

Robert Laurence Baber

Praktische Anwendbarkeit
mathematisch rigoroser
Methoden zum
Sicherstellen der
Programmkorrektheit



Walter de Gruyter
Berlin · New York 1995

Geleitworte

Die Qualität technischer Produkte muß gesichert und geprüft sein, bevor sie an einen Kunden ausgeliefert werden. Das gilt auch für Programme!

Bei der Programmentwicklung fehlte lange Zeit ein Ansatz, diese Forderung zu erfüllen. Es wurde drauflos programmiert und geglaubt oder gehofft, daß kein Fehler vorliegt, der die Qualität mindert oder das Programm eigentlich unbrauchbar macht. In der Praxis ist es weitgehend immer noch so. Die in den letzten beiden Jahrzehnten gewonnenen wissenschaftlichen Erkenntnisse zur Qualitätssicherung beim Entwurf und der Implementierung sowie dann auch der stichhaltigen, nachvollziehbaren Überprüfung haben sich kaum bei Praktikern eingeführt; es sei zu mathematisch, zu aufwendig, zu schwierig waren (und sind) häufig gehörte Meinungsäußerungen von Programmierern, Software-Entwicklern und Projektverantwortlichen.

Daß dieses Argument auf schwachen Füßen steht, zeigt diese Schrift. Zu einem ingenieurmäßigen Vorgehen beim Programmentwurf und der Programmierung sind kaum mehr mathematische Kenntnisse als die der Booleschen Algebra und der Mengenlehre erforderlich. Der Aufwand schlägt nicht zu Buche, wenn die Maßnahmen zur Qualitätssicherung beim Entwurf und der Programmierung sogleich mitbehandelt werden; es steigert sich sogar der Dokumentationswert, so daß auch eine qualitätsbezogene Überprüfung fast nebenbei abfällt. Und um Schwierigkeiten zu meistern, ist bekanntermaßen intensive Vorbereitung und stetige Übung der einfachste Weg; man muß es einfach tun, Erfahrung im Umgang mit den Formalien stellt sich ein, plötzlich sind die aufgezeigten Vorgehensweisen ganz selbstverständlich.

Jedem, der sich diesen Herausforderungen bei seiner praktischen Arbeit stellen will, sei diese Schrift zum Studium empfohlen.

Univ.-Prof. Dr.-Ing. Hans-Jürgen Hoffmann
Darmstadt, im Februar 1995

Seit der schrittmachenden Arbeit von Robert Floyd in den 60er Jahren sind uns die Grundideen der Methoden zum Sicherstellen der Korrektheit von sequentiellen Computerprogrammen bekannt. Mit den Ergänzungen von (u.a.) Mills, Majster-Cederbaum und de Bruijn haben wir allgemein anwendbare theoretische Grundlagen für dieses Gebiet gewonnen. Heute gibt es keinen theoretischen Grund, die Korrektheit von Programmen, die in der Praxis geschrieben werden, nicht zu beweisen. Trotz diesen erfreulichen Tatsachen muß festgestellt werden, daß diese Ideen in der Industrie sehr selten angewendet werden. Es stellt sich die Frage, "Warum hat sich die Anwendung von mathematischen Methoden im Bereich Software Engineering nicht weiter durchgesetzt?"

Meines Erachtens liegt das Hauptproblem in den bisher erarbeiteten Antworten auf die zwei Darstellungsfragen:

- Wie können wir die durch Programme verwirklichten Funktionen leserlich beschreiben?
- Wie können wir Datenzustände beschreiben?

⊗ Gedruckt auf säurefreiem Papier, das die US-ANSI-Norm über Haltbarkeit erfüllt.

Die Deutsche Bibliothek – CIP-Einheitsaufnahme

Baber, Robert Laurence:

Praktische Anwendbarkeit mathematisch rigoroser Methoden
zum Sicherstellen der Programmkorrektheit / Robert Laurence
Baber. – Berlin ; New York : de Gruyter, 1995
(Programmierung komplexer Systeme ; 8)
Zugl.: Darmstadt, Techn. Hochsch., Diss.
ISBN 3-11-014764-5
NE: GT

© Copyright 1995 by Walter de Gruyter & Co., D-10785 Berlin

Dieses Werk einschließlich aller seiner Teile ist urheberrechtlich geschützt. Jede Verwertung außerhalb der engen Grenzen des Urheberrechtsgesetzes ist ohne Zustimmung des Verlages unzulässig und strafbar. Das gilt insbesondere für Vervielfältigungen, Übersetzungen, Mikroverfilmungen und die Einspeicherung und Verarbeitung in elektronischen Systemen.

Druck: Werner Hildebrand, Berlin. – Buchbinderische Verarbeitung: Dieter Mikolaj, Berlin. – Printed in Germany

In den weit verbreiteten Arbeiten von Floyd, Hoare und Dijkstra wird angenommen, daß eine 1:1 Beziehung zwischen Bezeichnern und Programmvariablen besteht. Diese Annahme ermöglicht eine einfache Beschreibung der Datenzustände. Leider trifft diese Annahme bei den meisten in der Praxis benutzten Programmiersprachen nicht zu. Die Ansätze von Mills, Majster-Cederbaum, de Bruijn und anderen sind allgemeingültig, weil in diesen Arbeiten einfach von "states" gesprochen wird, ohne eine bestimmte Zustandsdarstellungsweise anzunehmen. Dies hat zur Folge, daß der Ingenieur, der diese Methoden anwenden will, seine eigene Darstellungsweise erfinden muß. Es hat sich herausgestellt, daß dies für die Mehrheit der Praktiker zu schwierig ist und daß sie deswegen auf die formale Überprüfung von Programmkorrektheit verzichten.

In dieser Arbeit von Robert Baber bekommen die Leser und Leserinnen praktische Vorschläge und Beispiele, die die Lücke zwischen Theorie und Praxis überbrücken. Es wird gezeigt, wie man die seit 20 Jahren bekannte theoretische Grundlage in einem praktikablen Verfahren anwenden kann.

In der vorliegenden Arbeit findet man Datenzustandsdarstellungen ohne die Einschränkung, die in den Ansätzen von Floyd, Hoare und Dijkstra impliziert sind. Man findet auch neue Vorschläge zur Beschreibung von Funktionen und Prädikaten. Viele Ingenieure werden die hier eingeführte Notation verständlicher als die klassische Schreibweise finden.

Dieses Buch baut auf viele frühere Arbeiten auf und faßt die verschiedenen Ansätze zusammen. Es wird allen in der Praxis beschäftigten Ingenieuren und Informatikern empfohlen. Das umfangreiche Literaturverzeichnis wird auch vielen Forschern behilflich sein.

Prof. Dr. rer. nat. (h.c.) David Lorge Parnas, Ph.D., FRSC
Department of Electrical and Computer Engineering
McMaster University
Hamilton, Ontario, Canada L8S 4K1

Danksagung

Meinem Doktorvater Prof. Dr.-Ing. Hans-Jürgen Hoffmann möchte ich vor allem für seine Geduld mit meinem nichtmuttersprachlichen Deutsch sowie für die Gelegenheiten, mit ihm verschiedene fachliche und technische Aspekte dieser Arbeit in persönlichen Gesprächen zu erörtern, danken. Diese Gespräche führten zu einer klareren und ausführlicheren Darstellung einiger Punkte.

Mein herzlicher Dank gehört dem Korreferenten Prof. Dr. David L. Parnas, der trotz erheblichen Termenschwierigkeiten, deren Überwindung mit Unannehmlichkeiten verbunden war, diese Aufgabe übernommen hat. Darüber hinaus danke ich ihm für mehrere sowohl schriftliche als auch persönliche fachliche Dialoge im Verlaufe der Jahre über dieses und verwandte Themen.

Meiner Frau Ursula und meinem Sohn Eric danke ich für stundenlanges Korrekturlesen. Meiner Tochter Ingrid danke ich für die rechtzeitige Bereitstellung plötzlich benötigter technischer Hilfsmittel.

Eher mittelbar jedoch im wesentlichen Maße haben auch viele andere Personen zur Entstehung dieser Arbeit beigetragen:

Prof. Dr. Hermann Dinges, Prof. Dr. Hilmar Drygas und Dr. John B. Ferebee verdanke ich eine Erweiterung und Vertiefung meiner mathematischen Kenntnisse sowie eine mathematischere Denkweise.

Prof. Dr. Edsger W. Dijkstra und Prof. Dr. C.A.R. Hoare danke ich für ihre Unterstützung zunächst durch ihre Bücher und Artikel, die mich in diesen Stoff eingeführt haben, und danach für ihren in persönlichen Kontakten übermittelten Zuspruch und ihre Ermunterung meiner Überlegungen, die u.a. zu dieser Arbeit geführt haben.

Drs. Willem Dijkhuis, der mich bei *Software Reflected* und meinen späteren Büchern mit wertvollem Rat und Tat unterstützte, bin ich sehr verbunden.

Für Unterstützung und Ermunterung bei verschiedenen ingenieurwissenschaftlichen Bemühungen danke ich Prof. Dr.-Ing. Rudolf Saal und Prof. Dr.-Ing. Adolf Schwab. Prof. Dr.-Ing. Rudolf Saal hat mir auch nützliche Hinweise auf klassische deutschsprachige Lehrbücher für Ingenieurstudenten gegeben.

Zusammenfassende Übersicht

Gegenstand dieser Arbeit ist eine mathematische Grundlage für die Softwareentwicklung, die den wissenschaftlichen Grundlagen der klassischen Ingenieurwissenschaften entspricht, und ihre praktische Anwendung sowohl bei der Programmkonstruktion als auch bei der Programmkorrektheitsbeweissführung. Die hier zusammengestellte Grundlage und Vorgehensweise unterscheiden sich von denen der bisherigen Literatur über die Programmkorrektheitsbeweissführung vor allem durch eine auf die Belange der Praxis gerichtete *Integration* ausgewählter theoretischer Kenntnisse. Die Betrachtung von Programmausführungszuständen z.B. als Folgen von Programmvariablen (Datenumgebungen genannt) ermöglicht die unmittelbare Behandlung der Vereinbarung und der Freigabe von Variablen sowie des Vorhandenseins gleichnamiger Variablen (wie sie z.B. bei blockstrukturierten Programmiersprachen und in rekursiven Programmen vorkommen) in Korrektheitsbeweisen. Weitere aus praktischer Sicht wichtige Aspekte der Programmkorrektheitsbeweissführung und -konstruktion, denen bisher in der einschlägigen Fachliteratur nicht bzw. nicht ausreichend Rechnung getragen worden ist, werden hier behandelt bzw. berücksichtigt, z.B. Rechnerarithmetik (Ungenauigkeit der Gleitkommaarithmetik und ganzzahlige Arithmetik auf beschränkten Intervallen), Definitionsbereiche von Programmanweisungen, das Streben nach Einfachheit sowohl der mathematischen Grundlage als auch ihrer praktischen Handhabung, die informale bis formale, teilweise bis vollständige Anwendung, nicht terminierende Programme, die Systematisierung der praktischen Handhabung usw.

Kapitel 1 behandelt das Problem fehlerhafter Software aus übergeordneter und gesamtgesellschaftlicher Sicht, steckt den Rahmen und die Zielsetzung dieser Arbeit ab und stellt die hier vertretenen Thesen auf. Kapitel 2 enthält eine historische Übersicht über das Gebiet der Programmkorrektheitsbeweissführung und leitet aus dem Gegensatz zwischen Theorie und der gegenwärtigen Anwendungspraxis Anforderungen an einen praxisgerechten Ansatz zur Programmkorrektheitsbeweissführung ab.

Kapitel 3 stellt das mathematische Grundmodell der Programmausführung vor und definiert Programmvariable, Datenumgebung, Ausführungsgeschichte, Programmanweisungen als Funktionen auf den Mengen der Datenumgebungen und Ausführungsgeschichten sowie Vor- und Nachbedingungen. Darauf aufbauend werden Beweisregeln aufgestellt, die der praktischen Beweissführung und der Programmkonstruktion dienen. Neu in dieser Arbeit sind die Begründung für die Struktur des Grundmodells der Programmausführung (insbesondere für die Definition der Datenumgebung), die Definition einer halbstrikten Vorbedingung und die Beweisregeln für halbstrikte und strikte Vorbedingungen. Darüber hinaus werden in Kapitel 3 verschiedene Aspekte der Programmkorrektheitsbeweissführung in der Praxis behandelt.

Kapitel 4 betrachtet die Anwendung der in Kapitel 3 vorgestellten Ideen und Konzepte auf die Konstruktion eines neuen Programms, die die unmittelbare Ableitung mehrerer Teile des zu konstruierenden Programmsegments aus den algebraischen Ausdrücken des Beweisentwurfs ermöglichen. Hinzuweisen ist auf den neuartigen Grad der Abstraktion und des Formalismus im Konstruktionsbeispiel.

Kapitel 5 diskutiert die Bedeutung der hier vorgestellten mathematischen Grundlage für die Konstruktionsänderung ("Wartung"). In Kapitel 6 werden Fragen des Anwendungsauf-

wands und maschineller Unterstützung diskutiert. Es stellt sich heraus, daß nur weitere und aufwendige Untersuchungen, für die die Voraussetzungen noch geschaffen werden müßten und die den Rahmen dieser Arbeit weit übersteigen, diese z.T. recht kontroversen Fragestellungen klären könnten.

Kapitel 7 behandelt spezielle Themen der Programmkorrektheitsbeweissführung: Rechnerarithmetik, bestimmte Aspekte der Nebenläufigkeit, die sich unmittelbar aus der Korrektheitsbeweissführung für sequentielle Programme ergeben, und die objektorientierte Programmierung. Neu in diesem Kapitel sind u.a. die Behandlung von Gleitkommaarithmetik in Korrektheitsbeweisen (insbesondere der Terminierungssatz in Abschnitt 7.1.2 und die Fehlerschranken für Gleitkommaoperationen in Anhang 3) sowie einige Vorschläge bezüglich der OOP-Sprache Eiffel, die verallgemeinert werden können.

Kapitel 8 enthält die wesentlichsten Schlußfolgerungen der vorliegenden Arbeit.

Neu in dieser Arbeit ist weiter der in Anhang 2 enthaltene Vorschlag zur Gliederung und Präsentation eines Korrektheitsbeweises. Darin werden die Struktur und der logische Aufbau des Beweises einerseits von den algebraischen Umformungsschritten andererseits streng getrennt, welches die Verständlichkeit, Übersichtlichkeit und Nachvollziehbarkeit erheblich verbessert. Dadurch wird z.B. die Nachprüfung durch andere Personen erleichtert und unterstützt.

Anhang 4 faßt die Korrektheitsbeweissführung zusammen mit einem Beweis der vollständigen Korrektheit für ein rekursives Unterprogramm; soweit für die aufgeworfene Fragestellung relevant werden alle für den praktischen Einsatz geforderten Eigenschaften erfaßt. Der bewiesene Korrektheitssatz enthält eine Aussage über die Rekursionstiefe, die als Maßstab für die maximale Speicherbelegung dienen kann, und sieht eine endliche Menge für die Zahlendarstellung vor.

Hinweise für den Leser

Zitate auf die Literatur sind im Text in eckigen Klammern [...] gesetzt. In solchen Zitaten erscheint zuerst der Name des Verfassers ggf. gefolgt von zusätzlichen Angaben, meist das Erscheinungsdatum. Ein Strich [—; ...] bedeutet, daß kein Verfasser angegeben ist; solche Eintragungen findet der Leser am Ende der Liste der Literaturhinweise. Steht in einem Zitat nur der Name des Verfassers ohne ergänzende Angaben, dann sind alle (falls mehrere) in den Literaturhinweisen angegebenen Veröffentlichungen des Verfassers zutreffend.

Das Zeichen ■ deutet auf das Ende eines Beweises hin.

In eckigen Klammern [] und rechtsbündig gesetzte Angaben sind Kommentare. Solche Kommentare kommen vor allem in Beweisen und in Folgen von algebraischen Umformungen vor.

Inhaltsverzeichnis

1. Einleitung	1
1.1 Problemstellung und Lösungsansatz	1
1.2 Zielsetzung und Abgrenzung des Themas	6
1.3 Thesen	7
2. Bisherige Entwicklungen auf dem Gebiet der Programmkorrektheitsbeweissführung	8
2.1 Historische Übersicht	8
2.1.1 Der Ansatz von Floyd zur Programmkorrektheitsbeweissführung	8
2.1.2 Die zunehmende Problematik der Softwareentwicklung	9
2.1.3 Die Korrektheitsansätze von Hoare, Dijkstra und Mills	9
2.1.4 Die Entwicklung von Spezifikationsprachen	10
2.1.5 Grundsätzliche Fragen zur Programmkorrektheitsbeweissführung	12
2.1.6 Anwendungsorientierung der Programmkorrektheitsfachliteratur	12
2.1.7 Beweissführungswerkzeuge	13
2.1.8 Operationale, denotationale und axiomatische Programmsemantik	14
2.1.9 Alternativen zur Softwarequalitätssicherung	14
2.1.10 Grenzgebiete der Programmkorrektheitsbeweissführung	16
2.1.11 Korrektheitsbeweissführung und objektorientierte Programmierung	17
2.1.12 Programmkorrektheitsbeweise und Normen	17
2.1.13 Gegenwärtige Forschungsprojekte	17
2.1.14 Korrektheitsbeweissführungstechniken im praktischen Einsatz	18
2.1.15 Schlußbemerkungen	19
2.2 Gegenwärtige Situation	20
2.2.1 Der Gegensatz zwischen Theorie und Anwendungspraxis	20
2.2.2 Anforderungen an einen praxisgerechten Ansatz zur Programmkorrektheitsbeweissführung	23
3. Eine praxisgerechte theoretische Grundlage für die Programmkorrektheitsbeweissführung	28
3.1 Grundbegriffe, -betrachtungen und Definitionen	31
3.1.1 Programmvariablen, Datenumgebungen und Ausführungsgeschichten	31
3.1.2 Werte von Variablen und Ausdrücken in Datenumgebungen	32
3.1.3 Programmanweisungen als Funktionen auf \mathbb{D}	32
3.1.4 Programmanweisungen als Funktionen auf \mathbb{D}^*	35
3.1.5 Vorbedingungen und Nachbedingungen	35
3.1.6 Nachbedingungen mit Bezug auf vorherige Variablenwerte	38

3.2 Allgemein gültige Sätze ("Beweisregeln")	39
3.2.1 Stärkung einer Vorbedingung, Schwächung einer Nachbedingung	39
3.2.2 Programmanweisungen und ihre Zusammensetzungen	39
3.2.3 Zerlegung von Vor- und Nachbedingungen	41
3.2.4 Programmsegment, Unterprogramm	41
3.2.5 Beweisregeln für strikte, halbstrikte und umfassende Vorbedingungen	42
3.3 Korrektheitsbeweissführung für sequentielle Programme	44
3.3.1 Voraussetzungen	44
3.3.2 Vorgehensweise ohne maschinelle Unterstützung	46
3.3.3 Anwendungsbeispiele	48
3.3.4 Potentielle Fallen	55
3.3.4.1 Herausnehmen eines Terms aus einer Reihe	55
3.3.4.2 Zuweisung zu einer Feldvariable	56
3.3.4.3 Nicht definierte Ausdrücke	59
3.3.4.4 Unterprogrammaufruf mit formaler Parameterübergabe	60
3.3.4.5 Computerarithmetik	61
3.3.4.6 Rekursion	62
3.4 Mathematische Anforderungen an den Softwareentwickler	63
3.4.1 Vorkenntnisse	64
3.4.2 Verwendete Notationsformen	65
3.5 Stufenweise Gestaltungsmöglichkeiten für unterschiedliche Anwenderkategorien	66
3.5.1 Mathematische Vorkenntnisse, Terminologie und Schreibweise	67
3.5.2 Informelle gegenüber formelle Anwendung	68
3.5.3 Abstraktionsgrad der Spezifikation und Aufgabenstellung	71
4. Bedeutung der Korrektheitsbeweissführung für die ingenieurmäßige Neukonstruktion eines Programms	74
4.1 Aus der Korrektheitsbeweissführung sich ergebende Anforderungen an und Leitlinien für die Konstruktion	75
4.2 Ansätze und Möglichkeiten zur Ableitung von Teilen eines Programms	79
4.3 Konstruktionsbeispiel	81
4.4 Spezifikationen bei der Konstruktion und im Korrektheitsbeweis	85
4.5 Anforderungen an die Dokumentation eines Unterprogramms	88
5. Bedeutung der Korrektheitsbeweissführung für die Konstruktionsänderung in instabiler Umgebung	91
5.1 Spezifikationen von Unterprogrammen	92
5.2 Eingrenzung der Auswirkungen einer Änderung	95
5.3 Konstruktionsleitlinien für die "Änderungsfreundlichkeit"	100

6. Anwendungsaufwand, Voraussetzungen und maschinelle Unterstützung für eine breitere Anwendungsakzeptanz	102
6.1 Lernphase	103
6.2 Programmverifikation	104
6.2.1 Manuelle Korrektheitsbeweissführung	105
6.2.2 Maschinelle Unterstützung	108
6.3 Programmkonstruktion	110
6.4 Konstruktionsänderung	111
7. Spezielle Aspekte der Programmkorrektheitsbeweissführung	112
7.1 Computerarithmetik	112
7.1.1 Ganzzahlenarithmetik auf einem beschränkten Intervall	112
7.1.2 Gleitkommaarithmetik	115
7.2 Nebenläufigkeit aus der Sicht der Korrektheitsbeweissführung	120
7.2.1 Wechselwirkung zwischen nebenläufigen Prozessen in einem Korrektheitsbeweis	121
7.2.2 Ein Korrektheitsbeweis für Kommunikation zwischen nebenläufigen Prozessen	123
7.2.2.1 Übertragungskanal ohne Puffer	123
7.2.2.2 Gepufferter Übertragungskanal	133
7.2.3 Vollständige Korrektheit und Verklemmung/Verhungern (lassen)	136
7.3 Objektorientierte Programmierung am Beispiel der OOP-Sprache Eiffel	137
7.3.1 Korrektheitsbeweissführungsrelevante Besonderheiten von Eiffel	138
7.3.1.1 Variablen und Objekte	138
7.3.1.2 Routinen	140
7.3.1.3 Zugriffseinschränkungen	141
7.3.1.4 Ausnahmebehandlung	142
7.3.1.5 Unbestimmte Bezüge auf Variablen und Funktionen	142
7.3.2 Die korrekttheitsbezogenen Konstrukte in Eiffel	143
7.3.3 Beispiel der Korrektheitsbeweissführung für eine Klasse	145
7.3.4 Verbesserungs- und Erweiterungsvorschläge zu den korrekttheitsbezogenen Konstrukten in Eiffel	152
8. Schlußfolgerungen	156
8.1 Erreichtes	156
8.2 Als offen Erkanntes	156
8.3 Handlungsbedarf	157

Anhang 1. Beispiele unterschiedlicher Spezifikationsstufen	159
A1.1 Kernreaktorüberwachung	159
A1.1.1 Allgemeine Beschreibung der Überwachungsaufgabe	159
A1.1.2 VDM-Spezifikationen unterschiedlicher Abstraktionsgrade	160
A1.1.3 Anwenderorientierte Vorgehensweise zur Erarbeitung der Spezifikation	162
A1.2 Kellerspeicher	165
A1.2.1 Abstrakte Spezifikationen eines Kellerspeichers (Funktionensystem)	165
A1.2.2 Abstrakte Spezifikation eines Systems parameterloser Unterprogramme	165
A1.2.3 Spezifikation durch Vor- und Nachbedingungen	167
A1.2.4 Konkrete Spezifikationsstufe 1 eines Unterprogrammsystems	168
A1.2.5 Konkrete Spezifikationsstufe 2 eines Unterprogrammsystems	168
Anhang 2. Dokumentationsbeispiel mit Korrektheitsbeweis für den Programmteil "Aufteilen"	171
Anhang 3. Einige Fehlerschranken für Gleitkommaarithmetik	186
A3.1 Definition und Axiome eines Gleitkommaarithmetiksystems	186
A3.2 Genauigkeit der Gleitkommaarstellung	188
A3.3 Genauigkeit der Gleitkommaaddition	189
A3.4 Genauigkeit der Gleitkommamultiplikation	190
A3.5 Rundungseigenarten	190
Anhang 4. Korrektheit eines rekursiven Unterprogramms	191
Literatur- und Quellenhinweise	201
Literaturhinweise	201
Persönliche Kommunikation	236

Abbildungen

Die Zustandsauslegungsfunktion ist nicht umkehrbar	29
Kellerspeicher gleichnamiger Variablen in einer Datenumgebung	33
Beziehungen zwischen dem Definitionsbereich einer Programmanweisung, einer gewöhnlichen Vorbedingung, einer strikten Vorbedingung, einer umfassenden Vorbedingung und dem Urbild der Nachbedingung	37
Die drei Arten von Vor- und Nachbedingungen eines Aufrufs auf ein Unterprogramm	88
Korrektes Funktionieren trotz fehlerhaftem Unterprogramm	99
Korrektes Funktionieren trotz fehlerhaftem Aufruf	99
Datenfluß über den Übertragungskanal ohne Puffer	124
Die einzelnen Zustandsübergangsdigramme für den Übertragungskanal ohne Puffer	131
Produktzustandsübergangsdigramm für den Übertragungskanal ohne Puffer	132
Petri-Netz mit Anfangsmarkierung für den Übertragungskanal ohne Puffer	133
Kernreaktorsystemübersicht	160
Schaltbild für die Ermittlung des Überwachungsmodus	163
Ermittlung des Soll-Zustands einer Anzeigelampe	163
Alternative Ermittlung des Soll-Zustands einer Anzeigelampe	164

1. Einleitung

1.1 Problemstellung und Lösungsansatz

In den etwa 50 Jahren, seitdem die ersten elektromechanischen Rechenanlagen konzipiert und gebaut wurden, sind solche Systeme beeindruckend weiterentwickelt worden. Sowohl die Leistungsfähigkeit (Geschwindigkeit und Speichergröße) als auch die Zuverlässigkeit der Hardware sind um Größenordnungen erhöht worden. Gleichzeitig sind die Kosten solcher Anlagen stark gesenkt worden, so daß viele neue Anwendungsgebiete wirtschaftlich geworden sind. Darunter befinden sich aus gesamtgesellschaftlicher Sicht besonders wichtige Anwendungen, z.B. die Abwicklung von vielfältigen Verwaltungs- und sonstigen Vorgängen in der Wirtschaft, die Flugsicherung usw. Seit einiger Zeit bieten sich auch sicherheitskritische Aufgaben als wirtschaftlich interessante Anwendungsgebiete an und mit der Erschließung dieses Potentials hat man bereits begonnen, z.B. für die Steuerung militärischer und Zivilflugzeuge (A320). Eine deutliche Zunahme sicherheitskritischer Anwendungen wird sich in den kommenden Jahren voraussichtlich abzeichnen, z.B. für Steuerungsaufgaben in Kraftwerken, Kernreaktoren, neuen Flugzeugmodellen [Rushby] und Fahrzeugen aller Art. In manchen solchen Systemen ist eine rechnergestützte Steuerung sogar unerlässlich, z.B. in aerodynamisch instabilen Hochleistungsflugzeugen.

Die Entwicklungen auf der Softwareseite haben bei weitem nicht Schritt gehalten, obwohl auch hier Fortschritte erzielt worden sind. Vor allem in Bezug auf die Zuverlässigkeit bleibt viel zu wünschen übrig. Wegen der mangelnden Zuverlässigkeit der Software können Rechnersysteme die Anforderungen sicherheitskritischer Anwendungen heute nicht erfüllen [Littlewood; 1993], [Dunn]. Potentiell sicherheitskritische Untersysteme mit eingebetteter Software werden oft durch Hardware-Sperren und -Umgehungen ergänzt, die die kritischen Funktionen übernehmen; auf diese Weise wird Software außerhalb der sicherheitskritischen Teile des Systems herausgehalten. Diese einschränkende Konstruktionsstrategie schließt offensichtlich viele potentiell interessante — darunter auch sicherheitserhöhende — Möglichkeiten aus. Der gegenwärtige Stand der Softwareentwicklungstechnik verhindert also die Ausschöpfung vieler hardwaretechnisch möglichen, wirtschaftlich vorteilhaften und systemtechnisch wünschenswerten Anwendungsmöglichkeiten.

Die Grenze für die Zuverlässigkeit rechnerbasierter Systeme wird heute also von der Software gesetzt. Aus technischer Sicht sollte diese Grenze stattdessen von physikalischen Eigenschaften der Hardware bestimmt werden, denn Hardwarekomponentenausfall nach der Inbetriebnahme (z.B. wegen Abnutzung, thermischer oder physischer Einwirkung, Veränderung der Eigenschaften durch Alterung usw.) kann grundsätzlich nicht ganz ausgeschlossen werden. Software, auf der anderen Seite, unterliegt derartigen Veränderungen nicht; sie kann nicht erst nach der Inbetriebnahme ausfallen. Ein von der Software verursachtes Versagen eines Systems während des Betriebs ist immer auf einen Fehler zurückzuführen, der sich von vornherein in der Software befand [VDI/VDE 3542 Blatt 4], [Grams; 1993 Juni] — also auf einen Entwurfsfehler [Gardener], [Littlewood; 1993]. Im Gegensatz dazu sind Hardwarefehler vorwiegend Komponentenausfälle; Hardwareentwurfsfehler kommen verhältnismäßig selten vor.

“Trotz beträchtlicher Fortschritte wird jedoch die systematische Erstellung von hinreichend zuverlässigen Programmsystemen unter ökonomischen Randbedingungen noch unvollständig beherrscht” [Fachbereich 2 der GI]. Die Probleme bei der Softwareentwicklung kennzeichnen einen Prozeß, den man intellektuell nicht in Griff hat [Cobb]. Man müßte auf die Softwarezuverlässigkeit mindestens so viel Wert legen wie auf die Softwareentwicklungsproduktivität [Yourdon; 1992], denn “bei der heutigen immer stärker werdenden Verflechtung der Computernetze mit ihren weitverzweigten Auswirkungen ist eine zuverlässige Software aber eine Existenzfrage für unsere Gesellschaft” [Zuse, Vorwort in VDI-GIS; 1993].

Fehlerhafte Software hat bereits nicht nur zu hohen finanziellen Verlusten sondern auch zu Todesfällen geführt [Joyce; 1987 May 15], [—; “Lethal dose”, 1987 Dec.], [Thomas; 1988 July 4], [Leveson; 1993 July], [Acklin]. Es gibt keinen Grund zu der Annahme, daß sich diese Problematik verringern wird; vielmehr muß mit einer Zunahme solcher Fälle gerechnet werden, solange Software auf die bisherige Weise entwickelt wird.

Über ernsthafte negative Auswirkungen von softwarebezogenen Unzulänglichkeiten und Fehlern in Rechnersystemen wird in der Rubrik “Risks to the Public in Computers and Related Systems” [Neumann] regelmäßig berichtet. Darüber hinaus erscheinen seit Jahren nicht nur in der öffentlichen Presse sondern auch in der Fachliteratur viele anekdotische Berichte über Pannen und Ausfälle verschiedener Art, die auf Softwarefehler zurückzuführen sind. Auch die Softwarefachwelt sorgt also dafür, daß diese Problematik bekannt wird und bleibt.

Auf die Problematik der mangelhaften Softwarezuverlässigkeit wird auf verschiedene Weise reagiert. Ein Konferenzteilnehmer (ein Softwareentwickler) schilderte die Strategie seines Arbeitgebers, “Es gibt nie die Zeit, es richtig zu machen, aber immer die Zeit, es nochmals zu machen.” Für die naheliegende Annahme, daß die typische Arbeitsweise in der Softwareentwicklung nicht nur zu einer niedrigen Qualität sondern auch zu einem Produktivitätsverlust führen muß, findet man in der wissenschaftlichen Literatur eine interessante Unterstützung: Eine positive statistische Korrelation zwischen Zuverlässigkeit und Produktivität ist beobachtet worden [Card].

Auf der rechtlichen Seite gibt es von Zeit zur Zeit immer wieder und in mehreren Ländern Bestrebungen dahingehend, Softwarelieferanten für die Konsequenzen von Fehlern in ihrer Software haftbar zu machen sowie Softwareentwickler sich prüfen und in ein Register eintragen zu lassen. Hinweise auf solche Bestrebungen in den U.S.A. in letzter Zeit enthalten z.B. [Davis], [Palermo] und [Trubow]. [Bartsch] und [VDI-GIS; Abschnitt 5.2] enthalten kurze Übersichten über Produkthaftung für Software und verwandte Rechtsfragen in Deutschland.

In der Kernreaktorindustrie reagiert man auf die Problematik der mangelhaften Softwarezuverlässigkeit mit einer Abneigung gegen den Einsatz von Software in sicherheitskritischen Systemteilen [Gardener]. Diese Einstellung vertritt auch [Littlewood; 1992 Nov., 1993]. Über eine nicht überzeugende Behandlung sicherheitskritischer Software an Bord von Flugzeugen bei der Bearbeitung eines Antrags auf Erteilung einer Flugtauglichkeitsbescheinigung berichtet [Mellor]: Effektiv werde Software dabei weitgehend ausgeklammert; man verzichte auf quantitative Nachweise der Zuverlässigkeit (nur) bei der Software.

In diesen Reaktionen kann man Beispiele der in [Baber; 1982, 1986] beschriebenen Zukünfte A (ruchlos, aberwitzig, engl. “audacious”) und B (reaktionär, beharrend, engl. “backward”) erkennen. Im Gegensatz dazu sind Anzeichen der Zukunft C (radikal, cherubinisch, engl. “celestial”) zur Zeit kaum erkennbar.

Die Beobachtung, daß Softwarefehler immer Entwurfsfehler sind (siehe oben), deutet daraufhin, daß das Wesen der eigentlichen Problematik nicht in der Unterscheidung zwischen Hardware und Software sondern in der Unterscheidung zwischen Entwurfsfehler und Komponentenausfall zu suchen ist. Sich auf die Unterschiede zwischen Hardware und Software zu konzentrieren lenkt die Aufmerksamkeit von den eigentlichen Ursachen des Problems fehlerhafter Software ab. Diese Überlegung führt zum Vorschlag, die in den klassischen Ingenieurdisziplinen typische Vorgehensweise beim Konstruieren (Entwerfen) als Vorbild und Basis für die Softwareentwicklung zu verwenden, d.h. den zu konstruierenden Gegenstand als mathematisches Objekt zu betrachten und die Eignung des Entwurfs für den beabsichtigten Zweck (Spezifikation, Pflichtenheft) vor dem Bau analytisch und rechnerisch zu prüfen. Siehe z.B. [Baber], [Barroca], [Borer], [Davis], [Dijkstra; 1982 (p. 273), 1989 May, 1992 Nov. 22], [Fößmeier], [Gerhart; 1989 Nov.], [Gries; 1991], [Grams; 1993 Juni], [Gruman; 1989 July], [Hoare; 1978, 1982 May, 1984 April], [Linger], [Mills], [Parnas], [Shaw], [Thomas; 1993], [Turski; 1978], [Weber].

Diese Idee ist grundsätzlich von allen befürwortet worden, aber im Detail gehen die Meinungen erheblich auseinander, insbesondere in Bezug auf die Frage ihrer eigentlichen Bedeutung für die Praxis sowie für die Lehre. Die gegenwärtig unter dem Begriff “Software-Engineering” umfaßte Sammlung von Konzepten, Erkenntnissen usw. befaßt sich überwiegend mit organisatorischen, projektmanagement- und führungsorientierten Aspekten der Softwareentwicklung [Card], [BWB], [DKE; 1987], [Evans], [Frühau], [IEC/SC 65A], [IEEE; *Software Engineering Standards*, 1987, 1989], [Kimm], [Macro], [Norris], [Somerville; 1985], [Wallmüller], [—; *Ticklt Guide*, 1990 Sept. 30]. So wertvoll diese Sammlung sein möge, sie trifft den Kern einer Ingenieurwissenschaft nicht und stellt nicht das dar, was im vorherigen Absatz gemeint ist. Die Beispiele und Geschichten der klassischen Ingenieurfachrichtungen zeigen, daß Entwurfsfehler durch die konsequente Anwendung einer theoretischen, wissenschaftlichen und mathematischen Grundlage (also durch *technische*, nicht organisatorische Maßnahmen) vermieden werden. Organisatorische und führungsorientierte Maßnahmen wirken dabei lediglich unterstützend; insbesondere erhöhen sie die Effizienz und Wirtschaftlichkeit, mit denen die Ingenieuraufgaben in der Praxis ausgeführt werden, d.h. mit denen die eigentliche — technische — Lösung in die Praxis umgesetzt wird. Diese Tatsache wird in der heutigen Softwarebranche weitgehend übersehen.

Der Mangel an der theoretischen Grundlage bzw. an ihrer praktischen Anwendung ist von manchen kritisiert worden, z.B.: “A severe problem in the software field is that, strictly speaking, there are no software engineers” [Parnas; 1988 May], siehe auch [Gries; 1991]. “Software engineering without formal methods as a basis should be considered as ludicrous as aeronautical engineering without the analytic and predictive capabilities of applied mathematics” [Gerhart; 1989 Nov.]. “put ... more engineering in software engineering” [Gruman; 1989 July]. Gerade auf dem Softwaregebiet fehlt weitgehend der Ausgleich zwischen Theorie (ratiocinatio) und Praxis (fabrica), den bereits vor 2000 Jahren [Vitruvius; Buch I, Kapitel 1] für eine Ingenieurdisziplin (damals Architektur bzw. Bauwesen) verlangte. Diese Kluft wird durch die Ansicht der Informatik-Fachliteratur klar ersichtlich: die überwiegende Mehrheit der Bücher, Artikel usw. weist entweder eine deutlich theoretische oder praktische Orientierung auf; diejenigen, die beide Aspekte ausgewogen behandeln, sind relativ selten. Man vergleiche z.B. die Literatur über “formale Methoden” und “Programm-Korrektheit” einerseits und “Software-Engineering” andererseits. Diese Kluft findet man bei den klassischen Ingenieurfachrichtungen nicht, siehe z.B. Lehr-

bücher für Studenten des Maschinenbaus [Föppl], [Meriam], [Keenan], der Elektrotechnik [Feldkeller], [Küpfmüller], [Oppelt], [Frank], [Guillemin], [Newton] usw., die typischerweise eine klare praktische Orientierung mit einem rigorosen theoretischen Inhalt vereinbaren. Auch die Abneigung gegen Mathematik schlechthin, die unter Softwarepraktikern verbreitet und sogar auch unter deutschen Diplom-Informatikern zu finden ist, ist unter Ingenieuren der klassischen Fachrichtungen nicht bekannt. Selbst auf der Hobbyseite kann man diesen Unterschied zwischen der "Computerei" und z.B. dem Amateurfunkwesen (selbst vor vielen Jahren) beobachten [American Radio Relay League].

Eine zusätzliche Problematik betrifft die Frage, ob eine Spezifikation "richtig" ist bzw. den eigentlichen Anwendungsbedarf ausreichend erfüllt. Dieser Aspekt der Systemgestaltung und -Konstruktion wird in der vorliegenden Arbeit lediglich am Rande angesprochen. U.a. sind Wechselwirkungen zwischen Hardware und Software dieser Kategorie zuzuordnen. Eine Situation, in der ein Strom- oder Hardwareausfall zu einer fehlerhaften Datenstruktur führt, obwohl dies durch eine geeignete Softwaregestaltung vermeidbar gewesen wäre, stellt ein Beispiel einer solchen für den Anwendungsbedarf nicht ausreichenden Spezifikation dar.

Der Begriff "software engineering" hat ihren Ursprung in einer vom NATO Scientific Committee geförderten Konferenz, die 1968 in Garmisch-Partenkirchen stattfand. Zur Erinnerung daran fand 25 Jahre später die European Software Engineering Conference 1993 ebenfalls in Garmisch-Partenkirchen statt. Dort wurde, vor allem in nicht veröffentlichten Beiträgen und Teilnehmerkommentaren, die Meinung vertreten, daß nach "25 Jahren Software-Engineering" immer noch nicht von "engineering" die Rede sein kann.

Trotz dieser Kluft zwischen der Softwareentwicklungspraxis und den klassischen Ingenieurdisziplinen können sich Computerspezialisten, darunter auch Softwareentwickler, in Großbritannien über die British Computer Society in das Ingenieurregister als "Chartered Engineer" (CEng) eintragen lassen. Eine ingenieurmäßige Vorgehensweise bei der Gestaltung von rechnerbasierten Systemen erhält dadurch Anerkennung sowie eine gewisse ideale Förderung. In den U.S.A. wird über einen vergleichbaren Schritt nachgedacht [Grujan; 1989 Nov.], [Davis]. In Deutschland ist die berufliche bzw. rechtliche Anerkennung als Ingenieur nicht von einer solchen Eintragung abhängig; nach dem Hessischen Ingenieurgesetz z.B. ist jeder berechtigt, die Berufsbezeichnung "Ingenieur" zu führen, wer eine entsprechende technische oder naturwissenschaftliche Ausbildung mit Erfolg abgeschlossen hat [Hessischer Landtag; 1970 Juli 21]. Im Gegensatz zu einigen anderen professionellen Berufen (z.B. Medizin, Recht) kommt es weder auf die ausgeübte Tätigkeit noch auf den Nachweis praktischer Berufserfahrung an. Ob die Eintragung in die Liste der Beratenden Ingenieure in Hessen auch Softwarespezialisten offen steht oder nicht, geht aus dem Text des Hessischen Ingenieurkammergesetzes nicht eindeutig hervor. Die Fachrichtung scheint jedoch keine Rolle zu spielen; die wesentlichen Voraussetzungen für die Eintragung sind die Berechtigung, die Berufsbezeichnung "Ingenieur" zu führen (die wiederum nur von der Ausbildung abhängt, siehe oben), eine dreijährige praktische Tätigkeit als Ingenieur und eine geschäftlich-gewerbliche Unabhängigkeit [Hessischer Landtag; 1986 Oktober 7].

Die theoretische Grundlage für ein echtes Softwareingenieurwesen wird von manchen in der Mathematik gesehen. (Siehe die oben zitierte Literatur sowie [Berzins], [Thomas; 1993 Jan.-Feb.]) Techniken, Methoden und Vorgehensweisen, die mathematische Beweise der Übereinstimmung von Programmen und Spezifikationen ermöglichen, werden oft "formale Methoden" genannt. Über erfolgreiche praktische Erfahrungen damit (z.B. Erhöhung der Qualität und Produktivität, Verbesserung der Kommunikation zwischen allen Beteilig-

ten) ist berichtet worden, aber auch teilweise starke Einwände gegen die praktische Anwendbarkeit formaler Methoden werden geäußert (siehe Abschnitte 2.1 und 2.2 unten). Dabei hängt der Erfolg nicht nur vom technischen Wert der formalen Methoden, sondern auch von kulturellen Faktoren [Gerhart; 1990 Sept.], der wirtschaftlichen Umgebung [Martin, Alain J.; 1992], geschäftlichen Prioritäten, Training, mathematischen Vorkenntnissen usw. ab. Auch die "Verpackung" (Präsentationsform) mathematisch rigoroser Methoden ist für ihre Akzeptanz und praktische Anwendung wichtig. Man muß feststellen, daß das Problem der geeigneten "Verpackung" formaler Methoden noch nicht ausreichend gelöst ist: "The scientific results ... must also be organized in a form that is useful to practitioners. Computer science has a few models and theories that are ready to support practice, but the packaging of these results for operational use is lacking" [Shaw]. Zur Ablehnung formaler Methoden trägt sicherlich auch die oben erwähnte Abneigung gegen Mathematik seitens vieler Softwareentwickler bei.

Auch auf dem Gebiet der Computerhardwareentwicklung gibt es Probleme mit Entwurfsfehlern, insbesondere hinsichtlich der Logik in komplexeren Untersystemen (z.B. Arithmetikeinheiten), die den Problemen mit fehlerhafter Software sehr ähneln. Über positive Erfahrungen mit "formaler Verifikation" (Analyse basierend auf diskreter und logischer Mathematik) zur Beherrschung dieser Probleme ist berichtet worden [Bryant], [—; "Formal Verification: ...", 1989 Dec.]; siehe auch [Goossens], [Kumar], [Musgrave], [Thuau]. Hauptsächlich wegen der Kostenstruktur erwartet [Martin, Alain J.; 1992], daß formale Verifikation bzw. Methoden zuerst für die Hardwarekonstruktion und erst später für die Softwarekonstruktion Anwendung finden werden.

Seit vielen Jahren sucht man alternative Lösungen zum Problem der Softwarefehler, ohne eindeutigen Erfolg. Trotz des berühmten Spruchs von Dijkstra, Testen kann nur das Vorhandensein von Fehlern, jedoch nicht die Fehlerfreiheit eines Programms zeigen [Buxton; S. 21, 85], wird immer weiter getestet und dabei ein sehr hoher Aufwand getrieben — obwohl niemand die Richtigkeit des Dijkstra-Spruchs ernsthaft in Frage gestellt hat. Testen bleibt die in der Praxis wichtigste Maßnahme zur Softwarequalitätssicherung.

Ein anderer Ansatz gilt der Fehlertolerierung (Fehlertoleranz, N-version programming, diversitäre Programmierung, dissimilar redundancy). Diese Vorgehensweise hat zwar in manchen Anwendungen zu einer Qualitätsverbesserung geführt [Anderson, Tom] und wird in der Praxis eingesetzt [Potocki], sie hat jedoch die ursprünglichen Erwartungen nicht erfüllt. Die unabhängige Programmentwicklung führt in der Regel nicht zu statistisch unabhängigem Versagensverhalten der fraglichen Programmteile [Eckhardt], [Knight], [Lindeberg], [Littlewood; 1989, 1993]. Wegen vergleichbarer Ausbildung, ähnlicher Erfahrung usw. sowie aus psychologischen Gründen neigen Programmierer dazu, gleichartige Fehler zu machen und in ihre Programme einzubauen [Grams; 1990, 1992].

Andere Ansätze zur Softwarequalitätsverbesserung, z.B. organisatorische Maßnahmen an verschiedenen (und allen) Stellen des Entwicklungsprozesses, haben vergleichbare Ergebnisse gebracht: gewisse Verbesserungen, aber keine grundlegende Lösung des Problems der Softwarefehler.

Es herrscht eine breite Übereinstimmung darüber, ein wesentlicher Aspekt des Problems liege in der Komplexität und im Umfang der Aufgaben, die typischerweise mittels Software gelöst werden sollen. Manche vertreten die Meinung, die grundsätzliche Lösung sei in einer Kombination von (1) bewußtem und gezieltem Streben nach Vereinfachung [Abrial], [Gardener], [Potocki], ("You don't understand the problem until you can simplify it" [Bennett]) und (2) Beherrschung der übriggebliebenen Komplexität mit Hilfe der

Mathematik [Thomas; 1993] zu suchen. “Let the symbols do the work” durch VLSAL, “Very Large Scale Application of Logic” [Dijkstra; S. 7, EWD 1041]. Diese Kombination stellt einen wesentlichen Teil des Ziels und Leitfadens für die vorliegende Arbeit dar.

1.2 Zielsetzung und Abgrenzung des Themas

Die Zielsetzungen dieser Arbeit sind:

- Eine Sammlung von mathematischen Grundsätzen ist zusammenzustellen, die als Basis für das Konstruieren fehlerfreier Programme und das Beweisen ihrer Korrektheit dienen kann.
- Diese Sammlung soll möglichst kompakt, übersichtlich, in sich weitgehend vollständig, leicht in Erinnerung zu behalten und in der Praxis leicht handhabbar sein.
- Sie soll die Durchführung in der Praxis typischer vorkommender Konstruktions- und Beweisaufgaben mit möglichst geringem Aufwand — ohne maschinelle Unterstützung — ermöglichen bzw. unterstützen.
- Dabei sollen möglichst wenige für den Anwender (den Softwareentwickler) neue mathematische Konzepte und Vorkenntnisse vorausgesetzt werden. Die gestellten Voraussetzungen sollen leicht zu erlernen und einfach anzuwenden sein.
- Die Notation soll aus Anwendersicht möglichst einfach bzw. bereits bekannt sein.
- Die Anwendung dieser Grundlage in der täglichen Arbeit soll möglichst wenige organisatorische Vorbereitungen und Unterstützung voraussetzen; sie soll idealerweise von einzelnen Softwareentwicklern einsetzbar sein, unabhängig davon, ob Kollegen sie einsetzen oder nicht.

Die Betonung liegt auf Verständlichkeit sowie auf der manuellen Anwendbarkeit und Verwendbarkeit. Rechnerunterstützung wird weder vorausgesetzt noch ausgeschlossen.

Im Vordergrund steht die effektive und effiziente Lösung praktischer Aufgaben. Dabei soll so viel (und nur so viel) Theorie herangezogen werden, wie zur Erreichung dieses Ziels nötig bzw. nützlich ist. Es geht hier nicht darum, möglichst viel Theorie anzuwenden bzw. möglichst viele Anwendungsmöglichkeiten für die zur Verfügung stehende Theorie aufzuzeigen.

Die Anforderung auf eine für die praktische Anwendung geeignete “Verpackung” dieses Stoffs (siehe Abschnitt 1.1 oben) soll beachtet werden. Je nach Zielgruppe, ihrer Vorbildung u.ä. kann die optimale Darstellungsform unterschiedlich sein.

Vorbilder für die Gestaltung und Darstellung dieser theoretischen Grundlage für die Softwareentwicklung sind die wissenschaftlichen Grundlagen der klassischen Ingenieurfächer, z.B. die Newton’schen Gesetze (Maschinenbau), die Maxwell’schen Gleichungen und Kirchhoff’schen Gesetze (Elektrotechnik) usw.

Gegenstand der Betrachtung bilden sequentielle Programme. Spezielle Themen wie Komplexitäts- und Betriebsmittelfragen (z.B. bezüglich der Verfügbarkeit ausreichender Speicherkapazität), Sonderaspekte von Echtzeitanwendungen usw. werden hier nicht behandelt. Grundsätzlich wird auch der breite und umfangreiche Themenkomplex parallele Verarbeitung sowie nebenläufige Programme und Prozesse ausgeklammert; lediglich bestimmte ausgewählte Aspekte dieses Themas, die sich unmittelbar aus der Betrachtung sequentieller Prozesse ergeben und auf eine daraus abgeleitete, besonders einfache Form der Nebenläufigkeit beschränken, werden angesprochen.

1.3 Thesen

Die bedeutsamsten Thesen dieser Arbeit sind:

- Mit relativ wenig und verhältnismäßig einfacher Mathematik kann man hinsichtlich der Vermeidung von Softwarefehlern viel erreichen.
- Die Komplexität typischer Softwaresysteme kann nur durch eine systematische, auf Mathematik basierte, ingenieurmäßige Vorgehensweise beherrscht werden.
- **Eine mathematische theoretische Grundlage für die Softwareentwicklung, die den jeweiligen Grundlagen der klassischen Ingenieurwissenschaften entspricht und eine vergleichbare Bedeutung für die praktische Tätigkeit hat, läßt sich zusammenstellen.**
- Diese Grundlage unterstützt sowohl das Konstruieren als auch das Verifizieren von Programmen.
- Durch die konsequente Anwendung dieser Grundlage läßt sich der gleiche Grad an Entwurfsfehlerfreiheit erreichen, der für die klassischen Ingenieurfächer typisch ist.
- Es ist möglich, mit bescheidenem Aufwand und ohne maschinelle Unterstützung diese Grundlage beim Konstruieren fehlerfreier Software in der Praxis anzuwenden.
- Durch die gekonnte Anwendung dieser Grundlage erhöht sich die Softwareentwicklungsproduktivität merklich.
- Die systematische Anwendung dieses Stoffs bei der Korrektheitsbeweismführung gewährleistet die Freiheit von einer wichtigen Klasse von Softwarefehlern, vor allem logischer Art.
- Diese Grundlage ist nicht schwieriger (eher sogar leichter) zu erlernen und anzuwenden als die entsprechenden Basen der klassischen Ingenieurwissenschaften.
- Eine Voraussetzung für die Akzeptanz dieses Stoffs durch den Praktiker ist ein klarer und konkreter Bezug auf *seine* Problemwelt, z.B. durch praxisnahe und anwendungsrelevante Beispiele, den Gebrauch von Fachbegriffen aus seinem Anwendungsgebiet, die Behandlung von aus seiner Sicht wesentlichen Problemen statt Nebensächlichkeiten usw.
- Durch die Anwendung dieser Grundlage in der Praxis wird die Softwareentwicklung von einer Kunst in eine Ingenieurwissenschaft verwandelt. Eine entsprechende Veränderung der Denkweise und der Arbeitsweise der Praktiker wird gefordert und gefördert.
- Die konsequente Anwendung dieses Stoffs führt zu einer vertieften und gründlicheren Einsicht in das Wesen eines Programms, des Programmierens und der Programmgestaltung.
- In Kombination mit traditionellen Maßnahmen zur Softwarequalitätssicherung wird die Anwendung dieser Konzepte den Einsatz von Software in sicherheitskritischen Systemteilen ermöglichen. Software wird dadurch einen genauso hohen — eventuell noch höheren — Stellenwert für kritische Anwendungen erreichen wie die Hardware heute hat.

2. Bisherige Entwicklungen auf dem Gebiet der Programmkorrektheitsbeweissführung

2.1 Historische Übersicht

Die ersten drei Programme, die für den Rechner EDSAC (University of Cambridge, England) 1949 geschrieben wurden, liefen zum ersten Mal fehlerfrei. Beim vierten Programm machte Wilkes die unerwartete aber jetzt zur Norm gewordene Erfahrung, daß ein neu geschriebenes Programm typischerweise mehrere Fehler verschiedener Art enthält. Das damals von Wilkes erstellte Programm war 126 Zeilen lang und enthielt 20 Fehler. Es wurde bald deutlich, daß Programmierungsfehler ein schwieriges Problem waren und voraussichtlich lange bleiben würden [Hayes].

2.1.1 Der Ansatz von Floyd zur Programmkorrektheitsbeweissführung

Die Programmkorrektheitsbeweissführung, die Fachliteratur darüber sowie ihre verschiedenen Forschungsrichtungen und Nebenthemen haben ihren Ursprung im viel zitierten Artikel [Floyd; 1967]. Darin ordnete Floyd jedem Pfad des Flußdiagramms eines Programms eine Bedingung (Proposition, Zusicherung) zu, die immer erfüllt (wahr) sein soll, wenn die Programmausführung über den fraglichen Pfad fließt. Für jede Programmanweisungsort legte er eine Beziehung ("Verifikationsbedingung", engl. "verification condition") zwischen den vor und nach ihrer Ausführung gültigen Propositionen fest. Eine solche Beziehung stellte die semantische Definition der Programmanweisung dar. Verzweigungs- und Verbindungsstellen in einem Flußdiagramm wurden auch auf diese Weise behandelt. Beweist man alle Verifikationsbedingungen, so beweist man effektiv durch Induktion die Beziehung zwischen der Vor- und der Nachbedingung des gesamten Programms, d.h. man hat die semantische Bedeutung des Programms ermittelt bzw. nachgewiesen. Floyd arbeitete vom Anfang zum Ende des Programms (des Flußdiagramms) hin und betrachtete für jede Anweisung die stärkste verifizierbare nach ihrer Ausführung gültige Proposition (engl. "strongest verifiable consequent").

Die Grundzüge der Vorgehensweise von Floyd bilden noch heute die Basis für die Programmkorrektheitsbeweissführung, wenn auch viele wichtige Ergänzungen und Vervollständigungen sowie wesentliche Veränderungen der Anschauung inzwischen zu verzeichnen sind. Z.B. arbeitet man heute statt vom Anfang zum Ende des Programms eher umgekehrt vom Ende zum Anfang oder, oft treffender beschrieben, von Außen nach Innen. Entsprechend beschäftigt sich die heutige Literatur mehr mit schwächsten Vorbedingungen als mit stärksten Nachbedingungen. Floyd betrachtete das Problem der Interpretation bzw. *Analyse* eines gegebenen Programms, während man inzwischen diese Ideen und Konzepte bereits bei der *Synthese* eines beweisbar korrekten Programms (vgl. Kapitel 4) einsetzt — eine von [Dijkstra; *BIT*, 1968] vorgeschlagene Vorgehensweise.

2.1.2 Die zunehmende Problematik der Softwareentwicklung

In der Softwareentwicklungspraxis wurden in den 1960er Jahren immer größere Systeme vor allem für kommerzielle, kaufmännische und verwaltungsorientierte Anwendungen entwickelt. Dabei gab es nicht nur beeindruckende Erfolge sondern auch sehr beunruhigende Mißerfolge, darunter auch finanzielle Katastrophen, worüber [Baber; 1982, 1986] und andere berichteten. Solche Mißerfolge führten zu Diskussionen über die "Software-Krise" und zur 1968 vom NATO Scientific Committee in Garmisch-Partenkirchen veranstalteten und inzwischen berühmt gewordenen Konferenz. Diese Konferenz führte in ihrem Titel den Begriff "Software Engineering" ein, um auf provokatorische Weise auf die Notwendigkeit hinzuweisen, die Softwareentwicklung auf einer für die etablierten Ingenieurwissenschaften typischen Kombination von theoretischen Grundlagen und praktischen Disziplinen zu basieren [Naur; 1969, S. 13].

2.1.3 Die Korrektheitsansätze von Hoare, Dijkstra und Mills

In einem klassischen Artikel stellte [Hoare; 1969] Axiome für Programmanweisungen bzw. Zusammensetzungen davon auf. Gegenstand dieser Axiome sind Beziehungen zwischen Vor- und Nachbedingungen (Zusicherungen, die vor bzw. nach Ausführung der fraglichen Anweisung erfüllt sind). Dabei wird eine Programmanweisung bzw. Zusammensetzung davon als eine Transformation zwischen Prädikaten betrachtet. Gedankliches Vorbild für diese Betrachtungsweise waren klassische Axiomsysteme für die Geometrie sowie andere Teilgebiete der Mathematik. Die Substanz des Artikels stammt im wesentlichen von Floyd (siehe oben), jedoch bezog sich Hoare auf den Programmtext statt auf das Flußdiagramm und vermittelte eine ziemlich andere Anschauung des Stoffs. Der in [Foley] veröffentlichte Beweis für den Algorithmus "Quicksort" ist ein Anwendungsbeispiel der in [Hoare; 1969] enthaltenen Basis für die Korrektheitsbeweissführung. Eine andere Anwendung des Inhalts von [Hoare; 1969] ist die Definition der semantischen Bedeutung der Anweisungen einer Programmiersprache, z.B. PASCAL [Hoare; 1973].

Es folgte eine Reihe verwandter Weiterentwicklungen, die zu Empfehlungen hinsichtlich Programmierstils (z.B. strukturierten Programmierung, modulare Programmierung usw.) führten, die z.T. in die Praxis übernommen wurden. Siehe z.B. [Dijkstra; *CACM*, 1968], [Hoare; 1972], [Dahl; 1972]. Das Beispiel des strukturierten Programmierens ist interessant und für die bis heute herrschende Einstellung typischer Softwareentwickler kennzeichnend: Strukturiertes Programmieren wurde von Theoretikern empfohlen, weil Korrektheitsbeweise dadurch vereinfacht werden; in die Praxis wurde jedoch nur das strukturierte Programmieren — ohne Korrektheitsbeweise — aufgenommen.

Dijkstra führte diese Forschungsrichtung weiter [Dijkstra; 1975, 1976]. Vor allem betonte er die schwächste Vorbedingung einer Anweisung und gab Regeln für ihre Ableitung an. Er beschränkte sich nicht auf deterministische Programme, sondern definierte die Zusammensetzungen von Programmanweisungen (if und do) auf nicht deterministische Weise. Die dadurch erreichte Verallgemeinerung führt in bestimmten Fällen zu einer Symmetrie und Vereinfachung. Er legte besonderen Wert auf eine einfache, klare und für die gegebene Aufgabe geeignete Schreibweise, die insbesondere in den Schriften derjenigen, die mit ihm gearbeitet oder unter ihm studiert haben, wiederzuerkennen ist, z.B. in [Cohen], [Feijen], [Kaldewaij]. Er vertrat auch die Ansicht, daß Korrektheitsbeweise durch

das Umformen *uninterpretierter* Ausdrücke erfolgen soll [Dijkstra; EWD 1041-7, EWD 1130-4].

In den U.S.A. erarbeitete Mills eine in manchen Aspekten andere Ansicht zum Thema Programmkorrektheit [Mills; 1975-1986, 1989], [Linger]. Seine Vorgehensweise basierte auf dem Flußdiagramm des Programms (wie die von Floyd). Nur bestimmte strukturelle Zusammensetzungen von Anweisungen wurden erlaubt. Vielleicht das hauptsächlichste Unterscheidungsmerkmal zwischen dem Ansatz von Mills einerseits und Hoare und Dijkstra andererseits war, daß Mills jede Anweisung bzw. Zusammensetzung von Anweisungen als Funktion betrachtete, die einen Programmausführungszustand in einen anderen abbildet. Analysen und Programmkorrektheitsbeweise nach der Methode von Mills sind oft durch relativ detaillierte Betrachtungen der Programmausführungsgeschichten (engl. "traces") und der Werte der fraglichen Variablen in den verschiedenen Ausführungszuständen gekennzeichnet. Diese Methode wurde mit anderen Qualitätssicherungsmaßnahmen, insbesondere statistischem Testen (wobei die Testfälle nach der im Betrieb vorkommenden Wahrscheinlichkeitsverteilung gewählt werden), kombiniert und unter dem Namen "Cleanroom" bei der IBM Federal Systems Division für mehrere größere Softwareentwicklungsprojekte eingesetzt. Über erhebliche Verbesserungen hinsichtlich Qualität ist berichtet worden [Mills; 1987-1991], [Cobb], [Currit], [Glass], [Poore], [Selby].

2.1.4 Die Entwicklung von Spezifikationsprachen

In den frühen 1960er Jahren begann im Wiener IBM-Laboratorium die Entwicklung einer Methodik, die jetzt VDM (Vienna Development Method) genannt wird [Lucas; 1987 und persönliche Kommunikation], [Rolland]. Die ursprüngliche Anwendung war die denotationale Beschreibung von Programmiersprachen [Jones; 1986, S. 289], z.B. PL/I, wodurch die Übersetzerentwicklung und Standardisierung gefördert werden sollten. Die denotationale Semantik basiert wiederum auf der bekannten Arbeit von Strachey und Scott [Sommerville; 1985, S. 42]. VDM unterscheidet sich von den oben aufgeführten Ansätzen und Vorgehensweisen insbesondere dadurch, daß eine spezifische formale Sprache für die Formulierung von Spezifikationen von Programmen definiert wurde. Diese Sprache basiert zwar auf in der Mathematik bekannten Konzepten und Notationsformen, enthält jedoch eine Vielzahl von Ergänzungen, eigenen Symbolen und Konstrukten. VDM beschäftigt sich überwiegend mit der schrittweisen Verfeinerung einer vergleichsweise abstrakten Spezifikation ggf. über mehrere jeweils konkreteren Spezifikationsstufen bis hin zum endgültigen Programm. Jeder Verfeinerungsschritt führt zu Beweisverpflichtungen (engl. "proof obligations"), die die Übereinstimmung der zwei Spezifikationen (vor und nach dem fraglichen Verfeinerungsschritt) zum Gegenstand haben. Die vollständige formale Anwendung von VDM kann zu einem hohen Zeitaufwand für die Aufstellung der Beweise führen [Henhapl; S. 210], [Fields; S. 290]. Es wird deshalb manchmal vorgeschlagen, die Spezifikationen formal niederzuschreiben, aber die Beweise (mindestens zum Teil) nur durch informelle (aber formalisierbare) Argumentation ("rigorose Argumentation") zu erledigen. Es wird oft behauptet, der größte Wert dieser und anderer formalen Methoden sei in der Praxis bereits durch die Formulierung einer präzisen Spezifikation gewonnen, da die Spezifikation die zu erledigende Aufgabe klar und eindeutig definiert. Allein dadurch würden viele Implementationsfehler vermieden.

Die Arbeit der Programming Research Group an der Oxford University in England führte zur formalen Spezifikationsprache Z. Gegenstand von Z ist nur die Spezifikation eines Programms; Eigenschaften davon, die aus der Spezifikation logisch folgen, werden auch in die Methode einbezogen. Z sieht die Strukturierung von Spezifikationen mit Hilfe des sogenannten "Schemas" vor. Die Sprache ist weiterentwickelt worden und eine gewisse Standardisierung hat stattgefunden [Potter], [Spivey]. Andere Gruppen haben Z als Basis für eigene Entwicklungen verwendet, z.B. für Anna, eine Zwischenstufe zwischen einer in Z festgehaltenen Spezifikation und einem in Ada geschriebenen Programm [Sankar], [Wood]. Über Qualitätserhöhungen und Produktivitätssteigerungen, die auf den Einsatz von Z zurückzuführen sind, ist berichtet worden, z.B. bis 40% weniger Fehler und 9% Kosteneinsparung bei der Anwendung von Z auf CICS beim IBM Hursley Laboratory in England [Potter; S. 277-278], [Glass; 1992], [—; "An unlikely couple", 1992 Nov. 26] sowie bei der Ergänzung des Transputers um Gleitkommaarithmetik [Potter; S. 278-280]. Siehe auch [Norris; S. 96] für eine kurze Übersicht über einige formale Methoden und Anwendungen davon.

Mehrere andere Spezifikationsprachen sowie dazu gehörende maschinelle Unterstützung sind entwickelt worden, darunter RAISE ("Rigorous Approach to Industrial Software Engineering", Lloyd's Register, London), LOTOS ("Language of Temporal Ordering Specification") [Wilbur], ESTELLE [Wilbur], LARCH [Gaudel], [Wing], OBJ/OBJ3 (SRI International) [Gaudel], [Goguen; 1988]. Jede dieser Sprachen ist für einen bestimmten Anwendungsbereich konzipiert, z.B. nebenläufige kommunizierende verteilte Systeme (LOTOS, ESTELLE), die algebraische Spezifikation von Datentypen und Operationen darauf (LARCH). OBJ/OBJ3 implementiert ausführbare Spezifikationen und basiert auf einer funktionalen Programmiersprache. RAISE, das kommerziell vertrieben wird, zielt auf die Softwareentwicklung in einer industriellen Umgebung, u.a. von sicherheitskritischer Software.

Eine vom National Physical Laboratory (NPL, U.K.) 1992 durchgeführte Umfrage [Austin] ergab, daß VDM und Z die Anwendungsrankliste formaler Methoden führen: 55% der industriellen Befragten hatten VDM eingesetzt oder ihren Einsatz in Erwägung gezogen und genauso viele, Z. Die Gruppe LOTOS, CSP (Communicating Sequential Processes [Hoare; 1985]) und CCS (Calculus of Communicating Systems [Milner]) folgte mit 18%. Formale Methoden wurden überwiegend für das Spezifizieren eingesetzt bzw. dafür in Erwägung gezogen (89%). Deutlich weniger häufig wurden die anderen Einsatzgebiete genannt: Beweise 39%, Verfeinerung 17%, Entwurf 10%. Bei der Interpretation dieser Ergebnisse muß berücksichtigt werden, daß sie nicht als statistisch repräsentativ betrachtet werden kann. Die Befragung wurde zwar grundsätzlich international ausgerichtet, umfaßte jedoch tatsächlich überwiegend britische Organisationen mit einem besonderen Interesse an fehlerarmer Software, z.B. für sicherheitskritische Anwendungen. Die Ergebnisse deuten auf den relativen Bekanntheitsgrad bzw. auf die relative Beliebtheit verschiedener formaler Methoden in Großbritannien hin. Eine andere, weniger quantitativ orientierte Untersuchung [Craig; 1993 March], die sich mit Anwendungen formaler Methoden vor allem in den Vereinigten Staaten, Kanada und Großbritannien befaßte, kam zu etwa ähnlichen Ergebnissen.

2.1.5 Grundsätzliche Fragen zur Programmkorrektheitsbeweissführung

Die verschiedenen Programmkorrektheitsbeweissführungsmethoden waren Gegenstand beweistheoretischer Untersuchungen, z.B. [Wand; 1978], [Apt; 1981], [Loeckx]. Dieses Forschungsgebiet liegt außerhalb des Rahmens der vorliegenden Arbeit; nur einige wenige Ergebnisse sollen hier festgehalten werden. Das Hoaresche System z.B. ist wohlbegründet aber wegen der Wahl der Prädikatenlogik erster Ordnung als Spezifikationsprache hinsichtlich Terminierung unvollständig. Bezüglich partieller Korrektheit besteht eine derartige Problematik nicht.

Gegen die Idee an und für sich, man könne in der Praxis die Korrektheit eines Programms beweisen, haben De Millo, Lipton und Perlis 1979 grundsätzlich Stellung genommen [De Millo]. Jahre später (1987-1988) wurde diese ablehnende Meinung verstärkt von [Fetzer] und [Weiser] ("Program proving will never be practical.") vertreten. Stark vereinfacht beschriebenen wesentlichen Teile dieser Argumente auf dem Konsensbildungsmodell der Mathematik. Nüchtern betrachtet treffen ihre Argumente genauso gut auf die längst etablierten Ingenieurwissenschaften zu, wo mathematische Modelle eine grundlegende Rolle spielen (z.B. für statische Berechnungen im Bauingenieurwesen) und zur gewohnten Zuverlässigkeit der Entwürfe führen. Zumindest viele Ingenieure würden die Existenz und den Erfolg ihrer Fachgebiete als Gegenbeispiele dieser Argumentation betrachten. [Dijkstra; EWD 1041-9] hat diese Ansicht von De Millo, Lipton und Perlis entkräftet.

2.1.6 Anwendungsorientierung der Programmkorrektheitsfachliteratur

Verschiedene Verfasser haben sich bemüht, die mathematische Basis für die Programmkorrektheitsbeweissführung in einer anwendungsorientierteren Form vorzustellen und sie einer breiteren Anwenderschicht zugänglich zu machen. Solche Bücher und Artikel fallen in zwei Kategorien. In den meisten Exemplaren der ersten und größeren Gruppe ist die Mathematik (nicht der Anwendungsbedarf) als Herkunft und Motivation der Abhandlungen deutlich erkennbar; viele der darin enthaltenen Beispiele (z.B. das häufig vorkommende Beispiel des größten gemeinsamen Teilers) entstammen der Mathematik, nicht der typischen Softwareentwicklungspraxis. Typische Beispiele der ersten Gruppe sind [Dijkstra; 1976, 1985], [Alagić], [Anderson, Robert B.], [Linger], [Jones], [Gries; 1981], [Berlioux], [Backhouse], [Hoare; 1987 Aug.], [Futschek], [Dromey; 1989], [Cohen], [Kalde-waij], [Morgan] und [Hohlfeld]. Etwas weniger typisch für diese Gruppe sind [Baber; 1987, 1990, 1991], worin die anwendungsorientierte Motivation stärker hervorgehoben ist; z.B. stammen fast alle darin enthaltenen Beispiele aus geschäftlichen Softwareentwicklungsprojekten.

Die zweite Gruppe besteht aus überwiegend älteren Büchern, die eine deutliche, meist ausschließliche praktische Anwendungsorientierung aufwiesen, oft nur eine kärgliche geringfügige mathematische Basis in stark vereinfachter Form vorstellten und auch ganz andere Aspekte der Programmierung behandelten. Die zweite Gruppe war Softwareentwicklungspraktikern viel leichter zugänglich, reichte jedoch wegen der unvollständigen oder mathematisch oberflächlichen Behandlung für eine ernsthafte professionelle Anwendung nicht aus; sie enthalten nur einführende Übersichten über die Programmkorrektheits-

beweissführung. Für die zweite Gruppe typisch sind [McGowan], [Kimm] und [Martin, James].

