEFROID, P.: *Model Checking for Programming Languages Using VeriSoft*; in: *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*; S. 174–186, Januar 1997
- [God03] GODEFROID, P.: *Software Model Checking: The VeriSoft Approach*; Formal Methods in System Design, August 2003
- [Goo97] GOODWIN, M. M.: *Adaptive Signal Models: Theory, Algorithms, and Audio Applications*; Dissertation; University of California at Berkeley, 1997
- [Hal93] HALBWACHS, N.: *Synchronous Programming of Reactive Systems*; Kluwer Academic Publishers, 1993
- [Har87] HAREL, D.: *Statecharts: A Visual Formalism for Complex Systems*; Science of Computer Programming; Bd. 8, S. 231–274, 1987
- [Hes93] HESS, W.: *Digitale Filter – Eine Einführung*; Teubner Verlag, 1993
- [HLL⁺03] HYLANDS, C.; LEE, E. A.; LIU, J.; LIU, X.; NEUENDORFFER, S.; XIONG, Y.; ZHAO, Y.; ZENGH, H.: *Overview of the Ptolemy Project*; Techn. Ber.; University of California at Berkeley, Juli 2003

- [HMU02] HOPCROFT, J. E.; MOTWANI, R.; ULLMAN, J. D.: *Einführung in die Automatentheorie, formale Sprachen und Komplexitätstheorie*; 2. Aufl.; Addison-Wesley, 2002
- [Hol91] HOLZMANN, G. J.: *Design and Validation of Computer Protocols*; Prentice Hall, 1991
- [Hol04] HOLZMANN, G. J.: *The Spin Model Checker*; Addison-Wesley, 2004
- [Hor04] HORWITZ, S.: *Construction of Compilers*; <http://www.cs.wisc.edu/~cs701-1/cs701.html>, September 2004
- [Hou02] HOUSSAIS, B.: *The Synchronous Programming Language SIGNAL*; IRISA, Espresso Project, April 2002
- [HP99] HAREL, D.; POLITI, M.: *Modeling Reactive Systems with Statecharts: The State-mate Approach*; I-Logix Inc., 1999
- [HU88] HOPCROFT, J. E.; ULLMAN, J. D.: *Einführung in die Automatentheorie, formale Sprachen und Komplexitätstheorie*; Addison-Wesley, 1988
- [HWLC97] HOFMANN-WELLENHOF, B.; LICHTENEGGER, H.; COLLINS, J.: *Global Positioning System – Theory and Practice*; 4. Aufl.; Springer-Verlag, 1997
- [Jen97] JENSEN, K.: *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use*; Bd. 1 von *Monographs in Theoretical Computer Science*; 2. Aufl.; Springer-Verlag, 1997
- [JLR97] JIANG, T.; LI, M.; RAVIKUMAR, B.: *Handbook of Computer Science and Engineering*; Kap. Formal Models and Computability; CRC Press, 1997
- [Kah74] KAHN, G.: *The Semantics of a Simple Language for Parallel Processing*; in: *Information Processing 74: Proceedings of the IFIP Congress* (Hg. Rosenfeld, J. L.); S. 471–475; North-Holland Publishing Company, August 1974
- [Kar95] KARLSSON, E.-A. (Hg.): *Software Reuse*; Wiley Series in Software Based Systems; John Wiley & Sons, 1995
- [KCF⁺02] KOÇ, A.; CORD, T.; FEHRER, D.; JUNG-DIEFENBACH, C.; KAUN, S.; SCHMIED, J.: *Systematische Planung effizienter Qualitätssicherung eingebetteter Software*; Automatisierungstechnische Praxis; Bd. 44 (8), S. 45–51, 2002
- [KM66] KARP, R. M.; MILLER, R. E.: *Properties of a Model for Parallel Computations: Determinacy, Termination, Queuing*; SIAM Journal of Applied Mathematics; Bd. 14 (6), S. 1390–1411, November 1966

- [KM77] KAHN, G.; MACQUEEN, D. B.: *Coroutines and Networks of Parallel Processes*; in: *Information Processing 77: Proceedings of the IFIP Congress* (Hg. Gilchrist, B.); S. 993–998; North-Holland Publishing Company, August 1977
- [Kok86] KOK, J. N.: *Denotational Semantics of Nets with Nondeterminism*; in: *Proceedings of the European Symposium on Programming (ESOP 86)*; S. 237–249, März 1986
- [KRB96] KLEIN, C.; RUMPE, B.; BROY, M.: *A Stream-Based Mathematical Model for Distributed Information Processing Systems – SysLab System Model –*; in: *Proceedings of the Workshop on Formal Methods for Open Object-Based Distributed Systems (FMOODS'96)*; S. 323–338, 1996
- [Köc04] KÖCK, J.: *Verifikation von Fifomat-Kommunikationsprotokollen mittels Spin*, November 2004; Interner technischer Bericht (Universität Passau)
- [Laf03] LAFUENTE, A. L.: *Directed Search for the Verification of Communication Protocols*; Dissertation; Universität Freiburg, Juni 2003
- [Lau02] LAUSSER, M.: *Spezifikation und Implementierung eines Systems zur verteilten Ausführung von Datenflußgraphen mit einfachen Kontrollstrukturen*; Diplomarbeit; Lehrstuhl für Rechnerstrukturen, Universität Passau, Dezember 2002
- [Le01] LE, Q. B.: *Spezifikation und Implementierung eines modularen Tools für die Simulation der Ausführung von Datenflußgraphen mit einfachen Kontrollstrukturen*; Diplomarbeit; Lehrstuhl für Rechnerstrukturen, Universität Passau, Juni 2001
- [Lee97] LEE, E. A.: *A Denotational Semantics for Dataflow with Firing*; Techn. Ber.; University of California at Berkeley, Januar 1997
- [Lee99] LEE, E. A.: *Embedded Software – An Agenda for Research*; Techn. Ber.; University of California at Berkeley, Dezember 1999
- [Lee02] LEE, E. A.: *Embedded Software*; Advances in Computers; Bd. 56, 2002
- [LP95] LEE, E. A.; PARKS, T. M.: *Dataflow Process Networks*; in: *Proceedings of the IEEE*; S. 773–801, Mai 1995
- [LSV96] LEE, E. A.; SANGIOVANNI-VINCENTELLI, A.: *The Tagged Signal Model*; Techn. Ber.; University of California at Berkeley, Juni 1996
- [LSV98] LEE, E. A.; SANGIOVANNI-VINCENTELLI, A.: *A Framework for Comparing Models of Computation*; IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems; Bd. 17 (12), Dezember 1998
- [LX01] LEE, E. A.; XIONG, Y.: *System-Level Types for Component-Based Design*; in: *First Workshop on Embedded Software (EMSOFT)*; LNCS 2211; S. 237–255; Springer-Verlag, Oktober 2001