Übersichten über die Entwicklung und den Stand der Wissenschaft der Programmkorrektheitsbeweissführung enthalten die Enzyklopädiebeiträge [Loomes], [Baber; 1994]. Eine grafische taxonomische Übersicht über Verifikationsmethoden enthält [Martin, James; S. 38].

2.1.7 Beweissführungswerkzeuge

Es wurde und wird noch an der Programmkorrektheitsbeweissführung oft beanstandet, daß die mathematischen Voraussetzungen umfangreich und schwierig sind und daß die praktische Anwendung sehr zeitaufwendig ist. Daraus entsteht das Verlangen nach "Werkzeugen" ("Tools"). In Forschungsumgebungen sind auch viele Systeme zur maschinellen Unterstützung der Programmkorrektheitsbeweissführung verwirklicht worden. Sie bestehen im wesentlichen aus zwei Komponenten: Ein Untersystem erzeugt die Lemmata, aus deren Wahrheit die Korrektheit des Programms folgt (engl. "verification condition generator"). Das zweite Untersystem beweist diese Lemmata bzw. unterstützt das Beweisen von ihnen (Beweiser). Zusätzliche Untersysteme dienen der Aufbereitung des Programms und der Spezifikation, der Verbindung mit anderen Programmiersprachenprozessoren, Hilfsmitteln usw.

Wahrscheinlich die bekanntesten früheren Systeme zur Unterstützung der Programmkorrektheitsbeweissführung sind das Verifikationssystem von King (1970), der Stanford PASCAL Verifier, GYPSY, AFFIRM und ein von Boyer und Moore entwickeltes System für FORTRAN-Programme. [Hohlfeld; 1988, 1992] beschreiben diese Systeme sowie PPPV von Marmier (ETH Zürich), Tatzelwurm (Universität Karlsruhe, siehe auch [Käuffl]), PASQUALE (AEG AG/Daimler Benz AG, Forschungsinstitut Ulm) und andere. PAMELA (Universität Kiel) [Buth] und Mural [Ayres], [Fields] sind neuere Systeme für die Unterstützung der Programmkorrektheitsbeweissführung. Manche dieser Systeme sollen in der Handhabung umständlich sein [persönliche Kommunikation: Cohen], siehe auch [Ayres; S. 659], [Buth], [Craig; 1987]. Die Abhandlung [Craig; 1991] über das Thema Verifikationsumgebungen enthält nähere Information über einige dieser Werkzeuge sowie eine Liste weiterer Systeme dieser und ähnlicher Art.

Der Satzbeweiser von Boyer und Moore sowie Varianten und Weiterentwicklungen davon werden als Basis für manche solche Systeme eingesetzt. [Bevier] berichtet z.B. über die maschinelle Korrektheitsbeweissführung für den Kern eines Betriebssystems, bei der ein Boyer-Moore-Satzbeweiser eingesetzt wurde.

SPADE von Program Validation Ltd. (Southampton, England) [Carré], [Ayres] und MALPAS von RTP Software Ltd. (Farnham, England) [Emonts], [Webb], [—; *Malpas Management Guide*, 1989 Sept.] sind anscheinend die einzigen Systeme zur Unterstützung der Programmkorrektheitsbeweissführung, die bisher auf dem kommerziellen Markt vertrieben wurden. Einige andere sollen für die Markteinführung vorbereitet werden [persönliche Kommunikation: Cohen]. MALPAS wurde in Deutschland angeboten, war jedoch offensichtlich kein Verkaufserfolg. Vermutlich ist der Markt dafür noch nicht reif [Ayres; S. 659-660]. Es ist nicht realistisch zu erwarten, daß ein System zur maschinellen Unterstützung der Programmkorrektheitsbeweissführung ohne gute Vorkenntnisse der relevanten

Theorie seitens der Benutzer erfolgreich eingesetzt werden kann, und Personal mit solchen Vorkenntnissen ist vergleichsweise selten zu finden.

Man muß davon ausgehen, daß nicht alle Systeme zur Unterstützung der Programmkorrektheitsbeweissführung öffentlich bekannt sind, denn ein frühes und wichtiges Anwendungsgebiet betraf (und vermutlich betrifft noch) sichere ("secure" im Gegensatz zu "safe") militärische Systeme, die der Geheimhaltung unterliegen. Deswegen sind auch einige öffentlich bekannte Systeme nur beschränkt verfügbar, z.B. GYPSY, AFFIRM und der Stanford PASCAL Verifier [Buth].

2.1.8 Operationale, denotationale und axiomatische Programmsemantik

Die verschiedenen Methoden zur Programmkorrektheitsbeweissführung basieren auf drei unterschiedlichen Ansätzen für die mathematische Definition der Semantik eines Programms [Schneider], [Sommerville; 1985, S. 41-42]. Beim *operationalen* Ansatz wird die Semantik eines Programms mittels Wirkungen einer abstrakten Maschine definiert. Dieser Ansatz ist relativ konkret und für manche Praktiker am verständlichsten, da er die Wirkungsweise wirklicher Maschinen, vor allem die schrittweise Ausführung eines Programms, widerspiegelt. Er verlagert die eigentliche Problematik jedoch nur auf eine neue Ebene, die der Definition der abstrakten Maschine. Bei der *denotationalen* Semantik werden Programmanweisungen und Programme als Funktionen auf einer abstrakten Wertmenge definiert. Das Konzept der schrittweisen Ausführung eines Programms entsteht dabei formal nicht; ein Programm wird als eine zusammengesetzte Funktion betrachtet. Bei der *axiomatischen* Semantik, die die Hoaresche Methode (siehe oben) kennzeichnet, wird jede Programmanweisung als eine Transformation zwischen Prädikaten betrachtet und durch ein entsprechendes Axiom definiert. Weitere Axiome definieren Zusammensetzungen (z.B. if-Konstrukte, Schleifen und die Folge) von Programmanweisungen, und dadurch auch gesamte Programme, durch Bezug auf die Prädikaten-Transformationen ihrer Bestandteile.

Formal sind die Unterschiede zwischen diesen drei Ansätzen klar gezogen, aber konzeptionell sind sie eng miteinander verwandt. Z.B. entsprechen die Schritte des Zusammensetzens der Funktionen (bei der denotationalen Semantik) bzw. der Prädikaten (bei der axiomatischen Semantik) der schrittweisen Ausführung eines Programms durch eine Maschine (bei der operationalen Semantik). Die Prädikaten-Transformationen, die bei der axiomatischen Semantik Axiome sind, können aus den Definitionen der denotationalen bzw. operationalen Semantik abgeleitet werden, d.h. als Sätze formuliert und bewiesen werden. Für die praktische Anwendung ist die Unterscheidung zwischen diesen drei Konzepten weniger wichtig als für die theoretische Forschung. Die in [Baber; 1987, 1990 (*Fehlerfreie Programmierung*), 1991 (*Error-free Software*)] enthaltene Betrachtung sowie vorgeschlagene Vorgehensweise zeigen z.B. deutlich den Einfluß aller drei Semantik-Ansätze.

2.1.9 Alternativen zur Softwarequalitätssicherung

Verschiedene Alternativen zur Programmkorrektheitsbeweissführung und Softwarequalitätssicherung im allgemeinen sind im Laufe der Jahre verfolgt worden. Testen, Fehlertolerierung und organisatorische Maßnahmen im weiteren Sinne (oft "Software-Engineering"

genannt) wurden im Abschnitt 1 bereits erwähnt. Über jeden dieser Bereiche existiert eine umfangreiche Literatur, z.T. mit einer theoretischen Basis, z.T. basierend auf gesammelter Erfahrung. Solche Alternativen und die Programmkorrektheitsbeweissführung werden gewöhnlich als konkurrierende Maßnahmen angesehen; sie sind jedoch besser als komplementäre, sich gegenseitig unterstützende Wege zur Softwarequalität zu sehen. Formale Verifikation kann z.B. Fehler aufdecken, die durch Testen nicht entdeckt würden, aber umgekehrt können Tests Fehler aufdecken, die mit formaler Verifikation nicht zu finden wären, z.B. Mißverständnisse über Schnittstellen sowie nicht erfüllte Annahmen und Axiome [Parnas; 1990 Dec.].

Eine andere Methode zur Verifikation der Korrektheit eines Programms ist die "symbolische Ausführung" [Dannenbergh], [Darringer], [Hantler], [Khanna], die besonders in den 1970er Jahren Forschungsgegenstand war. Bei dieser Methode der Programmanalyse werden die anfänglichen Werte der Programmvariablen als unabhängige Variablen betrachtet und die Werte der Programmvariablen nach den verschiedenen Ausführungsschritten, insbesondere am Ende der Ausführung, als Funktionen der anfänglichen Werte ermittelt. Dabei verfolgt man alle Ausführungspfade des Programms; anders betrachtet konstruiert man den Ausführungsbaum des Programms, der sehr groß — im Falle einer Schleife prinzipiell unendlich — werden kann. Durch induktive Analyse können auch Schleifen behandelt werden. Das Interesse an der symbolischen Ausführung scheint in den letzten Jahren geringer geworden zu sein, eventuell weil ihre Grundidee auf gewisse Weise von den oben bereits behandelten Methoden (insbesondere von der Methode von Mills) umfaßt und abgedeckt wird.

Statistisches Testen wurde in Zusammenhang mit "Cleanroom" (siehe oben) bereits erwähnt. Dieses Verfahren kann prinzipiell auch im nachhinein verwendet werden, um die Zuverlässigkeit eines Programms zu messen [Currit], [Littlewood; 1991], [Musa], [Poore], [VDI-GIS; Kap. 3 und Anhang C]. Streng genommen mißt bzw. schätzt man die Wahrscheinlichkeit der Eingabemenge, die das Programm fehlerhaft verarbeitet. In der Praxis ist diese Methode nur bei bescheidenen Zuverlässigkeitsansprüchen von Interesse, weil sonst die Anzahl der erforderlichen Testfälle unrealistisch groß sein kann [Butler], [Littlewood; 1991, 1993]. Soll z.B. für eine sicherheitskritische Anwendung eine Versagensrate (Versagensfälle pro Anforderung) von $<10^{-9}$ statistisch nachgewiesen werden, müssen $>10^9$ (je nach Vertrauensgrenze ggf. viel mehr) Testfälle ohne ein Versagen verarbeitet werden. Der Zeitaufwand und die Kosten dafür sind kaum zu rechtfertigen.

Ein gewisser Gegensatz in Zusammenhang mit statistischem Testen sollte hier erwähnt werden, da er zu Kontroverse in Fachkreisen über die Anwendbarkeit dieser Verfahren geführt hat. Auf dem Hardwaregebiet wird gewöhnlich die Entwurfsfehlerfreiheit angenommen [Littlewood; 1993]; Versagen und Ausfälle wegen Abnutzung, Verschleiß, Veralterung u.ä. werden empirisch ermittelt und statistisch ausgewertet. Im Falle von Software wird genau umgekehrt vorgegangen: Abnutzung, Verschleiß, Veralterung u.ä. können ausgeschlossen werden [Gardener]; nur durch Entwurfsfehler verursachte Versagensfälle werden empirisch erfaßt [—; PDCS, Vol. 3, Chapter 3, S. 2, 1991 May] (siehe auch Abschnitt 1.1). Dieser Unterschied bezüglich der Ausgangssituationen wirft Fragen über die Eignung vieler im Hardwarebereich bewährten statistischen Modelle für die Ermittlung der Softwarezuverlässigkeit auf. Solche Fragen werden an verschiedenen Stellen in der bereits zitierten Literatur diskutiert.

Die Überprüfung von Zusicherungen während der Ausführung eines Programms [Sankar] ist noch eine Methode zur Sicherstellung der Korrektheit eines Programms, die in der

Fachliteratur bereits vor langer Zeit vorgeschlagen wurde und in der Praxis gelegentlich eingesetzt wird. Streng genommen wird dadurch nicht die Korrektheit des Programms, sondern die Korrektheit *der jeweiligen Ausführung* des Programms überprüft. Einige Programmiersprachensysteme sehen einen solchen Ausführungsmodus vor, z.B. Eiffel (siehe Abschnitt 7.3.2). Der Effekt kann in jeder prozeduralen Programmiersprache durch das Einfügen entsprechender if-Anweisungen in das fragliche Programm erzielt werden. Meist ist die zulässige Form der Zusicherungen mehr oder weniger stark eingeschränkt, z.B. Quantoren, entsprechende Reihen u.ä. werden in manchen Implementierungen nicht unterstützt. Solche Einschränkungen begrenzen die Nützlichkeit dieser Methode.

Die Inspektion ist eine 1972 von Fagan eingeführte Methode zur (mathematisch informellen) Verifikation von Software. Über die praktischen Vorteile davon wird immer wieder (z.B. in Konferenzdiskussionen) berichtet, siehe auch [Fagan], aber die Methode wird trotzdem — offensichtlich aus psychologischen Gründen — von Beteiligten oft abgelehnt. Daß ein mathematischer Korrektheitsbeweis sowohl die Effizienz einer Inspektionsbesprechung erhöhen als auch die potentielle psychologische Drohung des Programmmentwicklers verringern könnte, scheint in der Literatur unerwogen geblieben zu sein. [Britcher; S. 39, 1988] erwähnte die Möglichkeit, Korrektheitsbeweise in den Inspektionsprozeß einzubeziehen, ging jedoch nicht näher darauf ein.

2.1.10 Grenzgebiete der Programmkorrektheitsbeweissführung

Konzepte der Programmkorrektheitsbeweissführung, insbesondere Vor- und Nachbedingungen sowie Schleifen- und andere Invarianten, sind in Forschungsprojekten mit anderen Zielsetzungen angewendet worden. Z.B. spielen diese Konzepte in den PBM's (Plan Building Methods) von [Waters] eine Rolle, die wiederum ein Bestandteil des "Programmer's Apprentice", eines Systems zur Unterstützung eines Softwareentwicklers, sind. Vor- und Nachbedingungen sind wesentliche Teile der Spezifikationen von Methoden (Programmsegmenten) im Vorschlag von [Jeng] zur Klassifizierung und Organisation wieder verwendbarer Softwarekomponenten in einer Programmbibliothek. Auch zwecks "debugging" wurde die Verwendung von Zusicherungen und Invarianten vorgeschlagen [Bourdoncle].

Das alte Konzept der gleichzeitigen Entwicklung eines Programms und seines Korrektheitsbeweises, wobei die Entwicklung des Korrektheitsbeweises den Prozeß führt [Dijkstra in Buxton; S. 85], wird in [Manna; 1992] formalisiert und ergänzt mit dem Ziel, ein Programm automatisch abzuleiten. Vgl. Kapitel 4, siehe auch [Abrial], [Baber; 1987, 1990, 1991], [Boiten], [Dromey], [Kaldewajj].

Die Korrektheitsbeweissführung für nebenläufige Prozesse ist ein aktives Forschungsgebiet, auch auf der Ebene der Grundlagentheorie. Obwohl dieses Thema außerhalb des Rahmens dieser Arbeit liegt (siehe Abschnitt 1.2), soll es hier als Grenzgebiet und Gegenstand weiterführender Forschung kurz erwähnt werden. Die zahlreichen Strukturen, die für die Koordination bzw. Kommunikation zwischen nebenläufigen Prozessen vorgeschlagen und in verschiedenen Systemen verwirklicht werden, haben zu entsprechend vielfältigen Forschungsrichtungen geführt, siehe z.B. [America], [Andrews], [Apt], [Broy; 1991, 1992], [de Boer], [Dijkstra; 1968], [Hoare; 1981, 1985], [Kearney], [Milner], [Olderog], [Owicki], [Peleska; 1991], [Pnueli], [Turksi; 1990], [Wieczorek]. Das Petrinetz [Reisig], [Herzog] ist ein bei der Analyse nebenläufiger Systeme häufig eingesetztes Hilfsmittel.

Aus den besonderen Aspekten der Korrektheitsbeweissführung, die berücksichtigt werden müssen, wenn ein Programm mit einem anderen gleichzeitig ausgeführt wird, kann man einige vereinfachende Leitlinien zur Gestaltung solcher nebenläufigen Prozesse ableiten. Darauf wird im Abschnitt 7.2 näher eingegangen.

2.1.11 Korrektheitsbeweissführung und objektorientierte Programmierung

Über verschiedene Aspekte der Korrektheitsbeweissführung und formaler Spezifikation bezogen auf die objektorientierte Programmierung sind in letzter Zeit einige Fachartikel erschienen, z.B. [America], [Chedgy], [Coleman], [Cusack], [de Boer], [Fischbach], [Grams; 1992], [Hogg], [Holt], [Minkowitz], [Schmitz], [Winkler]. Siehe auch Abschnitt 7.3, wo auf diese Beziehungen näher eingegangen wird. Andere Veröffentlichungen berühren gewisse Aspekte der Korrektheitsbeweissführung in Zusammenhang mit objektorientierten Themen, z.B. [Lieberherr], [Schäfer]. [Thomas; 1993, S. 34] berichtet, die Spezifikationsprache Z sei eine effektive Notationsform für Systeme, die mit objektorientierten Techniken implementiert werden sollen. Programmkorrektheitskonzepte haben die objektorientierte Programmiersprache Eiffel [Meyer] ersichtlich beeinflusst, obwohl diese Entwicklung noch nicht ausreichend weit gegangen ist (vgl. Abschnitt 7.3.4). Sowohl die Grundidee "programming by contract" als auch verschiedene Sprachkonstrukte in Eiffel, z.B. Klasseninvarianten, haben Wurzeln in der Programmkorrektheitsbeweissführung.

2.1.12 Programmkorrektheitsbeweise und Normen

Im Laufe der Jahre sind viele Normen für die Entwicklung und Qualitätsbeurteilung von Software bzw. von Systemen, worin Software ein maßgeblicher Bestandteil ist, verabschiedet worden, z.B. [BWB], [Europäische Gemeinschaften], [DKE], [IEC/SC 65A], [IEEE; 1987, 1989], [Ministry of Defence], [RTCA], [—; *TickIt Guide*, 1990 Sept. 30]. Der Normenreihe ISO 9000 und ihrer Anwendung auf Software wird in der Industrie zur Zeit viel Aufmerksamkeit geschenkt, siehe z.B. [Evans], [Frühau]. Mit Ausnahme des "Interim Defence Standard 00-55" [Ministry of Defence], der formale Methoden stark befürwortet, verlangen die oben zitierten Normen den Einsatz von formalen Methoden entweder gar nicht oder nur für die höchste Sicherheitsstufe, und dann oft nur teilweise, z.B. für die formale Spezifikation jedoch nicht für die Verifikation. Grundsätzlich neigen diese Normen wenig zu formalen Methoden, obwohl eine langsame Tendenz mindestens zur Anerkennung ihrer potentiellen Vorteile zu beobachten ist.

2.1.13 Gegenwärtige Forschungsprojekte

In Europa laufen seit einiger Zeit mehrere z.T. größere Forschungsprojekte, die Softwarezuverlässigkeit teilweise oder ganz zum Gegenstand haben und die mit EG- oder nationalen Forschungsmitteln finanziell unterstützt werden. Im Esprit-Programm z.B. befaßten sich die Projekte ProCos (Provably Correct Systems) und PDCS (Predictably Dependable Computing Systems) sowohl mit Hardware als auch mit Software. Schwerpunkt des Pro-

jekts DARTS (Demonstration of Advanced Reliability Techniques for Safety-Related Computer Systems) war die Softwarezuverlässigkeit. Zielsetzung von ProCos war es, zu Methoden für die zusammenhängende Entwicklung von Hard- und Software für eingebettete sicherheitskritische Echtzeitsysteme beizutragen [Esprit; S. 112-115, 1991 Jan.]. Das Projekt umfaßt Spezifikations- und Programmierungssprachen, einen Übersetzer und einen Kern zur Unterstützung der Ausführung übersetzter Programme. [Bjørner; 1993] ist ein Abschlußbericht des ProCos-Projekts. Zielsetzung von PDCS war es, eine entwurfsunterstützende Umgebung für die Konstruktion großer, zuverlässiger, fehlertoleranter und verteilter Echtzeitsysteme zu schaffen [Esprit; S. 104-107, 1991 Jan.]. Formale Methoden und Programmkorrektheitsbeweise spielen im PDCS eine untergeordnete Rolle [—; *Predictably Dependable Computing Systems*, Vol. 3, Chapter 3, 1991 May]; die Betonung liegt eher auf dem Messen der erreichten Zuverlässigkeit bzw. das Voraussagen derselben. Die Arbeit von PDCS wird jetzt im Nachfolgerprojekt PDCS2 weitergeführt. Zielsetzung des Projekts DARTS [Esprit; S. 66-67, 1990 Sept.] war es, zur Entwicklung von Methoden für die Beurteilung und Beschleunigung von sicherheitskritischen rechnerbasierten Systemen beizutragen. Kosten-Nutzen-Verhältnisse verschiedener solcher Methoden, darunter formaler Methoden [Ayres; S. 658-659], sollen ermittelt werden.

[Broy; 1993] gibt eine gute Übersicht über drei deutsche Projekte vergleichbarer Art. Ziel des Verbundprojekts KORSO (Korrekte Software) ist "die prototypische Anwendung von Werkzeugen und Methoden, die Umsetzung des Grundlagenwissens und die Anpassung der Werkzeuge und Techniken an die Bedürfnisse der Anwendungen." Das BSI-Projekt VSE (Verification Support Environment) verfolgt das Ziel, eine Methode und entsprechende Werkzeuge zur Sicherheitsüberprüfung von Software zu entwickeln. Das DFG-Projekt Deduktion "befaßt sich im Grundlagenbereich mit der Erforschung von Deduktionstechniken und Werkzeugen."

2.1.14 Korrektheitsbeweissführungstechniken im praktischen Einsatz

In meiner eigenen beruflichen Tätigkeit als Unternehmensberater bin ich in einigen Fällen, meist in Zusammenhang mit einem übergeordneten Projekt, beauftragt worden, ein Anwendungssoftwaresystem oder Teile davon zu programmieren. Bei diesen Projekten habe ich mit der Anwendung von Programmkorrektheitsbeweissführungstechniken sehr positive Erfahrung gemacht. Der Schwerpunkt dabei lag auf der Konstruktion des jeweiligen Programms, und zwar überwiegend auf informale Weise. Für einige kompliziertere Programmteile habe ich eine formale Analyse bzw. Konstruktion ausgeführt. Die besonderen Vorteile, die sich aus dieser Vorgehensweise ergaben, waren folgende: Nach der ursprünglichen jeweiligen Inbetriebnahme mußte der Klient mich nie wegen eines Fehlers in der von mir gelieferten Anwendungssoftware zur Hilfe rufen. In den später erstellten Systemen hat sich auch während der Testphase kein Fehler bemerkbar gemacht. Meine Kostenvorkalkulation wurde durch die weitgehende Vermeidung des typischerweise ungenau schätzbaren Test- und Fehlerkorrekturaufwands erheblich erleichtert und mein Kalkulationsrisiko entsprechend reduziert. Gegenüber einer traditionelleren Arbeitsweise wurde nicht nur die Zuverlässigkeit der Systeme sondern m.E. auch die gesamte Produktivität des jeweiligen Projekts deutlich gesteigert (wie von [Cobb], [Dijkstra; 1982, S. 347-348] und [Hall; Myth 5, 1990 Sept.] behauptet und von [Card; S. 848] beobachtet) und die Lieferfrist verringert. Meine Klienten haben betriebssichere Systeme bekommen, worauf

sie sich Jahre lang verlassen konnten (und haben). Der jeweilige Projektaufwand betrug (wegen der produktivitätssteigernden Effekte nur) ca. drei Mannwochen bis einige Mannmonate. Die Anwendungen waren Eingabeuntersysteme für ein Abrechnungs- und Finanzbuchhaltungssystem, Managementplanspiele für Führungskräfte-seminare, die Arbeitsplanung und -abwicklung, Berichterstattung und Leistungsabrechnung in einem medizinischen Labor sowie eine auf einem verbundenen Rechner verwirklichte Ergänzung zu einem Rechnersystem für die Kommunikationsnetzkonfigurierung.

In der industriellen und kommerziellen Praxis kann man eine bedeutende Anzahl von Anwendungen von formalen Methoden finden, siehe den Absatz über Z in Abschnitt 2.1.4 sowie [Downs; S. 201], [Kershaw], [Norris; S. 111], [Peleska; 1991 März], [Rushby], [Thomas; 1993], [—; "Formal education", 1993 May 6]. Der Anteil solcher Anwendungen an der gesamten Anzahl der Softwareentwicklungsprojekten ist jedoch verschwindend klein. Auf die Gründe dafür wird im Abschnitt 2.2.1 unten näher eingegangen.

2.1.15 Schlußbemerkungen

Softwarefehler waren und sind noch ein in der Praxis gravierendes Problem. Verschiedene Untersuchungen haben Softwarefehlerhäufigkeiten in der Größenordnung von 5 bis 30 Fehler pro tausend Zeilen Programmcode [Brown], [Joyce; 1989] festgestellt. Durch die konsequente Anwendung fortgeschrittener Methoden konnten deutlich niedrigere Fehleraten — bis auf ca. 0,1 Fehler pro tausend Zeilen Programmcode — erreicht werden [Cobb], [Joyce; 1989]. Aber selbst solch niedrige Fehlerraten stehen sicherheitskritischen Anwendungen im Wege. Potentiell besonders problematisch für solche Anwendungen sind die "5000-Jahre-Bugs" [Currit], [Littlewood; 1991]: Eine von Adams durchgeführte Untersuchung ergab, daß Programmfehler, die jeweils nur sehr selten zum Versagen führen (MTTF > 5000 Jahre), zusammen einen hohen Anteil (>30%) aller Programmfehler ausmachen. Das bedeutet wiederum, daß sie auf empirische Weise schwer auffindbar sind. Kein "5000-Jahre-Fehler" darf in einem Programm vorhanden sein, dessen Zuverlässigkeit ein Versagen pro 10^9 Std. oder besser sein soll (eine oft zitierte Zahl für sicherheitskritische Anwendungen), denn $5000 \text{ Jahre} = 4,38 \times 10^7 \text{ Std.}$

Die Softwarezuverlässigkeit stellt den Engpaß bei der Entwicklung moderner Systeme dar. Z.B. wurden im nicht veröffentlichten Diskussionsbeitrag von [Laprie] detaillierte Zahlen über MTBF für ein Flugzeug vorgetragen, woraus hervorging, daß Versagensfälle viel häufiger durch Softwarefehler als durch alle (nicht nur Computer-) Hardwareausfälle zusammen verursacht wurden. Eine vergleichbare Aussage enthält [Gruman; 1991] über das Patriot-Abwehraketensystem. Im Falle der Bestrahlungsanlage Therac-25 [Joyce; 1987], [—; "Lethal dose", 1987 Dec.] waren Softwarefehler die unmittelbare Ursache für die "worst series of radiation accidents in the 35-year history of medical accelerators" [Leveson; 1993], woraus der Schluß gezogen wurde, "A common mistake in engineering ... is to put too much confidence in software".

Wie im Abschnitt 1.1 bereits erwähnt ist von mehreren Seiten mittelbar oder unmittelbar angeregt worden, die Softwareentwicklung als Ingenieurwissenschaft zu etablieren, d.h. u.a. sie und ihre Praxis auf eine wissenschaftliche und mathematische Grundlage zu stellen. Demgegenüber steht, daß sich die überwiegende Mehrheit der Informatiker offensichtlich nicht als Ingenieure versteht: In einer Mitgliederbefragung der Gesellschaft für

Informatik 1991/92 z.B. gaben nur 10,2% der Befragten "Ingenieur" als Berufsbezeichnung bzw. 6,4% als Funktionsbezeichnung an [Fachauschuß 7.4 der GI].

Während des gesamten Computeralters herrscht die Problematik der Softwarefehler. Trotz erheblichen Bemühungen verschiedener Art sowie gewissen Verbesserungen ist in der Praxis weder das Problem beseitigt noch die Ursache behoben worden. Fehlerhafte Software ist also eine "etablierte Tradition" in der Informatik und in der Computerindustrie.

2.2 Gegenwärtige Situation

2.2.1 Der Gegensatz zwischen Theorie und Anwendungspraxis

Die Theorie der Programmkorrektheitsbeweissführung ist bereits weitgehend entwickelt worden (zumindest hinsichtlich sequentieller Programme). Von ihr wird in der Praxis jedoch enttäuschend selten Gebrauch gemacht [Austin], [Bradley], [Martin, Alain J.; 1992], [Weigele; 1991]. Diese Lücke zwischen Theorie und Praxis ist unter den technologischen Fächern atypisch groß. Vermutlich liegt ein großes, wirtschaftlich interessantes Nutzungspotential brach. Jetzt obliegt es der anwendungsorientierten softwareingenieurwissenschaftlichen Forschung und Lehre, durch entsprechende Gestaltung dieses Stoffs den zum praktischen Einsatz dieser theoretischen Grundlage erforderlichen Lern- und Anwendungsaufwand zu verringern sowie die aus ihrer Anwendung sich ergebenden Vorteile möglichst zu vergrößern.

Einige Erfolge sind bei der praktischen Anwendung von formalen Methoden bzw. Korrektheitsbeweistechniken zu verzeichnen (siehe Abschnitt 2.1). Hinweise auf einen langsamen Trend zu ihrer zunehmenden Anwendung können beobachtet werden. Insbesondere die hohen Anforderungen sicherheitskritischer Anwendungen werden voraussichtlich zu wachsendem Verlangen nach ihrem Einsatz beitragen.

Die Forschungsbestrebungen gehen überwiegend in die Richtung spezieller Methoden, wobei jede einen eigenen und mehr oder weniger umfangreichen Formalismus mit dazu gehörender Sprache aufweist. Anscheinend wird davon ausgegangen, größer sei besser. [Weigele; 1990] hat eine andere Erklärung: "kompliziert verkauft sich (wissenschaftlich) besser" (siehe auch [Hamming]). Angesichts der breiten — und oft beharrlichen — Ablehnung der bereits entwickelten formalen Methoden durch etablierte Softwarepraktiker sowie ihrer Klagen, diese seien mathematisch zu schwer, ihre praktische Anwendung sei zu zeitaufwendig und ihr Einsatz sei nur mit weitgehend automatisierter Unterstützung wirtschaftlich zu vertreten, stellt sich die Frage, ob die gegenwärtig eingeschlagene Forschungs- und Entwicklungsrichtung geeignet ist, eine breite Annahmefähigkeit für mathematisch fundierte Vorgehensweisen bei der Softwareentwicklung zu erreichen. Als Alternative müßte auch der umgekehrte Weg — in Vereinfachung — in Erwägung gezogen werden.

Softwareentwickler sowie Softwareunternehmen befinden sich in einer Interessenslage, die sie eigentlich entmutigt, formale Methoden bzw. Programmkorrektheitsbeweissführungstechniken sich anzueignen. Sie hätten nämlich überwiegend die Nachteile davon zu tragen (Investition in Training, danach geringere Auslastung wegen verringerten Test- und Fehlerkorrekturaufwands). Die Vorteile (bessere und ggf. preiswertere Software) würden

eher die Kunden genießen. Die Aussicht auf Anschlußaufträge, die für Softwarelieferanten geschäftlich und finanziell durchaus interessant sind [persönliche Kommunikation eines Klienten], [Dijkstra; EWD 1041-6, EWD 1130-3], würde durch die erfolgreiche Anwendung mathematisch rigoroser Softwareentwicklungstechniken eher ab- als zunehmen. Solange Softwarelieferanten und -entwickler die Folgekosten und Verluste, die ihre Softwarefehler verursachen, nicht tragen müssen, werden sie wenig Interesse am Einsatz formaler Methoden haben, es sei denn, der Kunde verlangt diesen im Vertrag.

Vielfältige Gründe werden dafür angegeben, daß formale Methoden bzw. Programmkorrektheitsbeweissführungstechniken nicht bzw. nur relativ selten in der Praxis eingesetzt werden. Manche aufgeführten Gründe sind sicherlich begründet, andere eventuell nicht. Es geht bei der Behandlung dieser Einwände jedoch weniger darum, ob sie rational und gerechtfertigt sind oder nicht; sie müssen auf jeden Fall zur Kenntnis genommen werden. Rationale Argumente reichen oft nicht aus, um psychologisch begründete Verhaltensweisen zu verändern.

Zusammenfassend deuten die Einwände auf eine Kluft zwischen den (wirklichen oder vermeintlichen) Voraussetzungen für die Anwendung und den in der Praxis vorhandenen Gegebenheiten sowie auf eine Kluft zwischen den Problemen einerseits, die mit diesen Methoden gelöst werden können bzw. sollen, und den Problemen andererseits, die in der Praxis zu lösen sind, hin. Ferner deutet einiges darauf hin, daß der Gegensatz zwischen Theorie und Anwendungspraxis mindestens teilweise auf eine Kommunikationslücke zwischen den Welten der Theorie und der Praxis zurückzuführen ist [Strachey in Hoare; S. 370, 1989]. Die Praktiker erkennen oft nicht, was die Theoretiker ihnen anbieten, und die Theoretiker erkennen oft nicht, was die Praktiker brauchen bzw. in welcher Form die Praktiker es nützen können.

Die oft angegebenen Gründe und Einwände gegen den praktischen Einsatz formaler Methoden bzw. Programmkorrektheitsbeweissführungstechniken sind im einzelnen:

- Die erforderliche Mathematik bzw. der geforderte Formalismus ist schwierig und umfangreich [Barroca], [Basili; Diskussionsbeitrag], [Goldson], [Juliff], [Musgrave], [Sommerville], [Thomas; 1993], [Wendt], [—; "Formal education", 1993 May 6], [persönliche Kommunikation mehrerer Gesprächspartner]. Die diskrete Mathematik und ihre Notation sind für viele ungewohnt, schwierig und unverständlich, sogar abstoßend [Barroca], [Bloomfield; 1986 Sept.], [Culik]. Z.B. sind die häufig vorkommenden \forall - und \exists -Konstruktionen für viele schwer verständlich und problematisch; selbst Logiker haben damit gelegentlich Schwierigkeiten [Dijkstra; EWD 1084].
- Die Anwendung formaler Methoden setzt hochqualifiziertes Personal und entsprechende Aus- bzw. Weiterbildung voraus. Der Lernaufwand und damit die anfängliche Investition sind hoch. [Austin], [Ayres], [Barroca], [Bloomfield; 1986 Sept.], [Bradley], [Henhapl], [Macro], [Norris], [Sommerville], [Thomas; 1993]
- Geeignete maschinelle Unterstützung ist erforderlich, aber nicht verfügbar. Eine geeignete integrierte Programmierungsumgebung fehlt. [Austin], [Ayres], [Berlioux], [Bloomfield; 1986 Sept.], [Bradley], [Loomes], [Nakajima], [Norris], [Wood]
- Die Anwendung formaler Methoden ist sehr schwierig oder sehr aufwendig [Austin], [Emonts], [Macro], [Nakajima], [Wood].
- Formale Verifikation (die Erstellung von Korrektheitsbeweisen) ist aufwendig und teuer; Beweise sind sehr lang [Berlioux], [Downs], [Hoare in Buxton; S. 21], [Fields], [Henhapl], [Loomes], [Norris], [Sommerville], [Thomas; 1993], [Wallmüller], [Wing; Gong].

- Softwareentwicklungskosten werden durch den Einsatz formaler Methoden erhöht [Austin], [Ayres], [Thomas; 1993].
- Es ist oft schwierig, die benötigten Zusicherungen (Zwischenbedingungen, Schleifeninvarianten usw.) zu finden [Wallmüller].
- Die Einführung formaler Methoden benachteiligt den erfahrenen und fähigen Softwareentwickler, weil seine bisherigen Wettbewerbsvorteile dadurch verlorengehen bzw. entwertet werden [Thomas; 1993].
- Die Einführung und Integration formaler Methoden in die gegebene Softwareentwicklungspraxis bereitet Schwierigkeiten. Der Schritt von der gegenwärtigen Praxis zu formalen Methoden ist zu groß; Zwischenstufen auf dem Wege dahin fehlen. [Anderson, Stuart], [Bradley], [Henhapl]
- Keine einzige formale Methode ist ausreichend umfassend [Barroca], [Bloomfield; 1986 Sept.]. Die Einschränkung auf eine einzige formale Sprache mit einer ein für allemal definierten Semantik und Syntax wäre konterproduktiv; kein Mathematiker würde eine derartige Beschränkung akzeptieren [Petrone].
- Aus der teilweisen Anwendung formaler Methoden ist kein oder kaum ein Vorteil zu ziehen; man muß bei der Anwendung alles oder nichts einsetzen [Thomas; 1993].
- Ein günstiges Kosten-Nutzen-Verhältnis ist nicht objektiv und glaubhaft nachgewiesen worden [Austin], [Bloomfield; 1986 Sept.], [Bradley], [Littlewood; 1993], [Martin, Alain J.; 1992].
- Einige Anforderungen lassen sich in formalen Spezifikationen nur schwer oder nicht erfassen [Austin], [Barroca], [Loomes], [Wendt].
- Für Klienten und Anwendungsgebietexperten sind formale Spezifikationen schwer verständlich, zu wenig intuitiv oder mehrdeutig interpretierbar [Austin], [Barroca], [Europäische Gemeinschaften; S. 30], [France], [Macro].
- Formalismus nützt wenig; mit Intuition kommt man viel weiter [Gruman; 1989 July].
- Formale Spezifikationen können fehlerhaft sein [Austin], [Barroca], [Downs], [Wallmüller].
- Die mathematische Beweisführung (sowohl manuell als auch maschinell geführt) ist fehleranfällig [Downs], [Macro], [Wallmüller], [Wing; Gong], [—; PDCS, Chapter 3, Section I-4.3, 1991 May].
- Übersetzer, Betriebssysteme, andere Systemsoftware, Verifikationswerkzeuge (z.B. Satzbeweiser) und sogar die Hardware können fehlerhaft sein; korrekter Quellcode reicht nicht aus [Barroca], [Downs].
- Klienten verlangen den Einsatz formaler Methoden nicht [Austin].
- Formale Methoden unterstützen die Beschreibung eines Systems, aber nicht seine Konstruktion [Austin], [Macro].
- Formale Methoden sind nur auf einfache, kleine oder künstliche Probleme anwendbar [Ayres], [Wood].
- Manche wichtigen praktischen Probleme werden von der Theorie nicht behandelt; manch ein von der Theorie gelöstes Problem ist aus praktischer Sicht kein Problem [Hamming], [Strachey in Hoare; S. 370, 1989].
- Formale Methoden und Korrektheitsbeweise modellieren die wirkliche Welt mit einem idealen mathematischen Modell. Dazwischen gibt es grundsätzliche und unvermeidbare Differenzen, die eine adäquate Behandlung der wirklichen Welt mit formalen Methoden ausschließt bzw. sehr schwierig und problematisch macht. Rechnerarithmetik, verschiedene Arten von Laufzeitfehlern und Wechselwirkungen mit der Hardware bzw. dem Betriebs-

system sind wichtige Beispiele davon. [Downs], [—; PDCS, Chapter 3, Sections I-4.3 and I-4.4, 1991 May]

- Formale Methoden sind von Theoretikern und Forschern für Theoretiker und Forscher, isoliert von der Mehrheit der Praktiker, entwickelt worden [Tierney; insbesondere S. 58-59], siehe auch [Weigele; 1990].
- “Die meisten Publikationen zum Korrektheitsbeweis von Software sind zu theoretisch, um in der Praxis etwas zu bewirken” [persönliche Kommunikation: Schneider].
- “As long as computing is taught in a manner that conveys the impression that formal methods are useless, students will believe that formal methods are useless” [Gries; S. 54, 1991].
- Es gibt zu viele formale Methoden; die Methodenvielfalt ist unübersichtlich [Fachbereich 2 der GI].

Aus dem Einwand, der Anwendungsaufwand sei zu hoch, ergibt sich die Frage, warum ein mehrfacher Aufwand für die N-Version-Programmierung (siehe Abschnitt 1.1) sowie für sehr umfangreiche Testverfahren und Dokumentation, z.B. bei sicherheitskritischen Anwendungen [Potocki], akzeptiert wird, jedoch für formale Methoden nicht. Diese Frage scheint nicht gestellt worden zu sein. Daraus könnte man die Vermutung folgern, dieser Einwand sei eventuell nicht ernst gemeint, sondern sei eventuell nur eine bequeme Ausrede.

Andere Diskussionen über Einwände gegen eine mathematisch fundierte Vorgehensweise bei der Softwareentwicklung bzw. Gründe für ihren Nicht-Einsatz enthalten [Baber; Abschnitt 1.1, 1987], [Dijkstra; 1978] und [Dijkstra; EWD 1130].

2.2.2 Anforderungen an einen praxisgerechten Ansatz zur Programmkorrektheitsbeweissführung

Aus den im Abschnitt 2.2.1 aufgeführten Gründen und Einwänden gegen den praktischen Einsatz formaler Methoden und Programmkorrektheitsbeweissführungstechniken lassen sich einige Eigenschaften ableiten, die ein praxisgerechter Ansatz zur Programmkorrektheitsbeweissführung aufweisen müßte. Diese werden im folgenden näher betrachtet. Dabei handelt es sich nicht um einen Ersatz für bestehende formale Methoden und Sprachen, sondern um ein für den Praktiker leichter akzeptierbares, einfacheres Verfahren, das dem Softwareentwickler zu spezifischen formalen Methoden (z.B. Z, VDM) verhelfen könnte (als Zwischenstufe), aber ihn nicht dazu zwingt.

Der als sehr hoch betrachtete Aufwand für die Korrektheitsbeweissführung führt zum Verlangen nach maschineller Unterstützung. Sie ist aber nur eine Möglichkeit, diese Problematik in den Griff zu bekommen. Eine andere, die nicht übersehen werden darf, ist die Verringerung des Aufwands durch Vereinfachung des Beweises selbst, des Beweisführungsprozesses und der Darstellung des Beweises. Diese zwei Alternativen — maschinelle Unterstützung und Vereinfachung — schließen sich gegenseitig nicht aus; sie können sich sogar gegenseitig unterstützen.

Für den Softwareentwickler, der die formal beweisbare Korrektheit seines Programms anstrebt, ist die Mathematik ein Mittel zum Zweck, nicht der Zweck selbst. Ihm geht es nicht um die Anwendung bestimmter mathematischer Kenntnisse oder Theorien sondern um die Lösung in der Praxis vorkommender aktueller Probleme — und zwar mit minimalem Aufwand. Deshalb soll man sich bei der Erarbeitung eines praxisgerechten Ansatzes

für die Programmkorrektheitsbeweissführung an den sich aus der Anwendung ergebenden Bedarf orientieren und nicht an die mathematische theoretische Herkunft des Stoffs. Dabei darf die theoretische Fundierung und Substanz nicht aufgegeben werden. Die klassischen ingenieurwissenschaftlichen Fächer sind in dieser Hinsicht gleich gelagert und können deshalb als ein praktikables Vorbild dienen.

Ein wesentlicher Aspekt der Gestaltung einer in der Praxis akzeptablen mathematischen Grundlage für die Entwicklung korrekter Software betrifft die Frage, inwiefern informale Vorgehensweisen berücksichtigt und aufgenommen werden sollen. Auf der einen Seite soll eine theoretisch fundierte Basis für eine wissenschaftliche Vorgehensweise bei der Softwareentwicklung geschaffen werden; auf der anderen Seite soll sie fachkulturell annehmbar sein, was nicht nur vom technischen Wert abhängt: "Formal methods' transition is culturally and economically based, not simply technology-driven, and the explanations behind success do not lie only in technical merit" [Gerhart; 1990]. Ein überlegter Kompromiß ist erforderlich; das informale Extrem würde nicht zur gezielten Verbesserung führen, und das formale Extrem würde keine Akzeptanz finden. Es muß davon ausgegangen werden, daß der optimale Kompromiß je nach Zielkreis, seinen Vorkenntnissen usw. sowie je nach Unterrichtsform (Hochschulstudium, Seminar für in der Praxis stehende Softwareentwickler, Selbststudium) unterschiedlich sein wird.

Ein Mentalitätsunterschied zwischen dem Theoretiker oder Wissenschaftler einerseits und dem Praktiker oder Ingenieur andererseits soll berücksichtigt werden. Der Theoretiker will wissen, *warum* etwas so funktionieren *muß*; der Ingenieur will wissen und verstehen, *wie* es funktioniert. Der Theoretiker sucht eine allgemeine Lösung zu einer Klasse von ähnlich gelagerten vergleichbaren Problemen; der Ingenieur konstruiert eine spezifische effiziente Lösung zu einer konkreten vorgegebenen Problemstellung. Die Arbeitswelt und -ziele des Theoretikers verlangen eine abstraktere, die des Ingenieurs eine konkretere Anschauungs-, Denk- und Ausdrucksweise. Der Ingenieur will manchmal z.B. einen Ausdruck in einer Spezifikation oder in einem Beweis in Bezug auf seinen konkreten Entwurf interpretieren können; dies soll möglich sein. Ein System oder ein Verfahren, das ihn daran hindert, wird er für unpraktisch und nicht hilfreich halten. Der Beweis darf jedoch nicht von solchen Interpretationen abhängen, denn dadurch können Fehler in den Beweis eingeführt werden. Vgl. [De Millo], [Fetzer], [Weiser], [Dijkstra; EWD 1041-7-9 und EWD 1130-4].

Ein praxisgerechter Ansatz für die Programmkorrektheitsbeweissführung muß dem Softwareentwickler möglichst leicht verständlich sein und nützlich erscheinen, und zwar hinsichtlich seiner Softwareentwicklungsprojektziele: Fertigstellen des spezifizierten Programms in einem einsatzfähigen Zustand bis zum geplanten Termin und innerhalb des vorgesehenen Aufwandsbudgets. Lehrbeispiele sollen bekannte, aus seiner Praxis stammende Aufgaben sein, denn nicht bereits bekannte Aufgaben erfordern seinerseits einen zusätzlichen Aufwand, nur um das Beispiel zu verstehen.

Aus diesen Überlegungen und den im Abschnitt 2.2.1 aufgeführten Einwänden gegen formale Methoden und Gründen für die Ablehnung ihres Einsatzes ergeben sich die folgenden spezifischen Anforderungen an einen praxisgerechten Ansatz zur Programmkorrektheitsbeweissführung und entsprechenden Programmkonstruktion. Siehe auch Abschnitt 1.2, "Zielsetzung und Abgrenzung des Themas".

- Die Grundlage und das Verfahren sollen dem Softwareentwickler eine wirkliche Hilfe und klar erkennbare Vorteile bieten, statt ihm eine zusätzliche Last aufbürden. Sie sollen ihm ein tieferes Verständnis von Software und ihrer Entwicklung vermitteln.

- Das Streben nach Einfachheit soll unterstützt und gefördert werden, weil "entbehrliche Komplexität den Blick auf die wirklichen Probleme und deren Lösungen verstellt und so Fortschritt verhindert" [Weigale; 1990]. Dies gilt nicht nur für die Analyse und die Korrektheitsbeweise sondern auch für die entworfenen Programme selbst [Leveson; 1993].
- Wenige und nur einfache mathematische Konzepte und Kenntnisse sollen vorausgesetzt werden, möglichst nicht mehr als Mengen, Funktionen und logische (Boolesche) Algebra [Thomas; 1993].
- Möglichst wenige für den Zielkreis neue oder ungewohnte Konzepte, Notationsformen usw. sollen verwendet werden. Z.B. soll nach Möglichkeit für \forall - und \exists -Konstrukte eine alternative verständlichere Notation geboten werden.
- Grundsätzlich sollen der Stoff, die Notation usw. an bereits vorhandene Kenntnisse und Wissen des Zielkreises anknüpfen und darauf aufbauen.
- Das Verfahren soll auch teilweise, halbformal und informal einsetzbar sein [Austin], [Barroca], [Basili; Diskussionsbeitrag], [Bradley], [Culik], [Sommerville]. Es darf keinen Zwang geben, offensichtlich wahre oder anerkannte Propositionen zu beweisen [Fields].
- Der Lernaufwand soll gering sein. Möglichst wenige Grundregeln, Konstrukte u.ä. sollen für die Anwendung erforderlich sein. Das Verfahren soll derart gestaltet sein, daß es leicht erlernbar ist, z.B. dadurch, daß die Zusammenhänge zwischen den verschiedenen Bestandteilen des Verfahrens logisch einfach und offensichtlich sowie leicht in Erinnerung zu behalten sind.
- Der Anwendungsaufwand soll gering sein.
- Probleme, die in der Praxis von Belang sind, sollen auf geeignete Weise betrachtet werden, auch wenn sie aus theoretischer Sicht von untergeordneter Bedeutung sind. Beispiele davon sind Laufzeitfehler (d.h. die Definitionsbereiche von Programmanweisungen), Ungenauigkeit der Rechnerarithmetik usw. Nicht terminierende Programme bzw. Prozesse sollen ausdrücklich zugelassen und berücksichtigt werden, denn sie kommen in der Praxis nicht selten vor.
- Probleme, die in der Praxis selten wichtig sind, sollen nicht hervorgehoben werden, selbst wenn sie theoretisch sehr wichtig sein können (z.B., ob eine Spezifikation widerspruchsfrei ist oder nicht, d.h. überhaupt eine Realisierung zuläßt).
- Die Anpassung des Abstraktionsgrads der Spezifikation, der Notationsformen u.ä. an die in der Praxis vorkommenden Gegebenheiten (z.B. die Experten auf dem fraglichen Anwendungsgebiet bzw. Kunden, die technische Umgebung des fraglichen Systems usw.) soll ermöglicht bzw. unterstützt werden. Das Verfahren und vor allem die Notationsform für Spezifikationen sollen aus der Sicht des Gesprächspartners konkrete Formulierungen fördern und seine Anschauungsweise widerspiegeln.
- Die Programmkorrektheitsbeweissführung soll ohne maschinelle Unterstützung durchführbar sein. Vgl. [Bloomfield; S. 992, 1986 Sept.], [Toetenel; Abschnitt 1.2].
- Die Korrektheitsbeweissführung soll weitgehend systematisiert werden, um ggf. die Fehleranfälligkeit zu minimieren und um das Problem hinsichtlich des Konsensbildungsmodells der Mathematik (siehe Abschnitt 2.1) zu umgehen. Der Beweis soll als eine Folge von einfachen Schritten mit einfachen und übersichtlichen Zusammenhängen strukturiert werden. Es ist weniger wichtig, daß ein Beweis kurz ist, als daß er einfach und übersichtlich strukturiert und dargestellt wird. Vgl. [Dijkstra; EWD 1041], VLSAL.
- Eine übersichtliche Gliederung des Beweises soll gewährleisten, daß der Leser (1) den Überblick behalten kann, (2) eine beliebige Stelle finden und überprüfen kann, ohne den gesamten Beweis durcharbeiten zu müssen und (3) jederzeit klar sieht, wo im gesamten

Beweis er sich befindet und was noch bewiesen werden muß. Dies könnte z.B. durch eine geeignete Kombination von linearen und hierarchischen Gliederungsmechanismen erreicht werden. Der Beweis sowie ausgewählte Teile davon sollen leicht überprüfbar und nachvollziehbar sein. Die Vereinfachung des Beweises hinsichtlich Präsentation und Kommunikation ist anzustreben. Man soll einzelne Ausdrücke und Schritte in einem Beweis intuitiv interpretieren und auf Programmteilmfunktionen beziehen können — aber nicht müssen (vgl. [Dijkstra; EWD 1041-7]). In der Lehre ist die übersichtliche Gliederung von Beweisen eine Voraussetzung für die effiziente Überprüfung von Lösungen zu gestellten Korrektheitsbeweissführungsaufgaben [Hoffmann; 1990 Okt. 1, Aufgabe 4].

- Anleitungen für die Ermittlung benötigter Zusicherungen (Zwischenbedingungen) sollen angegeben werden.
- Das Verfahren soll einerseits in sich vollständig sein (d.h. das effektive und effiziente Lösen wichtiger in der Praxis vorkommender Probleme ermöglichen), aber andererseits zur Erweiterung offen sein sowie einen möglichen Übergang zu anderen, speziellen formalen Methoden (wie Z, VDM usw.) bieten — aber nicht erzwingen. Eine derartige Erweiterung soll sich nach Bedarf prinzipiell auf die ganze Mathematik erstrecken können [Petrone].
- Zumindest eine Basis für temporale Logik oder die Betrachtung des Programmausführungsablaufs (der zeitlichen Ordnung) soll vorgesehen werden, damit Echtzeitprozesse berücksichtigt werden können.
- Nicht nur die nachherige Verifikation sondern auch die ursprüngliche Konstruktion eines Programms soll unterstützt und — wie in den klassischen Ingenieurwissenschaften üblich — hervorgehoben werden [Dijkstra; BIT, 1968], [Weigle; 1990].
- Formale Spezifikationen sollen durch informelle Erläuterungen ergänzt werden, damit sie leichter gelesen und von einem möglichst breiten Kreis verstanden werden können [Europäische Gemeinschaften; S. 30], [France].
- Die Aufmerksamkeit des Programmkonstruktors soll auf für die Korrektheit des Programms wesentliche Details (z.B. der Parameterübergabe bei Prozeduraufrufen) gelenkt werden. Details, die für die Korrektheit des Programms nicht wesentlich sind, sollen unterdrückt werden können.
- Eine klare Trennung zwischen verschiedenartigen Aspekten der Programmkonstruktion bzw. der Korrektheitsbeweissführung soll gefördert werden — z.B. zwischen Programmlogik und Rechengenauigkeit, zwischen logischer Korrektheit und Terminierung (partieller und vollständiger Korrektheit).
- Die Aufteilung einer umfangreichen Konstruktions- oder Beweissaufgabe (z.B. in der lange algebraische Ausdrücke vorkommen) in mehrere kleinere überschaubarere Aufgaben soll gefördert werden.
- Die eindeutige, präzise Beschreibung von Schnittstellen zwischen Programmteilen soll gefördert werden [Borer; S. 461, 462].
- Das mathematisch basierte Verfahren für die Programmkonstruktion und -korrektheitsbeweissführung soll der Koordination und Kommunikation zwischen allen an der Entwicklung des fraglichen Softwaresystems Beteiligten fördern [Berzins; S. 18 ff.].

Die folgenden Punkte betreffen nicht die Gestaltung des Verfahrens zur Programmkorrektheitsbeweissführung und -konstruktion sondern seine Vermittlung an in der Praxis stehende Softwareentwickler.

- Beim Lehren des Stoffs soll der Schwerpunkt auf die praktische Anwendung, nicht die Theorie und die theoretische Herkunft, gelegt werden. Ein Aspekt dieses Vorschlags ist

im Einwand von Gries “emphasis is on studying the logic — predicate calculus — not on using it” [Gruman; 1989 July] implizit enthalten.

- Konstruktive, engere Kontakte zwischen der Forschung und der industriellen und kommerziellen Anwendung sollen geknüpft werden [Shaw].

Das Verfahren soll summa summarum dem Softwareentwickler zu professionell befriedigenderen Ergebnissen verhelfen: befriedigender für ihn, für seine Klienten und für die Gesellschaft insgesamt.

3. Eine praxisgerechte theoretische Grundlage für die Programmkorrektheitsbeweisführung

In den Abschnitten 3.1 bis 3.3 dieses Kapitels wird eine Grundlage für die Programmkorrektheitsbeweisführung zusammengestellt, die die im Abschnitt 2.2.2 aufgeführten Anforderungen weitgehend erfüllen soll. Einige Aspekte ihrer praktischen Anwendung werden in Abschnitten 3.4 und 3.5 diskutiert.

In Form und Struktur spiegelt die unten vorgestellte Grundlage die in [Baber; 1987, 1990 (*Fehlerfreie Programmierung*), 1991 (*Error-free Software*)] enthaltenen Ausführungen wider, die wiederum inhaltlich auf [Dijkstra; 1976], [Gries; 1981] und [Hoare; 1969, 1987 Aug.-Sept.] basieren. (Siehe auch Abschnitt 2.1.6 für zusätzliche Literaturhinweise.) Es werden allerdings hier Ergänzungen und Abrundungen hinzugefügt, die zu einer integrierteren, kompakteren, abgeklärteren und abgeschlosseneren Basis für die praktische Programmkorrektheitsbeweisführung sowie -konstruktion führen. Dadurch wird u.a. eine weiterführende, verfeinerte und systematischere Behandlung von Definitionsbereichen von Programmanweisungen und -segmenten bzw. von Fragen ihrer vollständigen Korrektheit ermöglicht. Ferner werden dadurch die praktische Anwendung vereinfacht sowie die Anwendungsmöglichkeiten erweitert.

Die semantische Basis der hier vorgestellten Grundlage ist operational orientiert, damit sie für den Praktiker konkret und sein typisches gedankliches Modell der Ausführung eines Programms widerspiegelt. Formal jedoch basiert die Semantik auf Definitionen der Programmanweisungen und ihrer Zusammensetzungen als mathematische Funktionen, weil diese Vorgehensweise zu einer mathematisch einfachen Formulierung und Struktur führt. Die Hoare-Axiome der axiomatischen Semantik kommen hier als Beweisregeln (mathematische Sätze) vor. Vgl. Abschnitt 2.1.8. In einer informellen Einweisung in diesen Stoff, z.B. für in der Praxis stehende Softwareentwickler, die sich weniger für die mathematische Basis interessieren, kann unmittelbar mit informellen Plausibilitätsargumenten für die Beweisregeln begonnen werden.

Die Ausführung einer Programmanweisung wird als das Anwenden einer der fraglichen Programmanweisung entsprechenden Funktion betrachtet bzw. definiert. Zusammensetzungen von Programmanweisungen (z.B. if-Konstrukte, Schleifen, Folgen von Anweisungen) werden als entsprechende Zusammensetzungen der Funktionen, die die untergeordneten Programmsegmente darstellen, definiert. Solche Definitionen werden für eine Sammlung von Programmanweisungen und Zusammensetzungen davon aufgestellt. Obwohl diese Sammlung klein und einfach ist, können viele implementierte Programmiersprachen damit modelliert bzw. darauf basierend definiert werden. Sowohl statische als auch dynamische sowie verschachtelte Vereinbarungen von Variablen lassen sich damit ausdrücken.

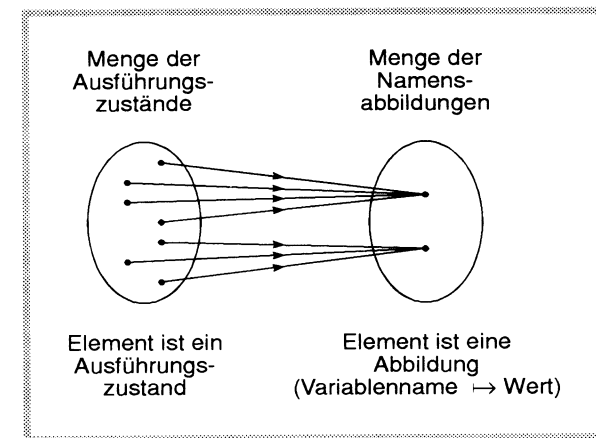
Der Zustand der Ausführung eines Programms wird in der Fachliteratur typischerweise als eine Funktion aufgefaßt, die Variablennamen in Werte abbildet (unten "Namensabbildung" genannt). Dieses Modell reicht jedoch nicht für alle Zwecke aus. In blockstrukturierten Programmiersprachen wie Algol-60 und ihren Nachfolgern sowie in Sprachen, die Rekursion zulassen, können beispielsweise Ausführungszustände entstehen, in denen mehrere gleichnamige Variablen gleichzeitig vorhanden sind. Ein solcher Zustand bestimmt auf naheliegender Weise eine Namensabbildung. Hinter einer Namensabbildung können je-

doch mehrere Ausführungszustände stehen, die sich dadurch voneinander unterscheiden lassen, daß sie zu unterschiedlichen Verhaltensweisen im späteren Verlauf der Programmausführung führen können. Als Beispiel betrachte die zwei Programmsegmente A und B

A: declare (x, Zeichenfolgen, "abc"); declare (y, Z, 1); declare (x, Z, 2)
B: declare (x, Z, 0); declare (y, Z, 1); declare (x, Z, 2)

Nach der Ausführung von A entsteht der Programmausführungszustand za und nach der (getrennten und unabhängigen) Ausführung von B entsteht der Zustand zb. Die Zustände za und zb sind durch die gleiche Namensabbildung ((x, 2), (y, 1)) gekennzeichnet. Die im Zustand za bzw. zb nicht beobachtbare Variable x mit dem Wert "abc" bzw. 0 hat keinen Einfluß auf die an dieser Ausführungsstelle herrschende Namensabbildung. Wird anschließend eine Programmanweisung ausgeführt, die die beobachtbare Variable x löscht, entsteht jeweils ein neuer Ausführungszustand, der durch die Namensabbildung ((x, "abc"), (y, 1)) bzw. ((x, 0), (y, 1)) gekennzeichnet ist. Dieser Unterschied könnte für die Programmkorrektheit relevant sein. Wir müssen deshalb zwischen zwei Ausführungszuständen wie za und zb unterscheiden können, obwohl sie durch dieselbe Namensabbildung gekennzeichnet sind.

Die Auslegung eines Programmausführungszustands als eine Namensabbildung ist also aus mathematischer Sicht eine nicht umkehrbare Funktion. Die Betrachtung der Namensabbildung erlaubt die erforderliche Unterscheidung zwischen Ausführungszuständen deshalb nicht.



Die Zustandsauslegungsfunktion ist nicht umkehrbar

Man muß auf andere Weise auf den zugrunde liegenden Programmausführungszustand zurückgreifen können. Deshalb wird hier ein Programmausführungszustand als eine Folge von Programmvariablen modelliert bzw. definiert; ein solcher Ausführungszustand wird *Datenumgebung* genannt. Durch diese Betrachtungsweise entsteht im wesentlichen für jeden Variablennamen ein Kellerspeicher mit den im allgemeinen verschiedenen Werten der gleichnamigen Variablen. Dieses Modell spiegelt viele Implementierungsmechanismen für Programmiersprachen wider, welches ein didaktischer Vorteil sein kann. Diese Definition

eines Zustands als eine Folge von Programmvariablen legt überflüssigerweise auch eine Ordnung zwischen unterschiedlich genannten Variablen fest. Dies wird deswegen in Kauf genommen, weil es in der Handhabung keinen Nachteil bringt und weil dessen Vermeidung einen zusätzlichen mathematischen Aufwand voraussetzen würde.

Die Ausführungsgeschichte (engl. "trace") einer Programmausführung wird als die Folge von Datenumgebungen definiert, die den einzelnen Ausführungsschritten entsprechen. Dadurch kann der zeitliche Verlauf der Programmausführung betrachtet werden, falls gewünscht oder erforderlich. Es bieten sich folglich zwei Möglichkeiten an, eine Programmweisung bzw. eine Zusammensetzung von Programmweisungen als Funktion zu definieren: die eine Funktion bildet eine Datenumgebung in eine Datenumgebung ab (unten DU-Funktion genannt); die andere, eine Ausführungsgeschichte in eine Ausführungsgeschichte (unten AG-Funktion genannt). Die einer Programmweisung bzw. Zusammensetzung von Programmweisungen entsprechenden DU- und AG-Funktionen sind miteinander eng verwandt; die Beziehung dazwischen wird unten definiert.

Die einem Programmsegment S entsprechende DU-Funktion wird auch mit S bezeichnet. Die entsprechende AG-Funktion wird mit S^* bezeichnet.

Jeder Ausdruck, der in einem Programm vorkommt, wird als eine Funktion aufgefaßt, die eine Datenumgebung in einen Wert abbildet.

In der bisherigen Literatur wird zwischen partieller und vollständiger Korrektheit unterschieden. Aus praktischer Sicht reicht diese Unterscheidung manchmal nicht aus; man will zwischen unendlichen Schleifen und anderen Fehlerarten (z.B. Überlauf, Typ-Widersprüchen, Bezugnahme auf eine nicht vereinbarte Variable usw.) unterscheiden können. Deshalb wird hier zwischen den zwei Definitionsbereichen bezüglich der zwei Funktionsarten (siehe den dritten Absatz oben) unterschieden. Dadurch kann zwischen drei Ausführungsverhaltensweisen unterschieden werden: (1) Die Ausführungsgeschichte ist wohl definiert und endlich lang (die Programmausführung terminiert mit einem definierten Ergebnis, d.h. liefert eine definierte Datenumgebung), (2) die Ausführungsgeschichte ist wohl definiert und unendlich lang (die Programmausführung terminiert nicht, z.B. wegen einer endlosen Schleife) und (3) die Ausführungsgeschichte ist nicht definiert (z.B. wegen eines Laufzeitfehlers). Nur im Falle 1 ist eine Ergebnisdatenumgebung definiert. Die vierte zunächst denkbare Kombination (Ergebnisdatenumgebung definiert, Ausführungsgeschichte nicht definiert) ist durch die Beziehung zwischen den zueinander gehörenden DU- und AG-Funktionen ausgeschlossen. In dieser Arbeit nicht explizit berücksichtigt sind abnormale Abbrüche der Programmausführung wegen nicht ausreichender Kapazität des ausführenden Rechners (z.B. Mangel an Speicherplatz), vgl. Abschnitt 1.2. Die Länge der Datenumgebung liefert einen Ansatz für eine Analyse des Speicherbedarfs; die Länge der Ausführungsgeschichte, für eine Analyse des Zeitbedarfs.

Ferner will man oft Programmfehlerarten (wie die oben genannten) feiner, genauer bzw. detaillierter modellieren können, als die Fachliteratur über die Programmkorrektheitsbeweissführung das typischerweise vorsieht. Als Beispiel sei die Zuweisung $x := A$ erwähnt, wo x ein Variablenname und A ein Ausdruck ist. Das Ergebnis der Ausführung dieser Zuweisung wird in typischen implementierten Programmiersprachensystemen nur dann definiert sein, wenn (1) der Wert des Ausdrucks A in der fraglichen Datenumgebung definiert ist, (2) eine Variable mit dem Namen x vereinbart ist und (3) der Wert von A eines geeigneten Typs ist, d.h., ein Element aus der Variable x zugeordneten Menge ist. Wenn die Frage des Definitionsbereichs einer Zuweisung in der Fachliteratur über Korrektheitsbeweise überhaupt behandelt wird, wird oft nur die Erfüllung der Bedingung (1)

oben verlangt, z.B. in [Dijkstra; 1976], [Hoare; 1987 Aug.-Sept.], [Cohen], [Kaldewaj]. In der hier vorgestellten Grundlage werden alle drei Bedingungen berücksichtigt.

Hier werden nur determinierte Programmweisungen und Konstrukte behandelt. In der Praxis wird das Thema der nichtdeterminierten Konstrukte nicht als wichtiges Problem angesehen, zumindest hinsichtlich der sequentiellen (im Gegensatz zur nebenläufigen) Ausführung eines Programms. Konzeptionell ist die determinierte Betrachtung etwas einfacher; ferner entspricht sie eher dem unter Softwareentwicklern gängigen Denkmodell der Programmausführung sowie den in der Praxis üblichen Programmiersprachen. Nachdem man die auf determinierte Konstrukte bezogene Programmkorrektheitsbeweissführung gelernt hat, kann man bei Bedarf das Gelernte relativ leicht auf nichtdeterminierte Konstrukte erweitern.

Vielen Praktikern sind die \forall - und \exists -Konstrukte der Prädikatenlogik nicht geläufig. Es fällt ihnen oft schwer, sich die Fähigkeit anzueignen, effektiv, richtig und fließend mit dieser Schreib- und Denkweise umzugehen. Die damit eng sinnverwandte Schreibweise $\sum_{i=1}^n$ ist in manchen Zielgruppen viel breiter und gut bekannt. Deshalb wird hier, insbesondere in Vor- und Nachbedingungen, als Alternativen für die \forall - und \exists -Konstrukte über abzählbare Mengen die Schreibweise **und** _{$i=1$} ^{n} bzw. **oder** _{$i=1$} ^{n} eingeführt, die auch der Kommunikation zwischen Softwareentwicklern und ihren Klienten dienen soll. Entsprechend wird auch die Schreibweise **und** und **oder** für \wedge bzw. \vee verwendet. In den metamathematischen Formeln, die bei der Kommunikation mit Nichtinformatikern kaum eine Rolle spielen, wird von der Schreibweise ($A_i: \dots : \dots$), ($E_i: \dots : \dots$), ($U_i: \dots : \dots$) usw. Gebrauch gemacht. Diese Schreibweise hat sich vor allem in der Fachliteratur über die Programmkorrektheitsbeweissführung etabliert, siehe z.B. [Dijkstra], [Gries; 1981], [Kaldewaj; Chapter 3]. Das Thema Schreibweise wird in Abschnitt 3.4.2 näher diskutiert.

Solche Unterschiede und Ergänzungen zu bereits eingeführten Definitionen, Notationsformen usw. sind grundsätzlich und aus theoretischer Sicht nicht von besonderer substantieller Bedeutung. Sie können das Erlernen und das praktische Anwenden dieses Stoffs jedoch signifikant erleichtern und fördern. Sie können deshalb von großer praktischer Bedeutung sein.

3.1 Grundbegriffe, -betrachtungen und Definitionen

3.1.1 Programmvariablen, Datenumgebungen und Ausführungsgeschichten

Eine *Programmvariable* (unten auch Variable genannt) ist ein Tripel (N, M, W) , bestehend aus einem Namen N , einer nicht leeren Menge M und einem Wert W , der ein Element aus der Menge M ist. Eine *Datenumgebung* ist eine (eventuell leere) Folge von Programmvariablen. Die Menge aller Datenumgebungen wird mit " \mathcal{D} " bezeichnet; es wird unterstellt, daß diese Menge existiert bzw. definiert werden kann (vgl. das Russell'sche Paradox). Eine *Ausführungsgeschichte* ist eine Folge von Datenumgebungen. Die Menge aller Ausführungsgeschichten ist \mathcal{D}^* , wobei $\mathcal{D}^* \triangleq (\cup_{n \in \mathbb{N}_0} \mathcal{D}^n) \cup \mathcal{D}^\infty$.

Zwei Datenumgebungen $d_0 = [(N_01, M_01, W_01), (N_02, M_02, W_02), \dots]$ und $d_1 = [(N_11, M_11, W_11), (N_12, M_12, W_12), \dots]$ sind *gleich*, wenn sie gleich lang sind und $N_{0i} = N_{1i}$, $M_{0i} = M_{1i}$ und $W_{0i} = W_{1i}$ für alle i . Die Datenumgebungen d_0 und d_1 sind

strukturell gleich, wenn sie abgesehen von den Werten der darin enthaltenen Variablen gleich sind, d.h. wenn sie gleich lang sind und $N0i=N1i$ und $M0i=M1i$ für alle i .

Die Datenumgebung $d0$ enthält eine Variable x , wenn mindestens ein i existiert derart, daß $N0i="x"$. Die Anführungszeichen deuten daraufhin, daß es sich um den Namen x handelt, nicht um den Wert von x .

Die Funktion "Menge" ermittelt die zu einer Programmvariable (N, M, W) gehörende Menge M : Menge." x ". d ist die Menge, die der ersten Variable mit dem Namen x in der Datenumgebung d zugeordnet ist. Enthält d keine Variable mit dem Namen x , dann ist Menge." x ". d die leere Menge.

Die Datenumgebung d enthält die Variable x genau dann, wenn Menge." x ". $d \neq \emptyset$.

3.1.2 Werte von Variablen und Ausdrücken in Datenumgebungen

Der Wert einer Variable x in einer Datenumgebung d ist der Wert der ersten in d vorkommenden Variable mit dem Namen x . Enthält d keine Variable x , dann ist der Wert von x in d undefiniert. Ein Variablenname x kann als eine Funktion aufgefaßt werden, die eine Datenumgebung d in einen Wert $x.d$ abbildet. Umgekehrt kann eine Datenumgebung als eine Funktion aufgefaßt werden, die einen Variablennamen in einen Wert abbildet.