- [LX04] LEE, E. A.; XIONG, Y.: *A Behavioral Type System and its Application in Ptolemy II*; Formal Aspects of Computing; Bd. 16 (3), S. 210–237, August 2004
- [Mat93] MATIYASEVICH, Y. V.: *Hilbert's Tenth Problem*; MIT Press, 1993
- [May99] MAYDL, W.: *Verwendung Neuronaler Netze (NARX und TDNN) als nichtlineare, dynamische Regler*; Diplomarbeit; Universität Passau, April 1999
- [May04a] MAYDL, W.: *Design Accompanying Analysis of Component-Based Embedded Software*; in: *Component-Based Software Engineering* (Hg. Crnkovic, I.); LNCS 3054; Springer-Verlag, Mai 2004
- [May04b] MAYDL, W.: *Model Checking for Component-Based Software Development for Embedded Systems*; in: *International Conference on Software Engineering and Applications*; S. 331–338, November 2004
- [May04c] MAYDL, W.: *A Novel Component Model for the Synchronous Dataflow Paradigm*; in: *International Conference on Software Engineering*; S. 172–177, Februar 2004
- [McM93] MCMILLAN, K.: *Symbolic Model Checking*; Kluwer Academic Publishers, 1993
- [McM99] MCMILLAN, K.: *Getting Started with SMV*; Cadence Berkeley Labs, März 1999
- [McM02] MCMILLAN, K.: <http://www-cad.eecs.berkeley.edu/~kenmcmil/smv/>, 2002
- [Mey99] MEYER, B.: *Every Little Bit Counts: Toward More Reliable Software*; IEEE Computer; Bd. 32 (11), S. 131–133, 1999
- [MG03] MAYDL, W.; GRAJCAR, M.: *Interface Type Checking for Component-Based Software Development for Embedded Systems*; in: *International Conference on Software Engineering and Applications*; S. 627–634, November 2003
- [MG05] MAYDL, W.; GRUNSKÉ, L.: *Component-Based Software Development for Embedded Systems – An Overview of Current Research Trends*; Kap. Behavioral Types for Embedded Software – A Survey, S. 82–106; LNCS 3778; Springer-Verlag, 2005
- [Mil78] MILNER, R.: *A Theory of Type Polymorphism in Programming*; Journal of Computer and System Sciences; Bd. 17, S. 348–375, 1978
- [ML02] MURTHY, P. K.; LEE, E. A.: *Multidimensional Synchronous Dataflow*; IEEE Transactions on Signal Processing; Bd. 50 (8), S. 2064–2079, 2002
- [MRSS01] MAYDL, W.; RAMSAUER, M.; SCHWARZFISCHER, T.; SICK, B.: *ICONNECT: Ein datenflußorientiertes, visuelles Programmiersystem für die Online-Signalverarbeitung*; in: *Engineering komplexer Automatisierungssysteme (EKA 2001)* (Hg. Schnieder, E.); S. 225–247, April 2001

- [MS00] MAYDL, W.; SICK, B.: *Recurrent and Non-recurrent Dynamic Network Paradigms: A Case Study*; in: *IJCNN 2000: Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks (Neural Computing: New Challenges and Perspectives for the New Millennium)*; Bd. 6; S. 73–78, 2000
- [MSG02] MAYDL, W.; SICK, B.; GRASS, W.: *Towards a Specification Technique for Component-Based Measurement and Control Software for Embedded Systems*; in: *Proceedings of the 28th Euromicro Conference*; S. 74–80; Dortmund, September 2002
- [MSG04] MAYDL, W.; SICK, B.; GRASS, W.: *Component-Based Software Development: Case Studies*; Kap. Component-Based Measurement and Control Software for Embedded Systems; World Scientific Publishing, März 2004
- [Mur89] MURATA, T.: *Petri Nets: Properties, Analysis and Applications*; Proceedings of the IEEE; Bd. 77 (4), S. 541–580, April 1989
- [NJN98] NORD, G.; JABON, D.; NORD, J.: *The Global Positioning System and the Implicit Function Theorem*; Siam Review; Bd. 40 (3), S. 692–696, September 1998
- [NLG99] NAJJAR, W. A.; LEE, E. A.; GAO, G. R.: *Advances in the Dataflow Computational Model*; Parallel Computing, 1999
- [Ode96] ODESKY, M.: *Challenges in Type Systems Research*; in: *ACM Workshop on Strategic Directions in Computing Research*; Bd. 28 (4es) von *ACM Computing Surveys*, 1996
- [Pal02] PALMER, R. D.: *Doppler Radar Signal Processing*; University of Nebraska, Dezember 2002
- [Par95] PARKS, T. M.: *Bounded Scheduling of Process Networks*; Dissertation; University of California at Berkeley, Dezember 1995
- [PCH99] PARK, C.; CHUNG, J.; HA, S.: *Extended Synchronous Dataflow for Efficient DSP System Prototyping*; in: *Tenth IEEE International Workshop on Rapid System Prototyping*; S. 196–202, Juni 1999
- [PPL95] PARKS, T. M.; PINO, J. L.; LEE, E. A.: *A Comparison of Synchronous and Cyclo-Static Dataflow*; in: *Proceedings of the IEEE Asilomar Conference on Signals, Systems and Computers*, Oktober 1995
- [Pra05] PRAML, R.: *Entwurf und Implementierung eines Algorithmus zur Stereobildverarbeitung in dem Signalverarbeitungswerkzeug Skylla*, Januar 2005; Interner technischer Bericht (Univerität Passau)
- [RB02] RAUSCH, A.; BROY, M.: *Evolutionary Development of Software Architectures*; Technology for Evolutionary Software Development, November 2002

- [Ree95] REEKIE, H. J.: *Realtime Signal Processing*; Dissertation; University of Technology at Sydney, School of Electrical Engineering, 1995
- [Rei91] REISIG, W.: *Petrinetze – Eine Einführung*; Springer-Verlag, 1991
- [Rei98] REISIG, W.: *Elements of Distributed Algorithms*; Springer-Verlag, April 1998
- [RF99] RINGER, M. A.; FRAZER, G. J.: *Waveform Analysis of Transmissions of Opportunity for Passive Radar*; in: *Fifth International Symposium on Signal Processing and its Applications (ISSPA)*; S. 511–514, August 1999
- [RM96] REHOF, J.; MOGENSEN, T.: *Tractable Constraints in Finite Semilattices*; in: *Third International Static Analysis Symposium*; LNCS 1145; S. 285–300; Springer-Verlag, September 1996
- [Rus04] RUSS, M.: *Embedded Quality*; <http://www.embedded-quality.de/>, 2004
- [Sam97] SAMETINGER, J.: *Software Engineering with Reusable Components*; Springer-Verlag, 1997
- [Sch94] SCHMIDT, D. A.: *The Structure of Typed Programming Languages*; The MIT Press, 1994
- [Sch97] SCHMIDT, D. A.: *Handbook of Computer Science and Engineering*; Kap. Programming Language Semantics, S. 2237–2254; CRC Press, 1997
- [Sch98a] SCHIFFER, S.: *Visuelle Programmierung – Grundlagen und Einsatzmöglichkeiten*; Addison Wesley, 1998
- [Sch98b] SCHMIDT, D. C.: *An Architectural Overview of the ACE Framework: A Case-study of Successful Cross-platform Systems Software Reuse*; in: *USENIX Login Magazine*, 1998
- [Sch02] SCHMIDT, K.: *Explicit State Space Verification*; Humboldt-Universität Berlin, November 2002
- [SH04] SPIES, C.; HEKEL, M.: *Design und Implementierung von Komponenten zur Qualitätskontrolle von Beilagscheiben in dem Signalverarbeitungswerkzeug Skylla*, Dezember 2004; Interner technischer Bericht (Universität Passau)
- [SLSV00] SGROI, M.; LAVAGNO, L.; SANGIOVANNI-VINCENTELLI, A.: *Formal Models for Embedded System Design*; IEEE Design and Test of Computers; Bd. 17 (2), S. 14–27, Juni 2000
- [SMH02] SCHURER, H.; MEYER, P.; HUNINK, J. J.: *Rapid Prototyping of Radar Signal Processing Systems Using ESPADON*; in: *Proceedings of the 3rd Progress Workshop on Embedded Systems*; S. 211–217, Oktober 2002