Der Wert eines Ausdrucks A in einer Datenumgebung d ist der Wert, der sich daraus ergibt, daß zuerst alle Programmvariablenamen durch ihre Werte in d ersetzt werden und danach der Wert des daraus resultierenden Ausdrucks berechnet wird. Wie im Falle eines Variablennamens kann ein Ausdruck A als eine Funktion aufgefaßt werden, die eine Datenumgebung d in einen Wert $A.d$ abbildet. Umgekehrt kann eine Datenumgebung als eine Funktion aufgefaßt werden, die einen Ausdruck in einen Wert abbildet.

Bei der Ermittlung des Werts einer Feldvariable $x(ia)$ in einer Datenumgebung d wird zuerst der Indexausdruck ia in d ausgewertet, um den wirklichen Namen der angesprochenen Variable (z.B. " $x(3)$ ") zu bestimmen. Formal, $(x(ia)).d \triangleq (x(ia.d)).d$. Auf die mit Feldvariablen verbundene potentielle Problematik wird in Abschnitt 3.3.4.2 näher eingegangen.

Ein Ausdruck mit Werten in {falsch, wahr} (ein Boolescher Ausdruck) wird auch eine Bedingung genannt. Eine Teilmenge Bm von \mathbb{D} entspricht einer Bedingung (einem Booleschen Ausdruck) Ba , falls

$$Bm = (\cup d : d \in \mathbb{D} \wedge Ba.d : \{d\})$$

Aufgrund dieser Definition wird unten eine Bedingung (Boolescher Ausdruck) manchmal als eine Funktion auf \mathbb{D} und manchmal als eine Teilmenge von \mathbb{D} betrachtet.

3.1.3 Programmanweisungen als Funktionen auf \mathbb{D}

Eine Programmanweisung wird als eine (im allgemeinen partielle) Funktion auf \mathbb{D} nach \mathbb{D} aufgefaßt (siehe "DU-Funktion" in Abschnitt 3 oben). Vier atomare Programmanweisungsarten (Zuweisung, Deklaration (Vereinbarung einer Variable), Löschung einer Variable und Nullanweisung) und vier Zusammensetzungen von Programmanweisungen (if-Anweisung, Folge von Anweisungen, while-Schleife und Prozeduraufruf ohne formale Parameterübergabe) werden hier definiert.

Das Ergebnis der Ausführung einer Zuweisung $x:=A$ auf eine Datenumgebung $d0$ ist die Datenumgebung $d1$, wo $d1=d0$ bis auf den Wert der ersten in $d0$ vorkommenden Variable mit dem Namen x . Dieser Wert in $d1$ ist der Wert des Ausdrucks A in $d0$, d.h. $x.d1=A.d0$. Formal, $(x:=A).d0 \triangleq d1$, wo $N1i=N0i$ und $M1i=M0i$ für alle i , $W1i=W0i$ für alle $i \neq j$ und $W1j=A.d0$, wobei $j=(\min k : N0k="x" : k)$. Diese Definition führt genau dann zu einem Ergebnis, das die Definition einer Datenumgebung erfüllt, wenn (1) $A.d0$ definiert ist, (2) $d0$ eine Variable mit dem Namen x enthält und (3) $A.d0 \in M1j$ (äquivalent: $A.d0 \in M0j$). Vgl. den Absatz über Programmfehlerarten im Abschnitt 3 oben. Man merke, diese Definition schließt durch die Auswertung des Ausdrucks A verursachte Nebeneffekte aus. Bei einer Zuweisung zu einer Feldvariable wird wie bereits beschrieben der Indexausdruck ausgewertet, um den wirklichen Namen der angesprochenen Variable zu bestimmen. D.h., $(x(ia):=A).d0 \triangleq (x(ia.d0):=A).d0$.

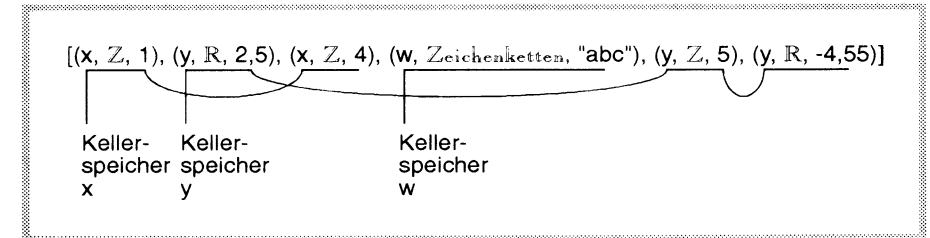
Durch die Ausführung einer Deklaration (Vereinbarung einer Variable mit Zuordnung einer Menge und einem Wert) auf eine Datenumgebung d wird eine neue Variable am Anfang der Datenumgebung eingefügt:

$$(\text{declare } (x, AM, AW)).d \triangleq [(x, AM.d, AW.d)] \& d$$

Dabei bedeutet $\&$ die Verknüpfung zweier Folgen. Diese Definition führt genau dann zu einem Ergebnis, das die Definition einer Datenumgebung erfüllt, wenn $AW.d \in AM.d$. Typischerweise ist die Menge AM eine Konstante auf \mathbb{D} , aber prinzipiell könnte sie der Wert eines entsprechenden Ausdrucks sein.

Durch die Ausführung einer Löschung einer Variable wird die erste Variable mit dem fraglichen Namen entfernt: $(\text{release } x).d0 = d1$, wo $N1i=N0i$, $M1i=M0i$ und $W1i=W0i$ für alle $i < j$ und $N1i=N0(i+1)$, $M1i=M0(i+1)$ und $W1i=W0(i+1)$ für alle $i \geq j$, wo $j=(\min k : N0k="x" : k)$. Diese Definition führt genau dann zu einem Ergebnis, wenn $d0$ eine Variable mit dem Namen x enthält.

Die Programmanweisungen `declare` und `release` entsprechen den Kellerspeicheroperationen "push" und "pop". Effektiv entsteht in der Datenumgebung für jeden Variablennamen ein Kellerspeicher.



Kellerspeicher gleichnamiger Variablen in einer Datenumgebung

Die Nullanweisung ist die Identitätsfunktion: $\text{null}.d \triangleq d$ für alle $d \in \mathbb{D}$.

Die Wirkung der Ausführung einer if-Anweisung wird wie folgt definiert:

$$(\text{if } B \text{ then } S1 \text{ else } S2 \text{ endif}).d \triangleq \begin{matrix} S1.d, & \text{falls } B.d \\ S2.d, & \text{falls } \neg B.d \end{matrix}$$

Die *Folge von Anweisungen* wird als die Zusammensetzung der entsprechenden Funktionen definiert: $(S1; S2).d \triangleq S2.(S1.d)$.

Entsprechend der iterativen Ausführung wird die Ausführung der *while-Schleife* rekursiv definiert:

$$(\text{while } B \text{ do } S \text{ endwhile}).d \triangleq \begin{array}{ll} (S; \text{while } B \text{ do } S \text{ endwhile}).d, & \text{falls } B.d \\ d, & \text{falls } \neg B.d \end{array}$$

Andere Schleifenarten können als Zusammensetzungen von while-Schleifen und anderen Anweisungen definiert werden. Z.B. wird die *repeat-Schleife* wie folgt definiert: $(\text{repeat } S \text{ until } B \text{ endrepeat}) \triangleq (S; \text{while } \neg B \text{ do } S \text{ endwhile})$. Die Schleife mit internem Ausgang (loop S1; if B then exit; S2 endloop) wird als $(S1; \text{while } \neg B \text{ do } S2; S1 \text{ endwhile})$ definiert.

Ein *Aufruf auf ein Unterprogramm* ohne formale Parameterübergabe hat den gleichen Effekt als die Ausführung des Unterprogramms. D.h., besteht das Unterprogramm Prg aus der (im allgemeinen zusammengesetzten) Programmanweisung S, dann gilt: $(\text{call Prg}).d \triangleq S.d$.

Ein Aufruf auf ein Unterprogramm mit formaler Parameterübergabe wird hier nicht definiert; siehe die Abschnitte 3.2.4 und 3.3.4.4.

Auch Eingabe- und Ausgabeanweisungen werden hier nicht formal definiert. Definitionen solcher Anweisungen müßten Details der Definition der betrachteten Programmiersprache mit einbeziehen. Dieses Thema wird in [Baber; 1987, Abschnitt 4.1] näher behandelt.

Jede Programmanweisung bzw. Zusammensetzung davon wurde oben als eine Funktion auf \mathbf{ID} nach \mathbf{ID} definiert. Folglich ist der Definitionsbereich einer Programmanweisung S das Urbild von \mathbf{ID} bezüglich S, d.h. $S^{-1}.\mathbf{ID}$.

Die Definitionsbereiche der verschiedenen Programmanweisungen folgen unmittelbar aus ihren Definitionen:

$$(x := A)^{-1}.\mathbf{ID} = (\cup d : d \in \mathbf{ID} \wedge A.d \in \text{Menge.} \text{``x''}.d : \{d\})$$

$$(\text{declare } (x, AM, AW))^{-1}.\mathbf{ID} = (\cup d : d \in \mathbf{ID} \wedge AW.d \in AM.d : \{d\})$$

$$(\text{release } x)^{-1}.\mathbf{ID} = (\cup d : d \in \mathbf{ID} \wedge \text{Menge.} \text{``x''}.d \neq \emptyset : \{d\})$$

$$(\text{null})^{-1}.\mathbf{ID} = \mathbf{ID}$$

$$(\text{if } B \text{ then } S1 \text{ else } S2 \text{ endif})^{-1}.\mathbf{ID} = B \cap S1^{-1}.\mathbf{ID} \cup \neg B \cap S2^{-1}.\mathbf{ID}$$

$$(S1; S2)^{-1}.\mathbf{ID} = S1^{-1}.(S2^{-1}.\mathbf{ID})$$

$$(\text{while } B \text{ do } S \text{ endwhile})^{-1}.\mathbf{ID} = (\cup n : n \in \mathbb{N}_0 : Y^n.Bf)$$

$$(\text{call Prg})^{-1}.\mathbf{ID} = S^{-1}.\mathbf{ID} \quad [\text{Prg besteht aus } S]$$

wo Bw die Menge aller Datenumgebungen d ist, für die B.d definiert und wahr ist, und Bf die Menge aller d ist, für die B.d definiert und falsch ist. Die Funktion Y wird wie folgt definiert: $Y^0.M \triangleq M$, $Y.M \triangleq Bw \cap S^{-1}.M$ und $Y^n.M \triangleq Y.(Y^{n-1}.M)$ für $n > 1$ und für jede Teilmenge M von \mathbf{ID} . $Y^n.Bf$ ist die Menge aller Datenumgebungen, worauf angewendet die while-Schleife nach genau n Ausführungen des Schleifenkerns terminiert. Folg-

lich umfaßt der oben angegebene Definitionsbereich einer while-Schleife nur Datenumgebungen, worauf angewendet die Schleife terminiert.

3.1.4 Programmanweisungen als Funktionen auf \mathbf{ID}^*

Die Programmanweisungen und ihre Zusammensetzungen können wie bereits erwähnt auch als Funktionen auf \mathbf{ID}^* nach \mathbf{ID}^* definiert werden (siehe "AG-Funktion" in Abschnitt 3 oben). Ist S eine atomare Programmanweisung (Zuweisung, Deklaration, Löschung oder Nullanweisung), dann fügt die Ausführung von S die neue Datenumgebung zur Ausführungsgeschichte hinzu: $S^*.g \triangleq g \& S.(\text{letz}.g)$ für jede nicht leere und endlich lange Ausführungsgeschichte $g \in \mathbf{ID}^*$. Dabei ist $\text{letz}.g$ die letzte Datenumgebung in der Ausführungsgeschichte g. Entsprechende Definitionen der AG-Funktionen für die zusammengesetzten Anweisungsarten (if-Anweisung, Folge von Anweisungen, while-Schleife und Prozeduraufruf) ähneln den bereits angegebenen Definitionen der entsprechenden DU-Funktionen hinsichtlich Form und Inhalt. Wesentlich dabei ist, daß die Ausführung einer Anweisung auf die letzte Datenumgebung der bisherigen Ausführungsgeschichte setzt und daß die letzte Datenumgebung in der resultierenden Ausführungsgeschichte das Endergebnis ist. Formal:

$$(A \ g, d : g \in \mathbf{ID}^* \wedge d \in \mathbf{ID} : S^*.g \& [d] = g \& S^*.[d])$$

$$(A \ d : d \in \mathbf{ID} : S.d = \text{letz}.(S^*.[d]))$$

Dabei wird vereinbart, daß $S^*.g = g$ und $g \& x = g$ für alle (auch undefinierten) x, falls g eine unendlich lange Folge ist. Aus praktischer Sicht betrachtet kommt es nie zu der Ausführung von S, wenn ein vorheriges Programmteil nicht terminiert; das Verhalten (die Ausführungsgeschichte) des gesamten Programms wird ausschließlich vom zuerst ausgeführten nicht terminierenden Programmteil bestimmt.

Aus der vorletzten Proposition oben folgt, daß $S^*.g$ genau dann definiert ist, wenn $\text{letz}.g \in \text{letz}.(S^{-1}.\mathbf{ID}^*)$ oder g eine unendlich lange Ausführungsgeschichte ist. Aus der letzten Proposition oben folgt, daß $S^{-1}.\mathbf{ID} \subseteq \text{letz}.(S^{-1}.\mathbf{ID}^*)$. Damit kann formal bestätigt werden, daß die Ausführungsgeschichte $S^*.g$ immer definiert ist, wenn die Datenumgebung S.letz.g definiert ist.

3.1.5 Vorbedingungen und Nachbedingungen

Eine Bedingung V ist eine (gewöhnliche) *Vorbedingung* der *Nachbedingung* P bezüglich der Programmanweisung S, falls die Wahrheit von V vor der Ausführung von S die Wahrheit von P danach sicherstellt, insofern die Ausführung von S überhaupt zu einem definierten Ergebnis führt. Man schreibt dafür $\{V\} S \{P\}$. Formal,

$$\{V\} S \{P\} \triangleq (A \ d : d \in V \cap S^{-1}.\mathbf{ID} : S.d \in P) \quad [V, P \text{ als Teilmengen von } \mathbf{ID}]$$

$$\text{bzw. } \{V\} S \{P\} \triangleq (A \ d : V.d \wedge d \in S^{-1}.\mathbf{ID} : P.(S.d)) \quad [V, P \text{ als Bedingungen}]$$

Ein Satz der Form $\{V\} S \{P\}$ wird hier *Korrektheitsaussage* genannt. Die Vorbedingung V und die Nachbedingung P zusammen können auch als *Spezifikation* des Programmsegments S aufgefaßt werden.

Eine *strikte* Vorbedingung V der Nachbedingung P bezüglich S ist eine Vorbedingung, die Teilmenge des Definitionsbereichs von S ist. Formal,

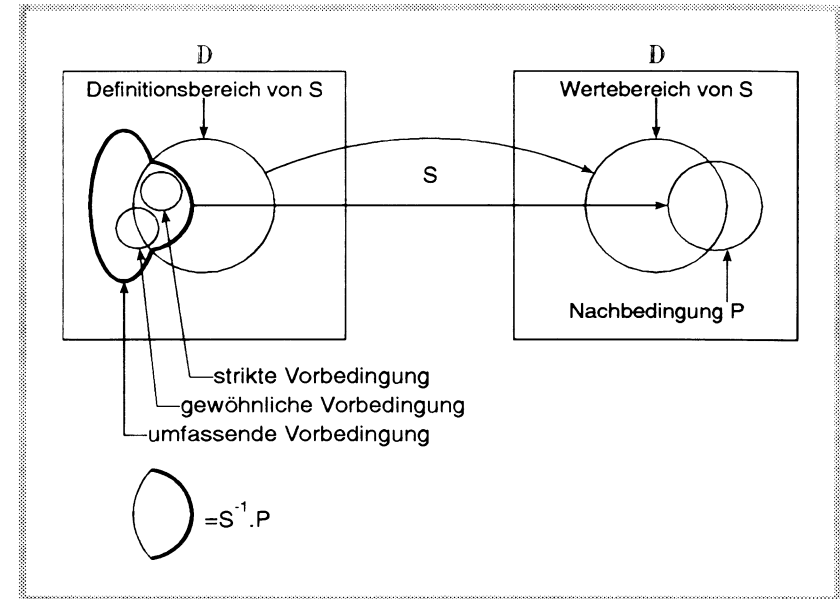
$$\{V\} S \{P\} \text{ strikt} \triangleq \{V\} S \{P\} \wedge V \subseteq S^{-1}.D$$

Eine *gewöhnliche* Vorbedingung stellt die *partielle* Korrektheit des fraglichen Programmsegments sicher. Eine *strikte* Vorbedingung stellt die *vollständige* (oder *totale*) Korrektheit sicher.

In der theoretischen Literatur interessiert man sich oft für die schwächste Vorbedingung. Wegen der detaillierteren Betrachtung des Definitionsbereichs hier ist die schwächste *gewöhnliche* Vorbedingung $(S^{-1}.P \cup (D - S^{-1}.D))$ oft umständlich zu ermitteln sowie nicht sonderlich bedeutungsvoll. Das gleiche gilt für die schwächste *strikte* Vorbedingung $(S^{-1}.P)$. Das Wesentliche an der schwächsten Vorbedingung ist es, daß sie eine notwendige Bedingung ist. Das bedeutet wiederum, daß sie das Urbild der Nachbedingung P bezüglich S umfassen muß. Deshalb wird unten anstatt der schwächsten Vorbedingung eine *umfassende* Vorbedingung betrachtet, die wie folgt formal definiert wird:

$$\{V\} S \{P\} \text{ umfassend} \triangleq \{V\} S \{P\} \wedge S^{-1}.P \subseteq V$$

Die Beziehungen zwischen den verschiedenen Arten von Vorbedingungen und bestimmten Teilmengen von D veranschaulicht das folgende Diagramm aus [Baber; S. 67, 1987]:



Beziehungen zwischen dem Definitionsbereich einer Programmanweisung, einer *gewöhnlichen* Vorbedingung, einer *strikten* Vorbedingung, einer *umfassenden* Vorbedingung und dem Urbild der Nachbedingung

Die Wahrheit einer *strikten* Vorbedingung vor der Ausführung eines Programmsegments S stellt sicher, daß die Ausführung von S mit einem definierten Ergebnis terminiert. Wie bereits erwähnt sind nicht terminierende Programmsegmente manchmal von Interesse. Es wird deshalb eine *halbstrikte* (semistrikte) Vorbedingung definiert, die nur sicherstellt, daß die Ausführungsgeschichte definiert ist, aber nicht, daß S terminiert. Formal,

$$\{V\} S \{P\} \text{ halbstrikt} \triangleq \{V\} S \{P\} \wedge V \subseteq \text{letz}t.(S^{*-1}.D^*)$$

Eine *strikte* Vorbedingung ist auch *halbstrikt*. Das umgekehrte gilt nicht, z.B. im Falle einer nicht terminierenden Schleife. Die Eigenschaft, daß $S.(\text{letz}t.g)$ definiert ist, ist stärker als die Eigenschaft, daß $S^*.g$ definiert ist.

Aus den Definitionen oben folgen einige wichtige Beziehungen:

$$\{V\} S \{P\} = (V \cap S^{-1}.D \subseteq S^{-1}.P)$$

$$\{V\} S \{P\} \text{ strikt} = (V \subseteq S^{-1}.P)$$

$$\{V\} S \{P\} \text{ umfassend} = (V \cap S^{-1}.D = S^{-1}.P)$$

$$\{V\} S \{P\} \text{ strikt und umfassend} = (V = S^{-1}.P)$$

$S^{-1}.P$ ist die schwächste *strikte* und die stärkste *umfassende* Vorbedingung.

3.1.6 Nachbedingungen mit Bezug auf vorherige Variablenwerte

Oft ist es nützlich, in der Nachbedingung Bezug auf den Wert einer oder mehrerer Variablen in der ursprünglichen Datenumgebung zu nehmen. Dies kann auf verschiedene Weisen erfolgen bzw. betrachtet werden. Dementsprechend müssen die oben angegebenen Definitionen ergänzt bzw. geändert werden. Drei Möglichkeiten, solche Nachbedingungen aufzufassen und zu behandeln, sind:

(1) Die Nachbedingung P bezieht sich auf Werte von Variablen vor und nach der Ausführung der fraglichen Anweisung S und ist somit eine Funktion von zwei Datenumgebungen d_0 und d_1 , die vor bzw. nach Ausführung von S gelten. Als Menge betrachtet ist P nicht mehr eine Teilmenge von \mathbf{ID} , sondern eine Teilmenge von $\mathbf{ID} \times \mathbf{ID}$ (eine Relation). Die Definition von Vor- und Nachbedingungen muß auf geeignete Weise geändert werden, z.B.: $\{V\} S \{P\} \triangleq (\mathbf{A} d : d \in V \cap S^{-1} \cdot \mathbf{ID} : (d, S.d) \in P)$ bzw. $\{V\} S \{P\} \triangleq (\mathbf{A} d : V.d \wedge d \in S^{-1} \cdot \mathbf{ID} : P(d, S.d))$ sowie für strikte, umfassende und halbstrikte Vorbedingungen entsprechend.

(2) Für jede Programmvariable, auf deren ursprünglichen Wert sich die Nachbedingung bezieht, wird eine neue Programmvariable eingeführt, die den Wert der Variable an der fraglichen Stelle festhält. Betrachte z.B. die Forderung $\{V\} S \{P(x, x')\}$, wo $P(x, x')$ ein Ausdruck ist, in dem x und x' (und eventuell andere Programmvariablen) vorkommen, und wo x' der Wert von x in der ursprünglichen Datenumgebung ist. Diese Forderung wird umformuliert in $\{V\} x' := x \{V \wedge x = x'\} S \{P(x, x')\}$. Hier ist x' eine Programmvariable, die weder in S noch in V angesprochen wird.

(3) Der fragliche ursprüngliche Wert wird als Parameter der Korrektheitsaussage betrachtet. Die Korrektheitsaussage soll für alle in Frage kommenden Werte dieses Parameters gelten. Z.B. ist $\{V\} S \{P(x, x')\}$ als $\{V \wedge x = x'\} S \{P(x, x')\}$ bzw. $(\mathbf{A} x' : x' \in M : \{V \wedge x = x'\} S \{P(x, x')\})$ zu verstehen. Dabei muß die Menge M aus dem Zusammenhang ersichtlich sein; in solchen Situationen schreibt man oft nur $(\mathbf{A} x' : : \dots)$. Die Variable x' hier ist keine Programmvariable; [Kaldewaj] nennt sie "Spezifikationsvariable". Diese Definition ist eine sinnvolle Verallgemeinerung der Definition von $\{V\} S \{P\}$ (siehe Abschnitt 3.1.5), denn $(\mathbf{A} x' : : \{V \wedge x = x'\} S \{P\}) = \{V\} S \{P\}$, wenn P nicht von x' abhängt und falls aus V folgt, daß die Variable x deklariert ist.

In dieser Arbeit wird von der letzten Möglichkeit Gebrauch gemacht, weil sie keine neuen, zusätzlichen Konzepte sowie keine Ergänzung des fraglichen Programmsegments voraussetzt und deshalb einfach ist. Dabei wird das Zeichen ' verwendet, um auf den ursprünglichen Wert einer Programmvariable Bezug zu nehmen.

Der manchmal zu sehende Gebrauch von Großbuchstaben für den ursprünglichen Wert einer Variable ist typografisch keine gute Lösung, weil in der Praxis Großbuchstaben nicht selten für Programmvariablenamen verwendet werden. In Z bezeichnet ' den neuen Wert einer Variable; der Variablenname ohne Zusatz bezieht sich auf ihren ursprünglichen Wert. In VDM bezeichnet der Pfeil \leftarrow über dem Variablennamen ihren ursprünglichen Wert; der Name ohne zusätzliches Zeichen bezieht sich auf den neuen Wert. In Eiffel schreibt man "old ..." für den ursprünglichen Wert der genannten Variable.

3.2 Allgemein gültige Sätze ("Beweisregeln")

Auf der Basis der in Abschnitt 3.1 aufgeführten Definitionen kann eine Reihe von Sätzen — auch *Beweisregeln* genannt — aufgestellt werden, deren konsequente Anwendung zu einer Systematisierung der Programmkorrektheitsbeweissführung führt. Die Beweisregeln dienen der iterativen Zerlegung einer zu beweisenden Korrektheitsaussage in Lemmata und der Ableitung einer Vorbedingung (siehe Abschnitt 3.3) sowie als Leitlinien für die Neukonstruktion eines Programmsegments (siehe Kapitel 4). Auch bei der Programmkonstruktion dienen die Beweisregeln der Zerlegung der Entwurfsaufgabe; dabei stellen sie sicher, daß die einzelnen konstruierten Programmteile zusammen die gewünschte Gesamtwirkung aufweisen werden.

Die Bezeichnungen der Beweisregeln sind [Baber; 1990 (*Fehlerfreie Programmierung*)] entnommen. Einige Beweisregeln gelten für alle Programmanweisungsarten; andere beziehen sich jeweils auf nur eine Anweisungsart.

3.2.1 Stärkung einer Vorbedingung, Schwächung einer Nachbedingung

Beweisregel B1: $[V \Rightarrow V1] \wedge \{V1\} S \{P1\} \wedge [P1 \Rightarrow P] \Rightarrow \{V\} S \{P\}$

Diese Beweisregel ermöglicht die Stärkung von Bedingungen, wenn man rückwärts durch ein Programm arbeitet, und die Schwächung von Bedingungen, wenn man vorwärts durch ein Programm arbeitet.

Die Klammern $[\]$ in der Schreibweise $[V \Rightarrow V1]$ (bzw. $[P1 \Rightarrow P]$) oben bedeuten die universale Quantifizierung über jede in Frage kommende Variable, in diesem Fall über alle Datenumgebungen aus \mathbf{ID} : $(\mathbf{A} d : d \in \mathbf{ID} : V.d \Rightarrow V1.d)$. Dieser Ausdruck ist gleichwertig mit $(\mathbf{A} d : d \in V : d \in V1)$ und damit auch mit der Teilmengenrelation $V \subseteq V1$. Diese Bemerkung gilt für die gleichartigen Implikationen in den folgenden Beweisregeln entsprechend.

3.2.2 Programmanweisungen und ihre Zusammensetzungen

Beweisregel Z1: $\{P_A^x\} x := A \{P\}$ sowie $\{P_A^x\} \text{declare } (x, M, A) \{P\}$

Dabei ist P_A^x die Bedingung, die sich daraus ergibt, wenn man die Variable x durch den in Klammern gesetzten Ausdruck A überall in P ersetzt. Bei der Bedingung P muß es sich um einen Ausdruck handeln, worin x für den Wert der Variable x steht (im Gegensatz zum Namen selbst). Diese Beweisregel dient der Ableitung einer Vorbedingung bezüglich einer Zuweisung oder einer declare-Anweisung. Bei der Anwendung dieser Beweisregel auf eine Zuweisung zu einer Feldvariable, z.B. $x(\text{ia}) := A$, muß darauf geachtet werden, daß $x(\text{ia})$ kein Variablenname ist; der Indexausdruck ia muß ausgewertet werden, um den eigentlichen Namen der angesprochenen Feldvariable zu ermitteln, der in der Nachbedingung zu ersetzen ist. Diese potentielle Falle wird in Abschnitt 3.3.4.2 näher behandelt.

Beweisregel Z2: $[V \Rightarrow P_A^x] \Rightarrow \{V\} x := A \{P\}$

sowie $[V \Rightarrow P_A^x] \Rightarrow \{V\} \text{declare } (x, M, A) \{P\}$

Diese Beweisregel ist eine Kombination der Beweisregeln B1 und Z1. Die Bemerkungen über Beweisregel Z1 treffen deshalb auch für Beweisregel Z2 zu. Die Beweisregel Z2 dient der Verifikation einer vorgegebenen Vorbedingung bezüglich einer Zuweisung oder einer declare-Anweisung.

$$\text{Beweisregel IF1: } (\{V \wedge B\}S1\{P\} \wedge \{V \wedge \neg B\}S2\{P\}) \\ \Rightarrow \{V\} \text{ if } B \text{ then } S1 \text{ else } S2 \text{ endif } \{P\}$$

Diese Beweisregel dient der Verifikation einer Korrektheitsaussage über eine if-Anweisung.

$$\text{Beweisregel IF2: } \{V1\}S1\{P\} \wedge \{V2\}S2\{P\} \\ \Rightarrow \{B \wedge V1 \vee \neg B \wedge V2\} \text{ if } B \text{ then } S1 \text{ else } S2 \text{ endif } \{P\}$$

Diese Beweisregel dient der Ableitung einer Vorbedingung bezüglich einer if-Anweisung. In der Fachliteratur erscheint manchmal als Vorbedingung statt $\{B \wedge V1 \vee \neg B \wedge V2\}$ der gleichwertige Ausdruck $\{(B \Rightarrow V1) \wedge (\neg B \Rightarrow V2)\}$. Der letztere Ausdruck spiegelt vielleicht die Logik des Beweises dieser Beweisregel eher wider; der erste ist jedoch in der Anwendungspraxis oft vorteilhafter.

$$\text{Beweisregel F1: } \{V\}S1\{P1\} \wedge \{P1\}S2\{P\} \Rightarrow \{V\} S1; S2 \{P\}$$

Diese Beweisregel dient der Ableitung oder der Verifikation einer Vorbedingung bezüglich einer Folge von Programmanweisungen.

$$\text{Beweisregel W1: } \{I \wedge B\}S\{I\} \Rightarrow \{I\} \text{ while } B \text{ do } S \text{ endwhile } \{I \wedge \neg B\}$$

Diese Beweisregel dient der Verifikation einer Korrektheitsaussage über eine while-Schleife. Die Bedingung I ist die *Schleifeninvariante*, die eine wichtige Entwurfsentscheidung verkörpert, und muß bereits (z.B. aus der Programmkonstruktion) bekannt sein.

Die Schleifeninvariante ist vor der Ausführung der Schleife erfüllt (meist auf triviale Weise). Bei eventueller Terminierung der Schleife ist sie auch erfüllt (vgl. $I \wedge \neg B$ als Nachbedingung in der Beweisregel W1). Die Anfangs- und Endsituationen sind also Spezialfälle der Schleifeninvariante. Anders herum betrachtet ist die Schleifeninvariante eine Verallgemeinerung der Anfangs- und Endsituationen. Diese Beobachtung liefert eine nützliche Faustregel für die Bestimmung einer geeigneten Schleifeninvariante.

$$\text{Beweisregel W2: } \{V\} \text{Init}\{I\} \wedge \{I \wedge B\}S\{I\} \wedge [I \wedge \neg B \Rightarrow P] \\ \Rightarrow \{V\} \text{Init; while } B \text{ do } S \text{ endwhile } \{P\}$$

Diese Beweisregel dient der Verifikation einer Korrektheitsaussage über eine while-Schleife mit Initialisierung oder der Ableitung einer Vorbedingung einer while-Schleife mit Initialisierung. Dabei muß die Schleifeninvariante I bereits bekannt sein. Die Initialisierung hat im Korrektheitsbeweis — und folglich auch im Programm selbst — nur die Aufgabe, die anfängliche Wahrheit der Schleifeninvariante sicherzustellen (vgl. Beweisregel W1).

$$\text{Beweisregel R1: } \{V \vee I \wedge \neg B\}S\{I\} \wedge [I \wedge B \Rightarrow P] \Rightarrow \{V\} \text{ repeat } S \text{ until } B \text{ endrepeat } \{P\}$$

Diese Beweisregel für die repeat-Schleife entspricht der Beweisregel W2. Die repeat-Schleife entspricht oft strukturell einer while-Schleife, deren Schleifenkern auch die Initialisierung ist bzw. enthält.

$$\text{Beweisregel R2: } \{I\}S\{I\} \wedge [I \wedge B \Rightarrow P] \Rightarrow \{I\} \text{ repeat } S \text{ until } B \text{ endrepeat } \{P\}$$

Diese Beweisregel ist ein Sonderfall der Beweisregel R1, wo $V=I$, was in der Praxis oft vorkommt.

3.2.3 Zerlegung von Vor- und Nachbedingungen

Manchmal entstehen bei der Korrektheitsbeweissführung lange Boolesche Ausdrücke. Mit Hilfe der folgenden Beweisregeln können solche langen Ausdrücke zerlegt werden, um die Beweisführung übersichtlicher zu gestalten.

$$\text{Beweisregel DC1: } \{V1\}S\{P1\} \wedge \{V2\}S\{P2\} \Rightarrow \{V1 \wedge V2\}S\{P1 \wedge P2\}$$

$$\text{Beweisregel DC2: } \{V1\}S\{P1\} \wedge \{V2\}S\{P2\} \Rightarrow \{V1 \vee V2\}S\{P1 \vee P2\}$$

$$\text{Beweisregel DC3: } \{V\}S\{P1\} \wedge \{V\}S\{P2\} \Rightarrow \{V\}S\{P1 \wedge P2\}$$

$$\text{Beweisregel DC4: } \{V1\}S\{P\} \wedge \{V2\}S\{P\} \Rightarrow \{V1 \vee V2\}S\{P\}$$

$$\text{Beweisregel DC5: } \{V\}S\{P1\} \wedge \{V\}S\{P2\} \Rightarrow \{V\}S\{P1 \vee P2\}$$

$$\text{Beweisregel DC6: } \{V\}S\{P1\} \vee \{V\}S\{P2\} \Rightarrow \{V\}S\{P1 \vee P2\}$$

Die Hypothesen der Beweisregeln DC1, DC2, DC5 und DC6 sind hinreichende, aber keine notwendigen Bedingungen. Bei der Anwendung dieser Beweisregeln kommt es im allgemeinen deshalb auf die spezifische Aufteilung der Vorbedingung in $V1$ und $V2$ bzw. der Nachbedingung in $P1$ und $P2$ an, ob der Beweis gelingt oder nicht.

Im Gegensatz dazu sind die Hypothesen der Beweisregeln DC3 und DC4 hinreichende und notwendige Bedingungen, d.h., es gilt, daß $\{V\}S\{P1\} \wedge \{V\}S\{P2\} \iff \{V\}S\{P1 \wedge P2\}$ sowie $\{V1\}S\{P\} \wedge \{V2\}S\{P\} \iff \{V1 \vee V2\}S\{P\}$ (\Leftarrow gilt jeweils wegen Beweisregel B1). Auf die spezifische Aufteilung der Nachbedingung in $P1$ und $P2$ bzw. der Vorbedingung in $V1$ und $V2$ kommt es deshalb nicht an, ob der Beweis gelingen wird oder nicht. U.a. aus diesem Grund sind in der Regel die Beweisregeln DC3 und DC4 die nützlichsten dieser sechs Beweisregeln.

$$\text{Beweisregel DC7: } \{V1\}S\{P1\} \wedge \{V1\}S\{P2\} \wedge \{V2\}S\{P1\} \wedge \{V2\}S\{P2\} \\ \Rightarrow \{V1 \vee V2\}S\{P1 \wedge P2\}$$

Auch diese Kombination der Beweisregeln DC3 und DC4 ist manchmal von praktischem Interesse, weil man damit zwei Beweiserlegungsschritte auf einmal durchführen kann.

Die Beweisregeln DC1 bis DC7 lassen sich auf offensichtliche Weise für beliebig viele Terme sowohl in der Vor- als auch in der Nachbedingung verallgemeinern.

3.2.4 Programmsegment, Unterprogramm

Beweisregel U1: Falls die Variablen, die in der Bedingung B vorkommen, durch die Ausführung des Programmteils S unverändert bleiben, dann gilt, daß $\{B\}S\{B\}$ bzw. $\{B\}$ call Prg $\{B\}$, wo das Unterprogramm Prg aus S besteht. Diese Beweisregel trifft immer für die Nullanweisung zu.

$$\text{Beweisregel U2: } \{B\}S\{B\} \wedge \{V\}S\{P\} \Rightarrow \{V \wedge B\}S\{P \wedge B\} \\ \text{sowie } \{B\}S\{B\} \wedge \{V\}S\{P\} \Rightarrow \{V \vee B\}S\{P \vee B\}$$

Die Vor- und Nachbedingungen V und P zusammen stellen die Spezifikation des Programmteils S bzw. der Schnittstelle zwischen dem übergeordneten Programmsegment und S dar. Die Bedingung B bezieht sich nur auf Variablen, die durch die Ausführung von S nicht verändert werden (vgl. Beweisregel U1). Diese Beweisregel ist ein auf den Aufruf eines Unterprogramms bezogener Sonderfall der Beweisregeln DC1 bzw. DC2 und dient der Verifikation einer Korrektheitsaussage über einen Unterprogrammaufruf oder der Ableitung einer Vorbedingung bezüglich eines Unterprogrammaufrufs. Die Spezifikation des Programmteils S, d.h. die Vor- und Nachbedingungen V und P müssen bereits bekannt sein.

Beweisregel U3: $\{B\}S\{B\} \wedge [V \Rightarrow V1] \wedge \{V1\}S\{P1\} \wedge [P1 \Rightarrow P] \Rightarrow \{V \wedge B\}S\{P \wedge B\}$
sowie
 $\{B\}S\{B\} \wedge [V \Rightarrow V1] \wedge \{V1\}S\{P1\} \wedge [P1 \Rightarrow P] \Rightarrow \{V \vee B\}S\{P \vee B\}$

Diese Beweisregel dient der Verifikation einer Korrektheitsaussage über einen Unterprogrammaufruf oder der Ableitung einer Vorbedingung bezüglich eines Unterprogrammaufrufs in Fällen, wo vom aufrufenden Programmteil eine stärkere Vorbedingung geleistet wird als die vom Unterprogramm benötigte bzw. wo eine schwächere Nachbedingung gefordert wird als die vom Unterprogramm geleistete. Diese Beweisregel ist eine Kombination der Beweisregeln U2 und B1. Wie bei manchen anderen Beweisregeln ist sie nur durch den Bedarf der Anwendungspraxis motiviert. Anhang 4 enthält Beispiele für die Anwendung der Beweisregel U3.

Für die Behandlung von Unterprogrammaufrufen mit formaler Parameterübergabe stehen zwei Möglichkeiten zur Verfügung:

(1) Der Unterprogrammaufruf mit formaler Parameterübergabe wird durch einen Aufruf ohne formaler Parameterübergabe zusammen mit anderen Anweisungen (z.B. Zuweisungen oder Deklarationen) definiert, d.h., der Aufruf mit formaler Parameterübergabe wird unter Verwendung nur der im Abschnitt 3.1.3 definierten Programmanweisungen modelliert. Ein zu beweisendes Programmsegment mit formaler Parameterübergabe wird in das äquivalente Programm ohne formale Parameterübergabe (zumindest gedanklich) übersetzt; die Korrektheitsbeweissführung setzt darauf. Dieses Verfahren hat die Vorteile, daß es einfach ist und den Softwareentwickler zwingt, sich mit dem Parameterübergabemechanismus auseinanderzusetzen, den er verwendet und wovon die Korrektheit seines Programms abhängt. Die eventuell notwendige zusätzliche Detailarbeit kann ein Nachteil sein, obwohl sie nur einmal vom Konstrukteur des Unterprogramms geleistet und nicht für jeden Aufruf wiederholt werden soll (siehe Abschnitt 4.5).

(2) Aufgrund der Definition des Parameterübergabemechanismus (siehe (1) oben) werden Beweisregeln für den Unterprogrammaufruf mit formaler Parameterübergabe zusammengestellt. Von Vorteil ist die Möglichkeit, solche Beweisregeln direkt anwenden zu können, ohne das fragliche Programmsegment umformulieren zu müssen. Nachteilig ist die Tatsache, daß die Beweisregeln ggf. programmiersprachen- bzw. sogar implementierungsspezifisch sind. Derartige in der bereits zitierten Literatur angegebene Beweisregeln sind oft im Detail etwas verzwickelt und weniger leicht in Erinnerung zu behalten als die hier eingeführten allgemeiner gültigen Beweisregeln.

Abschnitt 3.3.4.4 geht auf potentiell problematische Aspekte dieses Themas näher ein. Bezüglich rekursiver Unterprogramme siehe Abschnitt 3.3.4.6 und Anhang 4.

3.2.5 Beweisregeln für strikte, halbstrikte und umfassende Vorbedingungen

Aus den Definitionen (siehe Abschnitt 3.1.5) folgen einige Eigenschaften von Vorbedingungen. Jede Teilmenge einer Vorbedingung (jede stärkere Bedingung) ist eine Vorbedingung. Jede Teilmenge einer strikten Vorbedingung ist eine strikte Vorbedingung. Der Schnitt (die **und**-Verknüpfung) einer gewöhnlichen Vorbedingung mit dem Definitionsbereich des fraglichen Programmteils oder mit einer Teilmenge davon ist eine strikte Vorbedingung. Entsprechendes gilt für halbstrikte Vorbedingungen.

Wird eine umfassende Vorbedingung gestärkt, ist das Ergebnis nicht immer eine umfassende Vorbedingung.

Die mit Hilfe der Beweisregel Z1 ermittelte Vorbedingung ist eine umfassende Vorbedingung: $\{P_A^x\} x:=A \{P\}$ umfassend sowie $\{P_A^x\}$ declare $(x, M, A) \{P\}$ umfassend.

Der Beweis der Beweisregel Z1 basiert nur darauf, daß $x.((x:=A).d)=A.d$ und $y.((x:=A).d)=y.d$ für alle Variablennamen y außer x . Vom spezifischen Definitionsbereich der Zuweisung hängt der Beweis nicht ab. Deshalb ist auch für andere Definitionen der Zuweisung, die zu anderen Definitionsbereichen führen, P_A^x eine umfassende Vorbedingung von P bezüglich der Zuweisung $x:=A$. Falls der Definitionsbereich der Zuweisung die gesamte Menge D ist, ist P_A^x das Urbild von P bezüglich der Zuweisung $x:=A$, die schwächste strikte Vorbedingung, die einzige umfassende Vorbedingung sowie die schwächste Vorbedingung. Ferner ist dann jede Vorbedingung eine strikte Vorbedingung.

Einige strikte und umfassende Versionen der im Abschnitt 3.2.1 bis 3.2.4 vorgestellten Beweisregeln gelten. Sie sind in der Praxis jedoch nur von untergeordneter Bedeutung. Fast immer trennt man den Beweis der partiellen Korrektheit, in dem die Programmlogik eine wesentliche Rolle spielt und in dem nur mit gewöhnlichen Vorbedingungen gearbeitet wird, und den Beweis, daß der fragliche Programm zum Ende ausgeführt wird. Typischerweise hat der letztere Beweis überwiegend Wertebereiche bestimmter Programmvariablen und entsprechende Schranken zum Gegenstand. Effektiv wird an jeder Stelle die gewöhnliche Vorbedingung derart gestärkt, daß sie eine Teilmenge des Definitionsbereichs der unmittelbar darauffolgenden Anweisung ist. Z.B. wird vor einer if-Anweisung oder einer while-Schleife auf diese Weise sichergestellt, daß der Wert der if- bzw. der while-Bedingung definiert ist. Ähnlich wird vor einer Zuweisung $x:=A$ sichergestellt, daß A die Menge " x ".d.

Daß eine Überprüfung der Ausführbarkeit jeder einzelnen atomaren Anweisung sowie der Auswertbarkeit jeder if- und while-Bedingung ausreicht, um die Ausführbarkeit des ganzen Programmsegments zu verifizieren, geht formal aus den folgenden Beweisregeln hervor. Diese Beweisregeln sind Kombinationen der Definition einer strikten Vorbedingung und dem Definitionsbereich der jeweiligen Programmanweisung nach dem Muster $\{V\}S\{P\} \wedge V \subseteq S^{-1}.D \Rightarrow \{V\}S\{P\}$ strikt.

Beweisregel ZS: $\{V\} x:=A \{P\} \wedge [V \Rightarrow A \in \text{Menge. "x"}] \Rightarrow \{V\} x:=A \{P\}$ strikt

Beweisregel DS: $\{V\}$ declare $(x, M, A) \{P\} \wedge [V \Rightarrow A \in M] \Rightarrow \{V\}$ declare $(x, M, A) \{P\}$ strikt

Beweisregel RS: $\{V\}$ release $x \{P\} \wedge [V \Rightarrow \text{Menge. "x"} \neq \emptyset] \Rightarrow \{V\}$ release $x \{P\}$ strikt

Beweisregel NS: $\{V\}$ null $\{P\} \Rightarrow \{V\}$ null $\{P\}$ strikt

Beweisregel IFS: $[V \Rightarrow B \in \{\text{falsch, wahr}\}] \wedge \{V \wedge B\}S1\{P\}$ (halb)strikt $\wedge \{V \wedge \neg B\}S2\{P\}$ (halb)strikt $\Rightarrow \{V\}$ if B then $S1$ else $S2$ endif $\{P\}$ (halb)strikt

Die Schreibweise "(halb)strikt" bedeutet, daß die Beweisregel in zwei Versionen gilt: man darf entweder überall "halbstrikt" oder überall "strikt" für "(halb)strikt" einsetzen.

Beweisregel FS: $\{V\}S1\{P1\}$ (halb)strikt $\wedge \{P1\}S2\{P\}$ (halb)strikt $\Rightarrow \{V\} S1; S2 \{P\}$ (halb)strikt

Beweisregel WS: $[I \Rightarrow B \in \{\text{falsch, wahr}\}] \wedge \{I \wedge B\}S\{I\}$ (halb)strikt $\Rightarrow \{I\}$ while B do S endwhile $\{I \wedge \neg B\}$ halbstrikt

Um zu beweisen, daß $\{I\}$ while B do S endwhile $\{I \wedge \neg B\}$ strikt gilt, muß darüber hinaus gezeigt werden, daß (1) $\{I \wedge B\} S \{I\}$ strikt gilt (d.h., daß S keine unendliche Schleife enthält) und daß (2) die Schleife terminiert, d.h., daß die Bedingung B nach endlich vielen Ausführungen von S falsch wird.

Beweisregel US: $\{V\} S \{P\}$ strikt $\Rightarrow \{V\}$ call Prg $\{P\}$ strikt
wo Prg aus S besteht.

Diese Beweisregeln bedeuten, daß die Beweisführung der vollständigen Korrektheit eines Programms in der Praxis in die folgenden drei Schritten unterteilt werden darf. Um zu beweisen, daß $\{V\} S \{P\}$ strikt gilt, beweist man

- (1) daß $\{V\} S \{P\}$ gilt, d.h., daß V eine gewöhnliche Vorbedingung von P bezüglich S ist,
- (2) daß die Vorbedingung jeder atomaren Anweisung in S eine strikte Vorbedingung ist (d.h. daß jede atomare Anweisung ausgeführt wird) und daß der Wert jeder if- und jeder while-Bedingung definiert ist (d.h. daß aus der jeweiligen Vorbedingung $B \in \{\text{falsch, wahr}\}$ folgt, vgl. die Beweisregeln IFS und WS oben) und
- (3) daß der Schleifenkern jeder im Programm S befindlichen Schleife nur endlich viele Male ausgeführt wird.

In Schritt (1) wird die partielle Korrektheit von S bewiesen. Aus den Schritten (1) und (2) zusammen folgt, daß V eine halbstrikte Vorbedingung ist, d.h., daß die Ausführung des Programms S definiert (aber eventuell eine unendlich lange Ausführungsgeschichte) ist. Der dritte Schritt schließt die Möglichkeit einer unendlich langen Ausführungsgeschichte aus, d.h. stellt sicher, daß die Vorbedingung V eine strikte ist und folglich daß S vollständig korrekt ist für die gegebenen Vor- und Nachbedingungen V bzw. P.

Die oben angegebene Aufteilung der Beweisführung ist in der Praxis zweckmäßig, weil sich unterschiedliche Beweismethoden für die drei Schritten eignen. Im Schritt (1) geht es überwiegend um die Zerlegung der Korrektheitsaussage und die Verifikation der daraus entstehenden Implikationen. Im Schritt (2) wird überwiegend überprüft, daß Variablen, zu denen Werte zugewiesen werden, deklariert sind und daß Werte von Ausdrücken aus bestimmten Mengen sind (vgl. die Beweisregeln ZS, DS und RS oben). Im Schritt (3) geht es um die Bestimmung einer geeigneten *Schleifenvariante* (ein Ausdruck, dessen Wert jede Ausführung des Schleifenkerns verringert bzw. erhöht) und um die Überprüfung, daß es für sie eine geeignete Schranke gibt bzw. daß ihre Wertfolge im Verlaufe der Ausführung der Schleife aus einem bestimmten Wertebereich austritt.

3.3 Korrektheitsbeweissführung für sequentielle Programme

3.3.1 Voraussetzungen

Die wesentlichste Voraussetzung für die Programmkorrektheitsbeweissführung ist natürlich der zu beweisende Satz — die zu verifizierende Korrektheitsaussage. Diese besteht aus der Vorbedingung, der Nachbedingung und dem fraglichen Programmsegment. Die Vor- und Nachbedingungen zusammen stellen eine Spezifikation des Programmsegments sowie der Schnittstelle zwischen ihm und seiner Umgebung dar.

Die Spezifikation eines Programmsegments kann aus mehreren Korrektheitsaussagen, d.h. aus mehreren Paaren von Vor- und Nachbedingungen, bestehen, z.B. einer funktionalen und einer sicherheitsbezogenen Korrektheitsaussage. In anderen Fällen wird sich z.B. eine Korrektheitsaussage auf den "Normalfall" und eine andere auf "Fehlerfälle" beziehen. Aus der Sicht des Konstrukteurs sind solche "Fehler" Situationen, die gemäß der Spezifikation vom zu konstruierenden Programm zu behandeln sind, genau wie die "normalen" Situationen. Das bedeutet wiederum, daß die Spezifikation alle Fehlerarten (nämlich Datenfehler, Eingabefehler und alle anderen in Betracht zu ziehenden bzw. zu behandelnden Fehler) sowie sogenannte "Ausnahmen" (engl. "exceptions") berücksichtigen muß.

Auch zur Spezifikation gehören ggf. "Dateninvarianten". Dateninvarianten sind Bedingungen, die sich auf bestimmte Datengruppen, z.B. Dateien, beziehen. Meist stellen sie Konsistenzbedingungen bzw. -anforderungen dar. Dateninvarianten sind als Bestandteile der Vor- und Nachbedingungen aller Programmsegmente, die die fraglichen Daten ansprechen, zu betrachten und zu behandeln.

Vor- und Nachbedingungen beziehen sich überwiegend auf die Werte der relevanten Programmvariablen vor bzw. nach Ausführung des fraglichen Programmsegments. Auch benötigt, vor allem für die Sicherstellung der vollständigen Korrektheit, sind Aussagen über die Struktur der Datenumgebungen, insbesondere über die darin enthaltenen (vereinbarten) Variablen und die ihnen zugeordneten Mengen (die "Typen" der Variablen). Die Anwendung der Beweisregeln U1 bis U3 setzt eine Aussage darüber voraus, welche Variablen vom Programmsegment nicht verändert werden (vgl. die Bedingung B in den Beweisregeln U1, U2 und U3, siehe Abschnitt 3.2.4). Diese Aussage wird zweckmäßigerweise in der Form einer vollständigen Liste der Variablen, die vom Programmsegment verändert werden können bzw. dürfen, angegeben.

Für jede Schleife wird eine geeignete Schleifeninvariante benötigt (vgl. die Beweisregeln W1, W2, R1 und R2). Diese sollte ein Nebenprodukt der Programmkonstruktion sein; fehlt die Schleifeninvariante, dann muß der entsprechende Konstruktionsschritt nach- bzw. wiederholt werden.

Für jedes Unterprogramm, das das Programmsegment aufruft, dessen Korrektheit zu beweisen ist, werden eine Vor- und eine Nachbedingung benötigt. Der Beweis der Korrektheitsaussage über das Unterprogramm ist eine zur Entwicklung des Unterprogramms gehörende Aufgabe. Die Korrektheitsaussage über das Unterprogramm wird im Beweis der Korrektheit des aufrufenden Programmsegments als bereits bewiesener Satz angenommen und verwendet. Im Korrektheitsbeweis eines rekursiven Unterprogramms tritt diese Annahme im Induktionsschritt auf; hinsichtlich rekursiver Unterprogramme siehe Abschnitt 3.3.4.6 und Anhang 4.

Nützlich sind Zwischenbedingungen (auch Zusicherungen, engl. "assertions", genannt) an verschiedenen Stellen im fraglichen Programmsegment (vgl. die Bedingung P1 in der Beweisregel F1). Fehlende Zwischenbedingungen müssen bei der Korrektheitsbeweissführung abgeleitet werden. Vorhandene Zwischenbedingungen, die die Absichten des Programmkonstrukteurs ausdrücken, fördern und begrenzen die Stärkung von Vorbedingungen (vgl. die Beweisregel B1). Dadurch können algebraische Ausdrücke vereinfacht werden, ohne daß eine übermäßige Stärkung von Vorbedingungen zum Scheitern der Beweisführung an einer früheren Stelle im Programm führt.

"Werkzeuge" zur Unterstützung der Programmkorrektheitsbeweissführung werden nicht selten als Voraussetzung für die Anwendung dieser Techniken gefordert. Wie bereits er-

wähnt, wird hier die These vertreten, daß geeignete "Werkzeuge" durchaus hilfreich sein könnten, jedoch nicht erforderlich sind.

3.3.2 Vorgehensweise ohne maschinelle Unterstützung

Prinzipiell können in Zusammenhang mit der Korrektheitsbeweissführung vier Arten von Aufgaben vorkommen:

- (1) $\{V\} S \{P\} ?$ (Verifikation einer vorgegebenen Korrektheitsaussage)
- (2) $\{V?\} S \{P\}$ (Ableitung einer Vorbedingung)
- (3) $\{V\} S? \{P\}$ (Konstruktion eines Programmsegments)
- (4) $\{V\} S \{P?\}$ (Ableitung einer Nachbedingung)

Das eigentliche Ziel der Programmkorrektheitsbeweissführung im engeren Sinne ist die erste Aufgabe, die Verifikation einer vorgegebenen Korrektheitsaussage. Bei der Erarbeitung eines Beweises müssen in gewissen Fällen Vorbedingungen abgeleitet werden, also es treten dabei Aufgaben der zweiten Art auf. Gegenstand dieses Abschnitts sind deshalb die ersten zwei oben aufgeführten Beweisaufgaben.

Die dritte Aufgabenart hat die Konstruktion eines Programmsegments zum Gegenstand. Dieses Thema wird im Kapitel 4 behandelt.

Die vierte Aufgabenart hat zum Gegenstand die Ermittlung einer Nachbedingung für eine gegebene Vorbedingung und ein gegebenes Programmsegment, d.h. die Feststellung, was ein gegebenes Programmsegment macht bzw. errechnet. Diese analytische Aufgabe sollte in der ingenieurmäßigen Entwicklungspraxis selten oder nie vorkommen. Sie kommt bei der Anwendung der im Abschnitt 3.2 vorgestellten Beweisregeln auf Verifikationsaufgaben auch nicht vor. Deshalb wird diese Aufgabenart hier nicht behandelt. Die bereits zitierte Fachliteratur enthält Beweisregeln, die der Ermittlung einer Nachbedingung dienen.

Beweisaufgaben der ersten und zweiten oben aufgeführten Arten werden durch die Anwendung geeigneter Beweisregeln in Beweisaufgaben dieser Arten über kleinere Programmteile iterativ zerlegt, bis nur Korrektheitsaussagen über atomare Anweisungen übrig bleiben. Diese werden, auch durch die Anwendung geeigneter Beweisregeln, in rein algebraische Aufgaben zerlegt, die schließlich gelöst werden müssen.

Bei der Programmkorrektheitsbeweissführung müssen diese Beweisaufgaben für die im Abschnitt 3.1.3 definierten Programmanweisungen und Zusammensetzungen davon gelöst werden.

Wie bereits erwähnt kommen in Korrektheitsätzen verschiedene Arten von Aussagen vor. Vor- und Nachbedingungen beziehen sich auf

- die Werte von Programmvariablen,
- die Mengen, die den Programmvariablen zugeordnet sind, (die "Typen" der Variablen) und
- die Struktur der fraglichen Datenumgebungen, insbesondere darauf, welche Programmvariablen darin enthalten (vereinbart) sind.

Bei Aussagen über die Werte von Programmvariablen wendet man die im Abschnitt 3.2 vorgestellten Beweisregeln an, um die jeweilige Beweisaufgabe zu zerlegen (siehe die Tabelle unten). Bei Aussagen über die Struktur von Datenumgebungen wendet man die Definitionen der fraglichen Programmanweisungen unmittelbar an. Bei Aussagen über die den Programmvariablen zugeordneten Mengen wendet man bei declare- und release-An-

weisungen ihre Definitionen, bei anderen Anweisungsarten die Beweisregeln (siehe die Tabelle unten) an.

Die folgende Tabelle gibt für jede Anweisungsart und Beweisaufgabe an, welche Beweisregeln anzuwenden sind, um die jeweilige Beweisaufgabe zu zerlegen. Ferner weist sie auf die weiteren Beweisaufgaben, die nach jedem Zerlegungsschritt noch zu erledigen sind. Der Doppelstrich in der Mitte der Tabelle trennt die atomaren Programmanweisungen von den Zusammensetzungen von Anweisungen.

Beweisaufgabe	Regel	Zerlegungsergebnisse
$\{V\} x := A \{P\} ?$	Z2	$V \Rightarrow P_A^x ?$
$\{V?\} x := A \{P\}$	Z1	$V = P_A^x$
$\{V\} \text{ declare } (x, M, A) \{P\} ?$	Z2	$V \Rightarrow P_A^x ?$
$\{V?\} \text{ declare } (x, M, A) \{P\}$	Z1	$V = P_A^x$
$\{V\} \text{ release } x \{P\} ?$	U1, B1	$x \text{ nicht in } P ?$ $V \Rightarrow P ?$
$\{V?\} \text{ release } x \{P\}$	U1	$x \text{ nicht in } P ?$ $V = P$
$\{V\} \text{ null } \{P\} ?$	U1, B1	$V \Rightarrow P ?$
$\{V?\} \text{ null } \{P\}$	U1	$V = P$
$\{V\} \text{ if } B \text{ then } S1 \text{ else } S2 \text{ endif } \{P\} ?$	IF1	$\{V \wedge B\} S1 \{P\} ?$ $\{V \wedge \neg B\} S2 \{P\} ?$
$\{V?\} \text{ if } B \text{ then } S1 \text{ else } S2 \text{ endif } \{P\}$	IF2	$\{V1?\} S1 \{P\}$ $\{V2?\} S2 \{P\}$ $V = V1 \wedge B \vee V2 \wedge \neg B$
$\{V\} S1 \{V2\} S2 \{P\} ?$ (Zwischenbedingung vorgegeben)	F1	$\{V2\} S2 \{P\} ?$ $\{V\} S1 \{V2\} ?$
$\{V?\} S1 \{V2\} S2 \{P\}$ (Zwischenbedingung vorgegeben)	F1	$\{V2\} S2 \{P\} ?$ $\{V?\} S1 \{V2\}$
$\{V\} S1; S2 \{P\} ?$ (Zwischenbedingung nicht vorgegeben)	F1	$\{V2?\} S2 \{P\}$ $\{V\} S1 \{V2\} ?$
$\{V?\} S1; S2 \{P\}$ (Zwischenbedingung nicht vorgegeben)	F1	$\{V2?\} S2 \{P\}$ $\{V?\} S1 \{V2\}$
$\{V\} \text{ while } B \text{ do } S \text{ invariant } I \text{ endwhile } \{P\} ?$	W1, B1	$V \Rightarrow I ?$ $\{I \wedge B\} S \{I\} ?$ $I \wedge \neg B \Rightarrow P ?$
$\{V?\} \text{ while } B \text{ do } S \text{ invariant } I \text{ endwhile } \{P\}$	W1, B1	$\{I \wedge B\} S \{I\} ?$ $I \wedge \neg B \Rightarrow P ?$ $V = I$

$\{V\}$ Init; while B do S invariant I endwhile $\{P\}$?	W2	$\{V\}$ Init $\{I\}$? $\{I \wedge B\}$ S $\{I\}$? $I \wedge \neg B \Rightarrow P$?
$\{V?\}$ Init; while B do S invariant I endwhile $\{P\}$	W2	$\{I \wedge B\}$ S $\{I\}$? $I \wedge \neg B \Rightarrow P$? $\{V?\}$ Init $\{I\}$
$\{V \wedge B\}$ call Prg $\{P \wedge B\}$?	U3	$\{B\}$ call Prg $\{B\}$? $V \Rightarrow V1$? $\{V1\}$ call Prg $\{P1\}$? $P1 \Rightarrow P$?
$\{V? \wedge B\}$ call Prg $\{P \wedge B\}$	U3	$\{B\}$ call Prg $\{B\}$? $\{V1\}$ call Prg $\{P1\}$? $P1 \Rightarrow P$? $V = V1$

Der iterative Zerlegungsprozeß wird immer enden, denn jede zerlegte Beweisaufgabe ist entweder eine Beweisaufgabe der ersten oder zweiten Art (siehe oben) über ein kleineres Programmsegment oder eine nicht weiter zu zerlegende Aufgabe: eine Aufgabe der Booleschen Algebra (die Verifikation einer logischen Implikation oder die Bildung eines Ausdrucks) oder die Prüfung, ob ein gegebener Ausdruck Bezug auf eine bestimmte Programmvariable nimmt oder nicht. Der Beweiszerlegungsprozeß endet folglich mit einer Sammlung von noch zu verifizierenden logischen Implikationen ("verification conditions").

Die aus dem Zerlegungsprozeß entstandenen zu verifizierenden Implikationen können oft in kürzere Ausdrücke weiter zerlegt werden. Z.B. kann eine Implikation der Form $X \Rightarrow Y \wedge Z$ in die Lemmata $X \Rightarrow Y$ und $X \Rightarrow Z$ zerlegt werden, weil $(X \Rightarrow Y \wedge Z) \iff (X \Rightarrow Y) \wedge (X \Rightarrow Z)$.

Bei der Anwendung der Beweisregeln für Schleifen (siehe Beweisregeln W1, W2, R1 und R2) muß die Schleifeninvariante bereits bekannt sein. Ist das nicht der Fall, dann muß eine geeignete Schleifeninvariante bestimmt werden, z.B. durch Verallgemeinerung der Anfangs- und Endsituationen (etwa Vor- und Nachbedingungen), siehe die Bemerkung nach der Beweisregel W1 im Abschnitt 3.2.2 oben.

3.3.3 Anwendungsbeispiele

Als Beispiel eines Korrektheitsbeweises betrachte ein Programmsegment S, das ein Feld $D(1), \dots, D(n)$ nach einem bestimmten Wert K absucht und die Indexwerte der gleichen und ungleichen Feldelemente in getrennten Teilen eines anderen Felds $L(1), \dots, L(n)$ festhält.

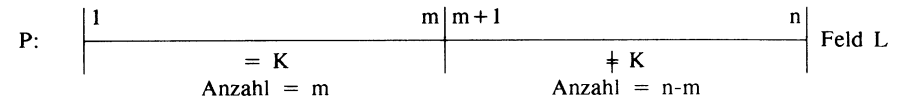
Die Vor- und Nachbedingungen V bzw. P sind:

$V: n \in \mathbb{Z}$ und $0 \leq n$ [n vereinbart, Wertebereich von n]
 $\text{und}_{i=1}^n K \in \text{Menge. "D(i)"} \text{ und } \{1, \dots, n\} \subseteq \text{Menge. "L(i)"} \\ [K, D(\cdot), L(\cdot) \text{ geeignet vereinbart}]$

$P: m \in \mathbb{Z}$ und $0 \leq m \leq n$ [Wertebereich von m]
 $\text{und}_{i=1}^m K = D(L(i))$ [L(1), ... L(m) zeigen auf gleiche Werte in D]
 $\text{und}_{i=m+1}^n K \neq D(L(i))$ [L(m+1), ... L(n) zeigen auf ungleiche Werte in D]
 $\text{und } ((\&_{i=1}^n [L(i)]) \text{ Perm } (\&_{i=1}^n [i]))$ [L ist eine Permutation der Ganzzahlen 1 bis n]

Die letzte Zeile in P stellt sicher, daß das Feld L genau einen Bezug auf jede Variable im Feld D enthält. Die Reihenform der Folgenverknüpfungsfunktion & oben entspricht der bekannten \sum -Schreibweise für Summen, vgl. Abschnitt 3.4.2. Der Infixoperator "Perm" (Permutation) bildet zwei Folgen in einen Booleschen Wert ab.

Wesentliche Aspekte der Nachbedingung P veranschaulicht das folgende Diagramm:



Die **Korrektheitsaussage** lautet:

- (1) $\{V\}$ S $\{P\}$ strikt und
 - (2) S.d = [(m, Z, .) & d, bis auf die Werte von m, L(1), ... L(n)]
- D.h., das Programmsegment S vereinbart die neue Programmvariable m, weist ihr und den Variablen L(1), ... L(n) Werte zu, die die Nachbedingung erfüllen, und terminiert, falls die Vorbedingung V anfangs erfüllt ist. Keine andere Variable wird durch die Ausführung von S verändert.

Das Programmsegment S ist wie folgt:

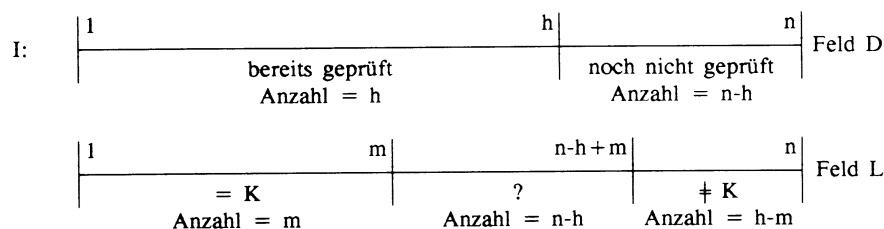
```

declare (m, Z, 0)
declare (h, Z, 0)
while h < n do
  h := h + 1
  if K = D(h) then m := m + 1; L(m) := h else L(n - h + m + 1) := h endif
invariant I
endwhile
release h
    
```

wobei die Schleifeninvariante I vom Programmkonstrukteur vorgegeben wurde:

$I: m \in \mathbb{Z}$ und $h \in \mathbb{Z}$ und $n \in \mathbb{Z}$ und $0 \leq m \leq h \leq n$ [Wertebereiche von m, h und n]
 $\text{und}_{i=1}^m K = D(L(i))$ [L(.) zeigen auf gleiche Werte]
 $\text{und}_{i=n-h+m+1}^n K \neq D(L(i))$ [L(.) zeigen auf ungleiche Werte]
 $\text{und } ((\&_{i=1}^m [L(i)] \&_{i=n-h+m+1}^n [L(i)]) \text{ Perm } (\&_{i=1}^h [i]))$ [Teil von L Permutation von 1 bis h]

Wesentliche Aspekte der Schleifeninvariante I veranschaulicht das folgende Diagramm:



Man merke, daß keine Anweisung im Programmsegment die Wahrheit der Vorbedingung V beeinflusst. Deshalb gilt V an jeder Stelle in S (d.h., V ist eine *Programminvariante*: eine an mehreren bestimmten oder an allen Stellen des Programms wahre Bedingung).

Der umfangreichste Teil des Beweises der Korrektheitsaussage beschäftigt sich mit der Aussage $\{V\} S \{P\}$, d.h., daß V eine gewöhnliche Vorbedingung von P bezüglich S ist. Eine Übersicht über den Beweis vermittelt der mit den Zwischenbedingungen ergänzte Programmtext:

```
{V}
declare (m, Z, 0); declare (h, Z, 0)
{I}
while h<n do
  {I und h<n}
  h:=h+1
  if K=D(h) then m:=m+1; L(m):=h else L(n-h+m+1):=h endif
  {1≤h und I}
  {I}
endwhile
{P}
release h
{P}
```

Anfangs ist der Wert von h Null. Der Schleifenkern erhöht den Wert von h um 1. Deshalb wird nach der ersten Ausführung des Schleifenkerns $1 \leq h$ gelten. Folglich darf man die Nachbedingung des Schleifenkerns entsprechend stärken (vgl. Beweisregel B1), ohne das Gelingen des Beweises zu gefährden. Diese Möglichkeit tritt bei jeder Schleife auf, in deren Schleifenkern der Wert der Schleifenvariable um einen festen Betrag erhöht oder verringert wird. Oft führt eine derartige Stärkung der Nachbedingung zu einer gewissen Vereinfachung der algebraischen Manipulationen im Korrektheitsbeweis des Schleifenkerns.

Beweis für $\{V\} S \{P\}$: Gemäß Beweisregel F1 gilt $\{V\} S \{P\}$, falls die folgenden zwei Aussagen erfüllt sind:

```
{V}
declare (m, Z, 0); declare (h, Z, 0)
{I}
while ... endwhile
{P} [1]
```

```
{P}
release h
{P} [2]
```

Gemäß Beweisregel U1 gilt die Aussage [2], weil die Bedingung P keinen Bezug auf die Variable h enthält.

Gemäß Beweisregel W2 für die while-Schleife mit Initialisierung sowie Beweisregel B1 gilt die Aussage [1], falls die folgenden vier Aussagen wahr sind:

```
{V} declare (m, Z, 0); declare (h, Z, 0) {I} [3]
```

```
{I und h<n} [4]
```

```
h:=h+1
```

```
if K=D(h) then m:=m+1; L(m):=h else L(n-h+m+1):=h endif
```

```
{1≤h und I}
```

```
1≤h und I ⇒ I [5]
```

```
I und nicht h<n ⇒ P [6]
```

wobei die Aussage [5] offensichtlich gilt.

Gemäß Beweisregeln F1, Z1 und Z2 gilt die Aussage [3], falls

```
V ⇒ (I0h)0m [7]
```

Gemäß Beweisregel F1 gilt die Aussage [4], falls die folgenden zwei Aussagen gelten:

```
{I und h<n} [8]
```

```
h:=h+1
```

```
{V1}
```

```
{V1} [9]
```

```
if K=D(h) then m:=m+1; L(m):=h else L(n-h+m+1):=h endif
```

```
{1≤h und I}
```

wobei die Vorbedingung $V1$ der if-Anweisung noch abgeleitet werden muß.

Gemäß Beweisregel Z2 gilt die Aussage [8], falls

```
I und h<n ⇒ V1h+1h [10]
```

Gemäß Beweisregeln IF2 und B1 gilt (vgl. Aussage [9]), daß

```
V1 = (K=D(h) und V1T oder K≠D(h) und V1E) [9a]
```

falls

```
{V1T} m:=m+1; L(m):=h {1≤m und 1≤h und I} [11]
```

```
1≤m und 1≤h und I ⇒ 1≤h und I [gilt offensichtlich]
```

```
{V1E} L(n-h+m+1):=h {m<h und 1≤h und I} [12]
```

```
m<h und 1≤h und I ⇒ 1≤h und I [gilt offensichtlich]
```

wobei V1T und V1E noch abgeleitet werden müssen. Hinsichtlich der Stärkung der jeweiligen Nachbedingung (d.h. des Hinzufügens der Terme $1 \leq m$ bzw. $m < h$) vgl. die Bemerkung über die Stärkung der Nachbedingung des Schleifenkerns oben.