- [Ste97] STEPHENS, R.: *A Survey of Stream Processing*; Acta Informatica; Bd. 34, S. 491–541, 1997
- [Sto03] STOCKINGER, J.: *Spezifikation und Implementierung eines Editors mit Analysekomponente für Datenflußgraphen mit einfachen Kontrollstrukturen*; Diplomarbeit; Universität Passau, September 2003
- [Str00] STROUSTRUP, B.: *The C++ Programming Language*; Addison-Wesley, 2000
- [SU96] VON DER SCHOOT, H.; URAL, H.: *Protocol Verification by Leaping Reachability Analysis*; in: *Proceedings of the IEEE*; S. 334–339, Oktober 1996
- [Szy02] SZYPERSKI, C.: *Component Software*; Addison-Wesley, 2002
- [Tei97] TEICH, J.: *Digitale Hardware/Software-Systeme*; Springer-Verlag, 1997
- [TTS00] THIELE, L.; TEICH, J.; STREHL, K.: *Regular State Machines*; Journal of Parallel Algorithms and Applications; Bd. 15, S. 265–300, Dezember 2000
- [Uri83] URICK, R. J.: *Principles of Underwater Sound*; 3. Aufl.; McGraw-Hill, 1983
- [VSVA04a] VARDHAN, A.; SEN, K.; VISWANATHAN, M.; AGHA, G.: *Actively Learning to Verify Safety for FIFO Automata*; in: *24th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS'04)*; LNCS 3328; Springer-Verlag, Dezember 2004
- [VSVA04b] VARDHAN, A.; SEN, K.; VISWANATHAN, M.; AGHA, G.: *Learning to Verify Safety Properties*; in: *6th International Conference on Formal Engineering Methods (ICFEM'04)*; Springer-Verlag, November 2004
- [WELP96] WAUTERS, P.; ENGELS, M.; LAUWEREINS, R.; PEPPERSTRAETE, J. A.: *Cyclo-Dynamic Dataflow*; in: *4th EUROMICRO Workshop on Parallel and Distributed Processing*; S. 319–326; Braga, Portugal, Januar 1996
- [Xio02] XIONG, Y.: *An Extensible Type System for Component-Based Design*; Dissertation; University of California at Berkeley, 2002
- [XL00] XIONG, Y.; LEE, E. A.: *An Extensible Type System for Component-Based Design*; in: *Sixth International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*; LNCS 1785; Springer-Verlag, März 2000

Index

- Ablaufsteuerung, 184, 190
- Adapterkomponente, 70, 77, 91, 184
- Aggregationstypen, 58, 98
- Aktualisierung einer Typbelegung, 117
- Analysierbarkeit, 15, 42, 43, 87
- Arithmetische Typconstraints, 109, 116

- Basistypen, 58, 98
- BDF, 15, 31
- Bedeutung eines Programms, 88
- Benutzergruppen, 181
- Blockieren, 141, 145, 165, 166, 170, 235
- Blockiertes Lesen, 16
- Blockiertes Schreiben, 16
- Boolescher Datenfluß, 15, 31

- Colored BDF, 80
- Colored DDF, 85
- Colored Merge, 85, 128, 134, 240
- Colored SDF, 78
- Colored Select, 81, 134
- Colored Switch, 80, 134
- Communicating Finite State Machine, 43
- Computation Tree Logic (CTL), 41
- Constraint-Pattern, 94
- Constraint-Propagationsregel, 110

- Data-driven, 35, 141
- Datenfluß, 15
- Datenflußdominant, 12
- Datenflußgraph, 12, 13, 211
- Datenflußkomponente, 12, 29, 203
- Datenflußparadigma, 13, 15, 29, 91
- DDF, 15, 31
- Deadlock, 16, 30, 36, 128
- Deblocker, 154

- Deblockieren, 141, 142, 145, 164, 166, 235
- Demand-driven, 35, 142
- Denotationelle Semantik, 88
- Depository, 176, 188
- Desired Wordmap, 164
- Determinismus, 31
- Domänenspezifische Programmiersprache, 13, 91
- Dynamischer Datenfluß, 15, 31