Gemäß Beweisregeln F1 und Z1 gilt (vgl. Aussage [11]), daß

$$V1T = ((1 \leq m \text{ und } 1 \leq h \text{ und } I)^{L(m)}_h)_{m+1}^m$$

Aber

$$(1 \leq m \text{ und } 1 \leq h \text{ und } I)^{L(m)}_h$$

=
 $1 \leq m \text{ und } 1 \leq h \text{ und } I^{L(m)}_h$
 = [L(m) durch h in I ersetzen, Gesamtausdruck vereinfachen, siehe unten]

$$m \in \mathbb{Z} \text{ und } h \in \mathbb{Z} \text{ und } n \in \mathbb{Z} \text{ und } 1 \leq m \leq h \leq n$$

$$\text{und}_{i=1}^{m-1} K = D(L(i)) \text{ und } K = D(h) \text{ und}_{i=n-h+m+1}^n K \neq D(L(i))$$

$$\text{und } ((\&_{i=1}^{m-1} [L(i)] \& [h] \&_{i=n-h+m+1}^n [L(i)]) \text{ Perm } (\&_{i=1}^{h-1} [i] \& [h]))$$

=
 [[h] aus beiden Folgen herausnehmen,
 Axiom: $(a \& b \& c \text{ Perm } d \& b \& e) = (a \& c \text{ Perm } d \& e)$]

$$m \in \mathbb{Z} \text{ und } h \in \mathbb{Z} \text{ und } n \in \mathbb{Z} \text{ und } 1 \leq m \leq h \leq n$$

$$\text{und}_{i=1}^{m-1} K = D(L(i)) \text{ und } K = D(h) \text{ und}_{i=n-h+m+1}^n K \neq D(L(i))$$

$$\text{und } ((\&_{i=1}^{m-1} [L(i)] \&_{i=n-h+m+1}^n [L(i)]) \text{ Perm } (\&_{i=1}^{h-1} [i]))$$

Bei der algebraischen Umformung oben mußte jeweils der in der **und**-Reihe vorkommende Term, der sich auf L(m) bezog, von den anderen getrennt werden, damit L(m) durch h ersetzt werden konnte. Auf diese potentielle Falle bei Bezugnahme auf eine Feldvariable, der durch die Ausführung einer Zuweisung ggf. ein neuer Wert zugeordnet wird, wird im Abschnitt 3.3.4.2 näher eingegangen.

Ersetzt man m durch m+1 im Ausdruck oben, erhält man

$$V1T$$

=
 $m \in \mathbb{Z} \text{ und } h \in \mathbb{Z} \text{ und } n \in \mathbb{Z} \text{ und } 1 \leq m+1 \leq h \leq n$

$$\text{und}_{i=1}^m K = D(L(i)) \text{ und } K = D(h) \text{ und}_{i=n-h+m+2}^n K \neq D(L(i))$$

$$\text{und } ((\&_{i=1}^m [L(i)] \&_{i=n-h+m+2}^n [L(i)]) \text{ Perm } (\&_{i=1}^{h-1} [i]))$$

=
 $m \in \mathbb{Z} \text{ und } h \in \mathbb{Z} \text{ und } n \in \mathbb{Z} \text{ und } 0 \leq m < h \leq n$

$$\text{und}_{i=1}^m K = D(L(i)) \text{ und } K = D(h) \text{ und}_{i=n-h+m+2}^n K \neq D(L(i))$$

$$\text{und } ((\&_{i=1}^m [L(i)] \&_{i=n-h+m+2}^n [L(i)]) \text{ Perm } (\&_{i=1}^{h-1} [i]))$$

Gemäß Beweisregel Z1 gilt (vgl. Aussage [12]), daß

$$V1E$$

$$=$$

$$(m < h \text{ und } 1 \leq h \text{ und } I)^{L(n-h+m+1)}_h$$

$$=$$

$$m \in \mathbb{Z} \text{ und } h \in \mathbb{Z} \text{ und } n \in \mathbb{Z} \text{ und } 0 \leq m < h \leq n$$

$$\text{und}_{i=1}^m K = D(L(i)) \text{ und } K \neq D(h) \text{ und}_{i=n-h+m+2}^n K \neq D(L(i))$$

$$\text{und } ((\&_{i=1}^m [L(i)] \&_{i=n-h+m+2}^n [L(i)]) \text{ Perm } (\&_{i=1}^{h-1} [i]))$$

Hier mußte jeweils der Term, der sich auf L(n-h+m+1) bezog, von den anderen getrennt werden, damit L(n-h+m+1) durch h ersetzt werden konnte. Vgl. die Ableitung von V1T oben sowie Abschnitt 3.3.4.2.

Kombiniert man V1T und V1E (siehe Aussagen [9] und [9a]), erhält man für die Vorbedingung V1 bezüglich der if-Anweisung

$$V1$$

=
 $m \in \mathbb{Z} \text{ und } h \in \mathbb{Z} \text{ und } n \in \mathbb{Z} \text{ und } 0 \leq m < h \leq n$

$$\text{und}_{i=1}^m K = D(L(i)) \text{ und}_{i=n-h+m+2}^n K \neq D(L(i))$$

$$\text{und } ((\&_{i=1}^m [L(i)] \&_{i=n-h+m+2}^n [L(i)]) \text{ Perm } (\&_{i=1}^{h-1} [i]))$$

Jetzt bleiben nur noch Boolesche Ausdrücke (Implikationen), die zu verifizieren sind. Aussage [6] gilt, weil

$$I \text{ und nicht } h < n$$

=
 $I \text{ und } n \leq h$

=
 $m \in \mathbb{Z} \text{ und } h \in \mathbb{Z} \text{ und } n \in \mathbb{Z} \text{ und } 0 \leq m \leq h = n$

$$\text{und}_{i=1}^m K = D(L(i)) \text{ und}_{i=n-h+m+1}^n K \neq D(L(i))$$

$$\text{und } ((\&_{i=1}^m [L(i)] \&_{i=n-h+m+1}^n [L(i)]) \text{ Perm } (\&_{i=1}^h [i]))$$

⇒
 $m \in \mathbb{Z} \text{ und } 0 \leq m \leq n$

$$\text{und}_{i=1}^m K = D(L(i)) \text{ und}_{i=m+1}^n K \neq D(L(i))$$

$$\text{und } ((\&_{i=1}^n [L(i)]) \text{ Perm } (\&_{i=1}^n [i]))$$

=
 P

Aussage [7] gilt, weil

$$(I^h_0)^m_0$$

=
 $n \in \mathbb{Z} \text{ und } 0 \leq n$

⇐
 V

Aussage [10] gilt, weil

$$\begin{aligned}
 & \forall_{h+1}^h \\
 & m \in \mathbb{Z} \text{ und } h \in \mathbb{Z} \text{ und } n \in \mathbb{Z} \text{ und } 0 \leq m < h+1 \leq n \\
 & \text{und } \bigwedge_{i=1}^m K = D(L(i)) \text{ und } \bigwedge_{i=n-h+m+1}^n K \neq D(L(i)) \\
 & \text{und } ((\bigwedge_{i=1}^m [L(i)] \ \& \ \bigwedge_{i=n-h+m+1}^n [L(i)]) \text{ Perm } (\bigwedge_{i=1}^h [i])) \\
 & m \in \mathbb{Z} \text{ und } h \in \mathbb{Z} \text{ und } n \in \mathbb{Z} \text{ und } 0 \leq m \leq h < n \\
 & \text{und } \bigwedge_{i=1}^m K = D(L(i)) \text{ und } \bigwedge_{i=n-h+m+1}^n K \neq D(L(i)) \\
 & \text{und } ((\bigwedge_{i=1}^m [L(i)] \ \& \ \bigwedge_{i=n-h+m+1}^n [L(i)]) \text{ Perm } (\bigwedge_{i=1}^h [i])) \\
 & I \text{ und } h < n
 \end{aligned}$$

Damit ist der Beweis vollständig, daß $\{V\} S \{P\}$. Es muß noch bewiesen werden, daß die Vorbedingung V strikt ist sowie daß Teil (2) der Korrektheitsaussage gilt (siehe die Korrektheitsaussage auf Seite 49). Teil (2) der Korrektheitsaussage kann leicht überprüft werden: Die Struktur der Datenumgebung $S.d$ ergibt sich aus dem Programmtext und den Definitionen der declare- und release-Anweisungen. Abgesehen von der lokalen Variable h , die am Ende des Programmsegments gelöscht wird, ändert S nur die Werte der Variablen m und der Feldvariablen L . Aus den Vorbedingungen der zwei Zuweisungen $L(.) := \dots$ folgt, daß der jeweilige Indexwert immer zwischen 1 und n einschließlich liegt. Man merke, daß das Programmsegment S eine eventuell vorher vereinbarte — in der ursprünglichen Datenumgebung d enthaltene — Variable mit dem Namen "h" nicht verändert. Entsprechend wird ggf. eine vorher vereinbarte Variable mit dem Namen "m" unverändert bleiben; sie wird jedoch durch die vom Programmsegment S neu vereinbarte Variable "m" verdeckt.

Die Tatsache, daß V eine strikte Vorbedingung ist, d.h. daß das Ergebnis der Ausführung jeder Anweisung und des ganzen Programmsegments definiert ist, folgt aus der Wahrheit von V gleich vor jeder atomaren Anweisung und vor jeder Auswertung einer if- oder while-Bedingung, aus der Vereinbarung der lokalen Variable h , aus den in der jeweiligen Vorbedingung enthaltenen Schranken für die in den Indexausdrücken vorkommenden Variablen sowie aus der Terminierung der Schleife. Die Schleife terminiert, weil jede Ausführung des Schleifenkerns den Wert der Variable h um 1 erhöht und n eine obere Schranke für h ist, vgl. die while-Bedingung sowie die Schleifeninvariante I . D.h., der Schleifenkern kann nicht unendlich wiederholt ausgeführt werden, ohne die Schranke zu verletzen. Etwas detaillierter: Sei die Schleifenvariante $SV \triangleq n-h$. Anfangs hat h den Wert 0 und folglich $SV = n$. Jede Ausführung des Schleifenkerns erhöht den Wert von h um 1 und damit verringert sie den Wert von SV um 1. Nach n Ausführungen des Schleifenkerns wird also SV gleich Null sein, woraus folgt, daß die while-Bedingung $((h < n) = (0 < n-h) = (0 < SV))$ falsch sein wird, weshalb die Schleife terminieren wird.

Als Beispiel der formalen Überprüfung, daß die Vorbedingungen strikt sind, betrachte die Zuweisung $L(m) := h$ und ihre Vorbedingung $(1 \leq m \text{ und } 1 \leq h \text{ und } I)_{L(m)}^h$ und V (siehe die Ableitung von VIT oben sowie die Bemerkung, daß V eine Programminvariante ist),

woraus folgt, daß $h \in \mathbb{Z}$ und $1 \leq m \leq h \leq n$ und $\{1, \dots, n\} \subseteq \text{Menge. "L(m)"}$. Daraus folgt, daß $h \in \text{Menge. "L(m)"}$. Gemäß Beweisregel ZS ist die Vorbedingung strikt.

Im Programmsegment und im Beweis tritt an mehreren Stellen die Menge Z aller Ganzzahlen auf. Sie könnte durch eine beliebige Menge ersetzt werden, die die Ganzzahlen $0, 1, \dots, n$ umfaßt, denn die betroffenen Variablen nehmen nur Werte in diesem Bereich an, vgl. die Schleifeninvariante I . Würde man diese Menge auf $\{0, 1, \dots, n\}$ einschränken, dann könnte nur während der Auswertung des Ausdrucks $(n-h+m+1)$ die Berechnung von Zwischenergebnissen u.U. zu einem Überlauf und folglich zu einem undefinierten Ergebnis führen, je nach der Reihenfolge der Ausführung der darin vorkommenden Operationen.

Anhang 2 enthält einen Korrektheitsbeweis für ein anderes Programmsegment und dient darüber hinaus als Beispiel einer Dokumentationsform, die das Überprüfen eines Beweises erleichtern soll. Die im oben geführten Beweis vorgekommene Struktur wird im Anhang 2 durch eine deutlichere und präzisere Gliederung klarer ausgedrückt. Siehe die sehr ähnliche in [Hoffmann; 1990 Okt. 1, Aufgabe 4] vorgegebene Struktur für einen Korrektheitsbeweis. Abschnitt 7.2.3 enthält weitere Korrektheitsbeweise bzw. Skizzen davon. Ein Beispiel eines Korrektheitsbeweises für ein übergeordnetes Programmsegment (ein Steuerprogramm für ein mittelgroßes Management-Planspielsystem) enthält [Baber; Abschnitt 6.10, S. 227 ff. und 300-301, 1987]. [Baber; Abschnitt 4.6, S. 83-87, 1990 (*Fehlerfreie Programmierung*)] veranschaulicht die Behandlung von Unterprogrammaufrufen in einem Korrektheitsbeweis.

3.3.4 Potentielle Fallen

Bei der Korrektheitsbeweisführung können einige Probleme auftreten, wenn gewisse Details außer Acht gelassen werden. Die wichtigsten und diejenigen, die manchmal als Gegenargumente gegen die Korrektheitsbeweisführung angesehen werden, werden in diesem Abschnitt diskutiert. Manche der hier behandelten Fallen machen sich zwar bei der Korrektheitsbeweisführung bemerkbar, sind jedoch auf Ursachen zurückzuführen, die völlig unabhängig davon und in ganz anderen Bereichen zu suchen sind.

3.3.4.1 Herausnehmen eines Terms aus einer Reihe

Bei der Umformung einer Reihe über einen Operator OP will man oft einen Term aus der Reihe herausnehmen, z.B.:

$$OP_{i=a}^b \text{ Term.i} = ((OP_{i=a}^{b-1} \text{ Term.i}) OP \text{ Term.b})$$

Nicht selten übersieht man, daß dieses Verfahren nur dann zulässig ist, wenn es überhaupt einen Term in der Reihe gibt, der herausgenommen werden kann (d.h., wenn $a \leq b$ im Beispiel oben), oder wenn OP Bestandteil einer Gruppe ist und für $a > b$ die Reihe auf geeignete Weise definiert ist.

Die Booleschen Operatoren **und** und **oder** bilden zusammen mit der Menge {falsch, wahr} keine Gruppen, sondern nur Halbgruppen, weil die Kehrwerte nicht existieren. Deshalb darf bei diesen Operatoren die oben geschilderte Umformung nur dann vorgenom-

men werden, wenn $a \leq b$ gilt. Unter Berücksichtigung dieser Einschränkung erhält man die folgenden allgemein gültigen Formeln:

$$\text{und}_{i=a}^b \text{Term}.i = (a > b \text{ oder } (\text{und}_{i=a}^{b-1} \text{Term}.i) \text{ und } \text{Term}.b)$$

$$\text{oder}_{i=a}^b \text{Term}.i = (a \leq b \text{ und } ((\text{oder}_{i=a}^{b-1} \text{Term}.i) \text{ oder } \text{Term}.b))$$

Wenn der Zusammenhang des fraglichen Teilausdrucks nicht sicherstellt, daß $a \leq b$ gilt, dann muß man die entsprechende Formel oben (oder eine äquivalente) anwenden, um einen Term aus einer Reihe herauszunehmen. Läßt man in einem solchen Fall " $a > b$ oder" bzw. " $a \leq b$ und" weg, dann wird der ermittelte Ausdruck im allgemeinen fehlerhaft sein. Diesen Fehler machen viele Anfänger bei der Korrektheitsbeweissführung.

3.3.4.2 Zuweisung zu einer Feldvariable

In Bezügen auf eine Feldvariable, sowohl in Nachbedingungen als auch im Programmtext, wird der Index häufig in der Form eines Ausdrucks, nicht in der Form einer Konstante, angegeben. Im wirklichen Namen selbst ist der Index jedoch eine Konstante, z.B. $x(2)$, $y(3)$. Die unterschiedlichen Formen können leicht zu Verwirrung und Fehlern bei der Anwendung der Beweisregeln Z1 und Z2 führen, wenn in einer Zuweisung einer Feldvariable ein neuer Wert zugewiesen wird.

Als Beispiel der potentiellen Problematik betrachte die Beweisaufgabe $\{V \text{ umf. } ?\} x(i) := 1 \{x(i) = x(j)\}$. Die bedachtlose "Lösung" wäre $\{1 = x(j)\}$, die jedoch keine umfassende Vorbedingung ist. Die bedachtlose "Lösung" zur anderen Beweisaufgabe $\{V ?\} x(i) := 1 \{x(j) = 5\}$ wäre $\{x(j) = 5\}$, die nicht einmal eine gewöhnliche Vorbedingung ist (Gegenbeispiel ist $i = j$). Das Problem stammt daher, daß " $x(i)$ " als der zu ersetzende Name betrachtet wird, die Feldvariablenamen jedoch die Form $x(1)$, $x(2)$ usw. aufweisen. Zu ersetzen sind alle in der Nachbedingung vorkommenden *Bezüge* auf $x(i)$, seien sie in der Form $x(i)$, $x(j)$ mit $i = j$, $x(3)$ mit $i = 3$ usw. Auf irgendeine Weise muß in den Beispielen oben zwischen den Fällen $i = j$ und $i \neq j$ unterschieden werden.

Die erforderliche Fallunterscheidung kann auf verschiedene Weise erfolgen, siehe z.B. [Baber; 1987] und [Gries; 1981]. Vielleicht die wichtigsten Lösungen zu diesem potentiellen Problem sind die folgenden vier Ansätze, die in der Handhabung unterschiedlich jedoch logisch äquivalent sind. In den Beschreibungen unten wird das Beispiel $\{V \text{ umf. } ?\} x(i) := 1 \{x(i) = x(j)\}$ behandelt.

• **1. Unmittelbare algebraische Ableitung der Vorbedingung:** Man unterscheide in der Schreibweise ausdrücklich zwischen den Werten der Variablen in der ursprünglichen Datenumgebung d und in der nachherigen Datenumgebung $Z.d$ (wo Z die fragliche Zuweisung ist) und wandle die Nachbedingung in einen Ausdruck um, in dem nur ursprüngliche Variablenwerte erscheinen. Dieser Ausdruck ist die gesuchte Vorbedingung. Das Beispiel oben wird auf diese Weise wie folgt gelöst, wobei ' Auswertung in der vorherigen Datenumgebung d bedeutet und " Auswertung in der nachherigen Datenumgebung ($x(i) := 1$).d. Aus der Definition einer Zuweisung folgt, daß $i = i'$, $j = j'$, $x''(k) = x'(k)$ für alle $k \neq i'$ und $x''(i') = 1$. Unter Berücksichtigung dieser Beziehungen läßt sich aus der Nachbedingung die Vorbedingung ableiten:

$$x(i) = x(j) \quad \text{[Nachbedingung]}$$

$$= \quad \quad \quad \text{[Nachbedingung in nachheriger Datenumgebung auszuwerten]}$$

$$= \quad x''(i') = x''(j')$$

$$= \quad x''(i') = x''(j')$$

$$= \quad \begin{matrix} 1 = x'(j'), & \text{falls } j' \neq i' \\ 1 = 1, & \text{falls } j' = i' \end{matrix}$$

$$= \quad 1 = x'(j') \wedge j' \neq i' \vee 1 = 1 \wedge j' = i'$$

$$= \quad 1 = x'(j') \vee j' = i' \quad \text{[nur Werte in der ursprünglichen Datenumgebung d]}$$

$$= \quad 1 = x(j) \vee j = i \quad \text{[Vorbedingung]}$$

• **2. Trennung der Bezüge auf Feldvariablen:** Meist einfacher und weniger umständlich geht die Ermittlung einer umfassenden Vorbedingung gemäß Beweisregel Z1 (bzw. Z2), wenn man die Nachbedingung derart umformt, daß jeder Bezug auf eine Variable des fraglichen Felds eindeutig entweder zu ersetzen oder nicht zu ersetzen ist. Kommen solche Bezüge auf Feldvariablen in **und**- bzw. **oder**-Reihen vor, dann nimmt man zweckmäßigerweise den Term, der sich auf die fragliche Feldvariable bezieht, aus der Reihe heraus (vgl. Abschnitt 3.3.4.1 oben). Sonst führt man die Fallunterscheidung dadurch ein, daß man die Nachbedingung bzw. den betroffenen Teil davon mit $(B \vee \neg B)$ **und**-verknüpft, wo B die relevante Bedingung der Fallunterscheidung ist, und den resultierenden Ausdruck expandiert. Auf diese Weise wird das Beispiel oben wie folgt gelöst:

$$= \quad (x(i) = x(j))_1^{x(i)}$$

$$= \quad ((j = i \vee j \neq i) \wedge x(i) = x(j))_1^{x(i)}$$

$$= \quad (j = i \wedge x(i) = x(j) \vee j \neq i \wedge x(i) = x(j))_1^{x(i)}$$

$$= \quad (j = i \vee j \neq i \wedge x(i) = x(j))_1^{x(i)}$$

$$= \quad j = i \vee j \neq i \wedge 1 = x(j)$$

$$= \quad j = i \vee 1 = x(j) \quad \text{[umfassende Vorbedingung]}$$

Ein Beispiel für das Herausheben eines Terms aus einer Reihe als Vorbereitung auf das Ersetzen eines Variablenamens durch einen Ausdruck enthält Abschnitt 3.3.3, siehe die Ableitung der Bedingungen V1T (Aussage [11]) und V1E (Aussage [12]).

• **3. Schreibweise für die Veränderung eines Feldelements:** Unter Verwendung einer in der Fachliteratur vor längerer Zeit eingeführten Schreib- und Betrachtungsweise wird aus einer Zuweisung zu einer Feldvariable eine Zuweisung ohne Besonderheit und damit die oben geschilderte Problematik umgangen. [Gries; S. 90 und 124 ff., 1981] schreibt $(b; i:e)$ für das Feld b , dessen i -tes Element in den Wert e verändert worden ist; d.h., $(b; i:e)(k) = e$, falls $k = i$, und $(b; i:e)(k) = b(k)$, falls $k \neq i$.

Die Zuweisung $b(i) := e$ zu einer Feldvariable wird als eine Zuweisung $b := (b; i:e)$ zum ganzen Feld aufgefaßt. Dadurch wird die Beweisaufgabe $\{V \text{ umf. } ?\} x(i) := 1 \{x(i) =$

$x(j)$ in $\{V \text{ umf. ?} \} x := (x; i:1) \{x(i) = x(j)\}$ umgewandelt. Gemäß Beweisregel Z1 ist die umfassende Vorbedingung

$$\begin{aligned}
 & (x(i) = x(j))_{(x; i:1)}^x \\
 = & (x; i:1)(i) = (x; i:1)(j) \\
 = & 1 = (x; i:1)(j) \\
 = & j = i \wedge 1 = (x; i:1)(j) \vee j \neq i \wedge 1 = (x; i:1)(j) \\
 = & j = i \vee j \neq i \wedge 1 = x(j) \\
 = & j = i \vee 1 = x(j)
 \end{aligned}$$

• **4. Algebraische Schreibweise mit if-Ausdruck:** Eine algebraische Schreibweise, die eine Fallunterscheidung in Teilausdrücken zuläßt, stellt noch eine Lösung zu dieser Problematik dar. Eine solche Schreibweise wurde z.B. in der Programmiersprache Algol-60 vorgesehen [Naur; 1962] und ist in anderer Form in der Mathematik gut bekannt. Unter Verwendung einer derartigen Schreibweise wird das Beispiel oben wie folgt gelöst:

$$\begin{aligned}
 & (x(i) = x(j))_{1}^{x(i)} \\
 = & 1 = (\text{if } j = i \text{ then } 1 \text{ else } x(j)) \\
 = & \text{if } j = i \text{ then wahr else } 1 = x(j) \\
 = & j = i \vee j \neq i \wedge 1 = x(j) \\
 = & j = i \vee 1 = x(j)
 \end{aligned}$$

Wegen problematischer Aspekte der Korrektheitsbeweissführung hinsichtlich Feldvariablen, insbesondere bezüglich des Umfangs der Beweise, hat [Mills; 1986 Feb.] von der Verwendung von Feldvariablen abgeraten (siehe auch [Ince; 1992]). Feldvariablen sind jedoch seit langem in der Softwareentwicklung fest etabliert. Es ist wahrscheinlich unrealistisch zu fordern, daß man auf sie verzichtet. Die oben erläuterten Lösungsmöglichkeiten zu dieser Problematik zeigen, daß die Korrektheitsbeweissführung den Verzicht auf Feldvariablen nicht voraussetzt.

Zeiger können als Indexe und die Variablen, auf die sie zeigen, als Feldvariablen aufgefaßt werden. Der einzige Unterschied zwischen ihnen (abgesehen von der Schreibweise) betrifft die Menge, aus der die Index- bzw. Zeigerwerte sind. Indexwerte sind typischerweise aus einer Menge von aufeinander folgenden Ganzzahlen; Zeigerwerte sind im Gegensatz dazu aus einer beliebigen Menge. Für die Korrektheitsbeweissführung ist dieser Unterschied unwesentlich.

Wenn Synonyme für Variablenamen zulässig sind, entsteht eine Problematik, die der oben geschilderten vergleichbar ist. Die Lösung basiert auf dem gleichen Prinzip: alle Bezüge auf die durch die Zuweisung veränderte Variable sind bei der Anwendung der Beweisregeln Z1 und Z2 zu ersetzen, unabhängig davon, ob ein solcher Bezug der origi-

nale Variablenname oder ein Synonym dafür ist. Die Fallunterscheidung ist dabei meist nicht erforderlich.

3.3.4.3 Nicht definierte Ausdrücke

Ist der Wert eines Teils eines Ausdrucks nicht definiert, dann tritt oft die Frage auf, ob der Wert des ganzen Ausdrucks definiert ist oder nicht. Der Ausdruck kann entweder im Programmtext oder in einer Vor- oder Nachbedingung vorkommen. Das eigentliche Problem stammt daher, daß es mehrere sinnvolle Möglichkeiten gibt, eine Funktion auf der Menge {falsch, wahr} (bzw. auf {falsch, wahr} × {falsch, wahr}, usw.) auf einer erweiterten Menge, z.B. {undef, falsch, wahr} fortzusetzen. Weder unter Programmiersprachenimplementierungen noch in der Mathematik gibt es eine allgemein akzeptierte Antwort auf Fragen dieser Art. Hauptsächlich um diese Problematik zu umgehen ist die Auffassung einer Vor- bzw. Nachbedingung als Teilmenge von \mathbb{D} im Abschnitt 3.1.5 eingeführt worden.

Ein typisches Beispiel dieser Problematik ist der Ausdruck $i < n \wedge x < A(i)$ in einem Zusammenhang, wo $i \geq n$ gilt und $A(i)$ nicht vereinbart ist. Der erste Teilausdruck ($i < n$) ist falsch. Der zweite Teilausdruck ist nicht definiert. Die Fortsetzung der \wedge -Funktion kann entweder so definiert werden, daß $\text{falsch} \wedge (\text{undefinierter Wert}) \triangleq \text{falsch}$, oder so, daß $\text{falsch} \wedge (\text{undefinierter Wert})$ undefiniert ist.

[Baber; 1987] führt drei verschiedene Möglichkeiten auf, die Booleschen Funktionen derart fortzusetzen; eine davon besteht aus den bekannten Funktionen **and** und **cor**, siehe z.B. [Dijkstra; 1976], [Gries; 1981]. [Bijlsma] stellt eine vierte Möglichkeit vor, die eine interessante Verallgemeinerung des Grundgedanken der dritten in [Baber; 1987] behandelten Möglichkeit darstellt. Der Vorschlag von Bijlsma führt jedoch zu einem Auswertungsmechanismus, der nicht mit der klassischen und gewöhnlichen Vorgehensweise zur Ermittlung des Werts eines Ausdrucks (nämlich durch schrittweises Berechnen und Kombinieren der Zwischenergebnisse) in Einklang gebracht werden kann. [Parnas; 1992 Feb. und 1993 Sept.] unterbreitet einen weiteren Vorschlag zur Auswertung von Ausdrücken, die eventuell undefinierte Teilausdrücke enthalten. Etwas vereinfacht formuliert basiert sein Vorschlag auf einer mengenbezogenen Interpretation von Prädikaten, in der ein undefinierter Wert eines Booleschen Ausdrucks in den logischen Wert *falsch* überführt wird.

Bei der Festlegung der Definitionen und Begriffen in Abschnitt 3.1, insbesondere der Definitionen der **if**-Anweisung und der **while**-Schleife, in denen die Bedingung B vorkommt, wurde unterstellt, daß $\neg B$ gleichwertig mit $B = \text{falsch}$ ist. Wird die Negation \neg für ein undefiniertes Argument anders definiert, dann müssen die erwähnten Stellen entsprechend geändert werden. Eine andere Vereinbarung, z.B. $\neg \text{undefiniert} = \text{wahr}$ (entsprechend der Betrachtung einer Bedingung als Teilmenge und der Negation \neg als Mengenkomplement), ist prinzipiell möglich, wäre jedoch ungewöhnlich.

Bei der Korrektheitsbeweissführung muß man auf die potentielle Problematik nicht definierter Ausdrücke achten. Die "Lösung" besteht aus der Wahl einer geeigneten Fortsetzung der Booleschen Funktionen und Ausdrücken sowie ihrer konsequenten und systematischen Anwendung. Dabei sind natürlich die spezifischen Eigenschaften der Programmiersprachendefinition bzw. -implementierung, die für die Ausführung des fraglichen Programms eingesetzt werden soll, zu berücksichtigen.

3.3.4.4 Unterprogrammaufruf mit formaler Parameterübergabe

Der Unterprogrammaufruf mit formaler Parameterübergabe ist ein für den Softwareentwickler vorteilhaftes und nützliches Konstrukt. Trotzdem birgt es einige Gefahren in sich.

In den verschiedenen Programmiersprachen sind unterschiedliche Mechanismen für die Parameterübergabe implementiert worden. Sie weichen oft in kleinen — aber hinsichtlich der Korrektheit des jeweiligen Programms sehr wesentlichen — Details voneinander ab, auch wenn die grundsätzlichen Prinzipien in zwei Kategorien weitgehend eingeteilt werden können: Wertübergabe (engl. "call by value") und Referenz bzw. Bezugnahme (engl. "call by name"). Nicht wenige Softwareentwickler verfügen über nur allgemeine Kenntnisse über diese Mechanismen; einige haben sogar falsche Vorstellungen davon. Da solche Details die Korrektheit eines Programms beeinflussen können, müssen sie im Korrektheitsbeweis entsprechend berücksichtigt werden. Das setzt voraus, daß die fraglichen Parameterübergabemechanismen auf geeignete Weise definiert sind. Günstig für die Korrektheitsbeweissführung ist es, wenn sie mit Hilfe der in Abschnitt 3.1.3 vorgestellten Programm-anweisungen und Zusammensetzungen davon definiert werden. Siehe [Baber; 1987, Abschnitt 4.0].

Auf der Basis von ausreichend präzisen Definitionen von spezifischen Parameterübergabemechanismen können darauf bezogene Beweisregeln für Unterprogrammaufrufe formuliert werden. [Gries; 1980 Oct. und 1981] und [Martin, Alain J.; 1983] enthalten Beispiele von solchen Beweisregeln für einige Parameterübergabemechanismen. Bei der Anwendung derartiger Beweisregeln muß der Softwareentwickler besonders sorgfältig darauf achten, daß das vorliegende Programmiersprachensystem die Voraussetzungen der jeweiligen Beweisregel erfüllt. Typische Beweisregeln dieser Art sind weniger allgemein gültig als die im Abschnitt 3.2 vorgestellten Beweisregeln.

Zur Dokumentation eines Unterprogramms gehört eine oder mehrere formale Korrektheitsaussagen, die die Spezifikation des Unterprogramms darstellen. Im Falle eines Unterprogrammaufrufs mit formaler Parameterübergabe ist eine solche Aussage eigentlich keine konkrete Korrektheitsaussage, sondern nur ein Schema dafür, denn die konkreten Namen der übergebenen Parameter können bei jedem Aufruf unterschiedlich sein. Oft hängt die Gültigkeit des Korrektheitsbeweises davon ab, ob verschiedene aktuelle Parameter die gleiche Programmvariable sind oder nicht. Ggf. muß eine entsprechende Bedingung zur Hypothese der fraglichen Korrektheitsaussage (bzw. des Schemas dafür) hinzugefügt werden. Als Beispiel betrachte die folgende Prozedur:

```
procedure proc(x, y)
  x:=y+1
endprocedure
```

und den Aufruf

```
call proc(a, b)
```

darauf, der ein Aufruf durch Namensübergabe (engl. "call by name" [Naur; 1962]) bzw. bei neueren Sprachen durch Referenz (engl. "call by reference") sein soll. D.h., die Anweisung `call proc(a, b)` wird als der Kern der Prozedur `proc` mit dem Namen `x` durch den Namen `a` und dem Namen `y` durch den Namen `b` textuell ersetzt definiert. Mit anderen Worten, die Anweisung `call proc(a, b)` hat definitionsgemäß die gleiche Wirkung wie die Anweisung `a:=b+1`. Der Korrektheitsatz, den Konstrukteure von Programmteilen, die

`proc` aufrufen, in ihren Korrektheitsbeweisen verwenden dürfen, wäre dann: $\{\text{wahr}\} \text{call proc}(a, b) \{a=b+1 \wedge b=b'\}$, wo `a` und `b` unterschiedliche aber sonst beliebige Variablenamen sind.

Der Satz gilt nämlich nicht, wenn die gleiche Variable für beide Parameter eingesetzt wird. Vgl. z.B. den Aufruf `call proc(z, z)`. In diesem Fall wird die Nachbedingung nie erfüllt, denn es gilt, daß $\{\text{falsch}\} \text{call proc}(z, z) \{z=z+1 \wedge z=z'\}$ umfassend. Es gilt sogar, daß $\{\text{falsch}\} \text{call proc}(z, z) \{z=z'\}$ umfassend bzw. $\{\text{wahr}\} \text{call proc}(z, z) \{z \neq z'\}$.

Im Beispiel oben muß der erste aktuelle Parameter ein Variablenname sein. Der zweite aktuelle Parameter darf typischerweise ein Ausdruck sein. Kommt die Variable, die der erste Parameter ist, im Ausdruck, der der zweite Parameter ist, vor, dann ist die Richtigkeit des Satzes nicht ohne weitere Voraussetzungen gewährleistet.

Um solche Probleme von vornherein aus dem Wege zu gehen sollte der Programmkonstrukteur bei Unterprogrammaufrufen mit formaler Parameterübergabe möglichst einfache Strukturen hinsichtlich der fraglichen Parameter verwenden. Dabei sind Zusammenhänge zwischen den Parametern zu vermeiden. Bei der Formulierung des Korrektheitsatzes (bzw. des Schemas) über Aufrufe auf das fragliche Unterprogramm ist die größte Sorgfalt geboten; besonders auf die Bedingungen hinsichtlich der Parameter und ihrer Zusammenhänge ist zu achten und ausdrücklich hinzuweisen. Bezüglich der Korrektheitsbeweissführung ist es natürlich am einfachsten, wenn Unterprogramme nur ohne formale Parameterübergabe aufgerufen werden.

3.3.4.5 Computerarithmetik

Die in wirklichen Rechnersystemen implementierte Arithmetik ist nur eine Annäherung an die in der Mathematik üblichen arithmetischen Funktionen bzw. eine Einschränkung davon auf endliche Mengen. Deshalb weisen die implementierten Funktionen oft andere Eigenschaften auf als die in der Mathematik definierten Funktionen. Oft werden die gleichen Zeichen für beide Funktionsarten verwendet, welches zu Verwirrung bei der Korrektheitsbeweissführung führen kann. In Korrektheitsbeweisen muß man sorgfältig darauf achten, daß nur tatsächlich erfüllte Eigenschaften unterstellt werden. Ein häufig unterlaufender derartiger Fehler (z.B. beim Kürzen) ist die implizite Annahme, die Gleitkommaaddition sei assoziativ.

Die Gleitkommaarithmetik birgt noch andere potentielle Fallen in sich, insbesondere weil sie Ergebnisse liefert, die von den Werten der entsprechenden mathematischen Funktionen auf den rationalen Zahlen abweichen können. Bei der Korrektheitsbeweissführung stehen mehrere Möglichkeiten zur Wahl, den unvermeidlichen Genauigkeitsverlust der Gleitkommaarithmetik zu berücksichtigen:

(1) Die mögliche Ungenauigkeit des Ergebnisses wird in den Vor-, Zwischen- und Nachbedingungen durch entsprechende Schranken ausgedrückt, z.B. $\{\dots\} \text{sum}:=x+y \{|\text{sum}-(x+y)| < \dots\}$.

(2) In den Vor-, Zwischen- und Nachbedingungen werden Aussagen über die genauen Ergebnisse formuliert, wobei zwischen den Operationen der Mathematik und der Implementierung ausdrücklich unterschieden wird, z.B. $\{\dots\} \text{sum}:=x \odot y \{ \text{sum} = x \odot y \}$. Die mögliche Ungenauigkeit des Ergebnisses (z.B. $|\text{sum}-(x+y)|$ bzw. $|(x \odot y)-(x+y)|$) wird völlig getrennt vom Programmkorrektheitsbeweis analysiert und abgegrenzt.

(3) Man vernachlässigt die mögliche Abweichung zwischen dem gewünschten und dem berechneten Ergebnis.

Möglichkeit (3) ist am einfachsten und liefert Aussagen, die für manche praktischen Zwecke ausreicht. Wenn die Ungenauigkeit der berechneten Ergebnisse von Belang sein könnte bzw. in Betracht gezogen werden soll, muß man eine der anderen Alternativen wählen. Wegen der Trennung der nicht miteinander verwandten Aspekten der Programmlogik und der numerischen Genauigkeit ist Möglichkeit (2) oben in der Durchführung typischerweise einfacher als (1). Sie führt zu einem übersichtlicheren und verständlicheren Beweis. Aus diesen Gründen ist sie in der Regel vorzuziehen. In Anhang 3 werden einige Fehlerschranken für gleitkommaarithmetische Berechnungen abgeleitet, die sich für derartige vom Korrektheitsbeweis getrennte Ungenauigkeitsanalysen eignen. Siehe auch Abschnitt 7.1.

Im Programmtext stehende Ausdrücke werden von vielen Systemen auf irgendeine Weise verändert und die veränderte Version des Programms ausgeführt. Dieses trifft besonders für optimierende Übersetzer zu, die oft sehr umfangreiche Analysen und Umformungen vornehmen, die sich über mehrere Programmanweisungen erstrecken können.

Manche Programmiersprachensysteme runden Ergebniswerte, z.B. im Falle von Indexausdrücken bei Feldvariablen oder bei Typenumwandlungen (Abbildungen von Werten aus einer Menge in Werte aus einer anderen Menge). Hinsichtlich der Richtung der Rundung besteht in der Praxis keine Einheitlichkeit; Rundung nach oben, nach unten, nach Null hin, von Null weg usw. kommen vor und sind in Normen vorgesehen [IEEE; 1987 Feb. und 1987 October 5]. Wo die Rundungsrichtung für die Korrektheitsbeweisführung wesentlich ist, sollte die Rundung im fraglichen Programmtext ausdrücklich geschrieben werden, z.B. $X(\text{int}((i+j)/2))$ oder $X(\text{int}((i+j+1)/2))$ oder $X(\text{int}((i+j)/2+1))$ statt $X((i+j)/2)$. Dadurch werden Fehler und die Notwendigkeit von zusätzlichen Annahmen in Korrektheitsbeweisen vermieden sowie die Übertragbarkeit des Programms auf andere, z.B. künftige Systeme erhöht.

Auch aus Gründen der Hardware-„Integrität“ wird im *Interim Defence Standard 00-55* vom Gebrauch der Gleitkommaarithmetik in sicherheitskritischer Software abgeraten [Ministry of Defence; Part 1, Abschnitt 30.1.3 und Part 2, Abschnitt 30.3.8]. Unter Hardware-„Integrität“ wird offensichtlich die Entwurfsfehlerfreiheit bzw. die funktionale Korrektheit der Hardware gemeint; im Anhang A zu Part 1 wird „Safety Integrity“ definiert als „The likelihood of a safety critical system, function or component achieving its required safety features under all the stated conditions within a stated measure of use“. Part 2, Abschnitt 30.3.8 bemerkt, daß „floating point coprocessors are not generally of the integrity required for safety critical hardware.“

3.3.4.6 Rekursion

Rekursion wird oft als problematisch betrachtet. Eine ablehnende Neigung gegen Rekursion ist z.B. im *Interim Defence Standard 00-55* ersichtlich [Ministry of Defence; Part 1, Abschnitt 30.1.3 und Part 2, Abschnitt 30.3.9]. Demgemäß ist ein Beweis für eine Schranke für die Rekursionstiefe und den sich daraus ergebenden Speicherbedarf Voraussetzung für die Verwendung von Rekursion in der Implementierung. Der frühere Entwurf vom 1989 Mai ging noch weiter und verbot die Verwendung von Rekursion als unsicher und schwer analysierbar (siehe Abschnitt 21, Punkt 2).

Hinsichtlich der Beweisführung über die partielle Korrektheit weist Rekursion keine besondere Problematik auf, abgesehen von den zusätzlichen Schritten eines induktiven Beweises. (Die Korrektheit eines rekursiven Programmsegments wird typischerweise induktiv bewiesen.)

Potentiell problematisch ist die Beweisführung der vollständigen Korrektheit, d.h., daß das fragliche Programmsegment überhaupt ausgeführt wird. Die Problemursache liegt darin, daß die Menge (Quantität) der für die Ausführung benötigten Betriebsmittel (Speicherkapazität) von Werten bestimmter Programmvariablen typischerweise funktional abhängt. Ferner hängt diese funktionale Beziehung vom jeweiligen Programmiersprachensystem sowie ggf. auch von der Konfiguration des programmausführenden Rechners ab. Wenn (und nur wenn) diese funktionale Abhängigkeit sowie geeignete Schranken für die verfügbare Speicherkapazität (die von der Speicherbelegung durch andere Prozesse abhängen kann) und für die Rekursionstiefe zur Konstruktionszeit bekannt sind, kann eine zuverlässige Aussage über die vollständige Korrektheit gemacht werden. Die Länge der Datenumgebung (die Anzahl der darin enthaltenen Programmvariablen) liefert einen Anhaltspunkt für eine Analyse des Speicherbedarfs bei der Ausführung eines rekursiven Unterprogramms.

Bei entsprechender Programmkonstruktion können oft Ergebnisse des Programmübersetzers oder sogar von geeigneten Testläufen eine zuverlässige Aussage darüber ermöglichen, ob ausreichende Speicherkapazität für die Ausführung eines rekursiven Programms zur Verfügung steht. Je nach dem, ob die verfügbare Speicherkapazität ausreicht oder nicht, wird das fragliche Programm ausgeführt werden können oder nicht — mit anderen Worten, vollständig korrekt bezüglich des fraglichen Ausführungssystems sein oder nicht.

Ein gleichartiges Problem entsteht auch in anderen Zusammenhängen. Z.B. beim Gebrauch von Programmiersprachen, die die dynamische Zuordnung von Variablen (z.B. von Feldern variabler Größe, Zeigervariablen usw.) zulassen, kann der Speicherbedarf von Werten der Eingabevariablen abhängen.

Die Korrektheit eines rekursiven Programms wird typischerweise durch Induktion bewiesen. [Baber; 1987, Abschnitt 5.8.1] enthält einen Korrektheitsbeweis für ein rekursives Unterprogramm, das die Fakultätsfunktion berechnet. Die programmlogischen bzw. rechnungsbezogenen Aspekte des Beweises sind in dem Falle genauso einfach wie die datenumgebungsstrukturellen Aspekte, so daß der Beweis am einfachsten durch unmittelbare Anwendung der Definitionen der einzelnen Programmanweisungen (also ohne Anwendung von Beweisregeln) durchzuführen ist. Im Gegensatz dazu zeigt das Beispiel in Anhang 4 ein Mischverfahren, in dem auch von Beweisregeln Gebrauch gemacht wird. In dem Beispiel wird eine Aussage bezüglich der Rekursionstiefe (genauer: über die maximale Länge einer während der rekursiven Ausführung entstehenden Datenumgebung) in den zu beweisenden Korrektheitsatz aufgenommen und bewiesen. Siehe auch [Hoffmann; 1990 Apr.-Juli, Abschnitt 7.8.3].

3.4 Mathematische Anforderungen an den Softwareentwickler

In Abschnitt 2.2.1 wurden Einwände gegen den praktischen Einsatz von Programmkorrektheitsbeweistechniken und in Abschnitt 2.2.2 daraus abgeleitete Anforderungen an einen praxisgerechten Ansatz zur Korrektheitsbeweisführung diskutiert. In diesem Abschnitt wird näher auf die Anforderungen mathematischer Art eingegangen.

In Aufsätzen und Diskussionen über die praktische Anwendung formaler Methoden wird von Vertretern der relevanten theoretischen Forschung oft hervorgehoben, daß der Praktiker, der diese Kenntnisse anwenden will, über bestimmte und weitgehende mathematische Vorkenntnisse verfügen und sich deshalb vorher eine entsprechende Denk- und Ausdrucksweise aneignen muß. Im Prinzip ist dies richtig; es wird jedoch oft übersehen, daß durch eine inhaltlich unwesentliche Umgestaltung des Stoffs die an den Praktiker zu stellenden mathematischen Anforderungen verlagert und damit der Lernprozeß für ihn erheblich erleichtert werden können. Dadurch kann Praktikern die Programmkorrektheitsbeweissführung viel zugänglicher gemacht werden.

Bei einer solchen Umgestaltung der theoretischen Grundlage soll zweckmäßigerweise betrachtet werden, daß der Praktiker eine wesentlich andere Zielsetzung, Orientierung und Wissensbasis hat als der Theoretiker. U.a. deshalb legt er eine andere Betonung auf die verschiedenen Aspekte des Stoffs. Er beschäftigt sich mit einer anderen Aufgabenstellung und Problematik. Diese Unterschiede sollen und meiner Erfahrung nach können in der (1) Präsentation der theoretischen und wissenschaftlichen Grundlage der Programmkorrektheitsbeweissführung, (2) mathematischen Schreibweise, (3) Form und Gliederung der Beweise und (4) Orientierung der Beweisregeln Ausdruck finden. Schließlich darf nicht außer Acht gelassen werden, daß es nicht nur eine Art Praktiker, sondern verschiedenartige Praktikerkreise gibt.

3.4.1 Vorkenntnisse

Die Anwendung von Programmkorrektheitsbeweistechniken setzt mathematische Vorkenntnisse voraus; die Meinung wird oft vertreten, daß sie die Kenntnisse und Fähigkeiten der meisten Softwareentwickler übersteigen, z.B. in [Ayres], [Barroca], [Dijkstra; 1982, S. 273-274], [Gries; 1980, 1991 March]. Auch Studenten haben Schwierigkeiten, die mathematische Basis für die Korrektheitsbeweissführung zu beherrschen [persönliche Kommunikation: Ehrenberger, McDermid, Mander].

Um die in den Abschnitten 3.1 bis 3.3 vorgestellte Grundlage für die Korrektheitsbeweissführung zu verstehen und sie anwenden zu können (siehe auch Anhang 2), muß der Softwareentwickler über allgemeine mathematische Kenntnisse (z.B. was ein Beweis ausmacht) sowie über Kenntnisse der folgenden Teilgebiete und Aspekte der Mathematik verfügen:

- mathematische Objekte wie Variablen, Folgen, Funktionen
 - Mengenlehre
 - Algebra, vor allem logische (Boolesche) Algebra und die Umformung deren Ausdrücke
- Weitergehende mathematische Kenntnisse, z.B. über Beweistheorie, mathematische Logik, Fixpunkte usw., sind nicht erforderlich.

Hinsichtlich der Tiefe der erforderlichen Kenntnisse muß zwischen verschiedenen Softwareentwicklerkreisen unterschieden werden. Der verantwortliche Softwaresystemkonstrukteur muß die oben aufgeführten Gebiete beherrschen und Spezifikationen (Vor-, Nach- und Zwischenbedingungen sowie Schleifeninvarianten) selbständig formulieren können. Er muß in der Umformung von Ausdrücken der Booleschen Algebra geübt sein. Sein technischer Assistent kann mit weniger tiefen Kenntnissen und vor allem ohne die Fähigkeit, selbst Vor- und Nachbedingungen formulieren zu können, bei der Codierung von Programmseg-

menten sowie bei der Nachprüfung bereits erstellter Programmentwürfe wertvolle Dienste leisten.

Die oben geschilderten mathematischen Voraussetzungen sind vergleichbar mit den mathematischen Voraussetzungen für die klassischen Ingenieurwissenschaften. Dabei sind die hierfür benötigten Vorkenntnisse eher weniger umfangreich und leichter anzueignen — man denke z.B. an die Differential- und Integralrechnung über komplexe Variablen und Funktionen, deren Einsatz in den klassischen Ingenieurfächern üblich ist.

3.4.2 Verwendete Notationsformen

Der Softwareentwickler, der sich mit der Programmkorrektheitsbeweissführung auseinandersetzen will, muß verschiedene Notationsformen für Ausdrücke der Booleschen Algebra kennen. Die Fachliteratur basiert zu einem großen Teil auf der Mathematik und weist eine entsprechende Schreibweise auf. Die Anwender, mit denen der Softwareentwickler Spezifikationen (Vor- und Nachbedingungen) entwickeln und abstimmen wird, sind Experten auf anderen Gebieten und sind oft an andere Schreibweisen gewöhnt — insofern sie überhaupt über Vorkenntnisse der logischen Algebra verfügen. Ferner führen verschiedene Spezifikationsprachen (z.B. Z, siehe [Spivey; 1989], und VDM, siehe [Jones; 1986]) ihre eigenen spezifischen Zeichen und Schreibweisen ein.

Die unten stehende Tabelle zeigt unterschiedliche Schreibweisen für die wichtigsten Funktionen, die in der Booleschen Algebra vorkommen. Auch wenn gute mathematische Gründe gegen die in der Spalte "Technik" aufgeführte Schreibweise bestehen, muß festgestellt werden, daß sie in der Hardwaretechnik seit langem verbreitet ist, siehe z.B. [Harrison]. Die Kommunikation mit manchen Ingenieuren aus dieser Fachrichtung wird durch den Gebrauch dieser Schreibweise erleichtert.

Die in der Spalte "lange Form" aufgeführte Schreibweise ist, besonders für den ungeübten Leser von Ausdrücken der Booleschen Algebra, am leichtesten zu verstehen. Dabei werden für die für viele unerfahrene Leser schwer verständlichen universellen und existentiellen Quantifizierungen (siehe die Einwände gegen den praktischen Einsatz formaler Methoden in Abschnitt 2.2.1) ggf. vorhandene Vorkenntnisse der \sum -Notation ausgenutzt. Selbst ohne vorherige Erklärung wird der typische Ingenieur oder Techniker die Schreibweise $\text{und}_{i=1}^n$ sofort richtig (oft unbewußt) interpretieren. Vgl. z.B. die folgenden bedeutungsgleichen aus [VDI-GIS; S. 192 und 225] stammenden Ausdrücke hinsichtlich Verständlichkeit:

$$[\text{Gefunden} \wedge (\exists i: i \in \mathbb{N}_1, i \leq n: A(i) = x)] \vee [\neg \text{Gefunden} \wedge (\forall i: i \in \mathbb{N}_1, i \leq n: A(i) \neq x)]$$

sowie

$$\text{Gefunden} \text{ und } (\text{oder}_{i=1}^n A(i) = x) \quad [\text{irgendein Element von } A = x]$$

oder

$$(\text{nicht Gefunden}) \text{ und } (\text{nicht } \text{und}_{i=1}^n A(i) \neq x) \quad [\text{kein Element von } A = x]$$

Verschiedene Leserkreise werden unterschiedliche Schreibweisen bevorzugen. Der kürzeste Ausdruck ist nicht unbedingt der verständlichste oder lesbarste; ungewohnte Symbole können abschreckend wirken [Barroca]. Die Schreibweise soll auf die jeweilige Leserzielgruppe angepaßt werden.

lange Form	Mathematik	Technik	Bedeutung
und	\wedge	$*, \cdot, \times$, nebeneinander schreiben	und-Funktion (Konjunktion)
oder	\vee	$+$	oder-Funktion (Disjunktion)
nicht	\neg, \sim	$-, \bar{\quad}$	Negierung
impl. \Rightarrow	\Rightarrow, \supset	\Rightarrow, \rightarrow	Implikation
wahr	wahr, true	1	Konstante wahr
falsch	falsch, false	0	Konstante falsch
und _{$i=1 \dots$}	\forall , ($\forall i: \dots : \dots$)	(kommt relativ selten vor)	für alle, für jede, (universelle Quantifizierung)
oder _{$i=1 \dots$}	\exists , ($\exists i: \dots : \dots$)	(kommt relativ selten vor)	es gibt, es existiert, für irgendein, (existenzielle Quantifizierung)

[Normenausschuß Informationsverarbeitungssysteme (NI) im DIN; DIN 66000] gibt für endliche **und**- und **oder**- Reihen die Schreibweise $\wedge(\phi_1, \dots, \phi_n)$ bzw. $\vee(\phi_1, \dots, \phi_n)$ für $n \geq 2$ an. Reihen mit unbegrenzt vielen Termen erwähnt diese Norm nicht.

3.5 Stufenweise Gestaltungsmöglichkeiten für unterschiedliche Anwenderkategorien

Der Unterschied zwischen der herkömmlichen Vorgehensweise bei der Softwareentwicklung und einer durch den vollständigen Einsatz formaler Methoden gekennzeichneten neuen Arbeitsweise ist zu groß, als daß der Übergang in einem Schritt verwirklicht werden könnte. Außerdem ist der vollständige Einsatz solcher Methoden eventuell nur für wenige Softwareentwicklungsprojekte wirtschaftlich vertretbar. Es besteht ein Bedarf an dazwischen liegenden Stufen, die für einige Softwareentwickler Endziele, für andere nur Zwischenstufen sind. Die verschiedenen Gestaltungsmöglichkeiten unterscheiden sich voneinander durch

- den Abstraktionsgrad der Formulierung der Problemstellung, der Analyse und der Darstellung,
- die Präzision der Betrachtung,
- die mathematische Strenge der Korrektheitsargumentation,
- den Umfang der schriftlich festgehaltenen Begründung der Korrektheit sowie
- den erforderlichen mathematischen Reifegrad seitens des Softwareentwicklers bzw. die an ihn zu stellenden mathematischen Anforderungen.

Jede solche Anwendungsstufe soll von der vorherigen Stufe leicht, d.h. ohne großen Lernaufwand, erreichbar sein sowie ein günstiges Verhältnis zwischen Aufwand und Nutzen aufweisen. Sonst wird sie aus praktischer Sicht uninteressant sein.

Hinsichtlich der Formalität des Einsatzes von Korrektheitsbeweissführungstechniken können die verschiedenen Möglichkeiten in drei grobe Kategorien eingeteilt werden. Bei der *informellen* Anwendung geht man nicht mathematisch rigoros vor. Vor-, Zwischen- und Nachbedingungen werden in natürlicher Sprache und mit anderen nicht rigorosen Hilfsmitteln (z.B. Diagrammen) ausgedrückt. Bei der oft "*rigoros*" genannten Anwendung werden Bedingungen usw. in einer Form festgehalten, die zwar nicht mathematisch formal ist, jedoch auf mehr oder weniger offensichtliche Weise formalisiert werden kann. D.h. inhaltlich ist die rigorose Anwendung prinzipiell vollständig formal bzw. formalisierbar, obwohl die äußere Form der Argumentation nicht formal ist. Für offensichtlich wahre Korrektheitsaussagen wird entweder kein oder nur ein skizzenhafter oder angedeuteter Beweis gegeben. Eine solche Vorgehensweise wird in den klassischen Ingenieurfachrichtungen oft praktiziert. Die *formale* Anwendung setzt voraus, daß alle Bedingungen, Ausdrücke, Beweisschritte usw. in mathematisch formaler Form festgehalten und ausgeschrieben werden, manchmal unter Verwendung einer speziellen Sprache, z.B. Z. Siehe [Barroca] und vgl. "Rigorous Argument" und "Formal Proof" im *Interim Defence Standard 00-55* [Ministry of Defence].

Die Aufstellung der Spezifikation, die Programmkonstruktion und der Korrektheitsnachweis können auf unterschiedliche Weisen behandelt werden, d.h. z.B., man kann die Spezifikation durch formale Vor- und Nachbedingungen angeben, daraus auf informelle Weise ein Programm entwerfen und Plausibilitätsargumente für die Korrektheit nur einiger Teile davon andeuten. Auch ein Korrektheitsbeweis kann auf unterschiedliche Weise gehandhabt werden, in dem man z.B. die partielle Korrektheit formal beweist und die vollständige Korrektheit informell behandelt. In der Praxis ist die Terminierung einer Schleife oft offensichtlich, in welchem Falle eine informelle Beweisskizze häufig genügt.

Die Spezifikation kann mehr oder weniger detailliert sein. In ihr können entweder nur bestimmte (z.B. sicherheitskritische) Eigenschaften, oder die volle Funktionalität des fraglichen Programms berücksichtigt werden. Sie kann sich entweder nur auf die Werte maßgeblicher Programmvariablen und Dateielemente oder auch auf die Struktur der Datenumgebungen vor und nach der Ausführung des Programms beziehen.

Um so detaillierter die Spezifikation und um so formaler die Vorgehensweise desto stärker wird die Aussage — und auch aufwendiger die Analyse — sein. Durch Auswahl einer geeigneten Kombination von Detail und Grad der Formalismus kann ein für den jeweiligen Fall vernünftiger Ausgleich zwischen der Aussagekraft der Analyseergebnisse einerseits und dem betriebenen Aufwand andererseits erreicht werden.

3.5.1 Mathematische Vorkenntnisse, Terminologie und Schreibweise

Bei ihrer informellen Anwendung führen die Konzepte und Grundsätze der Programmkorrektheitsbeweissführung zu einer gewissen Strukturierung und Systematisierung von lockeren Plausibilitätsargumenten für die Korrektheit eines Programms, wodurch ihre Aussagefähigkeit erhöht werden kann. Dabei ist das übergeordnete Ziel die Verringerung der Softwarefehlerrate. Außer einem gewissen Ansatz zu einer mathematischen Orientierung bzw. Denkweise sind keine besonderen mathematischen Vorkenntnisse nötig.

Am anderen Extrem des Spektrums liegt die vollständig formale Anwendung z.B. von Z oder VDM. Dabei muß der Softwareentwickler

- die mathematisch formale Spezifikation selbst verfassen,
 - sie schrittweise in immer konkretere Fassungen umwandeln,
 - die Übereinstimmung dieser verschiedenen Fassungen miteinander beweisen,
 - ein dazu passendes Programm entwerfen sowie
 - die Übereinstimmung des Programmentwurfs mit der letzten Fassung beweisen können.
- Dafür sind umfangreiche aktive Kenntnisse der Mathematik im allgemeinen und der spezifischen formalen Methode(n) insbesondere Voraussetzung. Zu den erforderlichen Kenntnissen der spezifischen formalen Methode gehört die Fähigkeit, mit ihrer speziellen Notation gewandt umzugehen.

Die Kluft zwischen den anspruchsvollen Anforderungen der fortgeschritteneren Anwendungsmöglichkeiten und den mathematischen Fähigkeiten selbst guter Softwareentwickler kann nicht nur durch die Förderung bzw. Forcierung der mathematischen Ausbildung gegenwärtiger und künftiger Softwareentwickler sondern auch durch die gezielte Vereinfachung des theoretischen Stoffs und ihrer Darstellung, wozu u.a. die Schreibweise gehört, überbrückt werden. Durch eine solche Vereinfachung kann die Aufmerksamkeit des Softwareentwicklers mehr auf den eigentlichen Inhalt des Stoffs und weniger auf seine äußere Form gelenkt werden. Dadurch kann wiederum die Motivation des Praktikers erhöht werden, sich mit diesem Stoff und mit der damit verbundenen Mathematik auseinanderzusetzen. Abschnitt 3.4.2 zeigt einige Beispiele alternativer Notationsformen, die den Lernaufwand verringern, den Lernprozeß erleichtern und die Kommunikation fördern können. Tabellen, Diagramme und Skizzen verschiedener Art sind zusätzliche Darstellungsmöglichkeiten, die praktische Vorteile aufweisen. Beispiele davon sind die bekannten "Entscheidungstabellen", "function tables" ([Cantin], [Parnas; 1992 Oct.], [Janicki], [Zucker]), diagrammatische Darstellungen von Feldern (wie in Abschnitt 3.3.3 verwendet, siehe auch [Reynolds]) und Zeichnungen von mit Zeigern verwirklichten verketteten Listen und verzweigten Datenstrukturen. Solche Darstellungsformen können informell, halbformal oder formal definiert werden.

3.5.2 Informelle gegenüber formelle Anwendung

Der folgende Auszug aus einem 1987 im Klientenauftrag erstellten Programm [Baber; unveröffentlichte Arbeitsunterlage, 1987] zeigt gleichzeitig verschiedene Formalitätsstufen hinsichtlich der Korrektheitsargumentation. Das in der (Programmtext-)Zeile 30000 beginnende Unterprogramm liest zwei "symbol files" und fügt die daraus entnommenen "port names" zusammen. Jedes Paar von "port names" wird in eine (Datei-)Zeile in der "connection file" geschrieben. Jede Symboldatei ist eine sequentielle Datei, die eine Folge von Namen enthält. Der erste Name ist der Name der Datei; jeder andere Name ist ein Portname. Ein Name ist eine Folge von Zeichen außer dem Leerzeichen. Zwei Namen werden entweder durch eine Folge von einem oder mehreren Leerzeichen oder durch ein Zeilenende getrennt.

Das in der Zeile 30000 beginnende Unterprogramm ruft zwei Unterprogramme auf. Das Unterprogramm 30600 fügt zwei Portnamen und eine geeignete Zeilennummer in eine Zeichenfolge zusammen, die in eine Zeile der Verbindungsdatei zu schreiben ist. Das lo-

gisch komplexere Unterprogramm 30700 zieht den nächsten Portnamen aus der angegebenen Symboldatei heraus.

Die in den Zeilen 30040 und 30043 festgehaltenen Vorbedingungen sind hier informell ausgedrückt. Der System- bzw. Programmkonstrukteur hätte sie vermutlich formal ausdrücken können, in welchem Fall sie als "rigoros" betrachtet werden können. Hier fehlt zunächst eine weitergehende Aussage, daß kein anderes Gerät auf irgendeinem numerierten Kanal geöffnet ist. Diese Tatsache folgt jedoch aus dem gesamten Zusammenhang, denn im gesamten Programm werden nur Dateien und COM-Porte über numerierte Kanäle angesprochen. Ähnlich ist die in den Zeilen 30050 bis 30056 enthaltene Nachbedingung als eine informelle Nachbedingung — bzw. als eine rigorose Nachbedingung, falls Spezifikationen der Dateiartern sowie der Beziehung zwischen zwei Symboldateien und einer Verbindungsdatei formal geschrieben werden können — zu sehen.

Die in den Zeilen 30180 bis 30290 erscheinenden Programminvarianten enthalten sowohl informelle bzw. rigorose als auch formale Teilbedingungen. Der Verfasser ging offensichtlich davon aus, daß die Ausdrucksweise einem mit der Aufgabe vertrauten Leser klar verständlich sein würde, obwohl Begriffe wie "have been extracted", "in prior lines", "in later lines" und "currently being assembled" mathematisch nicht eindeutig definiert worden sind. Auch nicht ganz eindeutig ist es, an welchen Stellen des Programms diese Programminvariante gelten soll; insofern spiegelt sie nur die Absicht des Programmkonstruktors bzw. seine algorithmische Strategie wider. Trotz diesen aus formaler Sicht festzustellenden Unzulänglichkeiten war diese Vorgehensweise ein Erfolg: weder bei den Abnahmetests noch im produktiven Einsatz hat sich ein Fehler in diesem Programmsystem bemerkbar gemacht.

Die Nachbedingung der Schleife (siehe die Zeilen 30760 bis 30780) kann formal aus der while-Bedingung in Zeile 30730 abgeleitet werden: die Nachbedingung ist die Negierung der while-Bedingung, vgl. Beweisregel W1 bzw. W2. Bei deren Ableitung wird von der Eigenschaft der implementierten Funktion MID\$(s\$, x, 1) Gebrauch gemacht, daß aus $x > \text{LEN}(s\$)$ folgt, daß $\text{MID}(s$, x, 1)$ die leere Zeichenfolge ist (für ganzzahlige x). Im vorliegenden Fall ging der Programmkonstrukteur tatsächlich umgekehrt vor: Aus der gewollten Nachbedingung leitete er die while-Bedingung ab, vgl. Abschnitt 4.2.

Eine detailliertere und formale Spezifikation (Vor- und Nachbedingungen) für ein der Subroutine 30700 ähnliches Unterprogramm enthält [Baber; *Fehlerfreie Programmierung*, 1990, Abschnitt 5.4].

```

30000 REM * Combine a pair of symbol files into a connection file
30003 REM Input parameters: SL$ name of left symbol file
30006 REM                      SR$ name of right symbol file
30010 REM                      CF$ name of connection file
30013 REM                      PL beg. pos. of left port name in connection file
30016 REM                      PR beginning position of right port name
30020 REM                      PN beginning position of line number
30023 REM                      LN length of line number in connection file
30026 REM                      ZS increment for line number
30030 REM Output parameter: R (0: no error detected, >0: error)
30033 REM Internal variables: F, S$, LC$, L$(1), L$(2), P(1), P(2), L, Z, Z$

```



```

30036 REM Precondition:
30040 REM Symbol files named SL$ and SR$ exist on the PC.
30043 REM No files or COM ports are open on any channel number.
30046 REM Postcondition:
30050 REM The connection file named CF$ is the combination
30053 REM of the pair (SL$, SR$) of symbol files.
30056 REM No files or COM ports are open on any channel number.
30100 REM ON ERROR GOTO 30900 ' Check for error on opening files
30110 OPEN SL$ FOR INPUT AS #1
30120 OPEN SR$ FOR INPUT AS #2
30130 OPEN CF$ FOR OUTPUT AS #3
30140 ON ERROR GOTO 0 ' Turn off error routine
30150 IF EOF(1) OR EOF(2) THEN CLOSE: R=31: RETURN
30170 Z=0: FOR F=1 TO 2: L$(F)="": P(F)=1: NEXT F
30180 REM Program invariant pertaining to input from the symbol files:
30190 REM For every F (F=1, 2):
30200 REM L$(F) is the last line input from file F
30210 REM AND port names have been extracted from L$(F) up to
30220 REM but not including position P(F)
30230 REM AND 1 <= P(F) <= LEN(L$(F)) + 1
30240 REM AND all port names in prior lines in file F have been extracted
30250 REM AND no port names in later lines in file F have been extracted
30260 REM Program invariant pertaining to output to the connection file:
30270 REM LC$ contains the line currently being assembled for
30280 REM subsequent writing to the connection file.
30290 REM AND Z is the next line number to be assigned.
30300 F=1: GOSUB 30700: LC$=S$ ' Get name of left symbol file
30310 F=2: GOSUB 30700 ' Get name of right symbol file
30320 GOSUB 30600 ' Combine LC$, S$ and Z into a line for the connection file
30330 PRINT#3,LC$;" CS321 symbol file names": PRINT#3," "
30340 F=1: GOSUB 30700: LC$=S$ ' Get first left port name
30350 F=2: GOSUB 30700 ' Get first right port name
30360 WHILE LC$ <> "" OR S$ <> ""
30370 GOSUB 30600 ' Combine LC$, S$ and Z into a line for the connection file
30380 PRINT#3,LC$
30390 F=1: GOSUB 30700: LC$=S$ ' Get next left port name
30400 F=2: GOSUB 30700 ' Get next right port name
30410 WEND
30420 REM Postcondition:
30430 REM LC$ and S$ are both empty strings, i.e. both input symbol
30440 REM files have been completely processed.
30450 CLOSE: R=0
30460 RETURN

```

```

30600 REM * Combine LC$, S$ and Z into a line for the connection file
30610 REM Output variables: LC$, Z (incremented)
...
30680 RETURN

30700 REM * Get next port name from file F
30710 REM See program invariant pertaining to input from the symbol files.
30720 REM Output variable: S$ (S$="": file F contains no more port names)
30730 WHILE (P(F)>LEN(L$(F)) AND NOT EOF(F)) OR MID$(L$(F),P(F),1)=" "
30740 IF P(F)>LEN(L$(F)) THEN LINE INPUT#F,L$(F): P(F)=1 ELSE P(F)=P(F)+1
30750 WEND
30760 REM Postcondition of loop:
30770 REM P(F) <= LEN(L$(F)) AND MID$(L$(F),P(F),1) <> " "
30780 REM OR P(F) > LEN(L$(F)) AND EOF(F)
30790 IF P(F) > LEN(L$(F)) THEN S$="": RETURN
30800 REM The above statement could be removed but is included for
30810 REM clarity and readability.
30820 REM Locate position of next blank; if none, then end of string:
30830 L=INSTR(P(F),L$(F)," "): IF L=0 THEN L=LEN(L$(F))+1
30840 S$=MID$(L$(F),P(F),L-P(F))
30850 P(F)=L
30860 RETURN

30900 PRINT "*** Error opening files while combining symbol files"
30905 PRINT " into a connection file."
30910 CLOSE: RESUME 30930
30920 REM *
30930 ON ERROR GOTO 0 ' Reset error trapping
30940 CLOSE: R=39
30950 RETURN

```

Anhang 2 enthält ein Dokumentationsbeispiel (über ein anderes Unterprogramm) mit einer formalen Spezifikation, einem formalen Beweis der partiellen Korrektheit, einem formalen Beweis der Terminierung einer Schleife, Beweisskizzen ("rigorosen" Beweisen) für die Terminierung der anderen Schleifen und Beweisskizzen der anderen Aspekte der vollständigen Korrektheit, nämlich, daß jede Anweisung mit einem definierten Ergebnis ausgeführt wird.

[Sommerville; 1985, Abschnitt 4.4.1] enthält ein Beispiel einer nicht formalen Korrektheitsargumentation, die eine weniger ausgeprägte Orientierung an Vor-, Nach- und Zwischenbedingungen aufweist.

3.5.3 Abstraktionsgrad der Spezifikation und Aufgabenstellung

Spezifikationen für Programmsysteme können in verschiedenen Formen ausgedrückt werden. Sie können Beziehungen zwischen

- abstrakten mathematischen Funktionen, die Systemfunktionen modellieren bzw. beschreiben sollen,
- Unterprogrammen des Systems oder
- Eingabe- und Ergebniswerten für jedes Unterprogramm festlegen. Gegenstand einer Spezifikation können
- mathematisch abstrakte Datenelemente,
- anwendungsbezogene Datenelemente oder
- programmiersprachenbezogene Datenelemente und Datentypen sein. Spezifikationen können sehr unterschiedliche Abstraktionsgrade aufweisen; z.B. können die Mengen und Typen der darin vorkommenden Variablen
- nicht festgelegt,
- nur insofern festgelegt, daß gewisse Eigenschaften angegeben werden,
- mathematisch festgelegt, jedoch nicht in einer programmierbaren Form angegeben, oder
- auf programmtechnisch geeignete Weise festgelegt sein. Spezifikationen können Bezug auf
- nur wenige besonders wesentliche Aspekte des extern beobachtbaren Verhaltens des fraglichen Systems,
- viele detaillierte extern beobachtbare Eigenschaften des Systems oder
- sogar die interne Struktur und Funktionsweise des Systems nehmen. Eine Spezifikation kann sich auf
- einzelne Systemfunktionen und -zustände (Datenumgebungen) oder
- lange Folgen davon (Ausführungsgeschichten) beziehen.

Dementsprechend werden sehr unterschiedliche Anforderungen an die mathematischen und anwendungsfachtechnischen Kenntnisse und Fähigkeiten aller Betroffenen (nicht nur der Softwareentwickler) gestellt. Beispiele verschiedenartiger Spezifikationen enthält Anhang 1.

Besonders in der Fachliteratur über VDM und Z wird über die stufenweise Verfeinerung einer abstrakten Spezifikation $S1$ zu einer konkreten Spezifikation S_n , die als Basis für die Konstruktion eines Programms P dient, gesprochen:

$$S1 \rightarrow S2 \rightarrow \dots \rightarrow Si \rightarrow \dots \rightarrow S_n \rightarrow P$$

Die Spezifikationen werden zunehmend (nach rechts im Diagramm oben) detaillierter und konkreter. Die Spezifikation $S1$ ist am allgemeinsten; sie schränkt die Konstruktionsfreiheit am wenigsten ein. Spätere Spezifikationen enthalten bzw. widerspiegeln bereits getroffene Entwurfsentscheidungen.

Hinsichtlich der Programmkorrektheitsbeweissführung wird typischerweise davon ausgegangen, daß die Übereinstimmung des Programms P mit der Spezifikation $S1$ bewiesen werden soll. Wird der Korrektheitsbeweis zusammen mit der Verfeinerung der Spezifikation erarbeitet, müßte die Übereinstimmung jeder Spezifikation mit ihrem Vorgänger sowie des Programms mit der letzten Spezifikation formal gezeigt werden:

$$S1 \Leftrightarrow S2 \Leftrightarrow \dots \Leftrightarrow Si \Leftrightarrow \dots \Leftrightarrow S_n \Leftrightarrow P$$

Diese Vorgehensweise kann zu einem hohen Aufwand für die Korrektheitsbeweissführung führen, worüber in der Fachliteratur auch berichtet wird (siehe Abschnitt 2.2.1 und Anhang 1, Abschnitt A1.1.2). Diese Beobachtung wird oft mit der Behauptung begegnet,

allein aus der sorgfältigen Erstellung einer Spezifikation (also ohne Beweise) könne viel Nutzen gezogen werden (siehe z.B. [Guttag]).

Der in Anhang 1 (vor allem in Abschnitt A1.2) veranschaulichte Verfeinerungsprozeß (engl. "data refinement", "reification") hat das Ziel, für abstrakte Datentypen konkretere Darstellungsformen festzulegen. Dadurch wird die Spezifikation von einer abstrakten in eine konkrete Form, die der Programmkonstruktion unmittelbar dienen kann, umgewandelt. Die Forderung, zu beweisen, daß jede Version der Spezifikation mit der vorherigen übereinstimmt, führt zu Beweisverpflichtungen (engl. "proof obligations"). Dieses Thema wird in der VDM- und Z-Literatur häufig angesprochen. Siehe auch [Gries; 1992] und [Jones; 1986].

Prinzipiell könnte man die Spezifikationsverfeinerungsschritte auf informelle, intuitive Weise ausführen, auf die formalen Korrektheitsbeweise für die Zwischenstufen verzichten und lediglich am Ende der Programmkonstruktion zeigen, daß P mit $S1$ (oder mit einer vereinbarten Zwischenspezifikation Si) übereinstimmt. Der Vorteil einer solchen Vorgehensweise könnte ein geringerer Aufwand für die Beweisführung sein. Nachteilig wäre ggf. der größere Schwierigkeitsgrad des Beweises wegen des größeren konzeptionellen Abstands zwischen $S1$ (bzw. Si) und P im Vergleich zu S_n und P . Ferner wäre das Risiko größer, daß Fehler erst später entdeckt würden.

Die Annahme, daß die abstrakteste Spezifikation $S1$ die geeignetste Basis für die Kommunikation zwischen den beteiligten Parteien und für die Leistungsvereinbarung ist, wird in der Fachliteratur kaum in Frage gestellt. Das im Anhang 1, Abschnitt A1.1 enthaltene Beispiel läßt jedoch diese Annahme fraglich erscheinen. In dem Beispiel ist die konkreteste Spezifikation (siehe Abschnitt A1.1.3) für Praktiker auf dem fraglichen Fachgebiet viel leichter verständlich als die abstrakteren Spezifikationen, weil sie (die Spezifikation des Abschnitts A1.1.3) die Kenntnisse, Erfahrungen und Ziele solcher Praktiker mehr anspricht und darauf aufbaut (siehe nähere Bemerkungen darüber am Ende des Abschnitts A1.1.2). Sie ist darüber hinaus — entgegen häufiger Erwartung — nicht länger als die abstrakteren Spezifikationen des Abschnitts A1.1.2. In der Praxis wäre es voraussichtlich viel leichter, eine Übereinstimmung der beteiligten Parteien auf der Basis der konkreten Spezifikation des Abschnitts A1.1.3 zu erreichen als auf der Basis einer der Spezifikationen des Abschnitts A1.1.2. In diesem Fall erübrigt sich die Notwendigkeit, die Übereinstimmung der verschiedenen Spezifikationsstufen miteinander zu beweisen; nur die Übereinstimmung des Programms mit der letzten (bzw. mit der einzigen) Spezifikation wäre zu beweisen ($S_n \Leftrightarrow P$ bzw. $S1 \Leftrightarrow P$).

4. Bedeutung der Korrektheitsbeweissführung für die ingenieurmäßige Neukonstruktion eines Programms

Die Idee, die Anforderungen eines Korrektheitsbeweises als Leitlinien für die Konstruktion eines korrekten Programms zu verwenden, ist nicht neu; sie erschien bereits Ende der 1960er Jahre in der Fachliteratur [Dijkstra; *BIT* 1968], [Buxton; p. 85]. Auf dieser Idee basierende Vorgehensweisen zur Programmkonstruktion sind im Laufe der Jahre immer wieder vorgeschlagen worden, z.B. in [Dijkstra; insbesondere 1975, 1976], [Gries; 1976 Dec., 1981], [Scherlis], [Berlioux], [Mili; 1986], [Back], [Dromey], [Baber; insbesondere 1987, 1990 (*Fehlerfreie Programmierung*)], [Hoffmann; 1990 Apr.-Juli], [Cohen], [Kalde-waij] und [Manna; 1992]. Derartige Vorschläge beziehen sich sowohl auf das manuelle als auch auf das automatische maschinelle Erstellen von Programmen.

Bei der Anwendung dieses Konzeptes werden der Korrektheitsbeweis und das Programm schrittweise und gleichzeitig entwickelt. Dabei führt eher die Entwicklung des Beweises die des Programms. Anders betrachtet erarbeitet man einen der Spezifikation passenden Korrektheitsbeweis; daraus entsteht das Programm in einem gewissen Sinne als Nebenprodukt [Pnueli; 1988].

Obwohl diese Idee mit Interesse aufgenommen worden ist, wird sie in der Software-entwicklungspraxis selten verwirklicht. Angesichts des an vielen Stellen in der Fachliteratur konstatierten hohen Aufwands der formalen Korrektheitsbeweissführung (siehe Abschnitt 2.2.1) sollte gerade der Programmkonstruktion als praktischem Anwendungsbereich dieses Stoffs mehr Aufmerksamkeit gewidmet werden.

Bei der ingenieurmäßigen Konstruktion von Gegenständen aller Art spielt die Unterteilung derselben in Untersysteme mit wohl definierten Schnittstellen dazwischen eine zentrale Rolle. Die (externe) Spezifikation eines Systems bzw. Untersystems ist nichts anderes als die Spezifikation der Schnittstelle zwischen ihm und seiner Umgebung. Falls die Spezifikation eines Programmteils in der Form von Vor- und Nachbedingungen vorliegt, kann die Übereinstimmung des fraglichen Programmteils mit seiner Spezifikation mit Hilfe der in Abschnitt 3.2 vorgestellten Beweisregeln auf systematische Weise bewiesen werden (vgl. Abschnitt 3.3 und Anhang 2). Deshalb stellen Vor- und Nachbedingungen eine nahe-liegende Form für die Spezifikation eines Programms bzw. eines Programmsegments dar (vgl. Abschnitt 3.5.3). Entsprechend besteht die Aufgabe des Systemkonstruktors aus der Festlegung der Vor- und Nachbedingungen für untergeordnete Programmsegmente, ausgehend von den Vor- und Nachbedingungen der übergeordneten Programmteile. Vor- und Nachbedingungen für einen bestimmten Programmteil (z.B. ein Unterprogramm) können als der technische Teil eines Vertrags zwischen dem Auftraggeber (z.B. Systemkonstrukteur) und dem Lieferanten (z.B. Programmierer) angesehen werden [Meyer].

In diesem Kapitel werden die Beweisregeln, insbesondere ihre Hypothesen, als Leitlinien für die Konstruktion eines Programmsegments betrachtet, wobei die Schnittstelle(n) zwischen dem fraglichen Programmsegment und seiner Umgebung durch Vor- und Nachbedingungen spezifiziert sind. Wie im Falle der Korrektheitsbeweissführung dienen die Beweisregeln der iterativen Zerlegung der durchzuführenden Konstruktionsaufgabe in Konstruktionsaufgaben über kleinere Programmteile. Die Beweisregeln stellen sicher, daß die

so konstruierten kleineren Programmteile derart zusammenpassen werden, daß die Spezifikation des ganzen erfüllt wird.

4.1 Aus der Korrektheitsbeweissführung sich ergebende Anforderungen an und Leitlinien für die Konstruktion

Ziel der Konstruktionsaufgabe ist es, für eine gegebene Vorbedingung V und eine gegebene Nachbedingung P ein Programmsegment S derart festzulegen, daß $\{V\} S \{P\}$. Diese Aufgabe wird unten $\{V\} S? \{P\}$ abgekürzt (vgl. Abschnitt 3.3.2).

Die Korrektheit eines Programmsegments wird durch Anwendung einer der Struktur des Programmsegments entsprechenden Beweisregel bewiesen. Die zu beweisende Korrektheitsaussage ist die These der Beweisregel, die gelten wird, falls die Hypothesen der Beweisregel gelten. Bei der Konstruktion soll deshalb sichergestellt werden, daß die Hypothesen der für das zu konstruierende Programmsegment geeigneten Beweisregel erfüllt sind. Diese Hypothesen stellen die Anforderungen an bzw. die Spezifikationen für die Bestandteile des zu konstruierenden Programmsegments dar, siehe die untenstehende Tabelle. Durch die Anwendung der Beweisregel wird die Konstruktionsaufgabe in Konstruktionsaufgaben über kleinere Programmteile zerlegt. Dieser Prozeß wird iterativ auf die untergeordneten Konstruktionsaufgaben fortgesetzt, bis alle atomaren Programmanweisungen und alle Bedingungen in if-Anweisungen und while-Schleifen festgelegt worden sind.

Für den erfahrenen Softwarekonstrukteur ist die Wahl der Art der Programmanweisung (Zuweisung, Vereinbarung usw.) bzw. Zusammensetzung von Programmanweisungen (if-Anweisung, Folge oder Schleife) oft offensichtlich. Damit wird die anzuwendende Beweisregel gleichzeitig gewählt (siehe die Tabelle in Abschnitt 3.3.2).

Steht bei der Konstruktion eines Programmsegments die Art der Programmanweisung bzw. Zusammensetzung von Programmanweisungen nicht fest, dann sucht der Konstrukteur eine Beweisregel, in der die Struktur der Vor- und Nachbedingungen bzw. der Hypothesen zu den vorgegebenen Vor- und Nachbedingungen paßt. D.h., es wird eine Beweisregel gesucht, die die zu erledigende Konstruktionsaufgabe in leicht lösbare Konstruktionsaufgaben zerlegt. Aus der Wahl der Beweisregel folgt die Art der Programmanweisung bzw. Zusammensetzung von Programmanweisungen für das zu konstruierende Programmsegment. Es gibt einige nützliche Hinweise für diese Wahl, aber letzten Endes bleibt sie eine wesentliche Entwurfsentscheidung, die dem Konstrukteur überlassen bleibt und von ihm manchmal Kreativität abverlangt.

Werden die Anforderungen eines Korrektheitsbeweises bei der Programmkonstruktion vollständig und rigoros berücksichtigt, dann entfällt im Prinzip die Notwendigkeit einer nachträglichen formalen Korrektheitsbeweissführung. In der Praxis könnte man zumindest bei nicht kritischen Anwendungen auf einen vollständigen formalen Korrektheitsbeweis verzichten. Auf jeden Fall entsteht eine Beweisskizze bei einer solchen konstruktiven Vorgehensweise. Darüber hinaus werden dabei viele der in einem Beweis benötigten Zwischenbedingungen entwickelt. Solche Nebenprodukte der Konstruktion sollten in der Dokumentation des Programms festgehalten werden (siehe Abschnitt 4.5).

Alle geforderten Eigenschaften des zu konstruierenden Programmsegments müssen in den Vor- und Nachbedingungen bzw. in der Korrektheitsaussage zum Ausdruck kommen. Sogenannte "fehlerhafte Eingaben", "Ausnahmefälle" usw. sind für den Konstrukteur keine

Fehler bzw. Ausnahmen, sondern Situationen, die das zu konstruierende Programmsegment auf vorher festgelegte Weise verarbeiten muß.

Grundsätzlich darf für eine gestellte Konstruktionsaufgabe $\{V\} S? \{P\}$ ein Programmsegment mit einer schwächeren Vorbedingung $V1 (V \Rightarrow V1)$ und/oder einer stärkeren Nachbedingung $P1 (P1 \Rightarrow P)$ erarbeitet werden, d.h. $\{V1\} S? \{P1\}$, vgl. Beweisregel B1.

Die folgende Tabelle gibt für jede Anweisungsart an, welche Beweisregel anzuwenden ist, um die jeweilige Konstruktionsaufgabe zu zerlegen (vgl. die Tabelle für das Zerlegen von Beweisaufgaben in Abschnitt 3.3.2). Ferner weist die folgende Tabelle auf die weiteren Konstruktionsaufgaben, die nach jedem Zerlegungsschritt noch zu erledigen sind, hin. Der Doppelstrich in der Mitte der Tabelle trennt die atomaren Programmanweisungen von den Zusammensetzungen von Anweisungen.

Konstruktionsaufgabe	Regel	zerlegte Konstruktionsaufgaben
$\{V\} x?: = A? \{P\}$	Z2	x und A so festlegen, daß $V \Rightarrow P^x_A$
$\{V\} \text{ declare } (x?, M?, A?) \{P\}$	Z2 DS	x und A so festlegen, daß $V \Rightarrow P^x_A$ M so festlegen, daß $V \Rightarrow A \in M$
$\{P\} \text{ release } x? \{P\}$	U1	verifizieren, daß x nicht in P vorkommt
$\{V\} \text{ null? } \{P\}$ (bzw. $\{V\} \{P\}$)	B1	verifizieren, daß $V \Rightarrow P$
$\{V\} \text{ if } B \text{ then } S1? \text{ else } S2? \text{ endif } \{P\}$ (Bedingung B bereits festgelegt)	IF1	$\{V \text{ und } B\} S1? \{P\}$ $\{V \text{ und nicht } B\} S2? \{P\}$
$\{V\} \text{ if } B? \text{ then } S1 \text{ else } S2? \text{ endif } \{P\}$ (eine Anweisung — S1 — bereits festgelegt)	IF1, B1	$\{V1?\} S1 \{P\}$ B so festlegen, daß $V \text{ und } B \Rightarrow V1$ [*] $\{V \text{ und nicht } B\} S2? \{P\}$
$\{V\} \text{ if } B? \text{ then } S1? \text{ else } S2? \text{ endif } \{P\}$	IF1	$\{V \text{ und } B?\} S1? \{P\}$ $\{V \text{ und nicht } B\} S2? \{P\}$
$\{V\} S1? \{V2\} S2? \{P\}$ (Zwischenbedingung bereits festgelegt)	F1	$\{V\} S1? \{V2\}$ $\{V2\} S2? \{P\}$
$\{V\} S1?; S2? \{P\}$ (Zwischenbedingung noch nicht festgelegt)	F1	V2 festlegen $\{V\} S1? \{V2\} S2? \{P\}$
$\{V\} \text{ Init?; while } B? \text{ do } S? \text{ endwhile } \{P\}$	W2	I festlegen $\{V\} \text{ Init? } \{I\}$ B so festlegen, daß $I \text{ und nicht } B \Rightarrow P$ [*] $\{I \text{ und } B\} S? \{I\}$

$\{V\} \text{ call Prg? } \{P\}$	U3	Prg wählen mit Spezifikation $\{Vs\} \text{ call Prg } \{Ps\}$ B, V1, P1 so festlegen, daß $V = V1 \text{ und } B$ $P = P1 \text{ und } B$ $\{B\} \text{ call Prg } \{B\}$ $V1 \Rightarrow Vs$ $Ps \Rightarrow P1$
----------------------------------	----	---

* Man merke, diese zwei Aufgaben weisen die gleiche Struktur auf. In Abschnitt 4.2 wird die Bestimmung von B derart, daß $I \text{ und nicht } B \Rightarrow P$, näher behandelt.

Bemerkungen zur Zuweisung: Bei der Konstruktion einer Zuweisung steht oft fest, welcher Variable ein neuer Wert zugeordnet werden soll. Wenn nicht, kommen meist nur wenige in Frage. Wenn z.B. eine oder mehrere **und**-Reihen in der Nachbedingung P, jedoch nicht in der Vorbedingung V erscheinen, ist es zweckmäßig, x und A derart zu wählen, daß diese Reihen leer werden. Das gilt insbesondere für die Initialisierung von Schleifen. Wenn die Variable x in P jedoch nicht in V erscheint, kommen für A eine Konstante oder ein Ausdruck, in dem nur in V vorkommende Variablen erscheinen, in Frage.

Bemerkungen zur Deklaration und zur Löschanweisung: Eine Deklaration hat die gleiche Auswirkung auf die Werte von Variablen wie die entsprechende Zuweisung. Der Unterschied dazwischen liegt nur in der Struktur der Ergebnisdatenumgebung. Um bei der Programmkonstruktion dazwischen entscheiden zu können, müssen in der Spezifikation stehende (implizite oder explizite) Angaben zur Struktur der Datenumgebung zu Rate gezogen werden. Meist kommen declare- und release-Anweisungen paarweise vor; viele Zielprogrammiersprachen erzwingen eine solche Programmstruktur.

Bemerkungen zur if-Anweisung: Oft wird die if-Anweisung gewählt, nachdem sich die Bedingung B der beabsichtigten Fallunterscheidung aus einer vorbereitenden Analyse ergeben hat; für diese Situation ist die erste if-Zeile in der Tabelle oben vorgesehen. Manchmal jedoch entwirft man bei einem Versuch, die Konstruktionsaufgabe $\{V\} S? \{P\}$ zu lösen, ein Programmsegment S1, das eine u.U. stärkere Vorbedingung V1 hat. Hierfür ist die zweite if-Zeile in der Tabelle oben geeignet.

Bemerkungen zur Folge von Anweisungen: Wenn keine andere einzige Struktur oder atomare Programmanweisung in Frage kommt, wird in der Regel die Konstruktionsaufgabe in eine Folge von Programmsegmenten aufgeteilt. Die für die Aufteilung wesentliche Entwurfsentscheidung verkörpert die Zwischenbedingung (V2 in der Tabelle oben). Allgemein soll die Zwischenbedingung hinsichtlich der Anzahl der darin angesprochenen Variablen, der Komplexität, des Umfangs o.ä. zwischen den Vor- und Nachbedingungen liegen. Soll das zu konstruierende Programmsegment z.B. Werte für die Variablen y und z berechnen, und hat die Nachbedingung P die Form $V(x) \wedge P1(x,y) \wedge P2(x,y,z)$ (bzw. läßt sich P in dieser Form schreiben), dann bietet sich $V(x) \wedge P1(x,y)$ als Zwischenbedingung V2 an: $\{V\} S? \{P\}$ wird in $\{V\} S1? \{V \wedge P1\}$ und $\{V \wedge P1\} S2? \{V \wedge P1 \wedge P2\}$ zerlegt.

Bemerkungen zur while-Schleife: Eine Schleife ist als Anweisungsart angezeigt, wenn eine Reihe in der Nachbedingung P, aber nicht in der Vorbedingung V erscheint, und die Reihe nicht unmittelbar leer gemacht werden darf (vgl. die Bemerkungen zur Zuweisung oben). Jede Ausführung des Schleifenkerns fügt der Reihe einen Term hinzu.

Vor allem für die Konstruktion einer while-Schleife liefert die entsprechende Beweisregel sehr nützliche Leitlinien. Durch ihre Anwendung wird die gesamte Konstruktionsaufgabe in drei Unteraufgaben sehr unterschiedlicher Art aufgeteilt. Jede Unteraufgabe kann völlig unabhängig von den anderen gelöst werden; die Beweisregel stellt sicher, daß die drei Teillösungen richtig zusammenpassen.

Die für die Konstruktion der Schleife wesentlichste Entwurfsentscheidung wird durch die Schleifeninvariante, die nicht selten das Ergebnis eines schöpferischen Denkprozesses darstellt, ausgedrückt. Eine geeignete Schleifeninvariante wird dadurch bestimmt, daß der Konstrukteur die Anfangs- und Endsituationen (etwa die Vorbedingung und die Nachbedingung) verallgemeinert, d.h. schwächt (vgl. Beweisregel W1 in Abschnitt 3.2.2 und siehe unten).

Nachdem die Schleifeninvariante festgelegt worden ist, können die Initialisierung der Schleife und die while-Bedingung unabhängig voneinander konstruiert werden. Oft wird die while-Bedingung sogar algebraisch abgeleitet, wie in Abschnitt 4.2 näher erläutert wird.

Nachdem die Schleifeninvariante I und die while-Bedingung B festgelegt worden sind, wird der Schleifenkern S derart konstruiert, daß $\{I \wedge B\} S \{I\}$ (eine der Hypothesen der Beweisregel W2). Ferner muß S für Fortschritt in die Richtung der Nachbedingung P sorgen. Diese letzte Anforderung ergibt sich aus der Terminierungsbeweissführung und nicht aus der Beweisregel W2, die sich nur mit der partiellen Korrektheit befaßt. Eine häufig verfolgte Konstruktionsstrategie für S fängt mit der Erhöhung oder Verringerung des Werts einer Variable an, um für Fortschritt in Richtung P (bzw. $\neg B$, denn $I \wedge \neg B \Rightarrow P$, eine Hypothese der Beweisregel W2) zu sorgen. Typischerweise wird dadurch die Wahrheit der Schleifeninvariante verletzt (wenn nicht, ist S fertig). Der Rest von S hat nur die Aufgabe, die Wahrheit von I wiederherzustellen.

Verschiedene Hinweise und heuristische Regeln für die Bestimmung der Schleifeninvariante durch Verallgemeinerung der Anfangs- und Endsituationen werden in der bereits zitierten Literatur (siehe die einleitenden Bemerkungen in Abschnitt 4 oben) angegeben. Unter den wichtigsten sind:

- Eine neue Variable wird eingeführt. An einigen Stellen in der Nachbedingung wird eine andere Variable durch die neue ersetzt, um eine geeignete Schleifeninvariante zu bilden. Auf diese Weise kann oft eine Reihe, die in der Nachbedingung, aber nicht in der Vorbedingung vorkommt, derart umgestaltet werden, daß durch eine geeignete Initialisierung der neu eingeführten Variable die fragliche Reihe leer wird. Um die Terminierungsbeweissführung zu erleichtern, sollte eine möglichst starke Bedingung über den Wertebereich der neu eingeführten Variable in die Schleifeninvariante aufgenommen werden. Eine solche Bedingung sollte mindestens eine untere und eine obere Schranke für den Wert der neuen Variable beinhalten.
- Besteht die Nachbedingung aus der **und**-Verknüpfung mehrerer Termen, wovon die Wahrheit einiger anfangs nicht sichergestellt werden kann, dann läßt man solche Terme weg, um einen Kandidaten für die Schleifeninvariante zu bilden. Die weggelassenen Terme zusammen bilden die Endbedingung (**nicht** B) der Schleife.
- Der Konstrukteur zeichnet (wie in Abschnitt 3.3.3) ein Diagramm, das die Nachbedingung darstellt (P-Diagramm) und in dem Bereiche unterschiedlicher Eigenschaften gebildet werden. Ein entsprechendes Diagramm wird für die anfängliche Situation (etwa Vorbedingung) gezeichnet (V-Diagramm). Typischerweise enthält das V-Diagramm einen Bereich, der im P-Diagramm nicht vorkommt. Gezeichnet wird schließlich ein drittes Diagramm (I-

Diagramm), in dem alle Bereiche der V- und P-Diagramme vorkommen. Das I-Diagramm wird dabei so gestaltet, daß es eine Verallgemeinerung der V- und P-Diagramme ist. Das I-Diagramm stellt die Schleifeninvariante I dar. Die while-Bedingung entspricht der Aussage, daß die im I-Diagramm vorhandenen aber im P-Diagramm fehlenden Bereiche nicht leer sind. Siehe die Beispiele in Abschnitt 3.3.3 und in Anhang 2, Abschnitte 1.3 und 2.3.

Siehe [Turski; 1984 March] für eine allgemeinere Behandlung des Themas Invarianten.

Bemerkungen zum Unterprogrammaufruf: Betrachte die Vorbedingung V und die Nachbedingung P , und nenne gemeinsame Teilausdrücke davon B . Sei $V1$ der Rest von V und $P1$ der Rest von P , d.h. $V = V1 \wedge B$ sowie $P = P1 \wedge B$. Man wähle ein Unterprogramm, dessen Vorbedingung $V1$ (oder eine schwächere Bedingung) ist, dessen Nachbedingung $P1$ (oder eine stärkere Bedingung) ist, und das keine in B vorkommende Variable verändert. Auch bestimmte Varianten dieser Vorgehensweise sind möglich und kommen oft vor, z.B. werden oft Teile von B auch in $V1$ aufgenommen, damit $V1 \Rightarrow Vs$ gilt. Für die Korrektheit wesentlich ist nur, daß $V \Rightarrow Vs$, $V \Rightarrow B$, $Ps \wedge B \Rightarrow P$, $\{Vs\}$ call Prg $\{Ps\}$ und $\{B\}$ call Prg $\{B\}$.

Steht ein geeignetes Unterprogramm nicht zur Verfügung, sollte der Konstrukteur die Spezifizierung eines neuen solchen Unterprogramms in Erwägung ziehen. Insbesondere wenn die gleiche oder sehr ähnliche Funktionen an mehreren Stellen des zu konstruierenden Programms gebraucht werden, kommt die Erstellung eines neuen Unterprogramms in Frage. Ebenfalls kommt die Ausgliederung der benötigten Funktion in ein Unterprogramm in Betracht, wenn die fragliche Spezifikation (Vs , Ps) bzw. ($V1$, $P1$) oben eine umfangreiche, in sich abgeschlossene Berechnung mit wenig oder einfacher Wechselwirkung mit der Umgebung darstellt — selbst wenn die Funktion an nur einer Stelle im übergeordneten Programm gebraucht wird.

Für nähere Leitlinien und Hinweise für die Festlegung einer Spezifikation und für die Konstruktion eines entsprechenden Unterprogramms siehe Abschnitte 4.4, 5.1 und 5.3.

4.2 Ansätze und Möglichkeiten zur Ableitung von Teilen eines Programms

Die Tabelle in Abschnitt 4.1 enthält Hinweise zur Konstruktion der verschiedenen Programmanweisungen und Zusammensetzungen davon, die mehr oder weniger formal angewendet werden können. Insbesondere die while-Bedingung läßt sich oft algebraisch ableiten. Aber ob man die Hinweise formal, halbformal oder informal einsetzt, geht es in jedem Fall um die formale Erfüllung einer bestimmten Hypothese einer Beweisregel.

Das Spektrum der Anwendungsmöglichkeiten zeigt die Bestimmung einer while-Bedingung, damit die entsprechende Hypothese der Beweisregel W2 (siehe Abschnitt 3.2.2) erfüllt wird. Diese Hypothese kann auf verschiedene äquivalente Weisen geschrieben werden:

$$\begin{array}{ll} I \wedge \neg B \Rightarrow P & [1] \\ I \wedge \neg P \Rightarrow B & [2] \\ \neg B \wedge \neg P \Rightarrow \neg I & [3] \\ I \Rightarrow B \vee P & [4] \\ I \Rightarrow (\neg B \Rightarrow P) & [5] \end{array}$$

$$I \Rightarrow (\neg P \Rightarrow B) \quad [6]$$

$$\neg B \Rightarrow \neg I \vee P \quad [7]$$

$$\neg B \Rightarrow (I \Rightarrow P) \quad [8]$$

$$(I \not\Rightarrow P) \Rightarrow B \quad [8a]$$

$$\neg B \Rightarrow (\neg P \Rightarrow \neg I) \quad [9]$$

$$\neg P \Rightarrow \neg I \vee B \quad [10]$$

$$\neg P \Rightarrow (I \Rightarrow B) \quad [11]$$

$$\neg P \Rightarrow (\neg B \Rightarrow \neg I) \quad [12]$$

Die für den Programmkonstrukteur bedeutendsten Formen oben deuten auf die folgenden Fragestellungen bzw. Vorgehensweisen, die der Ermittlung einer geeigneten while-Bedingung B dienen. In den meisten Fällen können die Fragen entweder aufgrund informaler Überlegungen, mit Hilfe geeigneter Diagramme, durch formale Algebra o.ä. beantwortet werden. Es ist oft hilfreich, die Implikationen (\Rightarrow) zunächst als Gleichheit (=) anzusehen und das vorläufige Ergebnis danach zu stärken bzw. zu schwächen.

Zu 1) Um welche Bedingung ($\neg B$) muß die Schleifeninvariante I verschärft werden, um die Wahrheit der Nachbedingung P sicherzustellen?

Zu 2) Welche Bedingung (B) muß gelten, falls die Schleifeninvariante I gilt, die Nachbedingung P aber nicht? Zur Beantwortung dieser Frage kann man den Ausdruck $I \wedge \neg P$ bilden und ihn vereinfachen bzw. schwächen.

Zu 4) Um welche Bedingung (B) muß die Nachbedingung P geschwächt (erweitert) werden, um die Schleifeninvariante I zu decken?

Zu 5) Man vereinfache bzw. stärke die Nachbedingung P unter der Annahme, daß die Schleifeninvariante I gilt. Das Ergebnis kommt für die Endbedingung ($\neg B$) in Frage.

Zu 6) Man vereinfache bzw. schwäche die Negierung der Nachbedingung ($\neg P$) unter der Annahme, daß die Schleifeninvariante I gilt. Das Ergebnis kommt für die while-Bedingung B in Frage.

Zu 8) Wann — d.h. unter welcher Bedingung ($\neg B$) — folgt die Nachbedingung P aus der Schleifeninvariante I? Alternativ (8a), welche Bedingung (B) muß gelten, falls die Nachbedingung P aus der Schleifeninvariante I nicht folgt?

In der Praxis ist die Fragestellung 8 oben wahrscheinlich am nützlichsten, wenn man in- oder halbformal vorgeht. Geht man formal vor, bieten sich vor allem die in den Bemerkungen 2, 5 und 6 oben angegebenen Vorgehensweisen an.

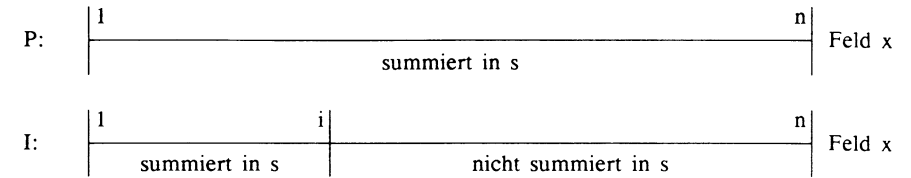
Prinzipiell darf an dieser Stelle der Schleifenkonstruktion eine mögliche while-Bedingung B geschwächt (bzw. eine Endbedingung $\neg B$ gestärkt) werden, ohne die partielle Korrektheit zu gefährden. Wird B zu viel geschwächt (bzw. $\neg B$ zu viel gestärkt), kann die Terminierung der Schleife u.U. nicht mehr sichergestellt werden (z.B. im Extremfall mit der logischen Konstante wahr für B). Um so schwächer B ist, desto schwieriger könnte auch die Konstruktion des Schleifenkerns S sein (wegen der schwächeren Vorbedingung $I \wedge B$).

Beispiel: Betrachte die Fragestellungen und Kommentare oben mit Hinblick auf das folgende bekannte Beispiel der Summierung der ersten n Elemente eines Felds x mit Nachbedingung P und Schleifeninvariante I:

$$P \triangleq s = \sum_{j=1}^n x(j)$$

$$I \triangleq n \in \mathbb{Z} \text{ und } i \in \mathbb{Z} \text{ und } 0 \leq i \leq n \text{ und } s = \sum_{j=1}^i x(j)$$

für die der Konstrukteur die folgenden Diagramme gezeichnet hat:



Aus den Diagrammen sieht man, daß P aus I folgt, wenn der Bereich "nicht summiert in s" in I leer ist, d.h. wenn $i=n$ (vgl. Ausdruck 8 oben). Eine geeignete while-Bedingung ist deshalb $i \neq n$ oder, unter der Wahrheit von I äquivalent, $i < n$.

Alternativ kann man B algebraisch ableiten (siehe Ausdruck 2 oben):

$$\begin{aligned} & I \wedge \neg P \\ = & n \in \mathbb{Z} \text{ und } i \in \mathbb{Z} \text{ und } 0 \leq i \leq n \text{ und } s = \sum_{j=1}^i x(j) \text{ und } s \neq \sum_{j=1}^n x(j) \\ \Rightarrow & i \leq n \text{ und } (\sum_{j=1}^i x(j)) \neq (\sum_{j=1}^n x(j)) \\ \Rightarrow & i \leq n \text{ und } i \neq n \\ = & i < n \blacksquare \end{aligned}$$

Entsprechende Beispiele für die Ermittlung der Bestandteile einer Zuweisung und einer if-Anweisung enthalten Abschnitt 4.3 sowie die bereits zitierte Literatur. Siehe auch [Smith]. Vgl. Abschnitt 3.3.3 und Anhang 2.

4.3 Konstruktionsbeispiel

Als Beispiel für die Anwendung der Beweisregeln auf die Programmkonstruktion soll hier ein Programmsegment PS entwickelt werden, das einen Wert für die bereits vereinbarte Variable j derart berechnet, daß $\{V\} PS \{P\}$. Das Programmsegment PS darf keine andere Variable verändern, d.h. $PS.d=d$ bis auf den Wert von j. V und P sind wie folgt vorgegeben:

$$V: M \in \mathbb{Z} \text{ und } 0 \leq M \text{ und } N \in \mathbb{Z} \text{ und } 0 \leq N$$

$$P: j \in \mathbb{Z} \text{ und } 0 \leq j \leq \max(0, M-N+1)$$

$$\text{und } \bigwedge_{k=0}^{j-1} (\text{oder } \bigwedge_{a=0}^{N-1} D(k+a) \neq K(a))$$

$$\text{und } (M-N < j \text{ oder } (j \leq M-N \text{ und } \bigwedge_{a=0}^{N-1} D(j+a) = K(a)))$$

wobei D und K nicht näher festgelegte Felder sind.

Es soll hier der üblichen Versuchung widerstanden werden, vorab die Zielsetzung des Programmsegments allgemein und informal zu beschreiben, denn eine solche Beschreibung ist für die Konstruktion dieses (und manch eines anderen) Programmsegments nicht not-

wendig, sie würde die Aufmerksamkeit von den wesentlichen Konstruktionsfragen ablenken und sie wäre, zumindest in diesem Falle, der Effizienz des Konstruktionsprozesses abträglich. Erst nachher wird eine informale und intuitive Interpretation der Aufgabe gegeben werden. Noch irrelevanter für die Konstruktionsaufgabe ist der Grund, warum und wozu dieses Programmsegment ausgeführt werden soll; dieser Grund ist nur für die Konstruktion des übergeordneten Programmteils von Belang.

Die Reihen, die in P aber nicht in V vorkommen, deuten auf eine Schleife als Grundstruktur für das zu konstruierende Programmsegment PS. Als erste Unteraufgabe ist dann die Festlegung der Schleifeninvariante I (siehe die Tabelle in Abschnitt 4.1). I muß eine Verallgemeinerung von den Anfangs- und Endsituationen sein, d.h. die Anfangs- und Endsituationen müssen Sonderfälle von I sein. P hat hier (wie typischerweise) die umfangreichere Struktur, weshalb P zunächst als die Basis für die Erarbeitung von I betrachtet wird. Wäre $j=0$, würden sich die ersten zwei Zeilen in P auf den Wert wahr reduzieren, wovon V ein Sonderfall ist ($V \Rightarrow$ wahr). Es bliebe dann nur die letzte und-Reihe übrig. Wäre $N=0$, würde diese Reihe verschwinden, aber der Wert von N ist vorgegeben und darf nicht verändert werden. Es liegt deshalb nahe, eine neue Variable i einzuführen und N in der Obergrenze der fraglichen und-Reihe durch i zu ersetzen. Der Wertebereich von i muß 0 und N umfassen, um die fragliche Reihe leer machen bzw. um die Nachbedingung abdecken zu können. Damit steht I fest:

$$\begin{aligned} I: & j \in \mathbf{Z} \text{ und } 0 \leq j \leq \max(0, M-N+1) \\ & \text{und } i \in \mathbf{Z} \text{ und } 0 \leq i \leq N \\ & \text{und } \bigwedge_{k=0}^{j-1} (\text{oder } \bigwedge_{a=0}^{N-1} D(k+a) \neq K(a)) \\ & \text{und } (M-N < j \text{ oder } (j \leq M-N \text{ und } \bigwedge_{a=0}^{i-1} D(j+a) = K(a))) \end{aligned}$$

Es bleiben noch die Konstruktionsaufgaben

$$\begin{aligned} \{V\} \text{ Init? } \{I\} \\ I \wedge \neg B? \Rightarrow P \\ \{I \wedge B\} S? \{I\} \text{ (wobei S für Fortschritt in Richtung P bzw. } \neg B \text{ sorgen muß)} \end{aligned}$$

zu erledigen.

Die Initialisierung geht aus den bereits getroffenen Überlegungen hinsichtlich der Schleifeninvariante I hervor: die Werte von i und j sind auf 0 zu setzen. Alternativ ergibt sich aus einer kurzen Betrachtung von I, daß $I = (0 \leq N)$ falls $i=0$ und $j=0$, d.h. $(I_0^i)_0^j = (0 \leq N)$. Weil $V \Rightarrow (0 \leq N)$, eignet sich eine entsprechende Initialisierung (vgl. Beweisregeln Z1, Z2 und F1). Der Wert einer etwa bereits vorhandenen Variable i darf nach den Konstruktionsvorgaben nicht verändert werden, weshalb i anfangs neu vereinbart und am Ende von PS gelöscht werden muß. Der erste Teilentwurf des Programmsegments wird also:

```
j:=0; declare (i, Z, 0)
while B? do S? endwhile
release i
```

Es bleiben noch die Konstruktionsaufgaben

$$\begin{aligned} I \wedge \neg B? \Rightarrow P \\ \{I \wedge B\} S? \{I\} \text{ (wobei S für Fortschritt in Richtung P bzw. } \neg B \text{ sorgen muß)} \end{aligned}$$

zu erledigen.

Die while-Bedingung B kann z.B. mit Hilfe des Ausdrucks $6 (I \Rightarrow (\neg P \Rightarrow B))$ in Abschnitt 4.2 abgeleitet werden. Dabei ist es günstig, $G(j, i)$ für die Reihe $\bigwedge_{a=0}^{i-1} D(j+a) = K(a)$ zu schreiben. Unter der Annahme der Wahrheit von I gilt:

$$\begin{aligned} & \text{nicht P} \\ = & \text{nicht } (M-N < j \text{ oder } (j \leq M-N \text{ und } G(j, N))) \quad [\text{wegen der Wahrheit von I}] \\ = & M-N \geq j \text{ und } (j > M-N \text{ oder nicht } G(j, N)) \\ = & j \leq M-N \text{ und nicht } G(j, N) \\ = & j \leq M-N \text{ und } G(j, i) \text{ und nicht } G(j, N) \quad [\text{wegen der Wahrheit von I}] \\ \Rightarrow & j \leq M-N \text{ und } i \neq N \\ = & j \leq M-N \text{ und } i < N \quad [\text{wegen der Wahrheit von I}] \end{aligned}$$

Jeder der letzten zwei Ausdrücke eignet sich für die while-Bedingung B. Traditionell wählt man der Robustheit wegen (defensive Programmierung) die stärkere Bedingung davon, d.h. $j \leq M-N$ und $i < N$. Manchmal wird dadurch die Terminierungsbeweissführung etwas erleichtert oder vereinfacht.

Es bleibt noch die Konstruktionsaufgabe

$$\{I \wedge B\} S? \{I\} \text{ (wobei S für Fortschritt in Richtung P bzw. } \neg B \text{ sorgen muß)}$$

zu erledigen.

Offensichtlich wird Fortschritt in Richtung P bzw. $\neg B$ (= Fortschritt in Richtung Terminierung der Schleife) dadurch erreicht, daß der Wert der Variable i oder j erhöht wird, vgl. die while-Bedingung oben. Es bieten sich also die Zuweisungen $i:=i+1$ und $j:=j+1$ als Kandidaten für den Schleifenkern S an. Daraus entstehen die Fragen, ob

$$I \wedge B \Rightarrow I_{i+1}^i ? \text{ bzw.}$$

$$I \wedge B \Rightarrow I_{j+1}^j ?$$

gelten (vgl. Beweisregel Z2). Wenn eine dieser Implikationen gilt, ist ein geeigneter Schleifenkern gefunden. Wenn nicht, müssen ausgehend von dem Unterschied zwischen $I \wedge B$ und I_{i+1}^i bzw. I_{j+1}^j weitere Programmanweisungen konstruiert werden. Dazu bildet man diese drei Ausdrücke (vgl. den Ausdruck für I oben):

$$\begin{aligned} I \wedge B = & j \in \mathbf{Z} \text{ und } 0 \leq j \leq \max(0, M-N+1) \\ & \text{und } i \in \mathbf{Z} \text{ und } 0 \leq i < N \quad [i < N \text{ wegen B}] \\ & \text{und } \bigwedge_{k=0}^{j-1} (\text{oder } \bigwedge_{a=0}^{N-1} D(k+a) \neq K(a)) \\ & \text{und } j \leq M-N \text{ und } \bigwedge_{a=0}^{i-1} D(j+a) = K(a) \quad [M-N < j \text{ entfällt wegen B}] \end{aligned}$$

$$\begin{aligned}
I_{i+1}^i &= j \in \mathbf{Z} \text{ und } 0 \leq j \leq \max(0, M-N+1) \\
&\text{und } i \in \mathbf{Z} \text{ und } -1 \leq i < N \quad [= i+1 \in \mathbf{Z} \text{ und } 0 \leq i+1 \leq N] \\
&\text{und } \bigvee_{k=0}^{j-1} (\text{oder}_{a=0}^{N-1} D(k+a) \neq K(a)) \\
&\text{und } (M-N < j \text{ oder } (j \leq M-N \text{ und } \bigvee_{a=0}^i D(j+a) = K(a))) \\
I_{j+1}^j &= j \in \mathbf{Z} \text{ und } -1 \leq j \leq \max(-1, M-N) \quad [j+1 \in \mathbf{Z} \text{ und } 0 \leq j+1 \leq \max(0, M-N+1)] \\
&\text{und } i \in \mathbf{Z} \text{ und } 0 \leq i \leq N \\
&\text{und } \bigvee_{k=0}^j (\text{oder}_{a=0}^{N-1} D(k+a) \neq K(a)) \\
&\text{und } (M-N < j+1 \text{ oder } (j+1 \leq M-N \text{ und } \bigvee_{a=0}^{i-1} D(j+1+a) = K(a)))
\end{aligned}$$

Aus dem Vergleich zwischen $I \wedge B$ und I_{i+1}^i ist es ersichtlich, daß I_{i+1}^i nicht aus $I \wedge B$ folgt. Vielmehr gilt, daß

$$I \wedge B \wedge D(j+i) = K(i) \Rightarrow I_{i+1}^i$$

und, gemäß Beweisregel Z2,

$$\{I \wedge B \wedge D(j+i) = K(i)\} \quad i := i+1 \quad \{I\}$$

Falls die Zuweisung $i := i+1$ in den Schleifenkern aufgenommen wird, bietet sich ihre Einbettung in eine if-Anweisung an (vgl. Beweisregel IF1, siehe Tabelle in Abschnitt 4.1), z.B. `if D(j+i)=K(i) then i:=i+1 else S2? endif`, wobei $\{I \wedge B \wedge D(j+i) \neq K(i)\} \quad S2? \quad \{I\}$.

Aus dem Vergleich zwischen $I \wedge B$ und I_{j+1}^j ist ersichtlich, daß I_{j+1}^j nicht aus $I \wedge B$ folgt. Zwei Merkmale stehen dem im Wege: die Obergrenzen der ersten und-Reihen und die letzten und-Reihen insgesamt, in deren Termen die Indexwerte um 1 versetzt sind ($j+1$ statt j). Der einfachste Weg, diese zwei Ausdrücke in Einstimmigkeit miteinander zu bringen, ist es, das letzte Glied der ersten und-Reihe in I_{j+1}^j als zusätzliche Bedingung aufzunehmen und die letzte und-Reihe in I_{j+1}^j leer zu machen. Es gilt also, daß

$$I \wedge B \wedge (\text{oder}_{a=0}^{N-1} D(j+a) \neq K(a)) \Rightarrow (I_{j+1}^j)_0^i$$

Aber einer der Terme in dieser oder-Reihe ist $D(j+i) \neq K(i)$, siehe die Vorbedingung von S2 im vorherigen Absatz. Formal,

$$\begin{aligned}
&I \wedge B \wedge D(j+i) \neq K(i) \\
= &I \wedge B \wedge 0 \leq i < N \wedge D(j+i) \neq K(i) \quad [0 \leq i < N \text{ ist ein Term in } I \wedge B, \text{ siehe oben}] \\
\Rightarrow &I \wedge B \wedge (\text{oder}_{a=0}^{N-1} D(j+a) \neq K(a)) \\
\Rightarrow &(I_{j+1}^j)_0^i
\end{aligned}$$

Daraus folgt gemäß Beweisregeln F1, Z1 und Z2, daß $\{I \wedge B \wedge D(j+i) \neq K(i)\} \quad i := 0; j := j+1 \quad \{I\}$. Die Folge $i := 0; j := j+1$ eignet sich also für S2.

Zusammenfassend gilt, daß

$$\{I \wedge B \wedge D(j+i) = K(i)\} \quad i := i+1 \quad \{I\} \text{ und}$$

$$\{I \wedge B \wedge D(j+i) \neq K(i)\} \quad i := 0; j := j+1 \quad \{I\}$$

woraus gemäß Beweisregel IF1 folgt, daß

$$\{I \wedge B\} \text{ if } D(j+i) = K(i) \text{ then } i := i+1 \text{ else } i := 0; j := j+1 \text{ endif } \{I\}$$

Damit ist die letzte Konstruktionsaufgabe erledigt. Das gesamte Programmsegment PS ist dann

```

j := 0; declare (i, Z, 0)
while j ≤ M-N und i < N do
  if D(j+i) = K(i) then i := i+1 else i := 0; j := j+1 endif
endwhile
release i

```

Dieses Programmsegment sucht die Folge $[K(0), \dots, K(N-1)]$ in der Folge $[D(0), \dots, D(M-1)]$. Auf herkömmliche Weise für diese Suchaufgabe konstruierte Programmsegmente sind typischerweise länger und enthalten oft zwei verschachtelte Schleifen — und programmlogische Fehler.

[Baber; 1990 (*Fehlerfreie Programmierung*), Abschnitt 5.3] enthält eine weniger formale und auf Diagrammen sowie Interpretation der Aufgabe basierte Konstruktion dieses Programmsegments. Die in den einführenden Bemerkungen in Abschnitt 4 zitierte Literatur enthält zusätzliche Konstruktionsbeispiele.

Bei der Konstruktion eines Programmsegments können Ausdrücke wie die Vor- und Nachbedingungen, Schleifeninvarianten, Zwischenbedingungen usw. anwendungsbezogen interpretiert werden. Das Beispiel oben zeigt jedoch, daß sie nicht so interpretiert werden müssen. Eine informale, auf Diagrammen und anwendungsbezogener Interpretation der Ausdrücke basierte Vorgehensweise bei der Konstruktion kann evtl. manchmal schneller zum Ziel führen. Eine formale Vorgehensweise, die weitgehend oder ganz auf anwendungsbezogene Interpretation der Ausdrücke verzichtet, ist in der Regel genauer und weniger fehleranfällig. Die Vor- und Nachteile der verschiedenen Vorgehensweisen ergänzen sich; der gute Softwareingenieur soll beides beherrschen.

4.4 Spezifikationen bei der Konstruktion und im Korrektheitsbeweis

Auf den klassischen ingenieurwissenschaftlichen Fachgebieten fördert bzw. ermöglicht die Festlegung präziser Schnittstellenspezifikationen für einzelne Komponenten

- die Austauschbarkeit der Komponente, z.B. bei Reparaturen nach ihrem Ausfall oder bei einer Konstruktionsänderung bzw. -verbesserung,
- die Wiederverwendbarkeit in anderen, z.B. später entwickelten Systemen — ohne Konstruktionsänderung oder sonstige Anpassung der Komponente,
- die getrennte, voneinander unabhängige und gleichzeitige Entwicklung mehrerer Komponenten, ohne daß diese bei der Systemintegration noch aufeinander anpaßt werden müssen, und
- die getrennte, voneinander unabhängige und gleichzeitige Fertigung mehrerer Komponenten mit der Gewißheit, daß diese in der Endmontage richtig zusammenpassen werden.

Einige dieser Gründe sind für Software nicht zutreffend, dennoch rechtfertigen die restlichen die Festlegung klarer, präziser Schnittstellenspezifikationen für Softwarekomponenten wie in der Fachliteratur mehrmals gefordert (siehe die Literaturhinweise in Abschnitt 1.1 bezüglich des Vorschlags, bei der Softwareentwicklung ingenieurmäßig vorzugehen).

Die externe Spezifikation (=Schnittstellenspezifikation) eines Unterprogramms (oder eines sonstigen Programmsegments im allgemein, das z.B. an mehreren Stellen des Programms "in-line" eingesetzt (eingebettet) werden soll) enthält alles, was der Konstrukteur eines verwendenden (z.B. aufrufenden) Programmteils über das Unterprogramm wissen und worauf er sich im Korrektheitsbeweis für den übergeordneten Programmteil beziehen darf. Insbesondere darf der Korrektheitsbeweis für den übergeordneten Programmteil nicht vom Programmtext des aufrufenden Unterprogramms abhängen (vgl. das Konzept "information hiding" von Parnas). Entsprechend enthält die Schnittstellenspezifikation alles, was der Konstrukteur des Unterprogramms über den aufrufenden Programmteil wissen und worauf er sich im Korrektheitsbeweis für das Unterprogramm beziehen darf. Der Korrektheitsbeweis für ein Unterprogramm darf nicht vom Programmtext des aufrufenden Programmteils abhängen. Die Schnittstellenspezifikation entkoppelt den aufrufenden Programmteil und das fragliche Unterprogramm voneinander.

Beim praktischen Einsatz der in dieser Arbeit behandelten Ansätze zur Softwareentwicklung gehören zur externen Spezifikation eines Unterprogramms UP

- eine Aussage über die Werte von Programmvariablen vor und nach der Ausführung des Unterprogramms — in der Regel in der Form einer strikten Vorbedingung und einer Nachbedingung,
- eine Aussage über die Struktur der Ergebnisdatenumgebung — typischerweise in der Form: $UP.d=d$ strukturell oder $UP.d=[(..., ..., ...), ...]\&d1$, wo $d1=d$ strukturell, und
- eine vollständige Liste der Variablen, die durch die Ausführung des Unterprogramms verändert werden können.

Die erste Aussage — bestehend aus den Vor- und Nachbedingungen — bildet den wesentlichen Teil der zu beweisenden Korrektheitsaussage; sie beschreibt die logische Funktionalität des Unterprogramms. Die dritte Aussage ist für die Anwendung der Beweisregeln U1 bis U3 (siehe Abschnitt 3.2.4) erforderlich (vgl. die darin vorkommende Bedingung B). Manchmal werden die zweite und dritte Aussagen zusammen in einem Ausdruck kombiniert, z.B. $UP.d=[(x, Z, ...), (y, Z, ...)]\&d$.

Im Falle eines Programms, das nicht terminieren soll (z.B. bei manchen Echtzeitanwendungen), übernimmt die Schleifeninvariante für die nicht terminierende Schleife die Rolle der Nachbedingung oben. Anstatt einer strikten Vorbedingung wird eine halbstrikte angegeben. Die Zielsetzung eines solchen Programms ist es, das System in einem gültigen Zustand — durch die Schleifeninvariante spezifiziert — zu halten.

Wie bereits erwähnt, sollen die Vor- und Nachbedingungen sogenannte "Fehler-" und "Ausnahmefälle" explizit berücksichtigen, wenn sie auf eine bestimmte Weise behandelt werden sollen. In solchen Situationen gibt es zwei (oder mehrere) Paare von Vor- und Nachbedingungen, z.B. $\{Vk\}$ UP $\{Pk\}$ und $\{Vf\}$ UP $\{Pf\}$, wo Vk "korrekte" und Vf "fehlerhafte" Eingaben darstellen; Pk und Pf sind die entsprechenden Nachbedingungen. Zwei (oder mehrere) solche Teilspezifikationen können in eine einzige Spezifikation kombiniert werden, denn

$$(\{Vk\} \text{ UP } \{Pk\} \wedge \{Vf\} \text{ UP } \{Pf\}) \iff \{Vk \vee Vf\} \text{ UP } \{(Vk' \implies Pk) \wedge (Vf' \implies Pf)\}$$

Wenn Vk und Vf sowie Pk und Pf disjunktiv sind, kann die kombinierte Spezifikation in anderen, für manche Zwecke einfacheren Formen geschrieben werden. Oft werden in den verschiedenen Nachbedingungen (Pk , Pf usw.) unterschiedliche Werte eines Rückmeldekodes festgelegt, so daß die Nachbedingungen disjunktiv sind.

Zu den Vor- und Nachbedingungen von zu konstruierenden Unterprogrammen gehören relevante Dateninvarianten (Konsistenzbedingungen für angesprochene Datengruppen und -sammlungen) und Programminvarianten (Bedingungen, die an mehreren bestimmten Stellen des Programms, von dem das fragliche Unterprogramm ein Bestandteil ist, wahr sein sollen).

Aus der Einschränkung, daß für die Konstruktion und Korrektheitsbeweissführung der jeweils anderen Seite einer Schnittstelle nur die Spezifikation maßgebend ist, folgt, daß eine Leistung der einen Seite (des Unterprogramms oder des aufrufenden Programmteils) nur dann von der anderen Seite der Schnittstelle genutzt werden darf, wenn die Leistung in der Schnittstellenspezifikation zum Ausdruck gebracht worden ist. Aus dieser Sicht ist zu empfehlen, spezifizierte Vorbedingungen für Unterprogramme möglichst schwach und spezifizierte Nachbedingungen möglichst stark zu formulieren. Ferner wird dadurch die Konstruktion sowie die Korrektheitsbeweissführung der aufrufenden Programmteile ggf. erleichtert.

Diese Betrachtungen führen u.a. zu den Spezifikationsleitlinien, in der Vorbedingung möglichst breite Wertebereiche für Eingabevariablen zuzulassen und in der Nachbedingung möglichst scharfe Aussagen über die Wertebereiche berechneter Variablen (z.B. mit unteren und oberen Schranken) vorzusehen. Ein Unterprogramm sollte möglichst wenige Variablen (idealerweise nur Ausgabevariablen) verändern; Hilfsvariablen sollten lokal vereinbart und vor dem Ende des Unterprogramms wieder gelöscht werden. Die Spezifikation sollte entsprechend formuliert werden.

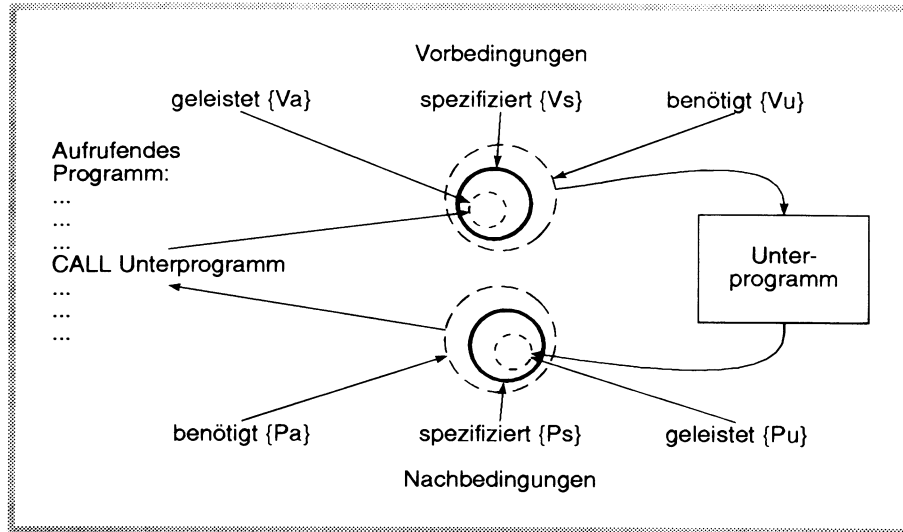
Betrachtet man den gesamten Prozeß des Aufrufens eines Unterprogramms, treten drei Arten von Vor- und Nachbedingungen des Aufrufs in Erscheinung:

```

...
vorheriger Teil des aufrufenden Programms
{Va} [vom aufrufenden Programmteil geleistet]
{Vs} [spezifizierte Vorbedingung des Unterprogramms]
{Vu} [vom Unterprogramm benötigt]
call Unterprogramm
{Pu} [vom Unterprogramm geleistet]
{Ps} [spezifizierte Nachbedingung des Unterprogramms]
{Pa} [vom aufrufenden Programmteil benötigt]
nachfolgender Teil des aufrufenden Programms
...

```

Die folgende Abbildung zeigt die Beziehungen zwischen den drei Arten von Vor- und Nachbedingungen eines Aufrufs auf ein Unterprogramm. Darin werden die verschiedenen Vor- und Nachbedingungen als Mengen dargestellt.



Die drei Arten von Vor- und Nachbedingungen eines Aufrufs auf ein Unterprogramm

V_a ist die Nachbedingung des vor dem Aufruf ausgeführten Teils des aufrufenden Programms; ihre Wahrheit wird vom vorherigen Teil des aufrufenden Programms sichergestellt. V_s ist die spezialisierte Vorbedingung des Unterprogramms. V_u ist die tatsächlich erforderliche — d.h. eine umfassende — Vorbedingung des Unterprogramms. Sind der aufrufende Programmteil und das Unterprogramm korrekt (erfüllen sie die Schnittstellenspezifikation), dann gilt, daß $(V_a \Rightarrow V_s) \wedge (V_s \Rightarrow V_u)$.

P_u ist die tatsächlich vom Unterprogramm erfüllte Nachbedingung. P_s ist die spezialisierte Nachbedingung des Unterprogramms. P_a ist eine umfassende Vorbedingung des nachfolgenden Teils des aufrufenden Programms. Sind der aufrufende Programmteil und das Unterprogramm korrekt, dann gilt, daß $(P_u \Rightarrow P_s) \wedge (P_s \Rightarrow P_a)$.

Bei der Konstruktion und im Korrektheitsbeweis des aufrufenden Programmteils dürfen nur V_a , V_s , P_s und P_a berücksichtigt werden bzw. vorkommen. Bei der Konstruktion und im Korrektheitsbeweis des Unterprogramms dürfen nur V_s , V_u , P_u und P_s berücksichtigt werden bzw. vorkommen. Die spezialisierten Vor- und Nachbedingungen V_s und P_s bilden also die Schnittstelle — und die Nahtstelle zugleich — zwischen dem aufrufenden Programmteil und dem aufgerufenen Unterprogramm.

4.5 Anforderungen an die Dokumentation eines Unterprogramms

Die ingenieurmäßige Dokumentation eines Gegenstands bzw. Systems soll zwei Aspekte davon festhalten: seine *extern* beobachtbaren Eigenschaften, die für die beabsichtigten Einsatzzwecke von Belang sind (siehe Abschnitt 4.4), und Details seiner *internen* Struktur und Zusammensetzung. Die Beschreibung der internen Struktur soll eine analytische rech-

nerische Überprüfung ermöglichen, ob das Objekt die externe Spezifikation erfüllt. Im wesentlichen geht es also darum, die Schnittstelle zwischen dem fraglichen Gegenstand und seiner Umgebung ausreichend präzise und vollständig festzuhalten sowie glaubhaft zu machen, daß sich der Gegenstand gemäß der Schnittstellenspezifikation verhalten wird.

Zur externen Spezifikation eines Unterprogramms (oder eines sonstigen Programmsegments) und folglich zu seiner Dokumentation gehören ggf. Hinweise auf relevante Dateninvarianten. Die Bedingungen selbst, woraus die jeweilige Dateninvariante besteht, sollten zweckmäßigerweise in einem getrennten Dokumentationsabschnitt festgehalten werden, denn eine Dateninvariante trifft für mehrere (oft sehr viele) Unterprogramme und Programmteile zu. Ähnliches gilt prinzipiell für Programminvarianten, obwohl diese nicht selten in die Spezifikationen der betroffenen Unterprogramme und Programmteile eingearbeitet werden.

Wenn das Unterprogramm durch Aufruf mit formaler Parameterübergabe ausgeführt werden soll, soll sich die Korrektheitsaussage explizit darauf beziehen. In der Dokumentation sollen alle zu berücksichtigenden Einschränkungen aufgeführt werden, siehe Abschnitt 3.3.4.4.

Über die interne Struktur und Zusammensetzung des Unterprogramms soll die Dokumentation wie oben erwähnt die für eine Überprüfung der Erfüllung der externen Spezifikation erforderlichen Angaben enthalten. Diese sind auch die wesentlichen Konstruktionsentscheidungen, -nebenprodukte und -ergebnisse. Außer der externen Spezifikation (siehe Abschnitt 4.4) und dem Programmcode des konstruierten Unterprogramms selbst sind die folgenden vom Unterprogrammentwickler getroffenen Konstruktionsentscheidungen für die Korrektheitsbeweissführung von Belang (vgl. die Tabelle in Abschnitt 4.1):

- für jede Schleife die Schleifeninvariante (für die Anwendung von Beweisregel W2 benötigt) und
- zumindest die nicht offensichtlichen, im Korrektheitsbeweis benötigten Zwischenbedingungen (siehe Abschnitt 4.1), auch verschärfte bzw. geschwächte Versionen davon (für die Anwendung der Beweisregeln F1 und U1 (release-Anweisung) benötigt).

Ein ggf. verlängerter Korrektheitsbeweis soll in einer möglichst übersichtlich gegliederten und leicht nachprüfaren Form dokumentiert werden. Die Struktur des Korrektheitsbeweises widerspiegelt die Konstruktionsvorgehensweise sowie die Struktur des Unterprogramms selbst.

Im Korrektheitsbeweis werden die Spezifikationen aufgerufener Unterprogramme (bzw. sonst verwendeter, z.B. eingebetteter Programmsegmente) als bereits bewiesene Sätze bzw. Lemmata betrachtet, siehe die Beweisregeln U2 und U3 in Abschnitt 3.2.4. Hinsichtlich Rekursion siehe Abschnitt 3.3.4.6 und Anhang 4.

Wie in Abschnitt 4.4 erläutert, darf der Konstrukteur eines Programmteils, der das fragliche Unterprogramm aufruft, nur von in der Schnittstellenspezifikation enthaltenen Angaben ausgehen. Dies gilt sowohl bei der Konstruktion als auch bei der Korrektheitsbeweissführung. Deshalb soll die Dokumentation über die interne Struktur des Unterprogramms nur einem beschränkten Leserkreis zugänglich sein. Aus diesem Grund soll die Gliederung der Dokumentation die Trennung der extern- und der internbezogenen Dokumentationssteile voneinander ermöglichen.

Anhang 2 zeigt eine mögliche Gestaltung der Dokumentation eines Unterprogramms, die den oben aufgeführten Anforderungen weitgehend genügt. Abschnitt 1 davon enthält die externe Spezifikation und soll allgemein zugänglich sein. Alles andere (Abschnitte 2-4 und die Anlage) enthält Angaben über interne Einzelheiten und soll nur dem Konstrukteur

und den Prüfern zur Verfügung stehen. Wenn kein formaler Korrektheitsbeweis verlangt worden wäre, würden in Anhang 2 die Abschnitte 3, 4.3.4, 4.3.5 und Anlage 1 fehlen. [Baber; 1990 (*Fehlerfreie Programmierung*), Abschnitt 3.5] enthält eine kurze Übersicht über die Anforderungen an die Dokumentation eines Unterprogramms.

5. Bedeutung der Korrektheitsbeweissführung für die Konstruktionsänderung in instabiler Umgebung

Für Änderungen an fertiggestellten und bereits im Einsatz befindlichen Programmen gibt es verschiedene Gründe und Ziele, wovon die folgenden die wesentlichen sind (vgl. [Baber; 1988, "Software + Wartung = Widerspruch ..."], [Bustard] und [Turski; 1982]):

- (1) Anforderungen auf neue Leistungen oder auf eine andere Funktionsweise, auch Korrektur einer als nicht geeignet festgestellten Spezifikation (Änderung der externen Spezifikation)

- (2) Veränderungen der Rechnersystemumgebung, z.B. der Hardware oder der Systemsoftware (Änderung der Spezifikation bestimmter Bestandteile des Programmsystems bzw. beanspruchter Ressourcen; Änderung der internen Spezifikation)

- (3) Verbesserung der Effizienz (z.B. Erhöhung der Laufgeschwindigkeit, Verringerung des Speicherbedarfs) bei unveränderter Spezifikation (Verbesserung von in der Spezifikation nicht festgelegten Eigenschaften des fraglichen Programms)

- (4) Korrektur von Fehlern im fraglichen Programm

In den Fällen (1) und (2) oben handelt es sich um Anpassungen an Veränderungen der externen (zum Klienten, Bedienten hin) bzw. der internen (zum Lieferanten, Bedienten hin) Umgebung des fraglichen Programms. Im Falle (2) oben kann es sich entweder um eine Änderung der Art und Weise, wie die (gleiche) Leistung beansprucht werden kann (Änderung der Form), oder um eine Änderung der Leistung bzw. des Leistungsumfangs selbst (Änderung der Substanz) handeln. In den Fällen (1) und (2) handelt es sich in einem gewissen Sinne um die Konstruktion eines neuen Programms für eine neue Aufgabe — die jedoch der alten Aufgabe ähnlich gelagert ist und für die viele Teile des alten Programms und des Korrektheitsbeweises noch geeignet sind oder durch geringfügige Änderungen geeignet gestaltet werden können.

Im Falle (3) oben handelt es sich um die nachträgliche Änderung einer Konstruktionsentscheidung. Die revidierte Konstruktionsentscheidung stand, zumindest prinzipiell, zur Konstruktionszeit zur Verfügung; eventuell wurde sie sogar damals in Erwägung gezogen und für den späteren Einsatz vorgesehen.

Im Falle (4) oben handelt es sich schlicht um die Behebung bzw. Beseitigung von Konstruktionsfehlern.

Die Programmkorrekttheitsbeweissführung und eine darauf basierte Konstruktionsvorgehensweise (z.B. wie im Kapitel 4 erläutert) haben in den verschiedenen oben aufgeführten Fällen unterschiedliche Bedeutung. In den Fällen (1) und (2) grenzen die Vor- und Nachbedingungen der verschiedenen Programmteile die Auswirkungen der Änderung ein (siehe Abschnitt 5.2). Sie führen den Konstrukteur zu den Programmteilen hin, die geändert werden müssen, und verdeutlichen, welche Programmteile nicht überprüft werden müssen. Typischerweise wird bei einer Änderung der externen Spezifikation (Fall 1 oben) die Nachprüfung auf die Notwendigkeit einer Programmänderung überwiegend von oben nach unten ("top-down") und bei einer Änderung der Spezifikation eines Bestandteils auf der unteren hierarchischen Ebene (Fall 2) von unten nach oben ("bottom-up") erfolgen. Im Gegensatz dazu vertritt [Wirth; 1972, S. 121] die Meinung, bei der Neukonstruktion sei

die "top-down" Vorgehensweise dominant und bei der Adaption eines Programms an eine leicht veränderte Zielsetzung überwiege die "bottom-up" Vorgehensweise.

Bei einer Änderung einer nicht spezifizierten (oder innerhalb eines vorgegebenen Rahmens offengelassenen) Eigenschaft (Fall 3 oben) stellen die betreffenden Vor- und Nachbedingungen sicher, daß sich die Änderung auf andere Programmteile nicht auswirkt. Dabei muß bemerkt werden, daß Mischfälle vorkommen können, wenn man z.B. die Spezifikation eines Unterprogramms ändert (vgl. Fall 2), um die gezielte Verbesserung zu erreichen.

Wenn der Fall 4 oben (Fehlerkorrektur) eintritt, geht es darum, den Programmcode in Übereinstimmung mit den betreffenden Vor- und Nachbedingungen zu bringen. Aber durch die Programmkorrektheitsbeweissführung bzw. eine darauf basierte Konstruktionsvorgehensweise soll die Häufigkeit von Programmfehlern so stark reduziert werden, daß der Fall 4 von untergeordneter Bedeutung (idealerweise sogar von keiner Bedeutung mehr) sein soll. Eine gewisse Unterstützung dieses Leitsatzes enthält eine auf einer umfangreichen Untersuchung basierende Feststellung von [Perry; S. 63]: Durch formale Methoden vermeidbare Fehler verursachen einen verhältnismäßig hohen Korrekturaufwand.

Der Zusammenhang zwischen der Programmkorrektheitsbeweissführung und formalen Methoden in diesem Sinne einerseits und Konstruktionsänderung und Softwarewartung andererseits scheint kaum erforscht zu sein. Eine entsprechende Literatursuche ergab wenige — und dann nur marginal — relevante Artikel; keiner befaßte sich direkt mit dem Schnitt dieser Teilgebiete. Falls und wenn Korrektheitsbeweissführungstechniken in der Praxis breiter eingesetzt werden, wird ein Bedarf für solche Forschung entstehen.

5.1 Spezifikationen von Unterprogrammen

Bei der Betrachtung der Bedeutung von Korrektheitsbeweissführungskonzepten für die Änderung eines Programms sollte man zwischen Programmen, für die keine Vor- und Nachbedingungen für die internen Schnittstellen vorliegen, und denjenigen, für die solche Vor- und Nachbedingungen vom Konstrukteur festgelegt worden sind, unterscheiden. Dieser Unterschied entspricht dem auf den klassischen technischen Gebieten bekannten Unterschied zwischen der traditionellen Handfertigung und der ingenieurmäßigen Konstruktion. Die handwerkliche Einzelfertigung kam ohne interne Spezifikationen und mit verhältnismäßig groben Toleranzen aus, weil jede Komponente an ihre unmittelbare Umgebung angepaßt wurde. Die hinsichtlich der Vorplanung und der engeren Toleranzen der einzelnen Komponenten aufwendigere ingenieurmäßige Konstruktion führt jedoch insgesamt zu wirtschaftlicheren Ergebnissen. Die vielen Schnittstellen müssen nicht einzeln bearbeitet — und nachgearbeitet — werden. Stattdessen werden sie (durch die Festlegung von Schnittstellenspezifikationen) in Standardklassen eingeordnet. Für jede Klasse wird nur eine Schnittstellenspezifikation definiert und für alle einzelnen Schnittstellen der Klasse verwendet. Bei einer nachträglichen Konstruktionsänderung wirkt sich diese Vorgehensweise effizienzerhöhend aus, weil die Anzahl der zu überprüfenden Schnittstellen (z.B. Unterprogrammaufrufe) verringert wird. Auch wird dadurch die Wiederverwendung einer Komponente in neuen Konstruktionen gefördert.