- Eingebettetes System, 11
- Erreichbarkeitsgraph, 41, 148
- Errorotyp, 108
- Extended Finite State Machine, 44

- Fifo, 13, 30, 41, 43, 130, 151
- Fifokapazität, 128, 130, 211
- Fifomat, 130
- For-Next-Schleife, 82

- Gekapselter Algorithmus, 74
- Gleichheitsconstraints, 109, 116

- Hierarchische Datenflußkomponente, 137

- If-Then-Else-Konstrukt, 81, 240
- Incrementalmap, 189, 237
- Initialisierungstoken, 16, 30, 33, 36, 70, 184
- Initialisierungswort, 130, 145
- Interface, 96
- Interface-Automaten, 42
- Interfacetyp, 96

- Künstlicher Deadlock, 49
- Kanal, 130
- Kommunikationsprotokoll, 41, 131, 137, 212

Kommutativität, 146, 157, 170, 177
 Komponente, 13, 29
 Komponentenbasierte Softwareentwicklung,
 18, 37
 Komposition, 135
 Kontrollflußdominant, 12
 Kontrollflußkomponente, 77

 Linear Temporal Logic (LTL), 41, 259

 Meet-In-The-Middle-Strategie, 185
 Model Checking, 40, 41
 Modellierungsmächtigkeit, 15, 87

 Nachrichtenalphabet, 130
 Nichtdeterminismus, 32, 70, 72, 75, 134

 Ordered Map, 190, 213, 237
 Overloading, 38, 108, 124

 Parametrischer Polymorphismus, 38, 108,
 124
 Partial Order Reduction, 47, 48, 145, 216,
 225, 237
 Partielle Ordnung, 99
 Physikalisches Signal, 25, 26, 62, 64
 Polymorphismus, 37
 Präfix, 158
 Priorität eines Debblockers, 155
 Priorität eines Knotens, 154
 Produktfifomat, 136
 Protokollkompatibilität, 137

 Qualitätsfalle, 18
 Qualitätssicherung, 18

 Rückkopplungskante, 30, 70, 82, 128, 131,
 139, 184, 211
 Redundanter Knoten, 144

 Scheduling, 35
 Schranken, 102
 SDF, 15, 30
 Signalmerkmale, 61, 93
 Signalsegment, 64, 70, 90

 Signalverarbeitungskomponente, 77
 Simulation, 139
 Speicherüberlauf, 16, 51
 Speicherbedarf, 30, 36, 128
 Speicherverwaltung, 184
 Spin, 48, 244
 Statemap, 148
 Statement Merging, 48
 Strom, 64, 77
 Suchstrategie, 44
 Suffix, 158
 Synchroner Datenfluß, 15, 30

 Tagged-Signal-Modell, 27
 Terminierungsproblem, 118
 Token, 13, 27, 61, 63, 76
 Tokenmaschine, 74, 77, 78, 128, 182, 195
 Transition, 131
 Transport-Overhead, 61
 Typ, 37
 Typbelegung, 115
 Typbestimmung, 40, 109, 113, 115
 Typchecker, 37
 Typconstraints, 21, 39, 94, 109, 116, 203
 Typdomäne, 38, 96, 97
 Typkonvertierung, 38, 123
 Typprüfung, 37
 Typsystem, 37
 Typtrait, 97, 187

 undefinierter Typ, 108

 Verband, 106
 Vergleichsoperationen, 150
 Vollständiger Verband, 106

 While-Schleife, 82, 240
 Wohldefiniertheit, 21, 74, 88, 89, 94
 Wordmap, 149
 Wort, 130

 Zustandsraum, 40
 Zustandsraumexplosion, 43, 47, 175, 188
 Zyklischer Schedule, 15, 30, 36, 128

Ich bin froh, seid Ihr es auch. [Johannes Paul II]