Um die Bedeutung von internen Spezifikationen für die Konstruktionsänderung klarer zu sehen, betrachte zwei Aufrufe auf dasselbe Unterprogramm, für das keine einheitliche Spezifikation vereinbart worden ist (vgl. Abschnitt 4.4.):

```
...
{Va1}
call S1
{Pa1}
...
{Va2}
call S1
{Pa2}
...
```

Eine Voraussetzung für die Korrektheit des Programms ist, daß $\{Va1\} S1 \{Pa1\}$ strikt und $\{Va2\} S1 \{Pa2\}$ strikt gelten. Diese Voraussetzung ist auch eine Randbedingung für eine Konstruktionsänderung von S1, wenn die Änderung von S1 keine Änderungen der aufrufenden Stellen mit sich ziehen soll. (Wird das Unterprogramm von mehreren Stellen aufgerufen, verschärft sich diese Randbedingung entsprechend: $\{Va1\} S1 \{Pa1\}$ strikt \wedge $\{Va2\} S1 \{Pa2\}$ strikt \wedge $\{Va3\} S1 \{Pa3\}$ strikt \wedge ...). Eine gemeinsame Spezifikation für S1, z.B. $\{Vs\} S1 \{Ps\}$ strikt, muß nicht unbedingt erfüllt sein (vgl. die Abbildungen am Ende des Abschnitts 5.2). Beschränkt man sich auf Nachbedingungen ohne Bezug auf vorherige Variablenwerte (vgl. Abschnitt 3.1.6), kann es sogar der Fall sein, daß es für die vorgegebenen Bedingungen Va1, Va2, Pa1, Pa2 und das vorgegebene Unterprogramm S1 keine Bedingungen Vs und Ps gibt, für die $[Va1 \Rightarrow Vs]$, $[Va2 \Rightarrow Vs]$, $[Ps \Rightarrow Pa1]$, $[Ps \Rightarrow Pa2]$ und $\{Vs\} S1 \{Ps\}$ strikt alle gelten. Sind z.B. Pa1 und Pa2 disjunktiv, dann kommt für Ps nur die logische Konstante falsch (bzw. die leere Teilmenge von \mathbb{D}) in Frage. Für Vs kommen nur $Va1 \vee Va2$ und schwächere Bedingungen in Frage. Aber $\{Vs\} S1 \{falsch\}$ strikt kann nur dann gelten, wenn Vs die logische Konstante falsch (bzw. die leere Teilmenge von \mathbb{D}) ist. Voraussetzung dafür ist, daß $Va1 \vee Va2 = \text{falsch}$, welches i.a. nicht zutrifft.

Läßt man aber Nachbedingungen mit Bezug auf vorherige Variablenwerte zu, dann kann eine gemeinsame Spezifikation immer aufgestellt werden, denn die Aussage

$$\{Va1\} S1 \{Pa1\} \text{ strikt } \wedge \{Va2\} S1 \{Pa2\} \text{ strikt } \wedge \dots$$

ist äquivalent zur Aussage

$$\{Va1 \vee Va2 \vee \dots\} S1 \{ (Va1' \Rightarrow Pa1) \wedge (Va2' \Rightarrow Pa2) \wedge \dots \} \text{ strikt}$$

Mangels einer gemeinsamen Spezifikation $\{Vs\} S1 \{Ps\}$ strikt muß bei einer Konstruktionsänderung von S1 darauf geachtet werden, daß die gesamte oben aufgeführte Randbedingung eingehalten werden muß. Das bedeutet, daß die neue Version von S1 auf die Einhaltung der Vor- und Nachbedingungen für *jeden* Aufruf — d.h. auf die Verträglichkeit mit *jedem* Aufruf — einzeln geprüft werden muß.

Die oben geschilderte, beim Fehlen einer festgelegten gemeinsamen Spezifikation auftretende Änderungsproblematik hat mathematisch gesehen ihre Ursache in der Auffassung einer Programmanweisung als eine Funktion, die eine Datenumgebung in eine Datenumgebung abbildet. Als natürliche und in der Mathematik übliche Erweiterung dieser Auffassung wird diese Funktion auch so betrachtet, daß sie eine Teilmenge von \mathbb{D} in eine Teilmenge von \mathbb{D} abbildet und umgekehrt (Bildmenge bzw. Urbild, Nach- bzw. Vorbedingung). Genauso wie eine Funktion f sich nicht durch nur ein Paar Elemente (x, f.x) charakterisieren läßt, läßt sich eine Programmanweisung S nicht durch nur ein Paar Bedin-

ungen (eine Vor- und eine Nachbedingung) charakterisieren. Wird ein Unterprogramm n mal in einem Programm aufgerufen, dann können n Paare von Vor- und Nachbedingungen (n Punkte der Funktion) erforderlich sein, um die Anforderungen des Programms an das Unterprogramm zu umfassen. Durch die Vereinbarung einer geeigneten zusätzlichen Einschränkung, etwa einer gemeinsamen Spezifikation $\{Vs\} S1 \{Ps\}$ strikt, wird effektiv festgelegt, daß im Programm nur von denjenigen Eigenschaften des Unterprogramms, die sich durch das Paar (Vs, Ps) charakterisieren lassen, Gebrauch gemacht werden darf bzw. daß nur solche Eigenschaften unterstellt werden dürfen.

Insofern schränkt die Festlegung einer einzigen Spezifikation für ein an mehreren Stellen aufgerufenes Unterprogramm seinen Einsatz und seine Verwendung und Ausnutzung theoretisch und prinzipiell ein. In der Praxis ist diese Einschränkung typischerweise kaum hindernd. Sie hat vielmehr eine vereinfachende und praktisch nützliche Wirkung bei der Neukonstruktion, bei der Korrektheitsbeweismführung und bei der nachträglichen Konstruktionsänderung.

Ist eine gemeinsame Spezifikation für $S1$, z.B. $\{Vs\} S1 \{Ps\}$ strikt, vom Konstrukteur festgelegt worden und ist das Programm bezüglich dieser Spezifikation korrekt, dann gelten die Bedingungen $[Va1 \Rightarrow Vs]$, $[Va2 \Rightarrow Vs]$, $[Ps \Rightarrow Pa1]$, $[Ps \Rightarrow Pa2]$ und $\{Vs\} S1 \{Ps\}$ strikt. Falls eine geänderte Version $S2$ des Unterprogramms $S1$ die gleiche Spezifikation erfüllt (d.h., es gilt $\{Vs\} S2 \{Ps\}$ strikt), sind Überprüfungen der verschiedenen Aufrufe auf $S1$ (bzw. $S2$) auf Korrektheit nicht erforderlich, denn aus $[Va1 \Rightarrow Vs] \wedge [Va2 \Rightarrow Vs] \wedge [Ps \Rightarrow Pa1] \wedge [Ps \Rightarrow Pa2] \wedge \{Vs\} S2 \{Ps\}$ strikt folgt gemäß der Beweisregel B1, daß $\{Va1\} S2 \{Pa1\}$ strikt $\wedge \{Va2\} S2 \{Pa2\}$ strikt. Ist also eine gemeinsame Spezifikation vorhanden, dann muß nur sie als Randbedingung für eine Konstruktionsänderung des Unterprogramms berücksichtigt werden. Die Vor- und Nachbedingungen der einzelnen Aufrufe sind in diesem Falle für eine Konstruktionsänderung des Unterprogramms nicht maßgebend und können außer Acht gelassen werden.

Sind gar keine Vor- und Nachbedingungen (weder Va , Vs , Vu , Pa , Ps noch Pu , siehe die letzten Absätze und die Abbildung in Abschnitt 4.4) vorgegeben, dann muß der Änderungskonstrukteur entweder geeignete Vor- und Nachbedingungen ermitteln oder die folgenden besonders strengen Randbedingungen beachten. Ohne Kenntnisse der Vor- und Nachbedingungen ist die einzige sichere Änderungsstrategie, ein Programmsegment $S1$ durch ein anderes $S2$ nur dann zu ersetzen, wenn $S2.d = S1.d$ für jede Datenumgebung d aus dem Definitionsbereich von $S1$. Mit anderen Worten: (1) die Funktion $S2$ beschränkt auf dem Definitionsbereich von $S1$ muß gleich $S1$ sein, und (2) der Definitionsbereich von $S2$ muß eine Obermenge des Definitionsbereichs von $S1$ sein:

$$S2|S1^{-1}.D = S1$$

$$S1^{-1}.D \subseteq S2^{-1}.D$$

Diese Anforderung kann auf verschiedene andere Weise formuliert werden, z.B.

$$(A V, P : V \subseteq D \wedge P \subseteq D : \{V\} S1 \{P\} \text{ strikt} \Rightarrow \{V\} S2 \{P\} \text{ strikt})$$

$$\text{oder } (A V, P : V \subseteq D \wedge P \subseteq D : V \subseteq S1^{-1}.P \Rightarrow V \subseteq S2^{-1}.P)$$

$$\text{oder } (A P : P \subseteq D : S1^{-1}.P \subseteq S2^{-1}.P)$$

Diese Beziehung zwischen $S1$ und $S2$ wird auch $S1 \subseteq S2$ geschrieben. Ein Programmsegment $S2$, das in dieser Beziehung zu $S1$ steht, ist allgemeiner als $S1$. Es kann immer an

die Stelle von $S1$ eingesetzt werden (es sei denn, ein undefiniertes Ergebnis $S1.d$ bzw. $S2.d$ ist beabsichtigt und gewollt). Siehe auch [Jeng] und [Naumann].

Die an den Änderungskonstrukteur für $S2$ gestellte Forderung

$$S1 \subseteq S2 \quad [\text{Randbedingung für Änderung ohne Vor- und Nachbedingungen}]$$

ist stärker als die Forderung

$$\{Va1\} S2 \{Pa1\} \text{ strikt} \wedge \{Va2\} S2 \{Pa2\} \text{ strikt} \wedge \dots \{Van\} S2 \{Pan\} \text{ strikt} \\ [\text{Randbedingung für Änderung ohne gemeinsame Spezifikation (Vs, Ps)}]$$

die wiederum stärker ist als die Forderung

$$\{Vs\} S2 \{Ps\} \text{ strikt} \quad [\text{Randbedingung für Änderung mit Spezifikation}]$$

für gegebene $S1$, $Va1, \dots, Van$, $Pa1, \dots, Pan$, Vs und Ps wie oben angegeben. Deshalb ist die Änderungsaufgabe am leichtesten, wenn eine gemeinsame Schnittstellenspezifikation (Vs, Ps) vorgegeben ist. Die Änderungsaufgabe ist schwieriger, wenn Vor- und Nachbedingungen für die verschiedenen Verwendungen von $S1$ bzw. $S2$ (z.B. Aufrufe darauf), aber keine gemeinsame Schnittstellenspezifikation, vorgegeben sind. Die Änderungsaufgabe ist am schwierigsten, wenn gar keine Vor- und Nachbedingungen festgelegt worden sind.

5.2 Eingrenzung der Auswirkungen einer Änderung

Der in der Praxis bei Konstruktionsänderungen (oft "Wartung" genannt) entstehende Aufwand ist zu einem erheblichen Teil darauf zurückzuführen, daß viele Stellen im fraglichen Programm in der Regel auf Auswirkungen einer Änderung geprüft werden müssen. In einigen davon müssen sekundäre Änderungen vorgenommen werden. Die Prüfung jeder solchen Stelle verursacht einen gewissen Aufwand, auch wenn sich keine Änderung als notwendig ergibt. Darüber hinaus verursacht jede notwendige sekundäre Änderung einen zusätzlichen Aufwand. Dabei ist zu bemerken, daß die zu prüfenden und zu ändernden Stellen nicht immer in einer unmittelbaren Beziehung zur primären Änderungsstelle stehen, sondern nur über Umwege ermittelt werden können. Festgelegte Vor- und Nachbedingungen, vor allem für die aufgerufenen Unterprogramme, können die Änderungsspanne eingrenzen (siehe [Baber; 1988, "Software + Wartung = Widerspruch ..."]) und dadurch den Prüf- und Änderungsaufwand verringern. Solche Eingrenzungsmöglichkeiten werden unten näher erläutert.

In Abschnitt 4.4 wurden die drei Arten von Vor- und Nachbedingungen eines Aufrufs auf ein Unterprogramm vorgestellt und in einem Diagramm abgebildet. Die Differenzen zwischen bestimmten Vor- bzw. Nachbedingungen bilden Puffer, die in einem gewissen Umfang nachträgliche Konstruktionsänderungen (und Programmierungsfehler, siehe unten) auffangen können. Die Differenzmenge $Vs \setminus Va$ (das Komplement von Va bezüglich Vs , auch $Vs \setminus Va$ geschrieben) z.B. stellt einen Puffer dar, der gewisse Änderungen der aufrufenden Stelle, die Va schwächen, auffangen kann, ohne daß das Unterprogramm geändert werden muß. Alternativ kann dieser Puffer eine gewisse durch eine Änderung der Schnittstellenspezifikation verursachte Stärkung von Vs auffangen, ohne daß die aufrufende Stelle des Programms entsprechend geändert werden muß. Die folgende Tabelle faßt diese Puffer und ihre Verwendungsmöglichkeiten zusammen:

Differenzmenge	Puffer für Änderung der/des	Änderungsart
Vs-Va	aufzufindende Stelle Spezifikation des Unterprogramms	Schwächung von Va Stärkung von Vs
Vu-Vs	Unterprogramm Spezifikation des Unterprogramms	Stärkung von Vu Schwächung von Vs
Ps-Pu	Unterprogramm Spezifikation des Unterprogramms	Schwächung von Pu Stärkung von Ps
Pa-Ps	aufzufindende Stelle Spezifikation des Unterprogramms	Stärkung von Pa Schwächung von Ps

Die anderen monotonen Änderungsarten (Änderung der aufrufenden Stelle mit Stärkung von Va und/oder Schwächung von Pa sowie Änderung des Unterprogramms mit Schwächung von Vu und/oder Stärkung von Pu) sind gemäß der Beweisregel B1 immer und in unbegrenztem Umfang zulässig.

Wenn eine Konstruktionsänderung an einem Unterprogramm, an einer aufrufenden Stelle oder an der Spezifikation eines Unterprogramms vorgenommen wird, müssen je nach Art der Änderung eventuell andere Stellen untersucht und ggf. geändert werden. Dafür gibt die folgende Tabelle Leitlinien an.

Änderungsgegenstand	Falls	sind die folgenden weiteren Maßnahmen erforderlich.
Unterprogramm (Vu in Vu2, Pu in Pu2 ändern)	$[Vs \Rightarrow Vu2] \wedge [Pu2 \Rightarrow Ps]$	keine
	sonst	Spezifikation so ändern, daß $[Vs2 \Rightarrow Vu2] \wedge [Pu2 \Rightarrow Ps2]$ [*]
Aufrufende Stelle (Va in Va2, Pa in Pa2 ändern)	$[Va2 \Rightarrow Vs] \wedge [Ps \Rightarrow Pa2]$	keine
	sonst	Spezifikation so ändern, daß $[Va2 \Rightarrow Vs2] \wedge [Ps2 \Rightarrow Pa2]$ [*]
Spezifikation (Vs in Vs2, Ps in Ps2 ändern)	$[Vs \Rightarrow Vs2] \wedge [Ps2 \Rightarrow Ps]$ $\wedge [Vs2 \Rightarrow Vu] \wedge [Pu \Rightarrow Ps2]$	keine
	$[Vs \Rightarrow Vs2] \wedge [Ps2 \Rightarrow Ps]$ $\wedge \neg([Vs2 \Rightarrow Vu] \wedge [Pu \Rightarrow Ps2])$	Unterprogramm (Vu, Pu) so ändern, daß $[Vs2 \Rightarrow Vu2] \wedge [Pu2 \Rightarrow Ps2]$ [*]
	$\neg([Vs \Rightarrow Vs2] \wedge [Ps2 \Rightarrow Ps])$ $\wedge [Vs2 \Rightarrow Vu] \wedge [Pu \Rightarrow Ps2]$	Jede aufrufende Stelle prüfen, ob $[Va \Rightarrow Vs2] \wedge [Ps2 \Rightarrow Pa]$. Falls ja, ist keine weitere Maßnahme erforderlich. Falls nein, aufrufende Stelle so ändern, daß $[Va2 \Rightarrow Vs2] \wedge [Ps2 \Rightarrow Pa2]$. [*]
	$\neg([Vs \Rightarrow Vs2] \wedge [Ps2 \Rightarrow Ps])$ $\wedge \neg([Vs2 \Rightarrow Vu] \wedge [Pu \Rightarrow Ps2])$	Unterprogramm wie oben ändern und jede aufrufende Stelle wie oben prüfen und ggf. ändern. [*]

* = oder die fragliche Änderung verwerfen.

In Worten bedeutet diese Tabelle folgendes: Wird ein Unterprogramm derart geändert, daß die neue Version seine ursprüngliche und unveränderte Schnittstellenspezifikation erfüllt, dann ist keine weitere Prüfung oder Änderung erforderlich. Wird die Vorbedingung Vu geschwächt und/oder die Nachbedingung Pu gestärkt, dann wird dies der Fall sein. Erfüllt die neue Version des Unterprogramms die Schnittstellenspezifikation nicht, dann muß die Spezifikation entsprechend geändert werden — mit allen sich daraus ergebenden Folgen, siehe die Tabelle oben und die Erläuterungen unten. Alternativ kann die Änderung des Unterprogramms verworfen werden.

Wird eine aufrufende Stelle derart geändert, daß die neue Version die ursprüngliche und unveränderte Schnittstellenspezifikation erfüllt, dann ist keine weitere Prüfung oder Änderung erforderlich. Wird die Vorbedingung Va gestärkt und/oder die Nachbedingung Pa geschwächt, dann wird dies der Fall sein. Erfüllt die neue Version der aufrufenden Stelle die Schnittstellenspezifikation nicht, dann muß die Spezifikation entsprechend geändert werden — mit allen sich daraus ergebenden Folgen, siehe die Tabelle oben und die Erläuterungen unten. Alternativ kann die Änderung der aufrufenden Stelle verworfen werden.

Wird eine Schnittstellenspezifikation derart geändert, daß die Vorbedingung V_s geschwächt und/oder die Nachbedingung P_s gestärkt ($[V_s \Rightarrow V_{s2}] \wedge [P_{s2} \Rightarrow P_s]$) werden, dann werden alle aufrufenden Stellen auch die neue Spezifikation erfüllen; sie müssen nicht überprüft werden. Anderenfalls muß jede aufrufende Stelle auf Einhaltung der neuen Spezifikation einzeln überprüft ($[V_a \Rightarrow V_{s2}] \wedge [P_{s2} \Rightarrow P_a]$?) und ggf. geändert werden. Darüber hinaus muß bei einer Spezifikationsänderung geprüft werden, ob das betroffene Unterprogramm auch die neue Spezifikation erfüllt ($[V_{s2} \Rightarrow V_u] \wedge [P_u \Rightarrow P_{s2}]$?). Wenn nicht, muß es entsprechend geändert werden. Alternativ kann die fragliche Änderung der Schnittstellenspezifikation verworfen werden. (Ende der Erläuterung der Tabelle)

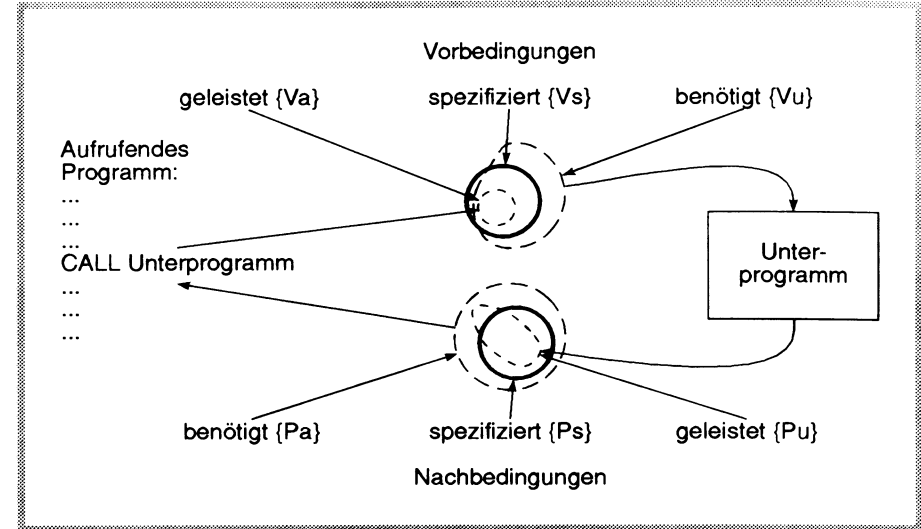
Die möglichen Auswirkungen einer sekundären Änderung müssen iterativ weiter verfolgt werden, bis keine Änderungen mehr erforderlich sind.

Ein Programmsegment in einer längeren Folge von Programmanweisungen kann entsprechend behandelt werden. Effektiv ist es ein Unterprogramm, das an nur einer Stelle des Programms aufgerufen wird.

Die verschiedenen Arten von Vor- und Nachbedingungen oben ermöglichen es, an bestimmten Stellen die Untersuchung nach möglichen Auswirkungen einer primären Änderung abzubrechen. Auf diese Weise können Zweige und Unterbäume in der Programmhierarchie ausgeschlossen werden, die im Falle eines Programms ohne Vor- und Nachbedingungen bzw. Schnittstellenspezifikationen ausführlich untersucht werden müßten. Besonders unveränderte Schnittstellenspezifikationen sind in dieser Hinsicht von Bedeutung, denn sie grenzen die mögliche Änderungsspanne stark ein und machen jeweils die Überprüfung einer Mehrzahl von aufrufenden Stellen überflüssig.

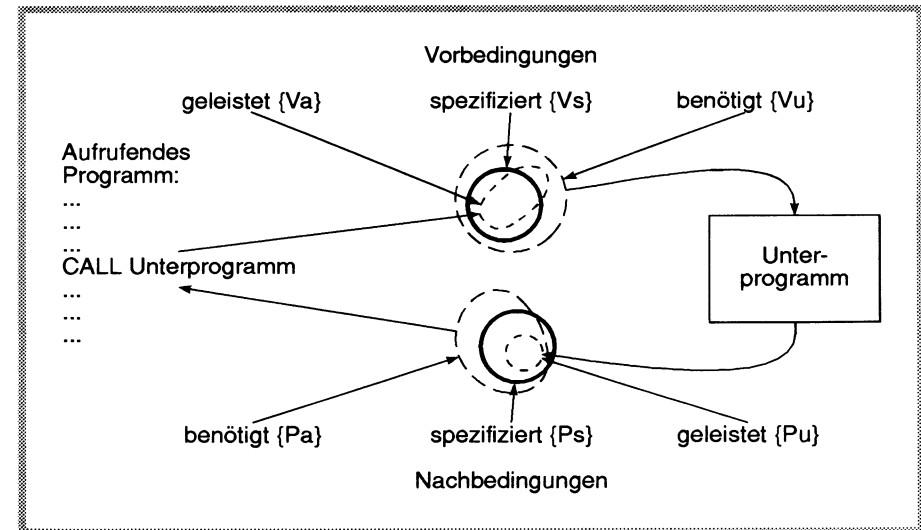
Falls die ursprüngliche Version des Programms korrekt war und alle Schnittstellenspezifikationen einhielt, wird die nach den oben aufgeführten Leitlinien geänderte Version auch korrekt sein und alle Schnittstellenspezifikationen einhalten.

Die Voraussetzung, daß das jeweilige Unterprogramm und alle Aufrufe darauf die fragliche Schnittstellenspezifikation einhalten, ist für die Anwendung der Änderungsleitlinien oben kritisch. Es reicht nicht aus, daß das Programm bzw. alle Aufrufe auf Unterprogramme sonst korrekt sind. Die folgenden Abbildungen veranschaulichen Situationen, in denen gilt, daß $\{V_a\}$ Unterprogramm $\{P_a\}$ — d.h., das Programm ist in dem Sinne korrekt, daß der Korrektheitsbeweis vollständig durchgeführt werden kann — obwohl die Schnittstellenspezifikation verletzt wird. Im ersten Beispiel verletzt das Unterprogramm die Schnittstellenspezifikation (V_s, P_s). Vgl. die Abbildung in Abschnitt 4.4.



Korrektes Funktionieren trotz fehlerhaftem Unterprogramm

Im zweiten Beispiel verletzt der Aufruf die Schnittstellenspezifikation (V_s, P_s):



Korrektes Funktionieren trotz fehlerhaftem Aufruf

Werden die Änderungsleitlinien oben auf eine solche Situation angewendet, kann ein fehlerhaftes Programm entstehen.

5.3 Konstruktionsleitlinien für die "Änderungsfreundlichkeit"

Aus den Betrachtungen und Überlegungen oben ergeben sich einige Konstruktionsleitlinien für die "Änderungsfreundlichkeit":

- Der Konstrukteur des gesamten Programms sollte grundsätzlich eine Spezifikation für jedes Unterprogramm und jedes Programmsegment, das eine in sich abgeschlossene Leistung erbringt bzw. Einheit bildet, festlegen. Jede Schnittstelle zwischen Programmsegmenten, die von verschiedenen Projektmitarbeitern erstellt werden sollen, sollte ebenfalls spezifiziert werden.
- Der Konstrukteur eines Unterprogramms sollte sich fragen, ob eine Schwächung der spezifizierten Vorbedingung oder eine Stärkung der spezifizierten Nachbedingung sinnvoll sein könnte. Wenn ja, sollte er von Anfang an das Unterprogramm für die schwächere Vor- und die stärkere Nachbedingung konstruieren und das in der Dokumentation des Unterprogramms festhalten. Ein klassisches Beispiel hiervon ist die Zulassung leerer Felder, leerer Zeichenfolgen u.ä., obwohl die Spezifikation nur ein nicht leeres solches Objekt vorsieht. Auch sollte der Konstrukteur es in Erwägung ziehen, nicht sinnvolle Variablenwerte ausdrücklich zuzulassen, sie auf eine naheliegende Weise zu verarbeiten und dementsprechend die Nachbedingung zu stärken.
- Der Konstrukteur eines Unterprogramms sollte sich fragen, welche künftige Änderungen voraussichtlich verlangt werden können, und entweder diese von Anfang an berücksichtigen (wie oben) oder dafür zumindest spätere Einbaumöglichkeiten vorsehen.
- Der Konstrukteur einer aufrufenden Stelle sollte sich fragen, welche in der spezifizierten Vorbedingung vorhandenen Möglichkeiten bzw. Freiheiten an der fraglichen Aufrufstelle nicht gebraucht bzw. nicht nützlich sind und ggf. die vor dem Aufruf geleistete Vorbedingung V_a entsprechend stärker formulieren.
- Der Konstrukteur einer aufrufenden Stelle sollte sich fragen, welche in der spezifizierten Nachbedingung erfaßten Leistungen des Unterprogramms an der fraglichen Aufrufstelle nicht erforderlich, nicht nützlich bzw. unwesentlich sind und ggf. die benötigte Nachbedingung P_a entsprechend schwächer formulieren.
- Der Konstrukteur des gesamten Programms sollte Schnittstellen zu zusätzlichen Programmteilen vorsehen oder sogar einbauen, die erst künftig voraussichtlich gebraucht werden. Diese Konstruktionsstrategie wird in den klassischen Ingenieurfächern oft praktiziert, z.B. für zusätzliche Zubehörteile oder Produktvarianten.
- Alle oben erwähnten Vor- und Nachbedingungen sollten dokumentiert werden. Zur Dokumentation des gesamten Programms gehören die Schnittstellenspezifikationen (V_s , P_s). Zur Dokumentation der jeweiligen aufrufenden Stelle gehören die Vor- und Nachbedingungen V_a und P_a . Zur internen Dokumentation des jeweiligen Unterprogramms gehören die Vor- und Nachbedingungen V_u und P_u .

Dabei sollte der Konstrukteur darauf achten, daß die oben angegebenen Maßnahmen zu einer Verringerung und nicht zu einer Erhöhung des zu erwartenden Gesamtaufwands führen, denn die Wirtschaftlichkeit, die ein wesentlicher Aspekt jeder ingenieurmäßigen Tätigkeit ist, fordert "so viel wie nötig, so wenig wie möglich". Nicht selten ist jedoch

die allgemeinere Problemstellung — einer schwächeren Vor- und einer stärkeren Nachbedingung entsprechend — einfacher zu verwirklichen und insgesamt weniger aufwendig.

6. Anwendungsaufwand, Voraussetzungen und maschinelle Unterstützung für eine breitere Anwendungsakzeptanz

Wie in Abschnitt 2.2.1 bereits erwähnt wird der Einwand gegen die Anwendung formaler Methoden erhoben, daß sie zu aufwendig sei. Gelegentlich wird zwischen Lern- und Anwendungsaufwand unterschieden, ansonsten wird dieser Urteil pauschal ohne weitere Unterscheidung gefällt. Es müßte jedoch mindestens zwischen (1) dem Lernaufwand, (2) dem Aufwand für die Programmverifikation und (3) dem Aufwand für die Programmkonstruktion unterschieden werden. Wenn nämlich der Lernaufwand — der wirtschaftlich eigentlich als eine Investition, nicht als ein Aufwand betrachtet werden sollte — dem ersten Anwendungsprojekt belastet wird, muß mit einem ungünstigen Verhältnis zwischen Kosten und Nutzen gerechnet werden, denn der Nutzen wird wegen mangelnder Anwendungserfahrung relativ gering sein, die Kosten werden jedoch in voller Höhe anfallen.

Bei der Beurteilung der Gesamtwirtschaftlichkeit müßte in jedem dieser drei Fälle der Aufwand mit einem geeigneten Maßstab sowie vor allem mit dem daraus entstehenden Nutzen verglichen werden. Entsprechende systematische Vergleichsstudien sind offensichtlich nicht durchgeführt worden, welches auch von mehreren Seiten bemängelt wird (siehe Abschnitt 2.2.1).

Dieser Unterscheidung dient die Gliederung dieses Kapitels: der Lernaufwand wird in Abschnitt 6.1, der Aufwand für die Programmverifikation in Abschnitt 6.2 und der Aufwand für die Konstruktion in Abschnitt 6.3 diskutiert. In der Konstruktionsänderungsphase entstehende Vorteile werden in Abschnitt 6.4 erörtert.

Bei der Betrachtung der Voraussetzungen für eine breitere Anwendungsakzeptanz muß man zwischen Maßnahmen, die den Praktiker in die Lage versetzen, Programmkorrektheitsbeweistechniken anwenden zu *können*, und denjenigen, die ihn in die Lage versetzen, sie anwenden zu *wollen*, unterscheiden. Befähigende Voraussetzungen sind in Abschnitt 3.4 (insbesondere in 3.4.1) besprochen worden; sie sind weitgehend mathematischer und technischer Natur.

Die motivierenden Voraussetzungen für eine breitere Anwendungsakzeptanz sind vielfältiger und komplexer als die befähigenden und zum größten Teil nichttechnischer Natur. Darunter sind gesetzliche Bestimmungen und sonstige juristische Richtlinien (die z.B. Haftung und Schadensersatz betreffen), Aspekte des geschäftlichen Umfelds (wie Kundenerwartungen bezüglich Fehlerfreiheit von Hard- sowie Software, Wettbewerbsdruck u.ä.), allgemeiner Druck der Gesellschaft nach zuverlässigerer (z.B. sicherer) Software usw. Solche nichttechnische Voraussetzungen für eine breitere Anwendung von Programmkorrektheitsbeweistechniken werden hier erwähnt, weil sie im Zweifel viel wichtiger und eher maßgebend sein werden als die technischen; sie sind jedoch nicht Gegenstand der vorliegenden Arbeit.

Eine technische motivierende Voraussetzung wird in Abschnitt 6.2.2 angesprochen: die maschinelle Unterstützung der Korrektheitsbeweistechnik. Ohne Zweifel könnten geeignete derartige "Werkzeuge" hilfreich sein. Es ist kontrovers, ob sie eine erforderliche Voraus-

setzung sind. Dabei spielen vermutlich psychologische Aspekte des Rechnereinsatzes zumindest eine gewisse Rolle.

Eine andere motivierende Voraussetzung für die breite Anwendungsakzeptanz wären Fallstudien, die das Verhältnis zwischen Aufwand und Nutzen klar zeigen. Wie bereits erwähnt, mangelt es an solchen Untersuchungen. Einige wenige quantitative Angaben über Qualitätsverbesserungen und Produktivitätserhöhungen sind bereits zitiert worden (siehe Abschnitt 2.1.4). Solche Ergebnisse stellen einen Schritt in die gewünschte Richtung dar, reichen jedoch bei weitem nicht aus, um Unentschlossene von den Vorteilen zu überzeugen. Ein grundsätzliches Hindernis steht wirklichkeitsnahen Fallstudien und Untersuchungen im Wege: eine gewisse Anzahl von Softwareentwicklern, die die fraglichen Programmkorrektheitsbeweistechniken beherrschen und in deren Anwendung ausgebildet *und erfahren* sind, ist Voraussetzung. Die Entstehung von ausreichend großen Gruppen solcher Softwareentwickler innerhalb einigen Organisationen setzt Investitionsentscheidungen voraus, die ohne eine überzeugende rationale Begründung unwahrscheinlich sind. Es besteht also ein stabiler Teufelskreis (vgl. das Huhn-Ei-Dilemma).

6.1 Lernphase

Die wichtigste und grundlegende Voraussetzung für die praktische Anwendung von Programmkorrektheitsbeweistechniken ist sicherlich die entsprechende mathematische Fähigkeit, siehe Abschnitt 3.4.1. Der Softwareentwickler muß über das erforderliche mathematische Wissen verfügen. Ferner muß er Ausdrücke der Booleschen Algebra schnell, gewandt und zielgerecht umformen können, genauso wie der Ingenieur einer klassischen Fachrichtung die Differential- und Integralrechnung beherrschen sowie mit Ausdrücken der reellen oder komplexen Algebra flink umgehen können muß. Darüber hinaus muß der kreative Anwender von Programmkorrektheitsbeweistechniken Anforderungen an die zu entwickelnde Software in der Sprache der Booleschen Algebra ausdrücken können.

Dieser Stoff ist eher etwas weniger schwierig als die Mathematik der klassischen Ingenieurfächer. Entsprechend kann davon ausgegangen werden, daß der zur Aneignung dieser Fähigkeiten erforderliche Lernaufwand etwas geringer ist, als bei der Ausbildung der Ingenieure der klassischen Fachrichtungen vorgesehen wird. Die Einbeziehung dieses Stoffs in die Informatikausbildung an Hochschulen dürfte also möglich und relativ problemlos sein, insbesondere wenn man berücksichtigt, daß ein Teil der erforderlichen Mathematik im Informatikstudienplan bereits enthalten ist. Wichtig dabei ist eine bewußte Betonung auf die Anwendung solcher mathematischer Kenntnisse mit entsprechenden Übungen; die bloße Vermittlung passiver Kenntnisse reicht für den künftigen Softwareentwicklungs-Ingenieur nicht aus.

Der Schwierigkeitsgrad und folglich der erforderliche Lernaufwand dürften jedoch so hoch sein, daß nur eine relativ kleine Minderheit der in der Praxis stehenden Softwareentwickler in der Lage ist, ohne bedeutsame Unterstützung sich diese Kenntnisse und Fähigkeiten anzueignen. Hier sind Arbeitgeber, private Weiterbildungsinstitute sowie staatliche Institutionen für die tertiäre Ausbildung hinsichtlich entsprechender fachlicher Weiterbildung gefordert.

Gewisse aber nicht wirklich befriedigende Anhaltspunkte für eine Abschätzung des erforderlichen Lehraufwands für in der Praxis stehende Softwareentwickler geben einige

Veröffentlichungen an. [Dyer; Abschnitt 3.5] empfiehlt drei "Workshops" für unterschiedliche Teilnehmerkreise:

- einen über die gewählte Spezifikationsmethode und den Spezifikationsprozeß für diejenigen, die für die Ermittlung der Anforderungen und die Festlegung der externen Spezifikation verantwortlich sind,
- einen über die Verifikation der Korrektheit von Entwürfen für Softwareentwickler und
- einen über statistische Methoden und Zuverlässigkeitsmeßverfahren für Mitarbeiter des Test- und Qualitätskontrollwesens.

Dyer erwähnt eine Dauer von je einer Woche bzw. 40 Stunden für den ersten und den dritten Workshop. Er gibt keine Dauer für den zweiten Workshop an, aber seine allgemeine Beschreibung deutet auf einen ähnlichen Umfang hin. Dabei geht er davon aus, daß die Teilnehmer bereits vorher über "working knowledge" von Mengenlehre und Algebra verfügen.

Der frühere Entwurf des "Interim Defence Standard 00-55" sieht umfangreicheres Training für verantwortliche Spezialisten für sicherheitskritische Software vor: dort ist von einer gesamten Dauer von 16 Wochen die Rede [Ministry of Defence; 1989 May 9, Annex N]. Es wird dort angeregt, entsprechende akademische Studiengänge auf dem Niveau des M.Sc. zu fördern.

[Bloomfield; 1986 Sept., S. 992] berichtet von einem viermonatigen Lernaufwand für VDM, Prolog und deren Anwendung auf das fragliche Problem.

[Thomas, Martyn; 1993 Jan.-Feb., p. 34] erwähnt Kurse über die Spezifikationssprache Z für in der Praxis stehende Softwareentwickler. Der Einführungskurs dauert 4 Tage, der Kurs für Fortgeschrittene, 1 Woche. Nachdem die Kursabsolventen praktische Erfahrung gesammelt haben, empfiehlt Thomas einen weiteren Workshop. Erst nach mehreren Monaten weiterer Erfahrung können sie als Experten angesehen werden.

Der Lernaufwand, den in der Praxis stehende Softwareentwickler betreiben müssen, um mathematisch rigorose Methoden mit Erfolg anwenden zu können, hält sich also in realisierbaren Grenzen, vorausgesetzt, unterstützende Aus- und Weiterbildungsmaßnahmen werden systematisch und konsequent durchgeführt. Informelle, unsystematische und augenblickliche Bemühungen sind nicht erfolgversprechend.

6.2 Programmverifikation

Die grundsätzliche Frage, ob die Programmkorrektheitsbeweissführung überhaupt Sinn hat, ist Gegenstand kontroverser Diskussionen gewesen, die in Abschnitt 2.1.5 angerissen wurden. Zur konkreteren Frage, ob der damit verbundene Zeitaufwand praktisch (d.h. wirtschaftlich) vertretbar ist bzw. sein könnte, sollen hier unterschiedliche Meinungen zusammenfassend betrachtet werden.

Am häufigsten wird die Meinung vertreten, die formale Verifikation sei viel zu zeitaufwendig, siehe Abschnitt 2.2.1. Auffallend bei solchen Behauptungen ist aber, daß kein Vergleichsmaßstab angegeben wird. Auch das ausführliche Testen (insbesondere statistisches Testen) kann sehr zeitaufwendig sein, aber ein hoher Testaufwand wird ohne weiteres akzeptiert und betrieben. Grobe qualitative Angaben in DIN IEC 880 über den Aufwand für Testen und für die Korrektheitsbeweissführung deuten darauf hin, daß diese zwei Verfahren ähnlich aufwendig seien [DKE; 1987 Aug., S. 54-55].

Gegen den mehrfachen Entwicklungsaufwand, den die N-Version-Programmierung (diversitäre Redundanz) zwangsläufig verursachen muß, werden kaum Einwände erhoben. In erster Linie wird ihre Effektivität, nicht ihre Effizienz, in Frage gestellt. In diesem Zusammenhang müßte gefragt werden, welches zeitaufwendige Verfahren wirtschaftlicher ist: ein Programm zu entwickeln und seine Korrektheit formal zu verifizieren oder mehrere Programme unabhängig voneinander zu entwickeln und sie in ein redundantes System zusammenzusetzen.

Gelegentlich wird auf der anderen Seite der Schluß gezogen, daß die konsequente Anwendung mathematisch rigoroser Methoden die Kosten tatsächlich verringern kann, z.B. in [Kershaw; S. 28]. Aber angesichts des bereits erwähnten Mangels an aussagefähigen vergleichenden Studien muß die Frage als noch nicht beantwortet betrachtet werden. Vor allem sind die Fragen noch offen,

- wie weit der Programmkorrektheitsbeweissführungsaufwand durch Übung, Erfahrung, Gliederungstechniken, Rechnerunterstützung u.ä. gesenkt werden kann,
- ob der übrigbleibende Aufwand durch sich aus der Anwendung von Korrektheitsbeweissführungstechniken ergebende Einsparungen an anderen Stellen des Entwicklungszyklus ausgeglichen werden und
- wie hoch der Zeitaufwand für die mathematisch rigorose Programmverifikation eigentlich ist bzw. sein muß.

6.2.1 Manuelle Korrektheitsbeweissführung

Die Einwände gegen den Zeitaufwand der formalen Programmverifikation beziehen sich meist auf die manuelle Korrektheitsbeweissführung. Damit wird oft die Erwartung verknüpft, daß automatisierte Rechnerunterstützung ("Werkzeuge") den menschlichen Zeitaufwand stark verringern wird und daß darin die Lösung des Problems zu suchen ist.

Zunächst sollte bemerkt werden, daß aus der heutigen Sicht eines Softwareentwicklers die formale Programmverifikation eine zusätzliche Aufgabe in einem bereits durch Zeitdruck gekennzeichneten Prozeß darstellt. Darüber hinaus erfordert diese zusätzliche Aufgabe spezielle Kenntnisse, Fähigkeiten und eine bestimmte Gewandtheit, die selten bereits vorhanden sind. Unter diesen Umständen ist es nicht verwunderlich, daß eine negative Einstellung entsteht, bewußt oder unbewußt.

Typische Korrektheitsbeweise sind aus der Sicht des Anfängers durch lange Ausdrücke und umfangreiche algebraische Umformungen gekennzeichnet. Besonders interessante mathematische Einsichten treten kaum auf. Wenn man die strukturellen und die algebraisch-manipulierenden Aspekte des Beweises nicht voneinander streng trennt, kann der Beweis unübersichtlich werden und verwirrend wirken.

In Bezug auf die Frage, wie hoch der Zeitaufwand für die Korrektheitsbeweissführung wirklich ist, geben einige eigene Projekte und Seminarerfahrungen gewisse, wenn auch nur sehr grobe, Anhaltspunkte [Baber; unveröffentlichte Beratungsberichte und projektinterne Unterlagen].

Im Projekt A handelte es sich um ein bereits erstelltes in Assemblersprache geschriebenes Unterprogramm, das eine bestimmte Hardwareschnittstelle steuert. Das Ziel des Projekts war die Erstellung eines mathematisch vollständigen Korrektheitsbeweises. Das Unterprogramm war 41 Zeilen lang. Der Projektaufwand, einschließlich der Zeit für das Studium der Hardwarebeschreibungen, für das Formulieren der Vor- und Nachbedingun-

gen, der Schleifeninvarianten und der sich aus den Anforderungen der Hardware ergebenden Zwischenbedingungen sowie für die Erstellung der endgültigen Dokumentation mit Hilfe eines marktgängigen Textverarbeitungsprogramms betrug 40 Mannstunden. Der Projektmitarbeiter hatte vorher weder mit der spezifischen Hardwareschnittstelle noch mit dem fraglichen Mikroprozessor Erfahrung oder Kenntnisse gehabt, obschon er über allgemeine und umfangreiche Kenntnisse über vergleichbare Hardwareeinheiten verfügte. Die in diesem Projekt erstellte Dokumentation umfaßte ca. 30 Seiten und beinhaltete nicht nur den Korrektheitsbeweis, sondern auch eine ausführliche Beschreibung der verwendeten Notationsformen, der Ableitung verschiedener Zwischenbedingungen aus den Hardwareanforderungen usw. Ein nicht unerheblicher Teil (evtl. ca. 20%) dieses Aufwands ist als Wiederholung der Konstruktionsaufgabe zu betrachten und war erforderlich, weil keine formale Spezifikation der Aufgabe, keine Schleifeninvarianten und nur unvollständige formale Angaben über die Hardwareeigenschaften vorgegeben wurden.

Im vergleichbaren Projekt B handelte es sich um ein vorgegebenes, in Pascal geschriebenes Programm, das eine Datei bestimmter Art in inhaltlich äquivalente Dateien in anderen Formaten und mit anderen Strukturen übersetzt. Das Ziel auch dieses Projekts war die Erstellung eines mathematisch vollständigen Korrektheitsbeweises. Das Programm war 105 Zeilen lang. Der Projektaufwand einschließlich der Zeit für das Studium von Unterlagen über die Dateiformate und -strukturen, für das Formulieren der Vor-, Nach- und Zwischenbedingungen und der Schleifeninvarianten sowie die Erstellung der endgültigen Dokumentation betrug ca. 60 Mannstunden. Die in diesem Projekt erstellte Dokumentation umfaßte ca. 40 Seiten und beinhaltete auch in diesem Falle mehr als nur den reinen Korrektheitsbeweis. Auch hier ist ein deutlicher Anteil des Aufwands als Wiederholung der Konstruktionsaufgabe anzusehen (Formulieren der Vor- und Nachbedingungen und der Schleifeninvarianten). In diesem Projekt führte ein Fehler im verwendeten Textverarbeitungsprogramm zu einem Zeitverlust von 10 bis 15 Mannstunden (ca. 20% des gesamten Projektaufwands). Dieser Zeitverlust ist im oben angegebenen Projektaufwand enthalten.

In meinem Seminar für in der Praxis stehende Softwareentwickler und in meinem Praktikum/Seminar für Informatikhochschulstudenten werden in Gruppenarbeit verschiedene Konstruktionsaufgaben gelöst. Zur Aufgabe gehört die Erarbeitung eines Korrektheitsbeweises. Die zu konstruierenden Programmsegmente sind vom Umfang und Komplexitätsgrad her vergleichbar mit den Programmsegmenten in Abschnitten 3.3.3 und 4.3, Anhang 1, Abschnitt A1.1.3, Anhang 2 und Anhang 4 (einige dieser Beispiele sind sogar Gegenstände solcher Gruppenaufgaben). In einer Gruppenarbeit von 3 bis 4 Stunden Dauer wird das Programmsegment konstruiert und bei den einfacheren Aufgaben ein mathematisch rigoroser Korrektheitsbeweis fertiggestellt. Im Falle der komplexeren Programmsegmenten wird in der verfügbaren Zeit im Seminar für in der Praxis stehende Softwareentwickler nur eine informale bis halbformale Beweisskizze erstellt. In einem Seminar hat eine Gruppe in der ersten Gruppenarbeit das Programmsegment konstruiert und einen informalen Beweis erstellt und in der zweiten Gruppenarbeit einen mathematisch vollständigen Korrektheitsbeweis dafür erarbeitet. Die Mitglieder dieser Gruppe waren alle Dipl.-Math., Dipl.-Inform. oder Dipl.-Ing.; insofern war die Zusammensetzung der Gruppe nicht typisch.

Am Anfang der Gruppenarbeiten verfügen die Gruppenmitglieder im Prinzip über die erforderlichen Kenntnisse der Programmkorrektheitsbeweisleitung in dem Sinne, daß Sie einen Vortrag bzw. Vorlesungen darüber gehört haben. Sie verfügen über keine Erfahrung bzw. Übung in der Anwendung dieses Stoffs; entsprechend fehlen Gewandtheit und Ge-

schicklichkeit bei der praktischen Anwendung. In den Gruppenarbeiten wird die jeweilige Gesamtaufgabe nur in geringfügigem Maße aufgeteilt und von verschiedenen Gruppenmitgliedern unabhängig voneinander gelöst. Vielmehr dient die Gruppenarbeit der gegenseitigen Unterstützung und dem Ideenaustausch, um den Lerneffekt zu erhöhen. Das bedeutet, daß man die Anzahl der Gruppenmitglieder mit der Gruppenarbeitszeit nicht multiplizieren darf, um den effektiven produktiven Zeitaufwand zu errechnen. Eher erscheint die Annahme gerechtfertigt, daß ein mit der Korrektheitsbeweisleitung vertrauter und darin erfahrener Softwareentwickler die Aufgabe allein in etwa der gleichen Zeit fertigstellen könnte (evtl. ohne Erstellung einer sauberen endgültigen Dokumentation).

Bei diesen Gruppenarbeiten stellt die Korrektheitsbeweisleitung im engeren Sinne kein besonders schwieriges Problem dar. Für die Teilnehmer deutlich schwieriger ist das Übersetzen der Aufgabenbeschreibung und des Aufgabenverständnisses in eine mathematisch präzise und ausreichend vollständige Formulierung der Spezifikation, d.h. in die Vor- und Nachbedingungen und in den Korrektheitssatz (die Korrektheitsaussage).

Die oben geschilderte Erfahrung steht in einem gewissen Widerspruch zum in der Fachliteratur angegebenen Zeitaufwand für die Beweisführung für die in Anhang 1, Abschnitt A1.1 beschriebene Aufgabe: zwei Mannwochen, siehe Anhang 1, Abschnitt A1.1.2. Ein Zeitaufwand von zwei Mannwochen dürfte für eine an einen praktischen Rahmen orientierte Analyse, Konstruktion und Korrektheitsbeweisleitung dieser doch recht einfachen Aufgabe übertrieben hoch sein.

Wenn man sich für die Verringerung des Zeitaufwands für die manuelle Korrektheitsbeweisleitung interessiert, muß man sich mit der Frage auseinandersetzen, welche Aspekte der Beweisführung die wesentlichen Teile des fraglichen Zeitaufwands verursachen. Die oben geschilderte Erfahrung deutet darauf hin, daß er überwiegend dafür aufgebracht wird, um (1) fehlende für die Beweisführung geeignete Zwischenbedingungen und Schleifeninvarianten zu bestimmen (diese gehören eigentlich zu den Soll-Ergebnissen der Konstruktionsaufgabe), (2) die algebraische Verifikation der sich aus der Zerlegung der zu beweisenden Korrektheitsaussage ergebenden Implikationen und (3) das Fertigstellen sauberer Dokumentation, insbesondere über die rein algebraischen Verifikationsschritte, in einer Form, die man an Externe, z.B. Prüfer, weiterreichen kann. Besonders die optisch klare und übersichtliche Gestaltung von langen Formeln kann einen nicht unerheblichen Zeitaufwand erfordern.

Der Zeitaufwand für die Korrektheitsbeweisleitung kann dadurch in Grenzen gehalten bzw. verringert werden, daß (1) die Konstruktion auf entsprechenden Konzepten basiert (siehe Kapitel 4) und Schleifeninvarianten und die wesentlichen Zwischenbedingungen in der Konstruktionsdokumentation festgehalten werden, (2) die Zerlegung der zu beweisenden Korrektheitsaussage so weit getrieben wird, daß nur Implikationen mit besonders einfachen (und ggf. kurzen) Thesen zur algebraischen Verifikation übrigbleiben und (3) die Dokumentation "erstellungsfreundlich" gestaltet wird und deren Erstellung mit geeigneten Hilfsmitteln unterstützt wird.

Beim Einsatz gängiger Textverarbeitungssystemen ist vor allem das Schreiben von Formeln mit hoch- und tiefgesetzten Zeichen und Zeichenfolgen eine relativ zeitraubende Angelegenheit, weil der jeweilige Zeilenabstand unterschiedlich sein soll, je nach dem, ob in der vorherigen Zeile tiefgesetzte Zeichen und/oder in der nachherigen Zeile hochgesetzte Zeichen vorkommen. Eine Schreibweise, die hoch- und tiefgesetzte Zeichen und Zeichenfolgen ganz vermeidet, würde folglich den Zeitaufwand für die Erstellung der Dokumentation verringern, z.B. (vgl. Anhang 2, Anlage 1, KA 1.4)

$il \in Z$ und $ir \in Z$ und $il \in Z$ und $ir \in Z$ und $il \leq ir$ und $ir \leq il$

und $(\text{und } k=il, il-1 : a(k) \leq a(il))$ und $(\text{und } k=ir+1, ir : a(k) \geq a(ir))$

und $(\& k=il, il-1 : [a(k)]) \& [a(il)] \& (\& k=ir+1, ir : [a(k)])$ Perm A'

statt

$il \in Z$ und $ir \in Z$ und $il \in Z$ und $ir \in Z$ und $il \leq ir$ und $ir \leq il$

und $\text{und}_{k=il}^{il-1} a(k) \leq a(il)$ und $\text{und}_{k=ir+1}^{ir} a(k) \geq a(ir)$

und $\&_{k=il}^{il-1} [a(k)] \& [a(il)] \&_{k=ir+1}^{ir} [a(k)]$ Perm A'

sowie

$(\text{ers } i \rightarrow il : (\text{ers } j \rightarrow ir : (\text{ers } g \rightarrow a(i) : IR)))$

oder

$IR [g \rightarrow a(i)] [j \rightarrow ir] [i \rightarrow il]$

statt

$((IR_{a(i)}^g)_{ir}^j)^i_{il}$

Sonst läßt sich ein deutlicher Anteil der mühseligen und fehleranfälligen Arbeit bei der Beweisführung mit Papier und Bleistift allein — vor allem das häufige Kopieren von Ausdrücken mit nur geringfügigen Änderungen — auch mit einfachen Textverarbeitungssystemen schnell und zuverlässig durchführen.

Bei der Korrektheitsbeweisführung und in der Dokumentation darüber ist es wichtig, die Zerlegung der zu beweisenden Korrektheitsaussage und die algebraische Verifikation der sich daraus ergebenden Ausdrücke streng voneinander getrennt zu halten (siehe die Bemerkung darüber am Anfang dieses Abschnitts 6.2.1). Werden diese zwei Aspekte miteinander vermischt, dann wird die Beweisführung unnötig kompliziert und zeitaufwendig. Hält man sie voneinander streng getrennt, dann bleiben auch umfangreiche Beweise übersichtlich. Sie können schneller und klarer vorgeführt werden. Studenten und Vortragshörer erheben viel weniger Einwände gegen eine so gegliederte Korrektheitsbeweisführung und den damit verbundenen vermuteten Zeitaufwand.

Weil Boolesche Ausdrücke für unterschiedliche Leserkreise ggf. unterschiedlich gestaltet werden sollten, käme maschinelle Unterstützung in Frage, die Formeln von einer (z.B. erfassungsfreundlichen) Schreibweise in mehrere andere leserfreundlichere Darstellungsformen umwandelt. Die Bestimmung sowohl erfassungsfreundlicher Schreibweisen als auch leserfreundlicher Darstellungsformen würde entsprechende, z.T. umfangreiche Untersuchungen voraussetzen. Im Gegensatz dazu wäre die Entwicklung solcher Umwandlungshilfsmittel voraussichtlich verhältnismäßig einfach.

6.2.2 Maschinelle Unterstützung

Wie bereits erwähnt, werden in Zusammenhang mit Einwänden gegen den für die Korrektheitsbeweisführung erforderlichen Zeitaufwand Werkzeuge dafür gefordert. Rechnerunterstützung für die Beweisführung könnte sicherlich prinzipiell nützlich sein, aber gewis-

se Warnsignale auf unrealistisch hohe Erwartungen bzw. Hoffnungen dürfen nicht übersehen oder vernachlässigt werden.

Seit dem Anfang des Computerzeitalters besteht eine gewisse Neigung, Aufgaben, die der Mensch nur unvollkommen versteht und eigentlich nicht wirklich in Griff hat, dem Computer zu übertragen in der impliziten Hoffnung, der mächtige Computer werde sie irgendwie beherrschen [Baber; 1982]. Die Ergebnisse entsprechender Einsatzversuche waren enttäuschend bis katastrophal. Insofern daß Softwareentwickler Werkzeuge für die Korrektheitsbeweisführung als Ersatz für das (nicht vorhandene oder nur mangelhafte) eigene Verständnis der Korrektheitsproblematik und ihrer Lösungen einsetzen wollen, muß mit einem Mißerfolg der Versuche gerechnet werden, egal wie gut die Werkzeuge sind. Insofern daß Softwareentwickler solche Werkzeuge als Verstärkung und Unterstützung der eigenen bereits gut entwickelten Fähigkeiten einsetzen wollen und gute geeignete Werkzeuge verfügbar sind, sind die Erfolgchancen gut.

Mehrere Systeme für die Rechnerunterstützung der Korrektheitsbeweisführung sind entwickelt worden, siehe Abschnitt 2.1.7. Aus technischer Sicht kann von Erfolgen gesprochen werden. Aus organisatorischer bzw. wirtschaftlicher Anwendungssicht kann der gegenwärtige Stand nicht so positiv beurteilt werden.

1971 bemerkte [Foley], daß satzbeweisende Systeme nicht ausreichend mächtig waren, um die Beweisführung komplexer Lemmata zu bewerkstelligen. Er hielt eine Zusammenarbeit zwischen Mensch und Maschine als Lösungsansatz für versprechend, zitierte aber einen Bericht von Burstall, daß der Prozeß mühselig und zeitaufwendig ("laborious") sei.

Inzwischen sind mehrere Systeme für die maschinelle Unterstützung der Programmkorrekttheitsbeweisführung entwickelt worden (siehe Abschnitt 2.1.7). Über Unzulänglichkeiten in deren Handhabung und Einsetzeinschränkungen wird jedoch immer noch berichtet [persönliche Kommunikation: Cohen und ein Konferenzteilnehmer], [Fields], [But], [Guaspari]. [Wing; 1990 Sept., "Experience with the Larch Prover"] warnt vor zu großen Erwartungen an solche Hilfsmittel: "Using a proof checker requires forethought, patience ..., users may easily be lured into thinking or hoping that the tool will find the proof for them. A proof checker does not decrease the amount of thinking required on the user's part; it can alleviate some of the bookkeeping and symbol pushing, but no more."

Wie in Abschnitt 2.1.7 bereits erwähnt, besteht ein System zur maschinellen Programmkorrekttheitsbeweisführung typischerweise aus zwei wesentlichen Komponenten. Das eine Untersystem zerlegt die zu beweisende Korrektheitsaussage in eine Sammlung von Prädikaten (Booleschen Ausdrücken, auch "verification conditions" genannt). Dabei werden effektiv Beweisregeln angewendet. Unter Anwendung der in Abschnitt 3.2 vorgestellten Beweisregeln im Zerlegungsprozeß weisen alle sich daraus ergebenden Ausdrücke die Form einer logischen Implikation auf. Dieser Zerlegungsprozeß ist leicht automatisierbar; ein dafür geeignetes in Prolog geschriebenes Programm wäre ca. 300-400 Zeilen lang.

Das zweite Untersystem übernimmt die Booleschen Ausdrücke vom ersten Untersystem und versucht, jeden einzelnen davon zu verifizieren, d.h. ihn in die logische Konstante wahr umzuwandeln. Hierfür wird ein Satzbeweiser eingesetzt. Die zu verifizierenden Ausdrücke sind einfacher Natur, können jedoch sehr lang und umfangreich sein, vgl. die verschiedenen Beispiele von Korrektheitsbeweisen an anderen Stellen in dieser Arbeit. In der Praxis kann auch ein sehr einfacher Satzbeweiser viele der fraglichen Ausdrücke verifizieren, aber nicht alle, und die übrigbleibenden stellen das praktische Problem dar.

Nicht selten kann der Mensch eine Strategie für die Beweisführung schnell sehen, wenn der Ausdruck ihm in einer verständlichen und einsichtigen Form vorgelegt wird. Bei

der Manipulation der Ausdrücke ist die Maschine viel schneller und zuverlässiger als der Mensch. Die für die Beweisführung wesentlichen Stärken und Schwächen des Menschen und der Maschine ergänzen sich also; eine Zusammenarbeit dabei erscheint angebracht und anstrebenswert.

Aus den oben erwähnten Unzulänglichkeiten bisheriger Systeme für die Programmkorrektheitsbeweisführung sowie aus der Notwendigkeit einer Zusammenarbeit dabei liegt die Schlußfolgerung nahe, daß erheblich mehr Betonung auf die Gestaltung der Zusammenarbeit und der Kommunikation zwischen Mensch und Maschine bei der kooperativen Beweisfindung und -führung gelegt werden sollte. Auch bei der Aufbereitung eines Programms für die maschinelle Korrektheitsbeweisführung ist eine für den Menschen möglichst günstige Lösung zu suchen. Eine Möglichkeit wäre z.B., Vor-, Nach- und Zwischenbedingungen sowie Schließeninvarianten im Programmtext vorzusehen und den Übersetzer so zu konstruieren, daß er das Programm zusammen mit den genannten Bedingungen in einer geeigneten Form an das System für die Korrektheitsbeweisführung übergibt, ohne daß eine weitere manuelle Bearbeitung erforderlich ist.

Nur wenn eine maschinelle Unterstützung der Korrektheitsbeweisführung zu einer verbesserten Wirtschaftlichkeit des Softwareentwicklungsprozesses führt, wird sie in der Praxis akzeptiert und eingesetzt. Eine vollkommene technische Lösung reicht dafür nicht aus, auch die praktische Handhabung muß effizient sein. Sie muß so gestaltet sein, daß sie psychologisch als eine nützliche Unterstützung — statt als zusätzliche Arbeit — empfunden wird.

6.3 Programmkonstruktion

Über den Zeitaufwand, der bei der Anwendung der Ideen und Konzepte der Programmkorrektheitsbeweisführung auf die Neukonstruktion eines Programms entsteht bzw. eingespart wird, wird in der Fachliteratur kaum berichtet, vermutlich weil diese Anwendungsmöglichkeit vergleichsweise selten ernsthaft in Betracht gezogen wird, siehe den letzten Absatz von Abschnitt 2.1.4 und die Einführung zum Kapitel 4. Die bereits zitierte Stelle in [Kershaw; S. 28] deutet vermutlich auf Einsparungen auch in der Entwurfsphase hin, ist aber in dieser Hinsicht weder spezifisch noch eindeutig.

Die Anwendung der Ideen und Konzepte der Programmkorrektheitsbeweisführung bei der Neukonstruktion setzt die zeitaufwendigen Aktivitäten der Beweisführung (siehe Abschnitt 6.2) nicht voraus. Diesbezügliche Einwände gegen die Korrektheitsbeweisführung betreffen deshalb ihre Anwendung bei der Erstkonstruktion nicht.

Meine eigene professionelle Erfahrung (siehe Abschnitt 2.1.14) deutet klar darauf hin, daß die Verwendung der Beweisregeln als Leitlinien für die Programmkonstruktion (siehe Kapitel 4) zu einer Verringerung der Entwicklungszeit führt. Dieser Effekt tritt hauptsächlich deswegen auf, weil die Entwicklungstätigkeit zielgerichteter verläuft. Irrwege (Versuche und Ideen, die doch nicht zu einem geeigneten Programmsegment führen) werden weitgehend vermieden oder wenigstens früher als solche erkannt, so daß unproduktive Aktivitäten und Zeit reduziert werden. Die Beweisregeln (ob informal oder formal angewendet) ermöglichen eine schnellere, aussagefähigere gedankliche Überprüfung der Teilentwürfe, die sonst (weil ungesteuert) oft in ein ineffizientes Grübeln übergeht.

Eventuell führt die Verwendung der Beweisregeln als Leitlinien für die Konstruktion auch deswegen zu einem geringeren Entwicklungszeitaufwand, weil die so konstruierten Programmsegmente oft kürzer sind.

Die wenigen, bereits zitierten Stellen in der Fachliteratur, die sich mit dieser spezifischen Frage beschäftigen, vertreten ähnliche Meinungen. Aber wissenschaftlich fundierte — z.B. durch kontrollierte Untersuchungen und systematische Vergleichsstudien gewonnene — Erkenntnisse darüber liegen offensichtlich nicht vor.

6.4 Konstruktionsänderung

Wenn man den Aufwand betrachtet, der bei der Anwendung von Korrektheitsbeweisführungstechniken auf die Konstruktionsänderung entsteht, sollte man — genau wie in Abschnitt 5.1 — zwischen Programmen, bei deren Neukonstruktion diese Techniken nicht angewendet wurden, und denjenigen, bei deren Neukonstruktion sie wohl angewendet wurden, unterscheiden. Im ersten Falle, d.h., wenn keine geeigneten internen Schnittstellenspezifikationen vorliegen, führt die in Abschnitt 5.1 aufgeführte Problematik zu einem höheren, eventuell sehr viel höheren Anwendungsaufwand als im zweiten Falle. Diese Problematik und der erhöhte Aufwand wird die Nützlichkeit der Anwendung auf die Konstruktionsänderung deutlich einschränken, wenn das fragliche Programm auf die herkömmliche Weise entwickelt worden ist.

Das Thema des Anwendungsaufwands bei der Konstruktionsänderung scheint noch weniger untersucht worden zu sein als bei der Programmverifikation und der Neukonstruktion, siehe Abschnitte 6.2 und 6.3. Dieses Untersuchungs- bzw. Forschungsthema wird voraussichtlich erst dann wirklich aktuell, wenn Korrektheitsbeweisführungstechniken in der Praxis viel breiter akzeptiert sind und eingesetzt werden als es heute der Fall ist. Dann wird es u.a. darum gehen, die folgenden Vermutungen zu bestätigen oder zu verwerfen:

- Bei Programmen, die unter Anwendung von Korrektheitsbeweisführungstechniken konstruiert worden sind, wird der Änderungsbedarf überhaupt viel geringer sein als bei auf die herkömmliche Weise entwickelten Programmen.
- Wenn jedoch ein solches Programm geändert wird, wird der Änderungsaufwand viel geringer sein als es heute der Fall ist, weil die Konstruktionsänderung zielgerichteter erfolgen wird (vgl. Abschnitt 6.3), weil die durch die Änderung beeinflussten Teile des Programms schnell und eindeutig identifiziert werden können und weil die Änderungsspanne klar abgegrenzt wird.
- Die gegenwärtige Problematik der bei Änderungen neu eingeführten Fehler wird weitgehend — eventuell sogar ganz — beseitigt werden.

7. Spezielle Aspekte der Programmkorrektheitsbeweissführung

7.1 Computerarithmetik

In Abschnitt 3.3.4.5 wurde die potentielle Problematik, die in Verbindung mit Computerarithmetik auftreten kann, diskutiert. Zusammenfassend kann gesagt werden, daß es keine in der Computerarithmetik inhärenten Probleme hinsichtlich der Korrektheitsbeweissführung gibt, sondern daß eine ungünstige Schreibweise, die zwischen verwandten aber verschiedenen Funktionen (z.B. zwischen der mathematischen Addition und der Gleitkommaaddition) nicht unterscheidet, einen verleitet, den computerarithmetischen Funktionen Eigenschaften zu unterstellen, die sie nicht aufweisen. Das wiederum kann natürlich zu fehlerhaften und nicht zutreffenden Beweisen führen.

Bei der Programmkorrektheitsbeweissführung ist es deshalb unerlässlich, daß man zwischen den mathematischen arithmetischen Funktionen und den verschiedenen computerarithmetischen Funktionen klar unterscheidet. Vor allem sollen unterschiedliche Zeichen für die unterschiedlichen arithmetischen Funktionen verwendet werden. Der Softwareentwickler sollte die unterschiedlichen Eigenschaften der verschiedenen relevanten arithmetischen Funktionen in Erinnerung behalten.

Die in der Praxis wichtigsten computerarithmetischen Systemen sind (1) Ganzzahlenarithmetik auf einer endlichen Menge (auf einem auf beiden Seiten beschränkten Intervall der Ganzzahlen) und (2) die Gleitkommaarithmetik. Grundsätzliche Bemerkungen zur Behandlung dieser Arithmetiksysteme in der Korrektheitsbeweissführung enthalten die folgenden Abschnitte 7.1.1 bzw. 7.1.2.

7.1.1 Ganzzahlenarithmetik auf einem beschränkten Intervall

Weil wirkliche Rechner nur endlich viele Zustände haben sowie aus Effizienzgründen wird die implementierte Ganzzahlenarithmetik meist absichtlich auf eine fest definierte endliche Menge beschränkt. (Computerarithmetiksysteme, die grundsätzlich die gesamte Menge der rationalen Zahlen abzubilden beabsichtigen und nur durch Rechnerkapazitätsbegrenzungen eingeschränkt sind, bilden eine Ausnahme.) Typischerweise wird die mathematisch übliche Ganzzahlenarithmetik angestrebt, wobei von vornherein davon ausgegangen wird, daß die auftretenden Werte der verschiedenen Variablen innerhalb geeigneter Bereiche liegen, so daß die berechneten Werte mit den mathematisch richtigen übereinstimmen. In der Korrektheitsbeweissführung muß entsprechend gezeigt bzw. durch geeignete Vorbedingungen sichergestellt werden, daß die fraglichen Variablenwerte tatsächlich in den erforderlichen Bereichen liegen. Dazu bedient man sich wesentlicher Eigenschaften des fraglichen implementierten Arithmetiksystems wie z.B. unten aufgeführt.

Typischerweise besteht ein auf einem Rechner verwirklichtes Ganzzahlenarithmetiksystem aus einer endlichen Menge aufeinanderfolgender Ganzzahlen $\mathbf{Ze} \triangleq \{\min, \min+1, \dots$

$\max\}$. Meist gilt ferner, daß $0 \in \mathbf{Ze}$, d.h., daß $\min \leq 0 \leq \max$. Die Additionsfunktion $\boxed{+}$ wird so definiert, daß

$$a \in \mathbf{Ze} \wedge b \in \mathbf{Ze} \wedge \min \leq a+b \leq \max \Rightarrow a \boxed{+} b = a+b \quad (1)$$

oder in alternativer Form,

$$a \in \mathbf{Ze} \wedge b \in \mathbf{Ze} \wedge a+b \in \mathbf{Ze} \Rightarrow a \boxed{+} b = a+b \quad (1a)$$

Dabei steht $+$ für die mathematisch exakte Addition.

Manchmal ist die implementierte Addition nur für a und b mit $a+b \in \mathbf{Ze}$ (d.h. mit $\min \leq a+b \leq \max$) definiert, in welchem Falle Werte von a und b mit $a+b \notin \mathbf{Ze}$ zu einem Laufzeitfehler (Über- bzw. Unterlauf) führen. Manchmal wird jedoch die Addition $\boxed{+}$ auf der gesamten Menge $\mathbf{Ze} \times \mathbf{Ze}$ definiert. Eine solche Möglichkeit führt dazu, daß auch die folgenden Bedingungen gelten:

$$a \in \mathbf{Ze} \wedge b \in \mathbf{Ze} \wedge a+b < \min \Rightarrow a \boxed{+} b = a+b + (\max - \min + 1) \quad (2)$$

$$a \in \mathbf{Ze} \wedge b \in \mathbf{Ze} \wedge \max < a+b \Rightarrow a \boxed{+} b = a+b - (\max - \min + 1) \quad (3)$$

In diesem Fall wird ein Überlauf über die obere Grenze \max in den Wert \min überführt, d.h. $\max \boxed{+} 1 = \min$ (falls $1 \in \mathbf{Ze}$). Entsprechend gilt $\min \boxed{+} -1 = \max$ (falls $-1 \in \mathbf{Ze}$). Eine so definierte Funktion $\boxed{+}$ und die Menge \mathbf{Ze} bilden eine kommutative Gruppe.

Wie in der Mathematik üblich wird die Subtraktion $\boxed{-}$ als die Addition mit dem Kehrwert definiert: $a \boxed{-} b \triangleq a \boxed{+} b^*$, wo b^* der Kehrwert von b in \mathbf{Ze} ist ($b^* = -b$, falls $-b \in \mathbf{Ze}$ (d.h. falls $\min \leq -b \leq \max$); $b^* = -b + (\max - \min + 1)$, falls $-b < \min$; $b^* = -b - (\max - \min + 1)$, falls $\max < -b$).

Entsprechend (1) und ggf. (2) und (3) oben gelten auch Beziehungen zwischen den anderen zum fraglichen Arithmetiksystem gehörenden Operationen $\boxed{-}$ und $\boxed{*}$ und den mathematischen Funktionen $-$ und $*$. Weil der Quotient zweier Ganzzahlen nicht immer eine Ganzzahl ist, wird die Beziehung zwischen $\boxed{/}$ und $/$ eine etwas andere Form annehmen.

Für die Korrektheitsbeweissführung wesentlich ist es, daß man in einem Beweis $a \boxed{+} b$ durch $a+b$ nur dann ersetzen darf, wenn die Hypothese der Implikation (1) oben erfüllt ist. Anderenfalls muß von einem gültigen Lemma wie (2) oder (3) oben Gebrauch gemacht werden.

In einem Programm kommen manchmal verschiedene Ganzzahlenarithmetiksysteme zur Verwendung, z.B. für 8-Bit-, 16-Bit-, 32-Bit-Zahlendarstellungen usw. Die Schreibweise sollte dazwischen unterscheiden, z.B. $\boxed{+8}$, $\boxed{+16}$ bzw. $\boxed{+32}$. Häufig gilt $\min=0$ und $\max=255$ für die 8-Bit-Arithmetik, aber auch die Vereinbarung $\min=-128$ und $\max=127$ bildet eine mögliche Basis. Für die 16-Bit-Arithmetik gilt gewöhnlich $\min=-32768$ und $\max=32767$, aber auch die Vereinbarung $\min=0$ und $\max=65535$ kommt vor allem in der Systemprogrammierung (im Gegensatz zur Anwendungsprogrammierung) vor.

Beispiel: In diesem Auszug aus dem Korrektheitsbeweis [Baber; unveröffentlichter Beratungsbericht] für ein hardwarebezogenes Unterprogramm zum Absenden einer Zeichenfolge über eine serielle Schnittstelle kommen die ganzzahlenarithmetischen Ausdrücken $r8 \boxed{-8} 1$ und $r16 \boxed{+16} 1$ vor. Die entsprechenden Mengen sind als $\mathbf{Z8} \triangleq \{0, \dots, 255\}$ bzw. $\mathbf{Z16} \triangleq \{0, \dots, 65535\}$ definiert und die Ergebnisse der Funktionen $\boxed{-8}$ und $\boxed{+16}$ sind für alle Argumentwerte aus $\mathbf{Z8}$ bzw. $\mathbf{Z16}$ definiert; die Eigenschaften (2) und (3) oben gelten und jedes Arithmetiksystem ist eine Gruppe. Da sich die Variablennamen $r8$ und $r16$ auf bestimmte 8- bzw. 16-Bit-Hardwareregister beziehen, gilt grundsätzlich, daß $r8 \in$

Z8 und $r16 \in \mathbf{Z16}$; diese Bedingungen wurden jedoch nicht formal und ausdrücklich im Beweis mitgeführt. Auch hardwarebedingt ist $\text{Spr}(\cdot)$ eine Feldvariable, deren Wert ein Element aus **Z8** ist ("Spr" bezieht sich auf den Hauptspeicher).

Von den Eigenschaften der ganzzahlenarithmetischen Operationen $\boxed{-8}$ und $\boxed{+16}$ ist für die Korrektheitsbeweissführung die folgende Variante von (1) oben nützlich:

$$1 \leq r8 \leq 255 \Rightarrow r8 \boxed{-8} 1 = r8 - 1 \quad (4)$$

In den hier relevanten Teilen des Korrektbeweises handelte es sich um die Verifikation der folgenden zwei Korrektheitsaussagen. Darin ist die Variable SG ("Sendegeschichte") eine Folge von Elementen aus **Z8**.

$$\{SG = (SG' \&_{i=0}^{\text{Spr}(r16')-r8} [\text{Spr}(r16' \boxed{+16} i)]) \quad (\text{KA1})$$

$$\text{und } r16 = (r16' \boxed{+16} (\text{Spr}(r16') - r8 + 1)) \text{ und } 1 \leq r8 \leq \text{Spr}(r16') \leq 255\}$$

$$r8 := r8 \boxed{-8} 1$$

$$\{SG = (SG' \&_{i=0}^{\text{Spr}(r16')-r8-1} [\text{Spr}(r16' \boxed{+16} i)])$$

$$\text{und } r16 = (r16' \boxed{+16} (\text{Spr}(r16') - r8)) \text{ und } 0 \leq r8 \leq \text{Spr}(r16') \leq 255\}$$

und

$$\{r16 = (r16' \boxed{+16} (\text{Spr}(r16') - r8)) \text{ und } 1 \leq r8 \leq \text{Spr}(r16') \leq 255\} \quad (\text{KA2})$$

$$r16 := r16 \boxed{+16} 1$$

$$\{r16 = (r16' \boxed{+16} (\text{Spr}(r16') - r8 + 1)) \text{ und } 1 \leq r8 \leq \text{Spr}(r16') \leq 255\}$$

Im wirklichen Programmtext würde man die Zuweisungen oben in der üblichen — nicht eindeutigen — Form $r8 := r8 - 1$ bzw. $r16 := r16 + 1$ schreiben.

Beweis der Korrektheitsaussage KA1: Man nenne die Vorbedingung der Korrektheitsaussage KA1 V1 und die Nachbedingung, P1. Gemäß Beweisregel Z2 wird KA1 gelten, falls

$$V1 \Rightarrow P1 \quad r8 \boxed{-8} 1$$

Aber

$$P1 \quad r8 \boxed{-8} 1$$

$$SG = (SG' \&_{i=0}^{\text{Spr}(r16')-(r8 \boxed{-8} 1)-1} [\text{Spr}(r16' \boxed{+16} i)])$$

$$\text{und } r16 = (r16' \boxed{+16} (\text{Spr}(r16') - (r8 \boxed{-8} 1))) \text{ und } 0 \leq r8 \boxed{-8} 1 \leq \text{Spr}(r16') \leq 255$$

← [vgl. V1]

$$SG = (SG' \&_{i=0}^{\text{Spr}(r16')-(r8 \boxed{-8} 1)-1} [\text{Spr}(r16' \boxed{+16} i)])$$

$$\text{und } r16 = (r16' \boxed{+16} (\text{Spr}(r16') - (r8 \boxed{-8} 1))) \text{ und } 0 \leq r8 \boxed{-8} 1 \leq \text{Spr}(r16') \leq 255$$

$$\text{und } 1 \leq r8 \leq 255$$

= [gemäß (4) oben]

$$SG = (SG' \&_{i=0}^{\text{Spr}(r16')-(r8-1)-1} [\text{Spr}(r16' \boxed{+16} i)])$$

$$\text{und } r16 = (r16' \boxed{+16} (\text{Spr}(r16') - (r8 - 1))) \text{ und } 0 \leq r8 - 1 \leq \text{Spr}(r16') \leq 255$$

$$\text{und } 1 \leq r8 \leq 255$$

=

$$SG = (SG' \&_{i=0}^{\text{Spr}(r16')-r8} [\text{Spr}(r16' \boxed{+16} i)])$$

$$\text{und } r16 = (r16' \boxed{+16} (\text{Spr}(r16') - r8 + 1)) \text{ und } 1 \leq r8 \leq \text{Spr}(r16') + 1 \leq 256$$

$$\text{und } 1 \leq r8 \leq 255$$

←

$$SG = (SG' \&_{i=0}^{\text{Spr}(r16')-r8} [\text{Spr}(r16' \boxed{+16} i)])$$

$$\text{und } r16 = (r16' \boxed{+16} (\text{Spr}(r16') - r8 + 1)) \text{ und } 1 \leq r8 \leq \text{Spr}(r16') \leq 255$$

=

$$V1 \quad \blacksquare$$

Beweis der Korrektheitsaussage KA2: Man nenne die Vorbedingung der Korrektheitsaussage KA2 V2 und die Nachbedingung, P2. Gemäß Beweisregel Z2 wird KA2 gelten, falls

$$V2 \Rightarrow P2 \quad r16 \boxed{+16} 1$$

Aber

$$P2 \quad r16 \boxed{+16} 1$$

=

$$(r16 \boxed{+16} 1) = (r16' \boxed{+16} (\text{Spr}(r16') - r8 + 1)) \text{ und } 1 \leq r8 \leq \text{Spr}(r16') \leq 255$$

=

$$[\text{Spr}(r16') - r8, 1, \text{Spr}(r16') - r8 + 1 \in \mathbf{Z8} \subset \mathbf{Z16} \text{ und (1a) oben}]$$

$$(r16 \boxed{+16} 1) = (r16' \boxed{+16} ((\text{Spr}(r16') - r8) \boxed{+16} 1)) \text{ und } 1 \leq r8 \leq \text{Spr}(r16') \leq 255$$

=

$$[\boxed{+16} \text{ assoziativ}]$$

$$(r16 \boxed{+16} 1) = ((r16' \boxed{+16} (\text{Spr}(r16') - r8)) \boxed{+16} 1) \text{ und } 1 \leq r8 \leq \text{Spr}(r16') \leq 255$$

=

$$[\Leftarrow \text{ offensichtlich, } \Rightarrow \text{ weil } \boxed{+16} \text{ Gruppenoperation}]$$

$$r16 = (r16' \boxed{+16} (\text{Spr}(r16') - r8)) \text{ und } 1 \leq r8 \leq \text{Spr}(r16') \leq 255$$

=

$$V2 \quad \blacksquare$$

(Ende des Beispiels)

Wie das Beispiel oben zeigt, verursachen die Unterschiede zwischen typischen implementierten Ganzzahlenarithmetiksystemen und den in der Mathematik üblichen arithmetischen Funktionen keine grundsätzlichen Schwierigkeiten bei der Korrektheitsbeweissführung. Man muß nur diesen Unterschied im Bewußtsein behalten und die tatsächlichen Eigenschaften der implementierten Arithmetik genau beachten. Eine unerläßliche Voraussetzung dafür ist eine Schreibweise, die zwischen den verschiedenen Funktionen klar und zwingend unterscheidet.

7.1.2 Gleitkommaarithmetik

Die Gleitkommaarithmetik ermöglicht bekanntlich eine für viele Anwendungen brauchbare Darstellung der reellen Zahlen als die oben diskutierte Ganzzahlenarithmetik. Die Genauigkeit der Gleitkommaarithmetik reicht für viele Zwecke aus. Dies führt manchmal

dazu, daß der Softwareentwickler auch dann die noch verbleibende Ungenauigkeit vernachlässigt, wenn sie zu unerwünschten Wirkungen führen kann und deshalb nicht vernachlässigt werden darf. Diese potentielle Falle wurde in Abschnitt 3.3.4.5 erörtert.

Wenn die Ungenauigkeit der Gleitkommaarithmetik von Belang sein kann, muß sie bei der Programmkorrektheitsbeweissführung explizit berücksichtigt und behandelt werden. Wahrscheinlich am einfachsten ist es, wenn man Genauigkeitsaspekte weitgehend aus der Korrektheitsbeweissführung im engeren Sinne ausklammert und sich getrennter gleitkommaarithmetikbezogener Lemmata und Hilfssätze bedient. Anhang 3 enthält mehrere solche Lemmata und Hilfssätze. Das folgende Beispiel zeigt eine solche Vorgehensweise. Siehe auch Abschnitt 3.3.4.5.

Beispiel: Betrachte den folgenden Auszug aus einem Programmsegment, das z.B. die Nullstelle einer Funktion suchen könnte. Darin werden gleitkommaarithmetische Funktionen durch die Zeichen \oplus , \ominus und \otimes explizit ausgedrückt. Die Zeichen $+$, $-$ und \times sowie das Nebeneinanderschreiben als Andeutung für die Multiplikation beziehen sich unten auf die idealen mathematischen Funktionen.

```

while eps < |x2  $\ominus$  x1|                                     [0 < eps]
do {x1 = x1' und x2 = x2'}                               [Anfangswerte von x1 und x2]
...
xn := (x1  $\oplus$  x2)  $\otimes$  0,5
{xn  $\ominus$  x1 < x2  $\ominus$  x1 und x2  $\ominus$  xn < x2  $\ominus$  x1}         [damit Schleife terminiert]
if ...
then ...
    x1 := xn
else ...
    x2 := xn
endif
{x2  $\ominus$  x1 < x2'  $\ominus$  x1'}
endwhile

```

Nach dem Prinzip der binären Suche wird die Länge des noch aktuellen Intervalls $[x1, x2]$ von jeder Ausführung des Schleifenkerns (etwa) halbiert.

Bei genauer Arithmetik würde die Schleife offensichtlich terminieren, denn die Folge $[|x2 - x1|.d, |x2 - x1|.d^2, |x2 - x1|.d^3, \dots]$, wo d eine Datenumgebung aus dem Definitionsbereich der Schleife und S der Schleifenkern sind, konvergiert gegen 0, eine Zahl, die kleiner als eps ist.

Weil die Menge der Gleitkommazahlen endlich ist, wird die Terminierung der Schleife dadurch sichergestellt, daß jede Ausführung des Schleifenkerns den Wert der Schleifenvariante $|x2 \ominus x1|$ verringert, siehe die Nachbedingung des Schleifenkerns oben. Aber wegen eventueller Rundung bei der Berechnung von xn ist es möglich, daß $xn = x1$ oder $xn = x2$, so daß der Wert der Schleifenvariante nicht verringert wird. Wegen eventueller Rundung bei der Berechnung von $|x2 \ominus x1|$ ist es möglich, daß $x2 \ominus x1 = x2' \ominus x1'$, selbst wenn $x1' < x1$ oder $x2 < x2'$. Um solche Rundungseffekte auszuschließen bzw. aufzufangen, muß das offene Intervall $(x1, x2)$ Element(e) aus G (die Menge der Gleitkommazahlen, siehe Anhang 3) enthalten. Es muß auch ggf. gewährleistet sein, daß das offene Intervall $(0, |x2 \ominus x1|)$ Element(e) aus G enthält. Das bedeutet, daß der Wert der Annäherungsgenauigkeitsschranke eps nicht zu klein angesetzt werden darf. Der Wert von eps muß in Bezug auf die genauigkeitsrelevanten gleitkommaarithmetischen Parameter R^{emin}

und δ ausreichend groß sein. (R^{emin} ist der kleinste darstellbare Betrag außer 0 und δ ist der (größte) Abstand zwischen benachbarten Mantissenwerten, siehe Anhang 3.) Die folgende Analyse gibt hinreichende untere Schranken für eps an.

Im folgenden wird der Einfachheit halber davon ausgegangen, daß $x1 \leq x2$. Bei entsprechender Initialisierung der Schleife oben ist diese Bedingung Teil der Schleifeninvariante. Die Werte der Variablen $x1$, $x2$ und xn sind Elemente aus G . Ferner wird angenommen, daß die Werte aller auftretenden gleitkommaarithmetischen Ausdrücke $(x1 \oplus x2)$ usw.) definiert sind, d.h., daß $|x1|$ und $|x2|$ nicht zu groß sind.

Lemma 1: $xn \ominus x1$ hat eine untere und eine obere Schranke wie folgt:

$$(1 - \delta)(xn - x1) - R^{\text{emin}} \leq xn \ominus x1 \leq (1 + \delta)(xn - x1) + R^{\text{emin}}$$

Beweishinweis: Die Aussage oben folgt aus der in Anhang 3 angegebenen Formel (3.5) und der Tatsache, daß $xn \ominus x1 = xn \ominus (-x1)$, siehe den letzten Absatz des Abschnitts A3.1 in Anhang 3. Alternativ kann die Aussage oben durch Anwendung der in Anhang 3 enthaltenen Formeln (1.7) und (2.4) bewiesen werden.

Lemma 2: $x2 \ominus xn$ hat eine untere und eine obere Schranke wie folgt:

$$(1 - \delta)(x2 - xn) - R^{\text{emin}} \leq x2 \ominus xn \leq (1 + \delta)(x2 - xn) + R^{\text{emin}}$$

Beweishinweis: Diese Aussage wird wie Lemma 1 bewiesen.

Lemma 3: $x2 \ominus x1$ hat eine untere und eine obere Schranke wie folgt:

$$(1 - \delta)(x2 - x1) - R^{\text{emin}} \leq x2 \ominus x1 \leq (1 + \delta)(x2 - x1) + R^{\text{emin}}$$

Beweishinweis: Diese Aussage wird wie Lemma 1 bewiesen.

Lemma 4: Falls $0,5 \in G$, dann hat $xn (\hat{=} (x1 \oplus x2) \otimes 0,5)$ eine untere und eine obere Schranke wie folgt:

$$\begin{aligned} & (x1 + x2) \times 0,5 - 0,5(3 + \delta)R^{\text{emin}} - 0,5\delta(2 + \delta)|x1 + x2| \\ & \leq xn \\ & \leq (x1 + x2) \times 0,5 + 0,5(3 + \delta)R^{\text{emin}} + 0,5\delta(2 + \delta)|x1 + x2| \end{aligned}$$

bzw.

$$\begin{aligned} & (1 + \delta)(x1 + x2) - (1 + \delta)(3 + \delta)R^{\text{emin}} - \delta(1 + \delta)(2 + \delta)|x1 + x2| \\ & \leq 2(1 + \delta)xn \\ & \leq (1 + \delta)(x1 + x2) + (1 + \delta)(3 + \delta)R^{\text{emin}} + \delta(1 + \delta)(2 + \delta)|x1 + x2| \end{aligned}$$

Beweis:

$$\begin{aligned} & |xn - (x1 + x2) \times 0,5| \\ = & |(x1 \oplus x2) \otimes 0,5 - (x1 + x2) \times 0,5| \\ \leq & \quad \quad \quad [\text{Anhang 3, (4.8), } a \rightarrow x1 \oplus x2, b \rightarrow 0,5, x \rightarrow x1 + x2, y \rightarrow 0,5] \\ & R^{\text{emin}} + \delta|(x1 + x2) \times 0,5| + (1 + \delta)|(x1 \oplus x2) \times 0,5 - (x1 + x2) \times 0,5| \\ = & \\ & R^{\text{emin}} + 0,5\delta|x1 + x2| + 0,5(1 + \delta)|(x1 \oplus x2) - (x1 + x2)| \\ \leq & \quad \quad \quad [\text{Anhang 3, (3.5)}] \\ & R^{\text{emin}} + 0,5\delta|x1 + x2| + 0,5(1 + \delta)(R^{\text{emin}} + \delta|x1 + x2|) \\ = & \end{aligned}$$

$$0,5(3+\delta)R^{emin} + 0,5\delta(2+\delta)|x_1+x_2|$$

Aus dieser gesamten Aussage ($|x_n-(x_1+x_2)\times 0,5| \leq 0,5(3+\delta)R^{emin} + 0,5\delta(2+\delta)|x_1+x_2|$) folgt die Aussage dieses Lemmas. ■

Lemma 5: Falls $\epsilon < x_2 \ominus x_1$, dann gilt, daß

$$\epsilon < (1+\delta)(x_2-x_1) + R^{emin}$$

Beweishinweis: Aus der oberen Schranke für $x_2 \ominus x_1$ (siehe Lemma 3) und der Hypothese dieses Lemmas folgt die Aussage oben.

Bemerkung: Weil $x_1 \leq x_2$ gilt (siehe den Absatz vor Lemma 1 oben), ist die while-Bedingung äquivalent zur Hypothese dieses Lemmas. Dabei wird unterstellt, daß $0 \leq x_2 \ominus x_1$ aus $0 \leq x_2 - x_1$ folgt, welches durch bestimmte Eigenschaften des Gleitkommaarithmetiksystems sichergestellt wird, siehe Anhang 3, (1.1), (1.2) und (1.3).

Satz über eine untere Schranke für eps: Falls

$$[(1+\delta)(3+\delta)+5]R^{emin} + \delta(1+\delta)(2+\delta)(|x_1| + |x_2|) + 4\delta|x_2-x_1| \leq \epsilon \quad [uS1]$$

und falls die while-Bedingung (die Hypothese des Lemmas 5) $\epsilon < x_2 \ominus x_1$ erfüllt ist, dann gilt, daß

$$x_n \ominus x_1 < x_2 \ominus x_1 \text{ und } x_2 \ominus x_n < x_2 \ominus x_1$$

Bemerkungen: Größere untere Schranken für eps (stärkere Bedingungen) sind

$$[(1+\delta)(3+\delta)+5]R^{emin} + 2\delta(1+\delta)(2+\delta)\max(|x_1|, |x_2|) + 4\delta|x_2-x_1| \leq \epsilon \quad [uS2]$$

$$[(1+\delta)(3+\delta)+5]R^{emin} + \delta[(1+\delta)(2+\delta)+4](|x_1| + |x_2|) \leq \epsilon \quad [uS3]$$

$$[(1+\delta)(3+\delta)+5]R^{emin} + 2\delta[(1+\delta)(2+\delta)+4]\max(|x_1|, |x_2|) \leq \epsilon \quad [uS4]$$

Diese unteren Schranken für eps können für bestimmte Anwendungszwecke Vorteile gegenüber der im Satz angegebenen Schranke aufweisen (siehe die Bemerkungen nach dem Beweis des Satzes unten). Typischerweise wird $x_1 \approx x_2$ (und somit $|x_2-x_1| < \max(|x_1|, |x_2|)$) gelten und δ viel kleiner als 0,1 sein. Entsprechende numerische Abschätzungen für die unteren Schranken [uS2] und [uS4] oben sind

$$8,5R^{emin} + 9\delta\max(|x_1|, |x_2|) \leq \epsilon \quad [\text{Abschätzung für uS2, } x_1 \approx x_2]$$

$$8,5R^{emin} + 13\delta\max(|x_1|, |x_2|) \leq \epsilon \quad [\text{Abschätzung für uS4}]$$

Beweis des Satzes: Die zwei Teile der Aussage des Satzes werden getrennt bewiesen. Eine hinreichende Bedingung wird ermittelt, die aus der Hypothese des Satzes und der Aussage des Lemmas 5 folgt.

$$x_n \ominus x_1 < x_2 \ominus x_1$$

$$\begin{aligned} &\Leftarrow \\ &= \\ &= \\ &= \end{aligned} \quad \begin{aligned} x_n \ominus x_1 &\leq (1+\delta)(x_n-x_1) + R^{emin} < (1-\delta)(x_2-x_1) - R^{emin} \leq x_2 \ominus x_1 \\ & \quad \text{[Lemma 1, obere Schranke; Lemma 3, untere Schranke]} \\ (1+\delta)(x_n-x_1) + R^{emin} &< (1-\delta)(x_2-x_1) - R^{emin} \\ (1+\delta)x_n &< (1-\delta)x_2 + 2\delta x_1 - 2R^{emin} \end{aligned}$$

$$(1+\delta)x_n < (1-\delta)x_2 + 2\delta x_1 - 2R^{emin}$$

$$\begin{aligned} &= \\ &\Leftarrow \\ &= \\ &= \\ &= \\ &\Leftarrow \\ &\Leftarrow \\ &\Leftarrow \\ &= \\ &\Leftarrow \\ &= \\ &= \\ &= \end{aligned} \quad \begin{aligned} 2(1+\delta)x_n &< 2(1-\delta)x_2 + 4\delta x_1 - 4R^{emin} \\ 2(1+\delta)x_n &\leq (1+\delta)(x_1+x_2) + (1+\delta)(3+\delta)R^{emin} + \delta(1+\delta)(2+\delta)|x_1+x_2| \\ &< 2(1-\delta)x_2 + 4\delta x_1 - 4R^{emin} \quad \text{[Lemma 4, obere Schranke]} \\ (1+\delta)(x_1+x_2) + (1+\delta)(3+\delta)R^{emin} + \delta(1+\delta)(2+\delta)|x_1+x_2| &< 2(1-\delta)x_2 + 4\delta x_1 - 4R^{emin} \\ [(1+\delta)(3+\delta)+5]R^{emin} + \delta(1+\delta)(2+\delta)|x_1+x_2| + 4\delta(x_2-x_1) &< (1+\delta)(x_2-x_1) + R^{emin} \\ [(1+\delta)(3+\delta)+5]R^{emin} + \delta(1+\delta)(2+\delta)(|x_1| + |x_2|) + 4\delta|x_2-x_1| &< (1+\delta)(x_2-x_1) + R^{emin} \\ [(1+\delta)(3+\delta)+5]R^{emin} + \delta(1+\delta)(2+\delta)(|x_1| + |x_2|) + 4\delta|x_2-x_1| &\leq \epsilon \\ &< (1+\delta)(x_2-x_1) + R^{emin} \quad \text{[Lemma 5]} \\ [(1+\delta)(3+\delta)+5]R^{emin} + \delta(1+\delta)(2+\delta)(|x_1| + |x_2|) + 4\delta|x_2-x_1| &\leq \epsilon \\ &< x_2 \ominus x_1 \\ &= \\ &\text{Hypothese des Satzes} \end{aligned}$$

Die Wahrheit des zweiten Terms ($x_2 \ominus x_n < x_2 \ominus x_1$) der Aussage wird auf die gleiche Weise bewiesen. ■

Hat die Programmvariable eps mindestens den im Satz angegebenen Wert für alle vorkommenden Kombinationen von Werten für x_1 und x_2 , dann wird jede Ausführung des Schleifenkerns den Wert der Schleifenvariante $x_2 \ominus x_1$ verringern. Dadurch wird wiederum sichergestellt, daß die Schleife terminieren wird. Für die Anwendung der unteren Schranken [uS2] und [uS4] oben genügt es, nur die ursprünglichen Werte der Variablen x_1 und x_2 zu berücksichtigen, denn die Ausführung des Schleifenkerns kann die Werte der darin vorkommenden Ausdrücke $\max(|x_1|, |x_2|)$ und $|x_2-x_1|$ nicht erhöhen.

(Ende des Beispiels)

Das Beispiel oben zeigt, daß gleitkommaarithmetische Ausdrücke durchaus einer mathematischen Analyse der Genauigkeit zugänglich sind. Dafür nützliche Lemmata und Hilfssätze enthält Anhang 3. Gewöhnliche, bekannte und einfache mathematische Techniken (wie die Anwendung der Dreiecksungleichung, das Zwischenschieben einer bekannten Größe, um eine Ungleichung zu beweisen bzw. zu erzwingen, u.ä.) reichen für eine solche Genauigkeitsanalyse weitgehend aus.

7.2 Nebenläufigkeit aus der Sicht der Korrektheitsbeweissführung

Der breite und umfangreiche Themenkomplex nebenläufiger Prozesse liegt außerhalb des Rahmens dieser Arbeit (siehe Abschnitt 1.2). Hier sollen nur bestimmte Aspekte davon angeschnitten werden, die sich unmittelbar aus der auf sequentielle Programme bezogenen Korrektheitsbeweissführung ergeben, siehe Abschnitt 7.2.1. Insbesondere wird für eine daraus abgeleitete, besonders einfache und nützliche Form der Nebenläufigkeit ein Korrektheitsbeweis vorgestellt, siehe Abschnitt 7.2.2.

Alle Behandlungen dieses Themas sehen vor, daß nebenläufige Prozesse auf gemeinsame Programmvariablen (oder Betriebs-, Hilfsmittel usw. im allgemeinen) zugreifen, denn ohne Bezug auf gemeinsame Objekte irgendeiner Art entstünden bei der gleichzeitigen Ausführung mehrerer Prozesse keinerlei Wechselwirkungen dazwischen, und keine neuen Aspekte der Programmausführung würden auftreten können.

In Programmiersprachen sind viele und verschiedenartige Konstrukte und Strukturen für nebenläufige Prozesse vorgesehen. Viele solche Konstrukte zielen darauf hin, nebenläufige Prozesse zu synchronisieren, d.h. die gleichzeitige Ausführung bestimmter Stellen oder Abschnitte der verschiedenen Prozesse zu erzwingen. Andere verfolgen das gegensätzliche Ziel: die gleichzeitige Ausführung bestimmter Abschnitte der nebenläufigen Prozesse zu verhindern. Vgl. z.B. Semaphore [Dijkstra; "A Constructive Approach to the Problem of Program Correctness", 1968], den "Rendezvous"-Mechanismus in Ada [—; *The Programming Language Ada Reference Manual*, 1981], PAR in Occam 2 [Wexler], den Monitor [Hoare; 1974 Oct.] (siehe auch die Beschreibung und den Vergleich von Monitoren in einigen Programmiersprachen in [Whiddett; Chap. 3]) usw. Die verschiedenen Programmiersprachenkonstrukte für nebenläufige Prozesse lenken die Aufmerksamkeit teils weniger, teils mehr auf die *Kommunikation* (die Übertragung von Variablenwerten) zwischen den fraglichen Prozessen.

In der auf sequentielle Programme bezogenen Korrektheitsbeweissführung (wie in dieser Arbeit behandelt) beziehen sich die verschiedenen Aussagen (Vor-, Nach-, und Zwischenbedingungen, Invarianten usw.) auf einzelne Ausführungszustände (Datenumgebungen) bzw. auf darin enthaltene Objekte (Programmvariablen). In der allgemeineren Form der auf nebenläufige Prozesse bezogenen Korrektheitsbeweissführung beziehen sich solche Aussagen nicht nur auf die einzelnen Ausführungszustände, sondern auf Folgen davon (Ausführungsgeschichten, "traces") bzw. auf Folgen von darin enthaltenen Objekten (Datenelemente, -strukturen usw.). Siehe z.B. [America; "Formal techniques for parallel object-oriented languages", 1991], [Broy; 1991 und 1992], [de Boer] und [Kearney] sowie die bereits zitierte Literatur. In dieser Arbeit wird nicht näher auf derartige Aussagen eingegangen.

In der in Abschnitt 7.2.2 behandelten Form von Nebenläufigkeit steht die Kommunikation zwischen den fraglichen Prozessen im Vordergrund der Betrachtung. Nur weil die Korrektheitsbeweissführung es voraussetzt, werden die nebenläufigen Prozesse so konstruiert, daß gewisse Abschnitte davon nicht gleichzeitig ausgeführt werden können. Eine in der Praxis wichtige und große Klasse von Anwendungen kann unter Verwendung nur dieses Mechanismus für die Wechselwirkung zwischen gleichzeitig laufenden Prozessen verwirklicht werden.

7.2.1 Wechselwirkung zwischen nebenläufigen Prozessen in einem Korrektheitsbeweis

In einem Korrektheitsbeweis für ein sequentielles Programm werden verschiedene Beweisregeln (Hilfssätze) angewendet, darunter z.B. die Beweisregel F1 für eine Folge (S1; S2) von Anweisungen. Die Definition der Wirkung der Ausführung einer solchen Folge sieht vor, daß S2 auf das Ergebnis der Ausführung von S1 angewendet wird, siehe Abschnitt 3.1.3. Ähnlich sieht die Definition einer if-Anweisung (if B then S1 else S2 endif) vor, daß S1 bzw. S2 auf die gleiche Datenumgebung angewendet wird, in der die Bedingung B ausgewertet wurde. Diese Annahmen können durch die gleichzeitige Ausführung verschiedener Programmsegmente, die sich auf gemeinsame Programmvariablen beziehen, verletzt werden, in welchem Falle der jeweilige Korrektheitsbeweis ggf. nicht mehr zutrifft.

Die oben erwähnte Annahme, daß die Anweisung S2 auf das Ergebnis der Ausführung der vorherigen Anweisung S1 angewendet wird, ist für den Beweis der Beweisregel F1 nicht erforderlich. Beweisregel F1 ($\{V\}S1\{P1\} \wedge \{P1\}S2\{P\} \Rightarrow \{V\} S1; S2 \{P\}$) gilt, falls S2 auf eine Datenumgebung angewendet wird, in der die Zwischenbedingung P1 wahr ist — eine wesentlich schwächere Voraussetzung, die eine Wechselwirkung zwischen nebenläufigen Prozessen erst zuläßt. Ein Korrektheitsbeweis, in dem die Beweisregel F1 angewendet wird, ist also auch dann gültig, wenn ein nebenläufiger Prozeß gemeinsame Variablen verändert, solange die Wahrheit der für den Korrektheitsbeweis maßgeblichen Zwischenbedingungen dadurch nicht beeinflußt wird. Entsprechendes gilt für die Beweisregeln für die anderen Anweisungsarten.

Für den Korrektheitsbeweis eines Programmsegments "maßgeblich" im obigen Sinne ist jede Bedingung, die für die lückenlose Korrektheitsbeweissführung erforderlich ist. Außer der Vorbedingung und der Nachbedingung sind diese die Bedingungen, die bei der Anwendung der Beweisregeln in den Korrektheitsbeweis eingeführt werden müssen. Die für den Korrektheitsbeweis eines Programmsegments PS "maßgeblichen" Bedingungen sind also: die Vorbedingung von PS, die Nachbedingung von PS, die Zwischenbedingung zwischen jedem Paar von zwei aufeinanderfolgenden Anweisungen, die Vorbedingung jedes then-Teils einer if-Anweisung, die Vorbedingung jedes else-Teils einer if-Anweisung, die Vorbedingung jedes Schleifenkerns und die Nachbedingung jedes Schleifenkerns. Durch diese Auflistung miterfaßt sind die Vor- und Nachbedingungen der Zusammensetzungen von Anweisungen (if-Anweisung, Folge von Anweisungen, while-Schleife und den Aufruf auf ein Unterprogramm). Falls an einer Stelle im fraglichen Programmsegment mehrere Bedingungen angegeben werden (vgl. die Beweisregeln B1, Z2, W2 usw.), muß nur die schwächste davon als maßgeblich betrachtet werden, da diese die eigentliche Vorbedingung der darauffolgenden Anweisung ist und es deswegen nur auf ihre Wahrheit ankommt.

Zu den für einen Korrektheitsbeweis maßgeblichen Bedingungen gehören nicht nur Boolesche Ausdrücke über die Werte von Programmvariablen, sondern auch Aussagen über die Struktur von Datenumgebungen und über die den Programmvariablen zugeordneten Mengen, vgl. Abschnitt 3.3.2. In der Praxis werden Programmvariablen, die von gleichzeitig laufenden Prozessen angesprochen werden, von diesen typischerweise nur durch Zuweisungen (und nicht durch declare- und release-Anweisungen) verändert, in welchem Falle nur Ausdrücke über die Werte solcher Variablen eine Rolle spielen.

Ein Korrektheitsbeweis für ein Programmsegment A wird durch einen nebenläufigen Prozeß B nicht gestört, falls für jede für den Korrektheitsbeweis für A maßgebliche Bedingung B_a und für jede nicht unterbrechbare Anweisung (bzw. Anweisungskomponente, siehe unten) S_b im Programm des Prozesses B gilt, daß die Ausführung von S_b die Wahrheit von B_a nicht verändert (stört). Eine hinreichende formale Bedingung dafür ist, daß die Aussage

$$\{B_a \wedge V_{sb}\} S_b \{B_a\}$$

gilt, wobei V_{sb} die maßgebliche Vorbedingung von S_b im Korrektheitsbeweis für B ist. Offensichtlich kann diese Prüfung auf diejenigen S_b beschränkt werden, die Programmvariablen verändern können, die in B_a vorkommen.

Für die Prüfung, ob der Korrektheitsbeweis für ein Programmsegment A durch ein nebenläufiger Prozeß B gestört wird, ist es nützlich, die folgenden Mengen von Programmvariablen zu betrachten [Andrews; S. 68]. Die *Bezugsmenge* BA von A ist die Menge aller Programmvariablen, worauf sich die für den Korrektheitsbeweis für A maßgeblichen Bedingungen beziehen. Die *Veränderungsmenge* VB von B ist die Menge aller Programmvariablen, die durch die Ausführung von in B enthaltenen Zuweisungen, declare-Anweisungen und release-Anweisungen verändert werden können. Man merke, daß *nicht* die Variablen, worauf sich der *Programmtext* des Programmsegments A bezieht, sondern die Variablen, worauf sich die *maßgeblichen Bedingungen im Korrektheitsbeweis* für A beziehen, dabei von Belang sind.

Eine Anweisung S_b ist nicht unterbrechbar, wenn sie vom Ausführungssystem (Systemsoftware oder Hardware) in einem einzigen Vorgang ausgeführt wird. Zugriffe auf bei der Ausführung von S_b ggf. entstehende Zwischenergebnisse oder Veränderung von gemeinsamen Variablen durch andere Prozesse (z.B. Prozeß A) sind während der Ausführung von S_b nicht zulässig. Wegen dieser Einschränkung auf nicht unterbrechbare Anweisungen muß manchmal bei der Prüfung auf Beweisstörung eine Anweisung in ihre nicht unterbrechbaren Ausführungskomponenten (die implementierungsabhängig sein können) unterteilt werden. Zu den nicht unterbrechbaren Vorgängen gehören typischerweise der Lesezugriff auf den Wert einer einzelnen Variable (engl. "fetch") sowie das Verändern des Werts einer einzelnen Variable (engl. "store"). Hier bedeutet der Begriff "einzelne" Variable eine Datengruppe, die vom Rechnersystem (ggf. auf Hardwareebene) in einem einzigen Schritt vom Speicher geholt bzw. gespeichert wird. Z.B. muß ggf. die Zuweisung $x := x + 1$, wo x eine gemeinsame Variable ist, in die Folge

```

lokalx := x           [Zugriff auf eine gemeinsame Variable ("fetch")]
{B1}
lokalx := lokalx + 1   [ausschließlich prozeßlokale Vorgänge]
{B2}
x := lokalx           [Veränderung einer gemeinsamen Variable ("store")]

```

unterteilt werden, wo die Programmvariable lokalx eine prozeßlokale Variable ist. D.h., weder der Programmtext noch der Korrektheitsbeweis eines anderen Prozesses darf auf die Variable lokalx Bezug nehmen. Nur die letzte Zuweisung oben und ihre Vorbedingung B2 kommen als S_b bzw. V_{sb} für die Prüfung auf Störung des Korrektheitsbeweises eines anderen nebenläufigen Prozesses in Frage. Die neu eingeführten Zwischenbedingungen B1 und B2 oben sind für den Korrektheitsbeweis dieses Programmsegments maßgeblich und

müssen bei der Prüfung auf Störung dieses Korrektheitsbeweises durch einen anderen nebenläufigen Prozeß entsprechend behandelt werden.

Das Thema der auf nebenläufige Prozesse bezogenen Korrektheitsbeweissführung wird u.a. in [Owicki], [Apt; 1991] und [Andrews] (siehe dort insbesondere die Abschnitte 2.2 und 2.3) näher und ausführlicher behandelt. [Andrews] enthält eine breite, umfassende und für den Praktiker relativ gut verständliche Abhandlung über viele Aspekte dieses Themenkomplexes.

Die oben geschilderte Prüfung kann sehr umfangreich und aufwendig sein, denn jedes Paar aus der fraglichen Sammlung von nebenläufigen Prozessen muß auf Beweisstörung in jede Richtung geprüft werden. Wenn ein neuer Prozeß einer bereits bestehenden Sammlung von nebenläufigen Prozessen hinzugefügt wird, muß der neue Prozeß mit jedem alten Prozeß auf Beweisstörung in jede Richtung geprüft werden. Nach einer Änderung eines Prozesses gilt das entsprechende. Der Aufwand wird in einem gewissen Umfang dadurch begrenzt, daß nur Anweisungen S_b berücksichtigt werden müssen, die Variablen verändern, worauf sich die jeweilige Bedingung B_a bezieht. Hierfür kommen grundsätzlich nur Variablen aus der Schnittmenge $BA \cap VB$ in Frage (siehe oben). Deshalb liegt die Konstruktionsstrategie nahe, jede solche Schnittmenge möglichst klein zu halten.

Der Aufwand kann auch durch andere Konstruktionstaktiken in Grenzen gehalten werden. Wenn jede gemeinsame Variable von nur zwei Prozessen angesprochen wird, wird die Anzahl der möglichen Paare zu prüfender Prozesse minimiert. Wenn dabei ein Prozeß die Variable verändert und der andere nur auf ihren Wert zugreift, ist die Wechselwirkung besonders einfach. Wenn die in den maßgeblichen Bedingungen auftretenden Bezüge auf gemeinsame Variablen nur in einem einheitlichen Ausdruck (in einer einzigen *globalen Invariante*) vorkommen, verringert sich der Aufwand für die Prüfung auf Beweisstörung, oft erheblich. Durch die Festlegung einer globalen Invariante bei der Konstruktion wird die Wechselwirkung zwischen den nebenläufigen Prozessen systematisiert und in einer einheitlichen Struktur gestaltet. Ein Beispiel dafür ist die Kanalvariante in Abschnitt 7.2.2.

Diese Überlegungen zur Vereinfachung der Korrektheitsbeweissführung führen zur Idee, die Wechselwirkung zwischen nebenläufigen Prozessen auf paarweise gemeinsame Variablen einzuschränken, die Kommunikation zwischen den zwei fraglichen Prozessen in nur eine Richtung bewirken. (Wenn Kommunikation in beide Richtungen möglich sein soll, wird eine zweite (unabhängige) Gruppe von gemeinsamen Variablen für die Kommunikation in die andere Richtung eingerichtet.) Dieses Konzept bildet eine Basis für *kommunizierende sequentielle Prozesse* (engl. communicating sequential processes, CSP). Siehe z.B. [Hoare; 1985], [Milner], [Lenders].

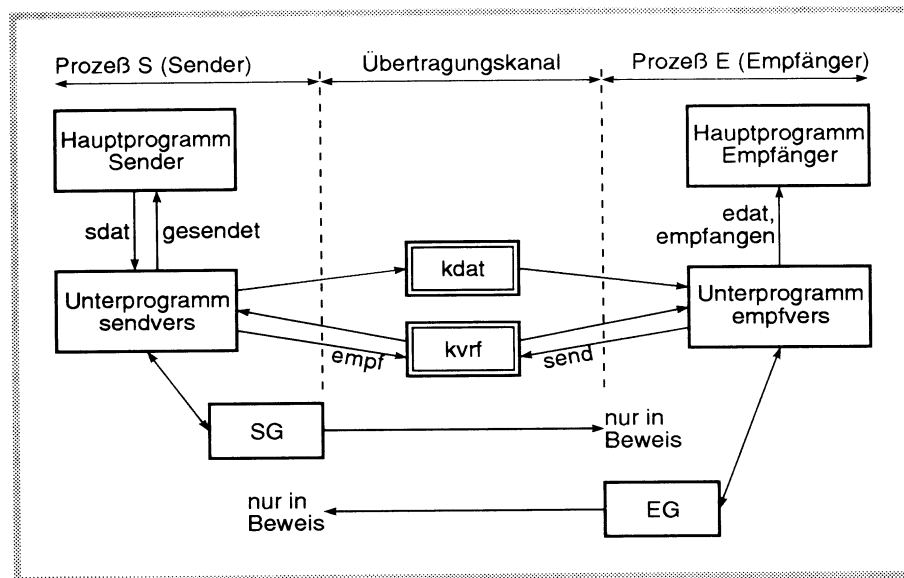
7.2.2 Ein Korrektheitsbeweis für Kommunikation zwischen nebenläufigen Prozessen

7.2.2.1 Übertragungskanal ohne Puffer

Die im letzten Absatz des Abschnitts 7.2.1 oben aufgeführten Überlegungen zur Vereinfachung der Korrektheitsbeweissführung für nebenläufige Prozesse führen zur folgenden Programmstruktur für die Kommunikation zwischen zwei Prozessen. Die gemeinsame Variable $kdat$ (Kanaldaten) überträgt Daten vom sendenden Prozeß zum empfangenden Prozeß. Die gemeinsame Variable $kvrf$ (Kanalverfügung) dient der Koordinierung der zwei

Prozesse und gibt an, ob der Kanal, d.h. insbesondere die Variable *kdat*, dem sendenden oder dem empfangenden Prozeß zur Verfügung steht. Falls *kvrf=send*, steht der Kanal dem sendenden Prozeß zur Verfügung; der Wert der Kanalvariable *kdat* ist bedeutungslos (der Kanal ist leer). Falls *kvrf=empf*, steht der Kanal dem empfangenden Prozeß zur Verfügung; der Wert der Kanalvariable *kdat* ist eine Nachricht, die vom empfangenden Prozeß übernommen (empfangen) werden soll. (Dabei sind *send* und *empf* (wie falsch und wahr) unterschiedliche Konstanten, keine Programmvariablen.) Die Variablen *SG* (Sendegeschichte) und *EG* (Empfangsgeschichte) halten die Folge der bereits gesendeten bzw. der bereits empfangenen Datenwerte fest; diese Variablen werden nur für die Korrektheitsbeweissführung benötigt und erscheinen deshalb nicht in der implementierten Version der fraglichen Programmsegmente.

Die übergeordnete Struktur der zwei Prozesse und ihre Wechselwirkung veranschaulicht das folgende Diagramm:



Datenfluß über den Übertragungskanal ohne Puffer

Unter "Hauptprogramm Sender" sind alle zum Prozeß S gehörenden Programmteile außer dem Unterprogramm *sendvers* (Sendeversuch) zu verstehen. Unter "Hauptprogramm Empfänger" sind alle zum Prozeß E gehörenden Programmteile außer dem Unterprogramm *empfers* (Empfangversuch) zu verstehen. Die Variablen *sd* und *gesendet* sind lokal zum Prozeß S. Die Variablen *edat* und *empfangen* sind lokal zum Prozeß E.

Die Kanalvariante (Kommunikationsvariante) *KI*, die wesentliche Aspekte der Wechselwirkung zwischen den Prozessen S und E festhält, wird wie folgt definiert:

$$KI \triangleq (kvrf=send \text{ und } SG=EG) \text{ oder } (kvrf=empf \text{ und } SG=EG\&\{kdat\})$$

Die Kanalvariante *KI* besagt, daß alles, das *gesendet* worden ist, entweder bereits empfangen wurde oder sich noch im Kanal befindet. Ferner besagt die Kanalvariante, daß

die Reihenfolge der empfangenen Datenwerte mit der Reihenfolge der gesendeten Datenwerte übereinstimmt. Die Kanalvariante ist eine Programmvariante, d.h., sie gilt an allen Stellen beider Prozesse.

Falls der Kanal dem Prozeß S zur Verfügung steht (*kvrf=send*), überträgt *sendvers* die zu sendende Nachricht *sd* an den Kanal und stellt den Kanal dem Prozeß E zur Verfügung. Dadurch wird die Nachricht effektiv gesendet. Steht der Kanal dem Prozeß S nicht zur Verfügung, sendet *sendvers* die Nachricht nicht. In jedem Fall teilt *sendvers* über die lokale Variable *gesendet* dem aufrufenden Programm mit, ob die Nachricht in *sd* gesendet wurde oder nicht.

Der Programmcode für *sendvers* mit den für die Korrektheitsbeweissführung maßgeblichen Vor-, Zwischen- und Nachbedingungen ist wie folgt:

```

{SG=SG' und KI} [S1]
if kvrf=send
then {kvrf=send und SG=SG' und KI} [S2]
    [= {kvrf=send und SG=SG' und SG=EG}]
    kdat:=sd [S3]
    {kvrf=send und kdat=sd und SG=SG' und KI}
    [= {kvrf=send und kdat=sd und SG=SG' und SG=EG}]
    (kvrf, SG):=(empf, SG&\{kdat\}) [implementiert durch kvrf:=empf]
    {SG=SG'\&\{sd\} und KI} [S4]
    gesendet:=wahr
    {(gesendet und SG=SG'\&\{sd\}
    oder nicht gesendet und SG=SG') und KI} [S5]
else {SG=SG' und KI} [S6]
    gesendet:=falsch
    {(gesendet und SG=SG'\&\{sd\}
    oder nicht gesendet und SG=SG') und KI} [S7]
endif
{(gesendet und SG=SG'\&\{sd\}
oder nicht gesendet und SG=SG') und KI} [S8]

```

Die Zuweisung oben der Form $(x, y):=(A1, A2)$ ist eine *Mehrfachzuweisung*, eine Verallgemeinerung der üblichen Zuweisung. Die Ausdrücke *A1* und *A2* werden in der gleichen ursprünglichen Datenumgebung ausgewertet und die errechneten Werte *x* bzw. *y* zugeordnet. Die entsprechende Verallgemeinerung der Beweisregeln *Z1* und *Z2* sieht vor, daß die Variablen *x* und *y* in der Nachbedingung gleichzeitig durch (*A1*) bzw. (*A2*) ersetzt werden. Weil in diesem Unterprogramm der zweite Teil der Mehrfachzuweisung nur für die Korrektheitsbeweissführung erforderlich ist und nicht in der implementierten Version des Programms erscheint (also eine fiktive bzw. idealisierte Anweisungskomponente ist), kann unterstellt werden, daß das Zuordnen des neuen Werts zur Variable *SG* im gleichen nicht unterbrechbaren Vorgang wie das Zuordnen des neuen Werts zur Variable *kvrf* erfolgt.

Falls der Kanal dem Prozeß E zur Verfügung steht (*kvrf=empf*), überträgt *empfers* die in *kdat* stehende Nachricht in die lokale Variable *edat* und stellt den Kanal dem Prozeß S zur Verfügung. Dadurch wird die Nachricht effektiv empfangen. Steht der Kanal dem Prozeß E nicht zur Verfügung, empfängt *empfers* keine Nachricht. In jedem Fall

teilt empfers über die lokale Variable empfangen dem aufrufenden Programm mit, ob eine Nachricht empfangen wurde oder nicht.

Der Programmcode für empfers mit den für die Korrektheitsbeweissführung maßgeblichen Vor-, Zwischen- und Nachbedingungen ist wie folgt:

```

{EG=EG' und KI} [E1]
if kvrf=empf
then {kvrf=empf und EG=EG' und KI} [E2]
    [= {kvrf=empf und EG=EG' und SG=EG&[kdat]}]
    edat:=kdat
    {kvrf=empf und kdat=edat und EG=EG' und KI} [E3]
    [= {kvrf=empf und kdat=edat und EG=EG' und SG=EG&[kdat]}]
    (kvrf, EG):=(send, EG&[kdat]) [implementiert durch kvrf:=send]
    {EG=EG'&[edat] und KI} [E4]
    empfangen:=wahr
    {(empfangen und EG=EG'&[edat]
    oder nicht empfangen und EG=EG') und KI} [E5]
else {EG=EG' und KI} [E6]
    empfangen:=falsch
    {(empfangen und EG=EG'&[edat]
    oder nicht empfangen und EG=EG') und KI} [E7]
endif
{(empfangen und EG=EG'&[edat]
oder nicht empfangen und EG=EG') und KI} [E8]

```

Der Kanal muß, bevor die Ausführung der Prozesse S und E anfängt, auf geeignete Weise initialisiert werden. Unter Berücksichtigung der Kanalvariante und der Bedeutung der Kanalvariablen bietet sich die Anweisungsfolge (SG:=[]; EG:=[]; kvrf:=send) als Initialisierung an.

Für das intuitive, informale Verständnis der Wechselwirkung zwischen den Prozessen S und E und dem Kanal ist es nützlich, das Konzept *Verantwortung* für die Nachricht einzuführen. Durch die Ausführung der Zuweisung kvrf:=empf im Prozeß S übergibt der Prozeß S die Verantwortung für die in kdat gespeicherte Nachricht an den Kanal. Durch die Ausführung der Zuweisung kvrf:=send im Prozeß E übernimmt der Prozeß E die Verantwortung für die in kdat gespeicherte Nachricht. Die ggf. bestehende Verantwortung des Kanals für die in kdat gespeicherte Nachricht widerspiegelt die Kanalvariante: falls kvrf=empf, befindet sich der letzte Term der gesendeten Folge SG noch im Kanal (SG=EG&[kdat]). Der Prozeß E hat die Verantwortung für diesen letzten Term noch nicht übernommen und der Kanal ist für diese noch im Kanal befindliche Nachricht verantwortlich. Falls kvrf=send, hat der Prozeß E die Verantwortung für jeden Term der gesendeten Folge SG übernommen (SG=EG); der Kanal ist für keinen Teil der gesendeten Folge SG verantwortlich. Bei der Übergabe der Verantwortung für eine Nachricht vom Prozeß S an den Kanal wird die fragliche Nachricht der Folge SG hinzugefügt. Bei der Übernahme der Verantwortung für eine Nachricht durch den Prozeß E wird die fragliche Nachricht der Folge EG hinzugefügt.

Einschränkungen hinsichtlich der Bezugnahme auf die verschiedenen Variablen im Programmtext und in Bedingungen der Korrektheitsbeweise werden wie folgt auferlegt:

Bezugnahme in ↓ auf Variablen →	kdat	kvrf	SG	EG	lokal zu S	lokal zu E
Programmtext Hauptprogramm Sender	N	N	N	N	SL	N
Programmtext sendvers	S	SL	SL	N	SL	N
Programmtext empfers	L	SL	N	SL	N	SL
Programmtext Hauptprogramm Empfänger	N	N	N	N	N	SL
Korrektheitsbeweis Hauptprogramm Sender	KI	KI	J	KI	J	N
Korrektheitsbeweis sendvers	J	J	J	KI	J	N
Korrektheitsbeweis empfers	J	J	KI	J	N	J
Korrektheitsbeweis Hauptprogramm Empfänger	KI	KI	KI	J	N	J

Die Abkürzungen bedeuten: N=nein (nicht erlaubt), J=ja (erlaubt), S=Schreibzugriff, L=Lesezugriff, KI=nur in KI als Teilausdruck.

Um die Korrektheit der nebenläufigen Prozessen S und E zu beweisen, muß man zuerst

- die Korrektheit des sequentiellen Prozesses S und unabhängig davon
 - die Korrektheit des sequentiellen Prozesses E beweisen.
- Gelingt das, dann muß man ferner beweisen, daß
- der Korrektheitsbeweis für Prozeß S durch den Prozeß E nicht gestört wird sowie daß
 - der Korrektheitsbeweis für Prozeß E durch den Prozeß S nicht gestört wird.

Der Korrektheitsbeweis für S gliedert sich in einen Teil für das Hauptprogramm Sender und in einen zweiten Teil für sendvers. In den für den Korrektheitsbeweis für das Hauptprogramm Sender maßgeblichen Bedingungen kommen zunächst nur die verschiedenen lokalen Programmvariablen sowie SG (aber weder kdat, kvrf noch EG) vor. Die für den Korrektheitsbeweis für sendvers maßgeblichen Bedingungen sind im Programmtext oben angegeben. Die Korrektheit des Hauptprogramms Sender und des Unterprogramms sendvers wird wie in dieser Arbeit bereits beschrieben bewiesen. Weil keine in KI vorkommende Variable von einer Anweisung im Hauptprogramm Sender verändert werden kann (siehe die Definition von KI und die Tabelle der vereinbarten Einschränkungen oben), darf "und KI" zu jeder für den Korrektheitsbeweis für das Hauptprogramm Sender maßgeblichen Bedingung hinzugefügt werden. KI muß so hinzugefügt werden, weil KI als Nachbedingung des Hauptprogramms Sender benötigt wird. Dies wiederum ist für den Beweis der korrekten Datenübertragung vom Prozeß S nach Prozeß E erforderlich.

Das entsprechende gilt für den Korrektheitsbeweis für E.

Für die Störung eines Korrektheitsbeweises durch den anderen Prozeß kommen grundsätzlich nur Anweisungen in Frage, die die gemeinsamen Variablen kdat, kvrf, SG und EG verändern können. (In den für den Korrektheitsbeweis für einen Prozeß maßgeblichen Bedingungen dürfen die lokalen Variablen eines anderen Prozesses nicht vorkommen, siehe die Tabelle der vereinbarten Einschränkungen oben.) Die gemeinsamen Variablen wer-

den nur in sendvers und empfvers und nicht in den Hauptprogrammen verändert. Es muß also grundsätzlich jede für den Korrektheitsbeweis des jeweiligen Programmteils maßgebliche Bedingung auf Störung durch die betreffende Zuweisung (vgl. Abschnitt 7.2.1) wie folgt geprüft werden:

Hauptprogramm Sender	mit kdat: = ... in empfvers	[1]
Hauptprogramm Sender	mit kvrf: = ... in empfvers	[2]
Hauptprogramm Sender	mit SG: = ... in empfvers	[3]
Hauptprogramm Sender	mit EG: = ... in empfvers	[4]
sendvers	mit kdat: = ... in empfvers	[5]
sendvers	mit kvrf: = ... in empfvers	[6]
sendvers	mit SG: = ... in empfvers	[7]
sendvers	mit EG: = ... in empfvers	[8]
empfvers	mit kdat: = ... in sendvers	[9]
empfvers	mit kvrf: = ... in sendvers	[10]
empfvers	mit SG: = ... in sendvers	[11]
empfvers	mit EG: = ... in sendvers	[12]
Hauptprogramm Empfänger	mit kdat: = ... in sendvers	[13]
Hauptprogramm Empfänger	mit kvrf: = ... in sendvers	[14]
Hauptprogramm Empfänger	mit SG: = ... in sendvers	[15]
Hauptprogramm Empfänger	mit EG: = ... in sendvers	[16]

Man merke, wie die in der Tabelle oben vereinbarten Einschränkungen die Durchführung dieser Prüfungen (siehe unten) erleichtern und den damit verbundenen Aufwand einschränken.

Die Notwendigkeit der Prüfungen [1] und [5] entfällt, weil in empfvers kdat nicht verändert wird. Die Prüfungen [3] und [7] entfallen, weil in empfvers SG nicht verändert wird. Auf die gleiche Weise erledigen sich die Prüfungen [12] und [16].

Die Prüfungen [2] und [4] bestehen aus vielen einzelnen Prüfungen der Form

$$\{Bs \text{ und } KI \text{ und } E3\} (kvrf, EG):=(send, EG\&kdat) \{Bs \text{ und } KI\} ?$$

wo Bs die jeweilige für den Korrektheitsbeweis für das Hauptprogramm Sender maßgebliche Bedingung ist. Weil weder kvrf noch EG in Bs vorkommen darf, gilt

$$\{Bs\} (kvrf, EG):=(send, EG\&kdat) \{Bs\}$$

Falls die Korrektheitsaussage

$$\{KI \text{ und } E3\} (kvrf, EG):=(send, EG\&kdat) \{KI\} \quad [2a]$$

gilt, werden gemäß der Beweisregel DC1 alle Prüfungen [2] und [4] erfüllt. Die Aussage [2a] kann durch Anwendung der oben erwähnten Verallgemeinerung der Beweisregel Z2 verifiziert werden. Die Prüfungen [2] und [4] sind damit bestanden.

Die Durchführung der Prüfungen [6] und [8] gliedert sich in das Verifizieren der folgenden Aussagen:

$$\begin{aligned} \{S1 \text{ und } E3\} (kvrf, EG):=(send, EG\&kdat) \{S1\} \\ \{S2 \text{ und } E3\} (kvrf, EG):=(send, EG\&kdat) \{S2\} \\ \{S3 \text{ und } E3\} (kvrf, EG):=(send, EG\&kdat) \{S3\} \\ \{S4 \text{ und } E3\} (kvrf, EG):=(send, EG\&kdat) \{S4\} \end{aligned}$$

$$\dots \\ \{S8 \text{ und } E3\} (kvrf, EG):=(send, EG\&kdat) \{S8\}$$

In den Bedingungen S1, S4, S5, S6, S7 und S8 kommen die Variablen kvrf und EG nur innerhalb des Teilausdrucks KI vor. Wegen [2a] gelten gemäß der Beweisregel DC1 die entsprechenden Aussagen oben.

Die Bedingungen (S2 und E3) und (S3 und E3) sind mit der logischen Konstante falsch äquivalent. Weil diese die universale Vorbedingung ist, gelten die entsprechenden Aussagen oben. Praktisch bedeutet das, daß die Ausführung der zwei Prozesse nie zu den Stellen S2 und E3 bzw. S3 und E3 gleichzeitig kommen kann. Dieses wird durch die zwei if-Bedingungen ausgeschlossen. Siehe auch unten.

Die Prüfung [9] gliedert sich in das Verifizieren der folgenden Aussagen:

$$\begin{aligned} \{E1 \text{ und } S2\} kdat:=sdat \{E1\} \\ \{E2 \text{ und } S2\} kdat:=sdat \{E2\} \\ \{E3 \text{ und } S2\} kdat:=sdat \{E3\} \\ \{E4 \text{ und } S2\} kdat:=sdat \{E4\} \\ \dots \\ \{E8 \text{ und } S2\} kdat:=sdat \{E8\} \end{aligned}$$

In den Bedingungen E1, E2, E4, E5, E6, E7 und E8 kommt die Variable kdat nur innerhalb des Teilausdrucks KI vor. Wegen (S2 \Rightarrow kvrf=send und KI) werden gemäß den Beweisregeln B1 und DC1 die entsprechenden Aussagen oben gelten, falls

$$\{kvrf=send \text{ und } KI\} kdat:=sdat \{KI\} \quad [9a]$$

welches durch Anwendung der Beweisregel Z2 verifiziert werden kann.

Die Bedingungen (E2 und S2) und (E3 und S2) sind mit der logischen Konstante falsch äquivalent. Weil diese die universale Vorbedingung ist, gelten die entsprechenden Aussagen oben.

Die Durchführung der Prüfungen [10] und [11] erfolgt analog zur Durchführung der Prüfungen [6] und [8] oben.

Die Prüfung [13] besteht aus vielen einzelnen Prüfungen der Form

$$\{Be \text{ und } KI \text{ und } S2\} kdat:=sdat \{Be \text{ und } KI\} ?$$

wo Be die jeweilige für den Korrektheitsbeweis für das Hauptprogramm Empfänger maßgebliche Bedingung ist. Weil kdat nicht in Be vorkommen darf, gilt

$$\{Be\} kdat:=sdat \{Be\}$$

Wegen [9a] oben gilt gemäß der Beweisregel B1 die Aussage

$$\{KI \text{ und } S2\} kdat:=sdat \{KI\}$$

Gemäß der Beweisregel DC1 sind damit alle einzelnen Prüfungen der Prüfungsgruppe [13] bestanden.

Die Durchführung der Prüfungen [14] und [15] erfolgt analog zur Durchführung der Prüfungen [2] und [4] oben.

Damit ist bewiesen worden, daß kein Prozeß die Korrektheitsbeweissführung des anderen Prozesses stört, d.h. ungültig macht.

Ein Beweis der Störungsfreiheit wie der oben geführte ist ein induktiver Beweis, auch wenn er nicht ausdrücklich die Form eines solchen aufweist. Im Induktionsschritt wird angenommen, daß die Ausführung der zwei Prozesse zur Stelle A im Prozeß S und zur Stelle B im Prozeß E gelangt ist, wobei die maßgeblichen Zwischenbedingungen an diesen Stellen wahr sind. Davon ausgehend wird bewiesen, daß

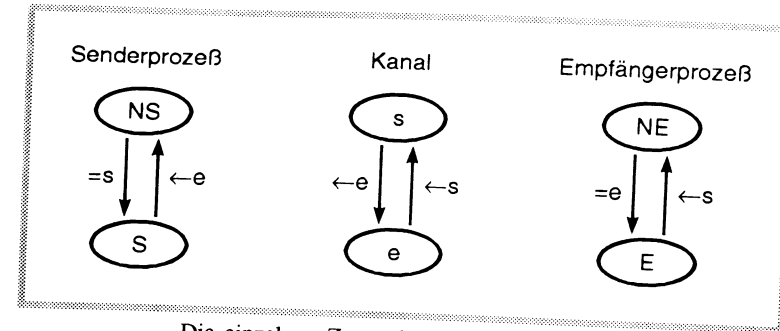
- nach der anschließenden Ausführung der nächsten nicht unterbrechbaren Anweisung bzw. Anweisungskomponente in S die für den Korrektheitsbeweis für E maßgebliche Zwischenbedingung an der Stelle B immer noch wahr ist sowie daß
- nach der anschließenden Ausführung der nächsten nicht unterbrechbaren Anweisung bzw. Anweisungskomponente in E die für den Korrektheitsbeweis für S maßgebliche Zwischenbedingung an der Stelle A immer noch wahr ist:

<u>Prozeß S</u>	<u>Prozeß E</u>
{VAs und VBe}	{VAs und VBe}
Ss	Se
{VBe}	{VAs}

Der Beweis wird prinzipiell für jedes Paar von Stellen A in S und B in E wiederholt.

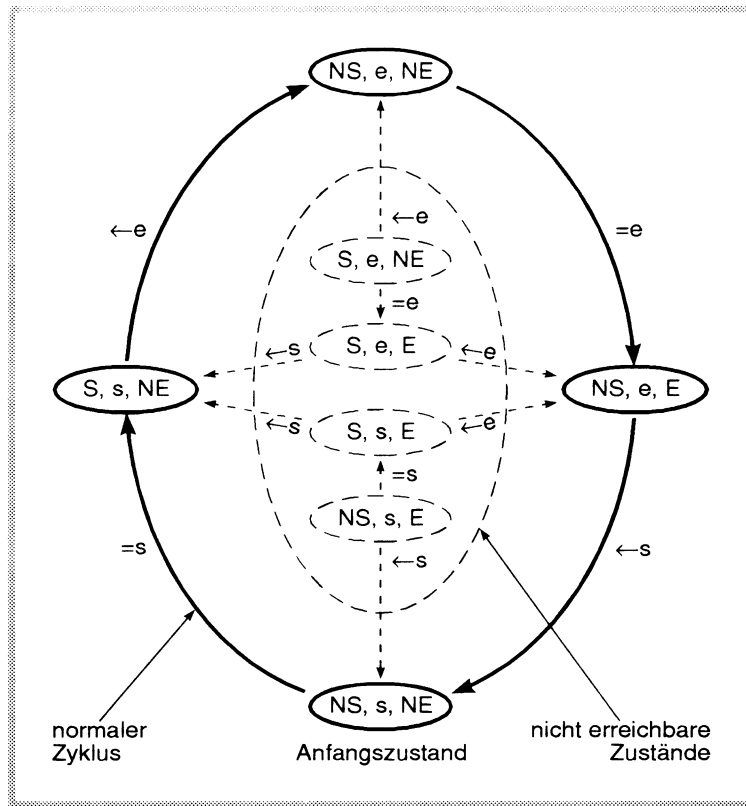
Die Basis für die Induktion bildet die Anfangssituation: Die Ausführung der nebenläufigen Prozesse S und E fängt am Anfang des Hauptprogramms Sender und am Anfang des Hauptprogramms Empfänger an, wobei die Kanalvariable KI anfangs wahr ist (z.B. mit $SG=[], EG=[]$ und $kvrf=send$, siehe die Bemerkungen zur Initialisierung oben). In diesem Beispiel sind diese Voraussetzungen Anforderungen an das übergeordnete ausführende System. Sind diese Anforderungen nicht erfüllt, dann trifft der oben angegebene Beweis der Störungsfreiheit auf die fragliche Ausführung nicht zu.

Für das intuitive, informale Verständnis des Ablaufs der Ausführung der Prozesse S und E ist es hilfreich, die Prozesse S und E sowie den Kanal als Automaten zu betrachten und entsprechende Zustandsdiagramme zu zeichnen. Wir definieren zwei Zustände für jeden Automaten. Der Prozeß S befindet sich im Zustand S während der Ausführung des ersten Teils des then-Zweigs der if-Anweisung im Unterprogramm sendvers, ansonsten befindet er sich im Zustand NS. Genauer: der Prozeß S geht vom Zustand NS in den Zustand S beim Lesezugriff auf den Wert send der Variable kvrf während der Auswertung der if-Bedingung in sendvers über. Der Prozeß S geht vom Zustand S in den Zustand NS bei der Zuweisung des Werts empf der Variable kvrf im then-Zweig der if-Anweisung in sendvers über. Die Zustände und die Zustandsübergänge des Prozesses E werden entsprechend definiert. Der Kanal ist im Zustand s genau dann, wenn die Kanalvariable kvrf den Wert send aufweist. Die einzelnen Zustandsübergangsdiagramme für die drei Automaten sind wie folgt:



Die einzelnen Zustandsübergangsdiagramme für den Übertragungskanal ohne Puffer

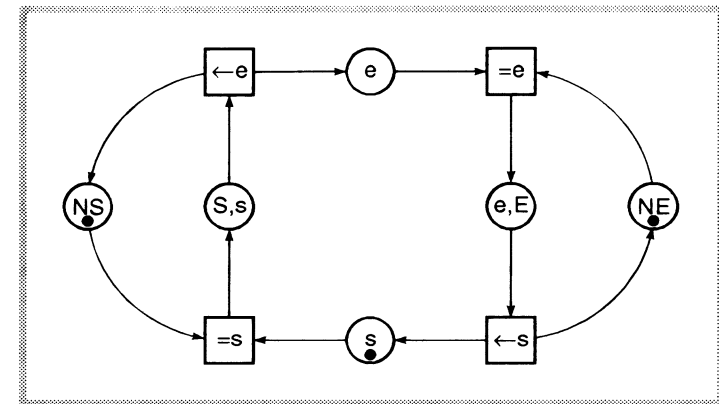
Bildet man das Produkt dieser drei Zustandsräume und betrachtet man alle möglichen Übergänge zwischen den Zuständen jenes Produktraums, so ergibt sich das folgende Zustandsübergangsdiagramm für den Produktautomaten:



Produktzustandsübergangsdiagramm für den Übertragungskanal ohne Puffer

Dieser Produktautomat zeigt, daß die bei der Ausführung der Prozesse S und E entstehende Folge der Zustände determiniert ist, vorausgesetzt, daß die Prozesse wie oben gefordert gestartet werden. Nicht determinierte Zustandsübergänge sind nur von Zuständen aus möglich, die vom Anfangszustand aus nicht erreichbar sind.

Die folgende Abbildung zeigt ein entsprechendes Petri-Netz:



Petri-Netz mit Anfangsmarkierung für den Übertragungskanal ohne Puffer

7.2.2.2 Gepufferter Übertragungskanal

Für die praktische Anwendung ist ein gepufferter Übertragungskanal gewöhnlich interessanter als der oben behandelte Kanal ohne Puffer. Die untenstehenden Versionen der Unterprogramme sendvers und empfvers verwirklichen einen gepufferten Übertragungskanal, in dem die zu kommunizierenden Nachrichten über das Feld kdat(.) übertragen werden. Das Feld umfaßt KN Elemente, genannt kdat(0), kdat(1), ... kdat(KN-1), wobei $KN \geq 2$. Diese Feldvariablen bilden einen kreisförmigen Puffer; dabei ist kdat(0) der Nachfolger von kdat(KN-1). Ein solcher Puffer wird auch *Ringpuffer* genannt. Diese und ähnliche Strukturen werden seit langer Zeit in implementierten Systemen (z.B. in Betriebssystemen) verwendet und sind in der Fachliteratur oft als Beispiele für parallele und nebenläufige Prozesse zu finden, z.B. [Zima], [Grams; pers. Komm.].

Die Kanalvariable ksend zeigt auf das erste Element in kdat, das dem Sender zur Verfügung steht, d.h. auf das Element, in das der Sender die nächste zu übertragende Nachricht ablegen soll. Die Kanalvariable kempf zeigt auf das erste Element in kdat, das dem Empfänger zur Verfügung steht, d.h. auf das Element, das der Empfänger als nächstes übernehmen soll. Der Bereich, der die in Übertragung befindlichen Nachrichten enthält, besteht aus den Feldvariablen $kdat(kempf \oplus 1)$, ... $kdat(ksend \ominus 1)$. Die Symbole \oplus und \ominus bedeuten hier Addition bzw. Subtraktion modulo KN. Falls $ksend = kempf$, dann ist der Übertragungspuffer leer.

Für die Formulierung der Kanalvariante (siehe unten) wird das Symbol \otimes (kreisförmige Konkatenation) wie folgt definiert:

$$\otimes_{i=a}^b [kdat(i)] \triangleq \begin{cases} [], & \text{falls } b = a \ominus 1 \\ \&_{i=a}^b [kdat(i)], & \text{falls } b \neq a \ominus 1 \text{ und } a \leq b \\ \&_{i=a}^{KN-1} [kdat(i)] \&_{i=0}^b [kdat(i)], & \text{falls } b \neq a \ominus 1 \text{ und } b < a \end{cases}$$

Die Folge von Werten, die im Kanalpuffer gespeichert sind, ist also

$$\bigotimes_{i=kempf}^{ksend} [kdat(i)]$$

Die Kanalvariante (Kommunikationsinvariante) KI für den gepufferten Kanal ist

$$KI \triangleq SG = EG \bigotimes_{i=kempf}^{ksend} [kdat(i)]$$

Die Initialisierung des Übertragungskanals muß für die anfängliche Wahrheit der Kanalvariante sorgen, z.B. dadurch, daß $kempf = ksend (=0)$ sowie SG und EG gleich der leeren Folge sind.

Die Spezifikationen der Übertragungsunterprogramme für den gepufferten Kanal (d.h. ihre Vor- und Nachbedingungen) sind die gleichen wie die für den ungepufferten Kanal in Abschnitt 7.2.2.1 oben.

Falls der Kanalpuffer nicht voll ist ($ksend < kempf$), überträgt `sendvers` die zu sendende Nachricht `sdat` in die erste freie Zelle des Kanalpuffers und fügt diese Zelle dem aktiven Teil des Kanalpuffers hinzu. Dadurch wird die Nachricht effektiv gesendet. Ist der Kanalpuffer voll, sendet `sendvers` die Nachricht nicht. In jedem Fall teilt `sendvers` über die lokale Variable `gsdat` dem aufrufenden Programm mit, ob die Nachricht in `sdat` gesendet wurde oder nicht.

Der Programmcode mit den für den Korrektheitsbeweis maßgeblichen Zwischenbedingungen für die gepufferte Version des Unterprogramms `sendvers` ist

```
{SG=SG' und KI}
if ksend < kempf [falls Kanalpuffer nicht voll]
then {ksend < kempf und SG=SG' und KI}
    kdat(ksend):=sdat
    {ksend < kempf und kdat(ksend)=sdat und SG=SG' und KI}
    (ksend, SG):=(ksend + 1, SG&[kdat(ksend)])
    {SG=SG'&[sdat] und KI}
    gesendet:=wahr
    {(gesendet und SG=SG'&[sdat]
    oder nicht gesendet und SG=SG') und KI}
else {SG=SG' und KI}
    gesendet:=falsch
    {(gesendet und SG=SG'&[sdat]
    oder nicht gesendet und SG=SG') und KI}
endif
{(gesendet und SG=SG'&[sdat] oder nicht gesendet und SG=SG') und KI}
```

Falls der Kanalpuffer nicht leer ist ($ksend = kempf$), überträgt `empfvers` die erste darin stehende Nachricht in `edat` und gibt diese Pufferzelle frei. Dadurch wird die Nachricht effektiv empfangen. Ist der Kanalpuffer leer, empfängt `empfvers` keine Nachricht. In jedem Fall teilt `empfvers` über die lokale Variable `empfangen` dem aufrufenden Programm mit, ob eine Nachricht empfangen wurde oder nicht.

Der Programmcode mit den für den Korrektheitsbeweis maßgeblichen Zwischenbedingungen für die gepufferte Version des Unterprogramms `empfvers` ist

```
{EG=EG' und KI}
if kempf = ksend [falls Kanalpuffer nicht leer]
then {kempf = ksend und EG=EG' und KI}
    edat:=kdat(kempf)
    {kempf = ksend und kdat(kempf)=edat und EG=EG' und KI}
    (kempf, EG):=(kempf + 1, EG&[kdat(kempf)])
    {EG=EG'&[edat] und KI}
    empfangen:=wahr
    {(empfangen und EG=EG'&[edat]
    oder nicht empfangen und EG=EG') und KI}
else {EG=EG' und KI}
    empfangen:=falsch
    {(empfangen und EG=EG'&[edat]
    oder nicht empfangen und EG=EG') und KI}
endif
{(empfangen und EG=EG'&[edat] oder nicht empfangen und EG=EG') und KI}
```

Der Beweis der Störungsfreiheit wird auf ähnliche Weise wie für den Übertragungskanal ohne Puffer geführt, siehe Abschnitt 7.2.2.1 oben.

Bei der Ausführung der Unterprogramme `sendvers` und `empfvers` oben werden stärkere Nachbedingungen einiger Anweisungen erfüllt als die oben angegebenen Zwischenbedingungen. Z.B. wird nach der Ausführung der Zuweisung $(ksend, SG) := \dots$ in `sendvers` die Bedingung $ksend < kempf$ zusätzlich erfüllt. Die Wahrheit dieser Bedingung kann jedoch von der Ausführung des Unterprogramms `empfvers`, insbesondere von der Ausführung der darin enthaltenen Zuweisung $(kempf, EG) := \dots$, gestört werden. Der gleiche Effekt tritt in die andere Richtung auf: Die Ausführung der Zuweisung $(ksend, SG) := \dots$ in `sendvers` kann die stärkere Nachbedingung $\{ksend < kempf \text{ und } EG = EG' \& [edat] \text{ und } KI\}$ bezüglich $(kempf, EG) := \dots$ in `empfvers` stören. Entsprechende Effekte treten auch in den Korrektheitsbeweisen für die nicht gepufferten Versionen von `sendvers` und `empfvers` in Abschnitt 7.2.2.1 auf. Die Korrektheitsbeweissführung für nebenläufige Prozesse kann also nicht nur von den fraglichen Programmen selbst, sondern auch von den in den einzelnen Korrektheitsbeweisen vorkommenden Zwischenbedingungen abhängen. Es ist durchaus möglich, daß ein Korrektheitsbeweis A1 für einen Prozeß A von einem anderen Prozeß B gestört, aber ein zweiter Korrektheitsbeweis A2 für den Prozeß A vom Prozeß B nicht gestört wird. Existiert für jeden Prozeß ein Korrektheitsbeweis, der von keinem anderen Prozeß gestört wird, dann ist die Korrektheit der nebenläufigen Kombination der Prozesse gewährleistet.

Der Beweisaufwand für nebenläufige Prozesse ist offensichtlich merklich höher als der für sequentielle Prozesse, auch wenn er durch geeignete Konstruktionsmaßnahmen und die Berücksichtigung bestimmter Konventionen in gewissen Grenzen gehalten werden kann. Es bietet sich deshalb die Konstruktionsstrategie an, Anwendungsprogramme möglichst nur als sequentielle Programmsegmente zu konstruieren und diese in eine geeignete nebenläufige Systemumgebung einzubetten, deren Korrektheit und gegenseitige Störungsfreiheit bereits bewiesen worden sind bzw. durch Systemkonventionen sichergestellt werden. Auf diese Weise könnten die Systemumgebung, ihre allgemein gültigen Unterprogramme (wie die Unterprogramme `sendvers` und `empfvers` in diesem Beispiel) und ihre Korrektheitsbeweise

oft und für mehrere Anwendungen "wiederverwendet" werden, um die gesamte Wirtschaftlichkeit zu verbessern bzw. zu gewährleisten.

7.2.3 Vollständige Korrektheit und Verklemmung/Verhungern (lassen)

Bei der Betrachtung eines sequentiellen Programms bzw. Prozesses beschäftigt man sich mit zwei Aspekten seiner Ausführung. Diese sind: die partielle Korrektheit (ein fehlerhaftes Ergebnis wird nicht erzeugt) und Terminierung (ein Ergebnis wird erzeugt). Aus diesen zwei Eigenschaften folgt die vollständige Korrektheit (ein richtiges Ergebnis wird erzeugt).

In einem System nebenläufiger Prozesse treten nicht nur diese Eigenschaften des jeweiligen Prozesses in Erscheinung, sondern auch übergeordnete systembezogene Eigenschaften, die in einem allgemeinen Sinne ähnlicher Natur sind. Diese werden oft in zwei Kategorien eingeteilt, sicherheitsbezogene und lebendigkeitsbezogene Eigenschaften (engl. safety and liveness properties [Andrews]). Lebendigkeitsbezogene Eigenschaften haben damit zu tun, daß erwünschte Ereignisse irgendwann eintreten werden (bzw. nicht eintreten werden). Sicherheitsbezogene Eigenschaften haben damit zu tun, daß unerwünschte Ereignisse nicht auftreten (bzw. auftreten). Einige solche Eigenschaften sind Funktionen des Ausführungszustands und andere sind Funktionen der Ausführungsgeschichte (der Folge von Ausführungszuständen) [Turski; "On Starvation ...", 1990].

Zu den oben angedeuteten Eigenschaften gehören z.B.

- Verklemmung (engl. deadlock): Ein oder mehrere Prozesse warten auf Ereignisse, die nicht mehr eintreten können, z.B. Prozeß A wartet auf die Erledigung einer Aufgabe durch den Prozeß B und B wartet auf die gleiche Weise auf A; weder A noch B wird je weiter fortschreiten.
- Verhungern (engl. starvation): Ein oder mehrere Prozesse warten auf Ereignisse, die künftig eintreten können, aber tatsächlich nicht eintreten (eventuell des Zufalls wegen). Die "unfaire" dynamische Zuteilung von Systemressourcen oder der Berechtigung zum Zugriff auf gemeinsame Variablen sowie die bevorzugte Behandlung bestimmter Arten von anstehenden Aufgaben sind klassische Ursachen des Verhungerns.

Diese und verwandte Aspekte der Nebenläufigkeit bilden ein sehr umfassendes und reichhaltiges Forschungsgebiet, das trotz langer und intensiver Arbeit noch nicht als abgeschlossen betrachtet werden kann. Die bereits zitierte Literatur über Nebenläufigkeit vermittelt Einblicke in dieses Gebiet. Allein über Verklemmung existiert z.B. eine sehr umfangreiche und vielfältige Literatur.

Weil dieser Themenkomplex zum Kern des Gebiets der Nebenläufigkeit gehört, das wiederum außerhalb des Rahmens dieser Arbeit liegt, wird hier nicht näher darauf eingegangen. Lediglich soll an dieser Stelle festgehalten werden, daß die vollständige Korrektheit jedes zu einem System gehörenden sequentiellen Programms nicht ausreicht, um die geforderte Lauffähigkeit und Eignung eines aus jenen Prozessen zusammengesetzten nebenläufigen Systems im allgemeinen zu gewährleisten. (Man kann z.B. mehrere Prozesse wie die in Abschnitt 7.2.2 behandelten derart zusammensetzen, daß sie sich gegenseitig verklemmen.) Die übergeordnete Zusammensetzung und die Art der Wechselwirkung zwischen den Prozessen können dafür maßgeblich sein, ob das System z.B. verklemmungsfrei ist oder nicht. Aus einer Analyse der einzelnen Prozesse allein lassen sich ausreichende

Aussagen über wesentliche Systemeigenschaften meist nicht machen. "Ein System ist mehr als die Summe seiner Bestandteile."

7.3 Objektorientierte Programmierung am Beispiel der OOP-Sprache Eiffel

In diesem Abschnitt werden Aspekte der objektorientierten Programmierung in Zusammenhang mit der Korrektheitsbeweissführung betrachtet. Ihre Berücksichtigung und Behandlung in einem Korrektheitsbeweis wird anhand eines Beispiels in Abschnitt 7.3.3 veranschaulicht. Die Basis der Betrachtung ist die objektorientierte Programmiersprache Eiffel, wie sie in [Meyer; *Eiffel ...*, 1992] definiert und in anderen Veröffentlichungen von [Meyer] näher beschrieben und erläutert wird. Beziehungen zwischen der objektorientierten Programmierung und verschiedenen Aspekten formaler Methoden, VDM, Z, formaler Spezifikationen usw. werden in [Alencar], [America], [Chedgely], [Cusack], [Hogg], [Holt], [Minkowitz], [Parnas; 1989 Oct.], [Robinson], [Thomas, 1993 Jan.-Feb.] und [Toetenel] z.T. angesprochen und z.T. detailliert behandelt.

In Zusammenhang mit der Korrektheitsbeweissführung sind insbesondere die folgenden Aspekte der objektorientierten Programmierung von Bedeutung:

- Abstrahierung,
- Datenkapselung (sowohl hinsichtlich der Gruppierung/Bindung als auch der Isolierung/Verbergung von Daten und Datenstrukturen),
- Vererbung und Polymorphismus.

Die Abstrahierung und die Datenkapselung führen typischerweise dazu, daß die externe Sicht über Datengruppen und Objekten (also vom Standpunkt der Entwickler aufrufender Programmsegmente, d.h. "Klienten", aus betrachtet) viel allgemeiner und weniger detailliert ist als die interne Sicht (also vom Standpunkt des Entwicklers der fraglichen Klasse aus gesehen). Unterschiedliche Spezifikationen (Vor- und Nachbedingungen) sind für diese zwei Gruppen von Softwareentwicklern angebracht. Diese Anforderung wird unten näher betrachtet.

Vererbung und insbesondere der damit zusammenhängende Polymorphismus führen dazu, daß ein Name, der eine Routine (ein Unterprogramm) identifizieren soll, die Routine nicht eindeutig bestimmt. Nur aus der Kombination eines Namens und der Klasse des Objekts, worauf die Routine angewendet wird, wird die angesprochene Routine (das aufgerufene Unterprogramm) effektiv bestimmt.

Die hier behandelten Eigenschaften, die als kennzeichnende Merkmale objektorientierter Programmiersprachen betrachtet werden, können, wenn auch in etwas anderer Form oder weniger ausgeprägt, in früheren nicht objektorientierten Programmiersprachen und -systemen gefunden werden. Darunter sind ALPHARD, CLU, EUCLID, GYPSY und LUCID, die in einem gewissen Sinne als Vorläufer von Eiffel angesehen werden können [Hoffmann; pers. Komm.].

[Meyer] sieht die Korrektheit als einen wesentlichen und zentralen Punkt bei der Programmkonstruktion. Diese Einstellung kommt an mehreren Stellen seiner Veröffentlichungen — teils explizit, teils implizit — zum Ausdruck. Meyers Konzept "design by contract" (auf einem "Vertrag" basierende Konstruktion) hängt sehr eng mit seinem Streben nach nachweisbarer Korrektheit zusammen. Tatsächlich geht Eiffel hinsichtlich der Unterstüt-

zung korrekttheitsbezogener Sprachelemente weiter als die meisten anderen Programmiersprachen — wenn auch kein völliges Neuland dabei betreten wird. Einige Unzulänglichkeiten, die z.T. auf erforderliche Ergänzungen und z.T. nur auf wichtige Erweiterungen hinweisen, sind jedoch zu vermerken, siehe Abschnitte 7.3.2 und 7.3.4. Zusammengefaßt kann festgestellt werden, daß Eiffel in Bezug auf die Korrektheitsbeweissführung und eine entsprechende Vorgehensweise bei der Programmkonstruktion erfreulich weit, aber doch nicht weit genug geht. Die aufgezeigten Mängel schließen jedoch die Anwendung der in dieser Arbeit vorgeführten Korrektheitsbeweissführung und korrekttheitsbezogenen Konstruktionsmaßnahmen keinesfalls aus. Lediglich muß der Konstrukteur sie außerhalb des formalen Rahmens der Sprache Eiffel anwenden — wie bei anderen gegenwärtigen Programmiersprachen auch.

7.3.1 Korrektheitsbeweissführungsrelevante Besonderheiten von Eiffel

Eiffel (wie andere objektorientierte Programmiersprachen) weist einige Besonderheiten auf, die bei der in dieser Arbeit beschriebenen Vorgehensweise zur Korrektheitsbeweissführung und zur Programmkonstruktion berücksichtigt werden müssen, auch wenn sie keine grundsätzlichen Hindernisse darstellen. Im folgenden Text basieren alle spezifischen Angaben über Eiffel auf [Meyer; *Eiffel* ..., 1992].

7.3.1.1 Variablen und Objekte

Bei der Korrektheitsbeweissführung ist es zweckmäßig, Variablen und Objekten eines in Eiffel geschriebenen Programms auf das in Kapitel 3 vorgestellte Grundmodell zurückzuführen. Eiffel kennt zwei Arten von Variablen, *expanded* und *reference*. (Für "Variable" wird in der Eiffel-Terminologie der englische Begriff "entity" oft verwendet.) Der Wert einer Variable der ersten Art ist der Wert des fraglichen Datenelements selbst. Eine solche Variable ist eine Programmvariable wie in Abschnitt 3.1.1 definiert. Der Wert einer Variable der zweiten Art ist ein Zeiger auf das fragliche Datenelement. Betrachtet man einen solchen Zeiger bzw. Zeigerwert als Indexwert zu einem Feld, dann paßt eine solche Variable in das in Abschnitt 3.1 festgelegte Schema hinein, siehe insbesondere Abschnitt 3.1.2. Die Eiffel-Variable `konto.saldo` z.B., wo `konto` eine Bezugsvariable (= "reference" Variable) ist, entspricht der Feldvariable `saldo(konto)`, wie in den Abschnitten 3.1.1 und 3.1.2 definiert und erläutert. Dabei ist die der Bezugsvariable `konto` zugeordnete Menge eine geeignete Menge von Indexwerten. Diese Menge muß typischerweise nicht näher definiert werden; sie ist implementationsabhängig und wird vom Programmiersprachensystem festgelegt und ggf. verwaltet. Ist das Objekt selbst ein Feld, stellt die Bezugsvariable, die auf das Objekt zeigt, eine zusätzliche Indexdimension dar. Z.B. kann das Attribut (die Variable) `st.x(3)` in Eiffel als `x(3, st)` (oder, alternativ, `x(st, 3)`) in der in dieser Arbeit sonst üblichen Schreibweise betrachtet werden.

In Eiffel werden Variablen bzw. Objekte, worauf Bezugsvariablen zeigen, von im System (z.T. oder ganz) vordefinierten Unterprogrammen, deren wesentlicher Teil eine oder mehrere `declare`-Anweisungen ist, dynamisch (d.h. zur Programmausführungszeit) erzeugt (engl. Eiffel-Begriff: "create"). Darin werden ein geeigneter Zeigerwert vom System ermittelt, die Bestandteile des zu erzeugenden Objekts vereinbart und der Zeigerwert der

Bezugsvariable, worauf die `Create`-Routine angewendet wurde, zugeordnet. Außer eventuell internen Systemvariablen werden keine anderen Variablen verändert. Als Beispiel betrachte man die Eiffel-Anweisung `!!p` in einer Umgebung, in der die folgenden Vereinbarungen zutreffen:

```
p: POINT;
...
class POINT feature
x, y, z: REAL;
...
```

Die Eiffel-Anweisung `!!p` entspricht dem folgenden zum Eiffel-System gehörenden Programmsegment:

```
{wahr}                                [{Aktivzeig=Aktivzeig'}]
p:=...
{p∈Syszeig-Aktivzeig'}
Aktivzeig:=Aktivzeig'∪{p}
declare (x(p), REAL, 0); declare (y(p), REAL, 0); declare (z(p), REAL, 0)
{p∈Syszeig-Aktivzeig' ∧ Aktivzeig=(Aktivzeig'∪{p})
 ∧ Menge."x(p)"=REAL ∧ x(p)=0
 ∧ Menge."y(p)"=REAL ∧ y(p)=0
 ∧ Menge."z(p)"=REAL ∧ z(p)=0}
```

In der ersten Zuweisung wird ein zulässiger und noch nicht zugeteilter Zeigerwert ermittelt und der Bezugsvariable `p` zugeordnet. Dabei ist `Syszeig` die Menge aller zulässigen Zeigerwerte; diese Menge ist implementationsabhängig und während der Ausführung eines Programms konstant. `Aktivzeig` ist die (veränderliche) Menge aller aktiven Zeigerwerte, d.h. die Menge der Zeigerwerte, die vom System bereits vergeben und noch nicht wieder freigegeben bzw. wieder eingesammelt worden sind. `Aktivzeig` wird nur durch die Ausführung von Anweisungen, die Objekte erzeugen, und durch das in Eiffel automatische Aufräumungsverfahren verändert. `Syszeig` und `Aktivzeig` müssen nicht explizit als Variablen, auch nicht als systeminterne Variablen, geführt werden, vgl. die Variablen `SG` und `EG` in Abschnitt 7.2.2.

Ist eine zu vereinbarende Variable eine Bezugsvariable, dann werden ihr die Menge `Syszeig` und der Wert `void` zugeordnet.

Es wird ferner unterstellt, daß jede systemeigene `Create`-Routine immer mit einem definierten Ergebnis ausgeführt wird (vollständig korrekt ist). Diese Annahme vernachlässigt nur die Möglichkeiten, daß die vorhandenen Systemressourcen nicht ausreichen (z.B. `Aktivzeit'=Syszeig`), vgl. Abschnitt 1.2, oder daß das ausführende Eiffel-System selbst nicht korrekt ist.

Für die Korrektheitsbeweissführung hinsichtlich der Erzeugung des Objekts in diesem Beispiel ist nur die Korrektheitsaussage

```

{wahr}                                     [{Aktivzeig = Aktivzeig'}]
!!p
{pεSyszeig-Aktivzeig' ∧ ((Aktivzeig' ∪ {p}) ⊆ Aktivzeig)
 ∧ Menge.“x(p)”=REAL ∧ x(p)=0
 ∧ Menge.“y(p)”=REAL ∧ y(p)=0
 ∧ Menge.“z(p)”=REAL ∧ z(p)=0} strikt

```

wesentlich. Diese Korrektheitsaussage stellt eine Annahme über die Auswirkung der Ausführung der Anweisung !!p durch das fragliche Eiffel-System dar.

Bei den Ausführungen oben wird unterstellt, daß die Werte von Zeigern auf Objekte aller Klassen Elemente aus einer gemeinsamen Menge (z.B. dem Speicheradreßraum) sind. Wird für jede Klasse bzw. für jede Gruppe von vererbungsverwandten Klassen eine eigene Menge von Zeigerwerten vom System zur Verfügung gestellt, dann müßten die Ausdrücke oben entsprechend revidiert werden.

7.3.1.2 Routinen

Aufrufe auf Routinen (Unterprogramme) mit formaler Parameterübergabe (in der Eiffel-Fachsprache besser “Argumentübergabe”) sind in Eiffel vorgesehen. Dieses Thema ist allgemeiner bezogen in Abschnitt 3.3.4.4 behandelt. Eiffel bringt in dieser Hinsicht keine besondere eigene Problematik mit sich. In Eiffel ist nur der Wertübergabemechanismus vorgesehen. Dabei kann natürlich der übergebene Wert ein Zeiger auf ein Objekt sein, wodurch der Effekt der Namensübergabe (engl. “call by name”) oder der Bezugnahme (engl. “call by reference”) erreicht werden kann. Siehe die Bemerkungen in Abschnitt 7.3.1.1 oben über Bezugsvariablen.

Wie in den einführenden Bemerkungen zum Abschnitt 7.3 oben erwähnt, kann Polymorphismus zu Mehrdeutigkeit beim Aufruf einer Routine führen. Die eigentlich angesprochene Routine wird vom System erst zur Ausführungszeit in Abhängigkeit vom Objekt, worauf im Aufruf Bezug genommen wird, ermittelt. Aus der Sicht der Korrektheitsbeweissführung besteht prinzipiell für jedes Objekt (genauer: für jede Klasse) eine eigene (und unterschiedliche) Routine mit dem fraglichen Namen. Bei der Anwendung der Beweisregeln U1, U2 und U3 (siehe Abschnitt 3.2.4) auf den Aufruf muß die tatsächlich angesprochene Routine bekannt bzw. zu bestimmen sein (z.B. mit Hilfe geeigneter Teilausdrücke in den relevanten Zwischenbedingungen), es sei denn, daß alle gleichnamigen Routinen die gleiche Spezifikationen (Vor- und Nachbedingungen) aufweisen oder daß die aufrufende Stelle nur von einer gemeinsamen (d.h. von allen in Frage kommenden Routinen erfüllten) Spezifikation (vgl. Abschnitt 5.1) ausgeht. Die Ermittlung der angesprochenen Routine läuft analog zur Ermittlung einer Feldvariable ab: abhängig vom Wert der Bezugsvariable (der Indexvariable bzw. des Indexausdrucks) wird die spezifische Routine (Feldvariable) ausgewählt. Effektiv bezieht sich der Name rt der Routine auf eine Familie von Routinen rt(.). Der Aufruf obj.rt bezieht sich also auf die Routine rt(obj) oder, genauer, rt(Klasse(obj)).

In Eiffel wird hinsichtlich der Vor- und Nachbedingungen von gleichnamigen (polymorphen) Routinen einer Ererbungslinie verlangt, daß eine in einer erbenden Klasse (Nachkommensklasse) redeclarierte Routine eine allgemeinere Spezifikation (vgl. “allgemeineres” Programmsegment in Abschnitt 5.1) als die der vererbenden Klasse (Vorfah-

rensklasse) aufweisen muß, d.h., daß die Vorbedingung der Nachkommensroutine mindestens so schwach und die Nachbedingung mindestens so stark sein müssen als die der Vorfahrensklasse. Diese Einschränkung verändert die Problematik der Ermittlung der angesprochenen Routine bei Polymorphismus nicht, bedeutet jedoch, daß die eigentlich angesprochene Routine ggf. nicht ermittelt werden muß. Seien z.B. v eine als VORFAHR vereinbarte Bezugsvariable und NACHKOMME eine Klasse, die ein Nachkomme (Erbe) der Klasse VORFAHR ist. Ferner kommen Routinen mit dem Namen rt in beiden Klassen vor. Der Wert der Variable v kann auf ein Objekt der Klasse VORFAHR oder auf ein Objekt der Klasse NACHKOMME zeigen. Ist ein bestimmter Aufruf auf rt in VORFAHR korrekt, dann kann dieser Aufruf durch einen Aufruf auf rt in NACHKOMME ersetzt werden (vgl. die Beweisregel B1 und Abschnitt 5.2, Schwächung von Vu und Stärkung von Pu). D.h., reichen die Vor- und Nachbedingungen von rt in VORFAHR (der für v vereinbarten Klasse) für die Korrektheitsbeweissführung aus, dann muß die tatsächlich angesprochene Routine rt nicht ermittelt werden. Falls jedoch die schwächere Vorbedingung oder die stärkere Nachbedingung bezüglich einer Nachkommensroutine für das Gelingen des Beweises erforderlich sind, dann muß bei der Korrektheitsbeweissführung die tatsächlich angesprochene Routine ermittelt und die entsprechenden Vor- und Nachbedingungen verwendet werden.

Beim Aufruf einer Routine wird die Variable “Current” vom Eiffel-System automatisch verwaltet. Der Wert von Current zeigt auf das Objekt des aktuellen Aufrufs, es sei denn, das fragliche Objekt ist der Art **expanded**, in welchem Falle Current das betroffene Objekt selbst ist. Die Variable Current ist innerhalb der aufgerufenen Routine ausdrücklich ansprechbar, darf aber nicht verändert werden. Ist z.B. obj eine Bezugsvariable, dann ist der Aufruf obj.rt äquivalent zur Folge (declare (Current, Syszeig, obj); call rt(Klasse(obj)); release Current).

Während der Ausführung einer Funktionsroutine wird auf ähnliche Weise die Ergebnisvariable Result automatisch geführt.

7.3.1.3 Zugriffseinschränkungen

In Eiffel herrschen verschiedene Einschränkungen hinsichtlich der Zugänglichkeit von Attributen (Variablen) und Routinen (Unterprogrammen). Z.B. sind nicht exportierte Attribute nur innerhalb der fraglichen Klasse zugänglich. Die in Kapitel 3 definierte Modellsprache, die die Basis für die in dieser Arbeit vorgestellten Beweisregeln bildet, erlegt derartige Einschränkungen nicht auf. Solche Einschränkungen können bei der Korrektheitsbeweissführung außer Acht gelassen werden, mindestens insofern es um die partielle Korrektheit geht. D.h., bei der Korrektheitsbeweissführung geht man davon aus, daß alles exportiert wird. Eine solche Vorgehensweise wird nur denjenigen Situationen nicht gerecht, in denen ein in Eiffel geschriebenes Programm nicht ausgeführt wird, weil es Bezüge auf nicht zugängliche Variablen enthält. Um solche Situationen ausführlich zu behandeln, muß eine zusätzliche, Eiffel spezifische Analyse durchgeführt werden, die typischerweise vom Programmiersprachensystem automatisch erledigt wird. Derartige Einschränkungen der Zugänglichkeit von Variablen sind keine Eigenart von Eiffel, sondern insbesondere in blockstrukturierten Programmiersprachen seit langem bekannt.

Mit anderen Worten, die Korrektheitsbeweissführung bezieht sich auf die Semantik des fraglichen Programms und setzt sich grundsätzlich auf ein syntaktisch korrektes Programm

auf. (Nur ein syntaktisch korrektes Programm kann überhaupt eine semantische Bedeutung haben.) Zur Syntax gehören Fragen der Zugänglichkeit der verschiedenen Variablen von den verschiedenen Stellen des Programms aus, der Typenverträglichkeit usw.

7.3.1.4 Ausnahmebehandlung

Eiffel (wie auch manche anderen Programmiersprachen) sieht die Ausführung von besonderen Programmsegmenten für die Behandlung von Ausnahmen ("exceptions") vor. Man kann darüber streiten, ob die Verwendung dieser Möglichkeit überhaupt in Einklang mit der Philosophie der Programmkorrektheitsbeweissführung oder der Konstruktion beweisbarer korrekter Programme steht, insbesondere dann, wenn die fragliche Ausnahme die Verletzung einer spezifizierten Vor-, Nach- oder Zwischenbedingung oder Invariante ist. (Wenn z.B. die Behandlung einer Verletzung einer Vorbedingung beabsichtigt ist, sollte eigentlich eine entsprechende Schwächung der Vorbedingung — und somit eine Erweiterung der Spezifikation — in Erwägung gezogen werden.) Auf einige strittige Aspekte der Ausnahmebehandlung weist [Meyer; 1988, S. 157] hin. Auf diese Kontroverse wird hier nicht näher eingegangen. [Arbib; 1979] betrachtet die Behandlung von Programmsegmenten mit mehreren Ausgängen in Korrektheitsbeweisen und gibt Beweisregeln dafür an; solchen Programmstrukturen entsprechen die in Eiffel vorhandenen Strukturen für die Ausnahmebehandlung. [Feder] behandelt das Thema Ausnahmebehandlung in objektorientierten Sprachen weiter.

Die Übertragung der Ausnahmebehandlungsstrukturen von Eiffel in entsprechende Anweisungen bietet eine im Prinzip einfache aber ggf. mühselige Möglichkeit für die Korrektheitsbeweissführung für ein gegebenes in Eiffel geschriebenes Programm, das von den in Eiffel vorgesehenen Ausnahmebehandlungsmechanismen Gebrauch macht. Gleichzeitig werden dabei die Ausnahmesituationen, worauf geprüft wird, deutlich und ausführlich zum Ausdruck gebracht. Im Gegensatz dazu halten die in Eiffel angebotenen Ausnahmebehandlungsstrukturen und -mechanismen die auslösenden Ausnahmesituationen selbst eher verborgen.

7.3.1.5 Unbestimmte Bezüge auf Variablen und Funktionen

Es ist oft nicht möglich, festzustellen, ob ein Bezug auf ein Merkmal ("feature") einer anderen Klasse (einer Lieferantenklasse) ein Bezug auf eine Funktion oder auf ein Attribut (eine Variable) ist. Diese Eigenschaft war bei der Planung der Sprache Eiffel offensichtlich beabsichtigt und hängt mit der Idee zusammen, klasseninterne Details verborgen zu halten ("information hiding"). Das wirkt sich insbesondere bei der Konstruktion aus, in dem die Festlegung bestimmter Implementationsdetails hinausgeschoben werden kann. In Abschnitt A1.2.4 des Anhangs 1 kann z.B. die mathematische Funktion Af als Variable in der fraglichen Spezifikationsstufe aufgefaßt, jedoch in der detaillierteren Spezifikationsstufe des Abschnitts A1.2.5 als Ausdruck betrachtet bzw. festgelegt und ggf. als Funktionsroutine implementiert werden. Die Definitionen der Werte einer Variable und eines Ausdrucks (= einer Funktion der darin vorkommenden Variablen) in Abschnitt 3.1.2 widerspiegeln die gleiche Unbestimmtheit; beide sind mathematisch gesehen Funktionen auf \mathbf{D} .

Siehe auch den Absatz über Bezüge auf Funktionsroutinen in Zusicherungen in Abschnitt 7.3.4, Seite 153.

7.3.2 Die korrektheitsbezogenen Konstrukte in Eiffel

Mehrere Sprachelemente von Eiffel zielen unmittelbar auf die Programmkorrektheitsnachweisführung sowie auf die Unterstützung der Korrektheitsbeweissführung. Wie bereits erwähnt und unten näher erläutert reichen diese korrektheitsbezogenen Konstrukte in der Regel jedoch nicht aus, um einen mathematisch vollständigen Beweis der Programmkorrektheit aufzustellen. Die korrektheitsbezogenen Sprachelemente von Eiffel sind:

- **require** (Vorbedingung)
- **ensure** (Nachbedingung)
- **check** (Zwischenbedingung)
- **invariant** (Klasseninvariante, Schleifeninvariante)
- **variant** (Schleifenvariante)
- **old** (vorheriger Wert eines Ausdrucks, vgl. Abschnitt 3.1.6)
- **strip** (zur Bildung bestimmter nützlicher Teilausdrücke in Nachbedingungen)
- **rescue, retry** (zur Ausnahmebehandlung)

Gegenstand der Konstrukte **require**, **ensure**, **check** und **invariant** ist ein wie sonst in Eiffel vorgesehener Boolescher Ausdruck oder ein Kommentar. In einem Booleschen Ausdruck sind die Quantifikatoren \forall und \exists sowie entsprechende Reihen nicht erlaubt. Derartige Teilausdrücke dürfen nur als Kommentare vorkommen, wo sie vom Eiffel-System nicht unterstützt werden. Diese Einschränkung ist aus praktischer Sicht als wesentlich anzusehen. Nicht selten verführt sie den Softwareentwickler, die nicht unterstützten Reihen nicht in Kommentare einzuführen, sondern ganz wegzulassen. Typischerweise sind gerade diese Teilausdrücke die wesentlichen sowie diejenigen, die die komplexeren logischen Aspekte der Aufgabe beinhalten. Die übrigen Teile der Vor-, Nach- und Zwischenbedingungen wirken oft fast inhaltslos oder trivial. Auf jeden Fall reichen sie für einen Korrektheitsbeweis nicht aus. Vgl. die verschiedenen Beispielen in [Meyer; *Object-oriented Software Construction*, 1988] und [Meyer; *Eiffel ...*, 1992] und Abschnitt 7.3.3 unten.

Zwei Arten von Invarianten sind vorgesehen, sie können aber weggelassen werden. Die Klasseninvariante ist eine Nachbedingung jeder zur fraglichen Klasse gehörenden Create-Routine und sowohl eine Vor- als auch eine Nachbedingung jeder anderen zur fraglichen Klasse gehörenden Routine. Die Schleifeninvariante in Eiffel ist die gleiche Schleifeninvariante, die in den Beweisregeln W1 und W2 eine zentrale Rolle spielt (siehe Abschnitt 3.2.2).

Gegenstand des Konstrukts **variant** (Schleifenvariante) ist ein Ausdruck mit nichtnegativen ganzzahligen Werten. In einem gewissen Sinne mag diese Einschränkung theoretisch hinnehmbar sein, sie ist trotzdem aus praktischer Sicht übertrieben restriktiv. Eine für das Beispiel in Abschnitt 7.1.2 natürliche Schleifenvariante könnte zwar im Prinzip in eine ganzzahlige Funktion umgewandelt werden, aber eine solche Funktion würde aus der Sicht der Problemstellung gezwungen und künstlich wirken. Ferner ist es oft einfacher, einen auf einer gewöhnlichen Konvergenzargumentation basierenden Terminierungsbeweis für eine Schleifenvariante mit reellen Werten aufzustellen als eine geeignete ganzzahlige Schleifenvariante zu ermitteln.

Der Eiffel-Begriff **old** entspricht dem in dieser Arbeit verwendeten Symbol \prime . Er ermöglicht es, in einer Nachbedingung Bezug auf den ursprünglichen Wert einer Variable oder eines Ausdrucks zu nehmen. Diese Möglichkeit ist für die praktische Anwendung wichtig, worauf verschiedene Beispiele in dieser Arbeit hinweisen.

Die Eiffel-Begriffe **strip** und **old** zusammen ermöglichen es anzugeben, daß die Werte gewisser Variablen des aktuellen Objekts durch die Ausführung einer Routine nicht verändert werden sollen. Eine solche Angabe erfolgt in der Nachbedingung der betroffenen Routine, z.B. im Ausdruck `equal(strip(x, y), old strip(x, y))`, der besagt, daß die Werte von Variablen des aktuellen Objekts (und ggf. der aktuellen Klasse, siehe [Meyer; Eiffel ..., 1992, Abschnitt 23.21]) — außer den Variablen x und y — nicht verändert werden dürfen. Weil die Verwendung von **strip** auf besondere Weise beschränkt wird, kann nur auf Umwege angegeben werden, daß andere (d.h. nicht aktuelle) Objekte bzw. Komponenten davon nicht verändert werden dürfen. In früheren Versionen von Eiffel war der Begriff "Nochange" für diesen Zweck vorgesehen. Es ist oft einfacher, die Objekte und Variablen anzugeben, die durch die Ausführung der fraglichen Routine verändert werden können, denn ihre Anzahl ist gewöhnlich kleiner als die Anzahl der dadurch nicht veränderten Objekte und Variablen, auch unter Berücksichtigung der von Eiffel auferlegten Zugriffseinschränkungen. Die Angabe einer vollständigen Liste der nicht veränderten Objekte und Variablen kann unpraktikabel und umständlich, wenn nicht sogar unmöglich, sein. Siehe auch den Absatz über **strip**, **old** und **maychange** in Abschnitt 7.3.4, Seite 154.

In Eiffel haben Korrektheitsbetrachtungen die Struktur der Ausnahmebehandlung beeinflußt und in einem gewissen Maße eingeschränkt. Nur zwei Ausgänge aus einer Routine sind vorgesehen: ein für die "normalen" Fälle und der andere für die Ausnahmesituationen, der zum betroffenen Programmsegment "rescue" führt. Ausdrückliches Ziel dieses Programmsegments ist es, für die Einhaltung der Nachbedingung und ggf. der Klasseninvariante, d.h. für die Konsistenz der betroffenen Daten, zu sorgen. Wird in einem **rescue**-Programmsegment die Anweisung **retry** ausgeführt, hat der vorherige Teil des Programms für die Wahrheit der Vorbedingung und ggf. der Klasseninvariante zu sorgen. Versagen solche Versuche, die Nachbedingung zu erfüllen, dann wird eine Ausnahme bei der Ausführung der aufrufenden Routine ausgelöst. Die Auslösung solcher Ausnahmen wird ggf. in höhere Ebenen der Ausführungshierarchie iterativ fortgesetzt. Wird die Ausnahme nirgendwo erfolgreich aufgefangen, dann endet die Ausführung des Programms abnormal, d.h. die Ausführung des Programms führt zu keinem definierten Ergebnis.

Auf Wunsch werden während der Ausführung eines Programms Vor-, Nach- und Zwischenbedingungen, Klassen- und Schleifeninvarianten sowie Schleifenvarianten ausgewertet. Wird dabei falsch als Wert einer Bedingung festgestellt oder verringert sich eine Schleifenvariante nicht, wird eine Ausnahme ausgelöst. Auf diese Weise kann die Korrektheit der jeweiligen Ausführung (des jeweiligen Programmlaufs) verifiziert werden — im Gegensatz zu einem Korrektheitsbeweis, der die Korrektheit des Programms, d.h. die Korrektheit aller möglichen Ausführungen, verifiziert. Merkwürdig und potentiell problematisch bei der Inanspruchnahme dieser Möglichkeit zur Ausführungsverifizierung ist die Tatsache, daß das Ergebnis des Programmlaufs durch eventuelle Nebenwirkungen beim Auswerten der Ausdrücke beeinflusst werden kann, d.h. unterschiedliche Ergebnisse können sich aus der Programmausführung ergeben, je nach dem, ob dieser Auswertungsmechanismus ein- oder ausgeschaltet wird. Der Nutzen dieser Möglichkeit wird ferner durch

die bereits erwähnten Einschränkungen hinsichtlich der Form der zulässigen Bedingungen (siehe oben) deutlich begrenzt.

7.3.3 Beispiel der Korrektheitsbeweissführung für eine Klasse

[Meyer; 1988, Abschnitt 4.7.5] gibt eine abstrakte Spezifikation für ein System von Funktionen an, das einen Kellerspeicher verwirklicht. Diese Spezifikation ist in Anhang 1, Abschnitt A1.2.1 dieser Arbeit wiedergegeben. Daraus entwickelt Meyer eine Reihe konkreterer Implementierungen, wovon STACK2 [Meyer; 1988, S. 118-119] der in Abschnitt A1.2.5 abgeleiteten Spezifikation ([1a] bis [6a]) entspricht. Auffällig bei der in [Meyer; 1988] enthaltenen Entwicklung der Spezifikationen und Implementierungen ist das Fehlen wesentlicher Teile der Vor- und Nachbedingungen, z.B. der Teile der Nachbedingungen, die sicherstellen, daß die unteren Elemente des Kellerspeichers nicht verändert werden. Diese Teile sind erforderlich für den Beweis, daß die ursprüngliche abstrakte Spezifikation tatsächlich erfüllt wird, vgl. Abschnitte A1.2.1 bis A1.2.5.

Die (in einer älteren Version von Eiffel geschriebene) Klasse STACK2 ist nach der Korrektur offensichtlicher Schreibfehler, der Umnennung der Variablen `max_size` und `nb_elements` in `nmax` bzw. `n` sowie der Umnennung formaler Parameter und Argumente wie folgt. Dabei tritt die Frage auf, ob das Exportieren der Variable `n` wirklich sinnvoll ist und im Einklang mit der Philosophie der objektorientierten Programmierung und der Informationsverbergung ("information hiding") steht.

```
class STACK2 [X] export
  push, pop, top, n, empty, full
feature
  implementation: ARRAY [X];
  nmax: INTEGER;
  n: INTEGER;

  Create(crsize: INTEGER);
    -- Allocate stack for a maximum of crsize elements
    -- (or for no elements if crsize<0)
  do
    if crsize>0 then nmax:=crsize end;
    implementation.Create(1, nmax)
  end; -- Create

  empty: BOOLEAN is
    -- Is stack empty?
  do
    Result:=(n=0)
  end; -- empty

  full: BOOLEAN is
    -- Is stack full?
  do
    Result:=(n=nmax)
  end; -- full
```

```

pop is
  -- Remove top element
  require
    not empty -- i.e. n>0
  do
    n:=n-1
  ensure
    not full;
    n= old n - 1
  end; -- pop
top: X is
  -- Top element
  require
    not empty -- i.e. n>0
  do
    Result:=implementation.entry(n)
  end; -- top
push(ax: X) is
  -- Add ax on top
  require
    not full
    -- i.e. n<nmax in this representation
  do
    n:=n+1;
    implementation.enter(n, ax)
  ensure
    not empty;
    top=ax;
    n= old n + 1
  end; -- push
end -- class STACK2

```

Die in Abschnitt A1.2.5 des Anhangs 1 angegebene eingeschränkte Spezifikation kann auf einfache Weise in eine für eine Implementierung als Eiffel-Klasse geeignete Form umgewandelt werden. Dabei wird die Spezifikation des Unterprogramms UPnew um die in Abschnitt A1.2.5 erwähnte Initialisierung (Erzeugung der Objektvariablen) ergänzt. Es wird festgelegt, daß das Unterprogramm UPempty nur die Ergebnisvariable Aempty verändert, daß UPnew nur die Variablen n, nmax und die Feldvariablen x(i), i=1, ... nmax, verändert, daß UPPush nur die Variable n und Feldvariablen x(.) verändert, daß UPPop nur die Variable n und ggf. Feldvariablen x(.) verändert, daß UPTop nur die Ergebnisvariable Ax verändert und daß UPfull nur die Ergebnisvariable Afull verändert. In der vorgegebenen generischen Eiffel-Klasse ARRAY [Meyer; 1988, S. 452-453] werden die Feldelemente mit dem Namen "entry" angesprochen; entsprechend wird der Feldname x in entry umgewandelt. Dadurch wird die endgültige Spezifikation wie folgt (KI=Klasseninvariante, siehe unten):

```
{KI} empty {KI und Result=(n=0)} strikt [KA1]
```

```

{crsize∈INTEGER} Create {KI und n=0} strikt [crsize ist Argument, KA2]

{KI und ax∈X und n<nmax} [(KI und n<nmax) = (KI und not full),
ax ist Argument]

push {KI und n= old n + 1
undi=1n-1 implementation.entry(i)=old implementation.entry(i)
und implementation.entry(n)=ax} strikt [bzw. top=ax, KA3]

{KI und 0<n} [(KI und 0<n) = (KI und not empty)]

pop {KI und n= old n - 1
undi=1n implementation.entry(i)
=old implementation.entry(i)} strikt [KA4]

{KI und 0<n} [(KI und 0<n) = (KI und not empty)]

top {KI und Result=implementation.entry(n)} strikt [KA5]

{KI} full {KI und Result=(n≥nmax)} strikt
[KI ⇒ ((n≥nmax)=(n=nmax)), KA6]

```

wobei die Klasseninvariante KI wie folgt definiert wird:

```

KI ≜ nmax∈INTEGER und {0, ... nmax} ⊆ Menge. "n"
und n∈INTEGER und 0≤n≤nmax
und implementation.lower=1 und implementation.upper=nmax
undi=1nmax X ⊆ Menge. "implementation.entry(i)"
undi=1n implementation.entry(i)∈X

```

Streng genommen müßten alle Bezüge oben auf die Variablen n und nmax durch die Bezugsvariable Current qualifiziert werden, d.h. jeder Bezug auf n oben ist als Current.n bzw. n(Current) zu verstehen. Jeder Bezug auf ein Feldelement entry(i) ist als Current.implementation.entry(i) bzw. entry(i, implementation(Current)) zu verstehen. Weil alle Bezüge oben auf n, nmax und entry(.) so zu verstehen sind, kann die zusätzliche Schreibarbeit gespart werden. Kämen andere Bezüge auf diese Variablen oder Feldelemente vor, z.B. stacka.n, stackb.implementation.entry(.), stackc.impa.entry(.) usw., dann müßten ggf. alle Bezüge vollständig qualifiziert werden, um die potentielle Falle bei einer Zuweisung zu solchen Variablen (die effektiv Feldvariablen sind) zu vermeiden, vgl. Abschnitt 3.3.4.2.

Die Klasseninvariante KI ist die Vorbedingung oder ein Teil der Vorbedingung jeder Klassenroutine außer Create. Die aufrufende Stelle ist nicht dafür verantwortlich (und ist in der Regel nicht in der Lage), die vorherige Wahrheit der Klasseninvariante KI sicherzustellen. Ihre Wahrheit wird durch die implizite Programminvariante

```
(A z : z∈Aktivzeig : (Klasse.z=STACK2) ⇒ KI.z)
[. bedeutet hier funktionale Anwendung]
```

gewährleistet, die durch die Kombination der Klassenroutinen (insbesondere ihrer Vor- und Nachbedingungen) und absichtlich in Eiffel eingebauter Einschränkungen aufrechterhalten wird. Diese Einschränkungen sind insbesondere, daß (1) ein Wert für irgendeine Bezugsvariable, der auf eine Ausprägung (engl. Eiffel-Begriff: "instance") einer Klasse C zeigt, nur durch die Ausführung der Create-Routine (in späteren Versionen von Eiffel durch die Ausführung irgendeiner Create-Routine) der Klasse C entstehen kann und daß (2) Variablen der Klasse C nur durch Routinen der Klasse C (unmittelbar) verändert werden können. Falls jede Create-Routine die Wahrheit der eigenen Klasseninvariante sicherstellt, jede Routine die Wahrheit der eigenen Klasseninvariante aufrechterhält und jede in der Klasseninvariante vorkommende Variable ausschließlich auf Veranlassung eigener Routinen verändert werden kann (eine wesentliche Einschränkung, vgl. unten), dann wird die Programminvariante oben an jeder Stelle außerhalb von Routinen der fraglichen Klasse gelten. Dabei ist die Wahrheit der Klasseninvariante bezogen auf jede Ausprägung der Klasse gemeint, nicht nur die Wahrheit der Klasseninvariante bezogen auf die aktuelle ("Current") Ausprägung. Ferner müssen mögliche Umwege betrachtet werden, z.B. daß eine Routine einer dritten Klasse über eine Bezugsvariable, deren Wert auf eine zu STACK2 gehörende Ausprägung der Klasse ARRAY zeigt, die Routine enter der Klasse ARRAY veranlaßt, den Wert eines Feldelements zu verändern. Dieses wäre möglich, wenn die Variable implementation exportiert wäre.

Für jede Klasse wird (unter den bereits angedeuteten Voraussetzungen) eine Programminvariante wie oben gelten. Entsprechend kann die Programminvariante oben allgemeiner formuliert werden:

$$(A \ z : z \in \text{Aktivzeig} : (\text{Klasseninvariante}(\text{Klasse}.z)).z)$$

Bei der Korrektheitsbeweismführung für die Routine Create müssen die besonderen Eigenschaften von Create-Routinen beachtet werden. Nach der Definition der Programmiersprache Eiffel wird am Anfang jeder Create-Routine die systemeigene Initialisierung (siehe Abschnitt 7.3.1.1 oben) automatisch durchgeführt. Danach werden die in der Create-Routine ausdrücklich angegebenen Anweisungen ausgeführt. Die Routine Create der Klasse STACK2 ist deshalb äquivalent zu:

```
call automatische Initialisierung der Klasse STACK2;
if crsize > 0 then nmax := crsize end;
implementation.Create(1, nmax)
```

Die automatische Initialisierung erfüllt die folgende Spezifikation (vgl. Abschnitt 7.3.1.1), wobei obj der Name der Bezugsvariable ist, worauf die Routine Create der Klasse STACK2 angewendet wurde (d.h., der Aufruf war obj.Create):

```
{wahr} [ {Aktivzeig = Aktivzeig' } ]
call automatische Initialisierung der Klasse STACK2
{CurrentSyszeig-Aktivzeig' ^ ((Aktivzeig' U {Current}) ⊆ Aktivzeig)
 ^ Menge."Current.implementation" = Syszeig ^ Current.implementation = void
 ^ Menge."Current.nmax" = INTEGER ^ Current.nmax = 0
 ^ Menge."Current.n" = INTEGER ^ Current.n = 0
 ^ obj = Current } strikt
```

Entsprechend erfüllt die Routine enter der Klasse ARRAY die folgende Spezifikation, wenn sie innerhalb der Klasse STACK2 aufgerufen wird:

```
{wahr} [Aktivzeig = Aktivzeig' ]
implementation.Create(1, nmax) [ARRAY-Objekt erzeugen]
{implementation ∈ Syszeig-Aktivzeig'
 ^ ((Aktivzeig' U {implementation}) ⊆ Aktivzeig)
 ^ Menge."implementation.lower" = INTEGER ^ implementation.lower = 1
 ^ Menge."implementation.size" = INTEGER ^ implementation.size = nmax
 ^ Menge."implementation.upper" = INTEGER ^ implementation.upper = nmax
 und_{i=1}^{nmax} X = Menge."implementation.entry(i)" } strikt
```

Die Routine enter der Klasse ARRAY weist effektiv einer Feldvariable einen Wert zu. Für die Korrektheitsbeweismführung ist die für einen Aufruf innerhalb der Klasse STACK2 gültige Spezifikation

```
{i ∈ INTEGER und implementation.lower ≤ i ≤ implementation.upper und wert ∈ X }
[i und wert sind beliebige Variablen]
implementation.enter(i, wert)
{implementation.entry(i) = wert } strikt
```

in Kombination mit der Forderung, daß enter keine andere Variable der aufrufenden Umgebung verändert, wesentlich.

Nach diesen Vorbereitungen können die Korrektheitsaussagen [KA1] bis [KA6] (siehe Seite 146) unmittelbar bewiesen werden. Mit Ausnahme der Routine Create sind die Korrektheitsbeweismführungen einfach bis trivial.

Beweis der Korrektheitsaussage KA1 (Routine empty): Die Korrektheitsaussage KA1 gilt gemäß den Beweisregeln Z2 und ZS, falls die folgenden Aussagen gelten:

$$KI \Rightarrow (KI \text{ und } \text{Result} = (n=0)) \text{Result} \text{ (n=0)} \quad [1.1]$$

$$KI \Rightarrow (n=0) \in \text{Menge}.\text{"Result"} \quad [1.2]$$

Weil der Variablenname Result in KI nicht vorkommt, reduziert sich [1.1] auf eine Tautologie. Auch der Beweis von [1.2] ist einfach:

$$\begin{aligned} & (n=0) \in \text{Menge}.\text{"Result"} \\ = & \text{[Programmcode für empty, Definition von Eiffel]} \\ & (n=0) \in \text{BOOLEAN} \\ \Leftarrow & \\ & n \in \text{INTEGER} \\ \Leftarrow & \\ & KI \blacksquare \end{aligned}$$

Beweis der Korrektheitsaussage KA2 (Routine Create): Betrachte das folgende Beweisschema für die Korrektheitsaussage KA2:


```

{crsize ∈ INTEGER} [B21]
    call automatische Initialisierung der Klasse STACK2;
{Menge.“nmax” = INTEGER und nmax = 0
und Menge.“n” = INTEGER und n = 0
und crsize ∈ INTEGER} [B22]
    if crsize > 0
    then
    nmax := crsize
    end;
{nmax ∈ INTEGER und {0, ... nmax} ⊆ Menge.“n”
und n ∈ INTEGER und 0 = n ≤ nmax und nmax = max(0, crsize)} [B23]
    implementation.Create(1, nmax)
{nmax ∈ INTEGER und {0, ... nmax} ⊆ Menge.“n”
und n ∈ INTEGER und 0 = n ≤ nmax und nmax = max(0, crsize)
und implementation.lower = 1 und implementation.upper = nmax
und  $\bigcup_{i=1}^{nmax} X \subseteq \text{Menge.“implementation.entry(i)”}$ } [B24]
= [Definition von KI]
{KI und n = 0 und nmax = max(0, crsize)}
⇒ [gilt offensichtlich]
{KI und n = 0}

```

Die Korrektheitsaussage 2 wird gemäß den relevanten Beweisregeln gelten, falls die folgenden Aussagen gelten:

```

{B21} call automatische Initialisierung der Klasse STACK2 {B22} strikt [2.1]
B22 und crsize > 0 ⇒ B23nmaxcrsize [2.2]
B22 und crsize ≤ 0 ⇒ B23 [2.3]
{B23} implementation.Create(1, nmax) {B24} strikt [2.4]
B22 ⇒ (crsize > 0) ∈ BOOLEAN [2.5]
B22 und crsize > 0 ⇒ crsize ∈ Menge.“nmax” [2.6]

```

Aussage [2.1] gilt gemäß den Beweisregeln U3 und B1. Dabei ist B21 die Bedingung B der Beweisregel U3 (die automatische Initialisierung verändert die Variable crsize nicht).

Aussagen [2.2], [2.3], [2.5] und [2.6] lassen sich leicht algebraisch verifizieren. Dabei wird von der Eigenschaft der Menge INTEGER Gebrauch gemacht, daß sie aus darauf folgenden Ganzzahlen besteht, insbesondere daß aus a, b ∈ INTEGER folgt, daß alle zwischen a und b liegenden Ganzzahlen auch aus INTEGER sind.

Aussage [2.4] gilt gemäß den Beweisregeln U3 und B1. Dabei ist B23 die Bedingung B der Beweisregel U3 (die darin angesprochenen Variablen werden von implementation.Create nicht verändert). ■

Beweise der Korrektheitsaussagen KA3 bis KA6: Die anderen Korrektheitsaussagen KA3 bis KA6 können auf die gleiche Weise bewiesen werden. ■

Die Korrektheitsbeweissführung oben zeigt die Korrektheit der Routinen selbst, jedoch ohne den Aufrufmechanismus zu betrachten. Ein Aufruf auf die Routine push soll als Beispiel eines entsprechend erweiterten Korrektheitsbeweises dienen. Darin werden der Klarheit wegen Bezüge auf Current ausdrücklich angegeben. Aus der Sicht einer aufrufenden Klasse hat push die Spezifikation

```

{werte X und not stk.full}
stk.push(wert)
{not stk.empty und stk.top = wert und stk.n = old stk.n + 1}

```

wo stk eine Bezugsvariable ist, deren Wert auf das fragliche Objekt der Klasse STACK2 zeigt. Es sollte jedoch bemerkt werden, daß aus der klassenexternen Sicht diese Spezifikation weniger aussagefähig ist als die in den Abschnitten A1.2.4, A1.2.3, A1.2.2 und A1.2.1 des Anhangs 1 enthaltenen übergeordneten Spezifikationen.

Die Klasseninvariante KI kann zur Vor- und zur Nachbedingung hinzugefügt werden, da sie wie bereits erläutert eine Programmvariante ist. Nach der Definition der Programmiersprache Eiffel hat der Aufruf stk.push(wert) die gleiche Wirkung wie

```

declare (Current, Syszeig, stk); declare (ax, X, wert);
Programmcode für push;
release ax; release Current

```

Unter Berücksichtigung des Aufrufmechanismus ist die zu beweisende Korrektheitsaussage zusammen mit den für den Beweis geeigneten Zwischenbedingungen wie folgt:

```

{KI.stk und werte X und not stk.full und stk.n = old stk.n}
    declare (Current, Syszeig, stk); declare (ax, X, wert);
{KI.Current und ax ∈ X und not Current.full
und Current.n = old Current.n und ax = wert und Current = stk}
= [Spezifikation von full]
{KI.Current und ax ∈ X und Current.n < Current.nmax
und Current.n = old Current.n und ax = wert und Current = stk}
    Programmcode für push; [push verändert weder ax, wert, Current noch stk]
{KI.Current und Current.n = old Current.n + 1
und Current.implementation.entry(n) = ax
und ax = wert und Current = stk}
    [Spezifikation von top]
{KI.Current und Current.n = old Current.n + 1 und Current.top = ax
und ax = wert und Current = stk}
{KI.stk und stk.n = old stk.n + 1 und stk.top = wert
und ax = wert und Current = stk}
    release ax; release Current
{KI.stk und stk.n = old stk.n + 1 und stk.top = wert}

```

= $[0 \leq \text{old stk.n}$ (vgl. Vorbedingung), Spezifikation von empty]
 $\{ \text{KI.stk und stk.n} = \text{old stk.n} + 1 \text{ und stk.top} = \text{wert und not stk.empty} \}$

Diese Korrektheitsaussage wird auf ähnliche Weise wie oben bewiesen. Dabei ist zu berücksichtigen, daß die in der Korrektheitsaussage KA3 über die Routine push vorkommenden Bezüge auf Variablen der Klasse STACK2 bezüglich der Systemvariable Current zu verstehen sind. Im Beweis, daß die Vorbedingung strikt ist, kommt die Bedingung `stkeSyszeig` als Voraussetzung vor, die aus der Annahme folgt, daß `stk` eine (auf ein Objekt der Klasse STACK2 zeigende) Bezugsvariable ist (siehe oben).

7.3.4 Verbesserungs- und Erweiterungsvorschläge zu den korrektheitsbezogenen Konstrukten in Eiffel

Vorab sollte bemerkt werden, daß die in der Programmiersprache Eiffel, [Meyer; 1988] und [Meyer; 1992] eingeschlagene Einstellung hinsichtlich der Korrektheitsbeweissführung und darauf basierender Konstruktionsvorgehensweisen pragmatisch und undogmatisch ist. Auf der einen Seite wird diese Einstellung, realistisch gesehen, die Akzeptanz der in der Praxis stehenden Softwareentwickler eher gewinnen und sie zu einer breiten Anwendung von Korrektheitsbeweistechniken bewegen als eine extremere, dogmatische Haltung. Auf der anderen Seite ermöglichen die in Eiffel implementierten Sprachelemente und Mechanismen die wirkliche Programmkorrektheitsbeweissführung nicht. Die in Eiffel verkörperte Einstellung kann vielleicht als eine gute, sogar möglicherweise erforderliche Übergangslösung angesehen werden, die für Fortschritt in eine wünschenswerte Richtung sorgt, aber sie ist als langfristiges Ziel nicht ausreichend.

Für die in Eiffel spezifizierten Mechanismen für die Auswertung von Vor- und Nachbedingungen, Invarianten usw. (Zusicherungen, engl. "assertions") sowie für die Ausnahmebehandlung sind zwar Korrektheitsbetrachtungen Beweggründe und Korrektheitsbeweissführungstechniken eine wesentliche Grundlage, aber diese Mechanismen sind Hilfsmittel nur für das Testen (bzw. "debugging") eines Programms, nicht für die Korrektheitsbeweissführung. Für die Korrektheitsbeweissführung ist ein Zusicherungsauswertungsmechanismus zur Programmausführungszeit irrelevant und nur ein Placebo; wie bereits erwähnt beweist er die Korrektheit der jeweiligen Ausführung des Programms, aber nicht des Programms selbst. Der Mechanismus für die Ausnahmebehandlung führt effektiv if-Strukturen mit Bedingungen, die implizite Teilausdrücke umfassen und die umfangreich sein können, in das Programm ein. Dadurch wird die Korrektheitsbeweissführung eher erschwert als erleichtert.

Der wichtigste Mangel Eiffels hinsichtlich der Korrektheitsbeweissführung ist die zu starke Einschränkung hinsichtlich der zulässigen Ausdrücke. Unbedingt erforderlich ist die *Erweiterung der Zusicherungssprache auf Reihen*, zumindest **oder-** und **und-**Reihen (die Quantifizierungen \exists und \forall), d.h. auf die Prädikatenlogik erster Ordnung. Sehr nützlich wäre die Erweiterung auf Reihen verschiedener (möglichst beliebiger) Arten, z.B. über die Addition (Σ), Multiplikation (Π), Konkatenation (&) usw., d.h. die Erweiterung auf die Quantifizierung über jede sinnvolle Funktion. Dabei wäre eine Einschränkung der Menge, über die quantifiziert wird, auf eine abzählbare oder endliche Menge annehmbar. Auch sehr hilfreich wäre die Möglichkeit, neue Funktionen (d.h. Hilfsfunktionen, die nicht Programmroutinen sind, z.B. **Perm** in Abschnitt 3.3.3 und **gcd** unten) zu definieren und Axi-

ome dafür anzugeben, die die algebraische Umformung von Ausdrücken ermöglichen, in denen die neu definierten Funktionen vorkommen.

Dieser Mangel ist offensichtlich von vielen bereits und früh beanstandet worden. Die Gegenargumentation von [Meyer; 1988, S. 155-156] ist aus der Sicht der Programmiersprachenkonstruktion durchaus verständlich, aber das ändert nichts daran, daß eine Erweiterung der Zusicherungssprache für die ernsthafte Korrektheitsbeweissführung unerlässlich ist, siehe die Bemerkungen oben sowie [Fischbach]. Auch aus einigen Beispielen in [Meyer; 1988] ist ersichtlich, daß die Zusicherungssprache Eiffels nicht ausreicht. Z.B. auf Seite 142 erscheint ein wesentlicher Teil der Schleifeninvariante für die Ermittlung des größten gemeinsamen Teilers zweier Zahlen als Kommentar:

invariant

$x > 0; y > 0;$

-- (x, y) have the same greatest common divisor as (a, b)

Für die formale Korrektheitsbeweissführung muß diese Invariante vollständig mathematisch ausgedrückt werden, z.B. wie folgt:

$x > 0 \text{ und } y > 0 \text{ und } \text{gcd}(x, y) = \text{gcd}(a, b)$

Ferner müssen für die Korrektheitsbeweissführung geeignete Eigenschaften (z.B. Axiome) der Funktion **gcd** bestimmt werden. In diesem Beispiel wären diese (1) $\text{gcd}(x, y) = \text{gcd}(y, x)$ und (2) $x > y \Rightarrow (\text{gcd}(x, y) = \text{gcd}(x-y, y))$ für alle positiven Ganzzahlen x und y .

Bei der mathematischen Beweissführung geht es nicht um Auswertung der verschiedenen Ausdrücke, sondern um ihre algebraische Umformung. Für die maschinelle Unterstützung der Korrektheitsbeweissführung in Eiffel wäre also *die algebraische Umformung von Zusicherungen* als Komponente eines Satzbeweislers eine Voraussetzung, vgl. Abschnitt 6.2.2. Entsprechende Verfahren könnten entweder in Eiffel selbst oder in einem getrennten aber verwandten Nebensystem eingebaut werden.

Im Zweifel sollten *Bezüge auf Funktionsroutinen in Zusicherungen nicht zulässig* sein, denn sie sind selbst Gegenstand der Korrektheitsprüfung bzw. -beweissführung, und nicht Hilfsmittel dazu. Diese in Eiffel gegebene Möglichkeit wird von [Meyer; 1988, S. 156-157] selbst in Frage gestellt. Entgegen dieser Einschränkung steht allerdings die in Eiffel klare Philosophie, daß es aus klassenexterner Sicht keinen Unterschied zwischen einer Variable und einer Funktionsroutine geben soll, und daß sie sich (wiederum aus klassenexterner Sicht) gleichartig verhalten sollen. (Diese Gleichstellung in Eiffel hat eine Parallele in den in Abschnitt 3.1.2 erläuterten Eigenschaften von Variablen und Ausdrücken: sowohl ein Variablenname als auch ein Ausdruck können als Funktionen auf **ID** aufgefaßt werden.) Eventuell könnten diese entgegengesetzten Anliegen dadurch in Einklang miteinander gebracht werden, daß die *klasseninternen Zusicherungen keine Bezüge auf eigene Funktionsroutinen* enthalten dürfen, daß aber *Zusicherungen in Klientenklassen doch Bezüge auf Funktionsroutinen von (anderen) Lieferantenklassen* enthalten dürfen. Dabei sollten Bezüge auf Funktionsroutinen mit Nebenwirkungen ggf. nicht zulässig sein. Die für die Korrektheitsbeweissführung benötigten mathematischen Eigenschaften der fraglichen Funktionsroutine müssen aus ihrer Spezifikation (Vor- und Nachbedingungen) hergeleitet werden. Dieser Vorschlag wäre leichter zu verwirklichen, wenn die (in Eiffel gegenwärtig einzige) Klasseninvariante in zwei Klasseninvarianten — eine interne und eine externe — aufgeteilt werden würde, siehe unten.

Die in den Abschnitten 3.5.3 und 4.4, Kapitel 5 sowie Abschnitt A1.2 des Anhangs 1 enthaltenen Betrachtungen führen zur Empfehlung, für jede exportierte Routine eine klassenexterne und eine klasseninterne Spezifikation (Vor- und Nachbedingungen) vorzusehen. Auch [Schmitz; S. 67] unterbreitet diesen Vorschlag. Wesentlicher Unterschied dazwischen wäre, daß sich die klassenexterne Spezifikation nur auf exportierte Merkmale ("features") der Klasse beziehen darf, während sich die klasseninterne Spezifikation natürlich auf alle Merkmale der eigenen Klasse beziehen darf. Da die Klasseninvariante zur Vorbedingung jeder eigenen Routine außer Create-Routinen und zur Nachbedingung jeder eigenen Routine gehört, müßte auch eine klassenexterne und eine klasseninterne Klasseninvariante vorgesehen werden. Diese Empfehlung stellt eine konsequente Vervollständigung des Konzepts der Darstellungsinvariante ("representation invariant") von [Meyer; 1988, S. 131-132] dar.

Eiffel sieht Zusicherungen auf den Ebenen der einzelnen Stellen im Programm, der einzelnen Routinen und der einzelnen Klassen vor. Übergeordnete Zusicherungen, wie z.B. Dateninvarianten (Beziehungen zwischen Daten und Datenstrukturen in Objekten verschiedener Klassen) und Programminvarianten (Bedingungen, die an mehreren Stellen des Programms gelten sollen), sind in der Syntax von Eiffel nicht vorgesehen. Diese Tatsache schließt die Aufstellung solcher Zusicherungen zwar nicht aus, aber der Mangel an einer Eiffel spezifischen Struktur dafür kann dazu führen, daß die Aufmerksamkeit des Softwareentwicklers auf die vorgesehenen Invariantenarten zu Ungunsten übergeordneter Zusicherungen abgelenkt wird. Außerdem kann die Einführung solcher übergeordneten Zusicherungen zu Umständlichkeiten führen, weil sie nicht gut in die Philosophie und Struktur von Eiffel (und anderen objektorientierten Programmiersprachen auch) passen. Klassenübergreifende Dateninvarianten z.B. können durch Wiederholung in allen betroffenen Klassen (als Teil der jeweiligen Klasseninvariante) eingeführt werden. Man würde jedoch dann beim Betrachten einer Klasse nicht unbedingt, auf jeden Fall nicht leicht, die Querverbindungen mit anderen Klassen sehen, die die fragliche Dateninvariante zum Ausdruck bringen soll. Dadurch würde die nachträgliche Änderung einzelner Klassen erschwert. Aus diesen Gründen wird die Einführung geeigneter Strukturen für klassenübergreifende Invarianten vorgeschlagen.

In Abschnitt 7.3.2 (siehe dort **variant**) wurde angeführt, daß die in Eiffel auf ganzzahlige Werte beschränkte Schleifenvariante zu restriktiv ist. Es wird deshalb vorgeschlagen, Ausdrücke mit reellen Werten als Schleifenvarianten zuzulassen. Noch konsequenter wäre eine Erweiterung auf Ausdrücke mit Werten aus einer linear geordneten aber sonst beliebigen Menge, obwohl der praktische Grenzwert dieser Erweiterung vergleichsweise gering wäre.

In Abschnitt 7.3.2 (siehe dort **strip** und **old**) wurde erwähnt, daß es oft einfacher ist, eine vollständige Liste aller Variablen anzugeben, die durch die Ausführung einer Routine verändert werden können, als eine vollständige Liste aller Variablen anzugeben, die durch die Ausführung einer Routine nicht verändert werden. Es wird deshalb vorgeschlagen, einen Eiffel-Begriff wie "maychange" oder "changes" einzuführen, der auf der syntaktischen Ebene von **require** und **ensure** eingefügt werden sollte. Alle Merkmale ("features") der eigenen Klasse, die durch Ausführung der fraglichen Routine verändert werden können, müssen unter **maychange** bzw. **changes** aufgeführt werden. Auch Merkmale anderer Klassen, die auf Veranlassung der fraglichen Routine verändert werden können, sind aufzuführen. Insbesondere sind Merkmale anderer Klassen, die in der Nachbedingung (einschließlich Klasseninvariante(n)) der fraglichen Routine vorkommen, in diesem Zusammen-

hang wichtig, vgl. `implementation.entry(i)` in der Invariante KI der Klasse `STACK2` im Beispiel in Abschnitt 7.3.3.

Aus der Sicht der Programmkorrektheitsbeweissführung ist die Wahl des Worts "check" als Eiffel-Begriff für eine Zusicherung ungünstig. Es drückt unterschwellig Unsicherheit seitens des Programmentwicklers aus und deutet auf eine Suche nach Bestätigung hin. Es wird deshalb vorgeschlagen, "check" in "assert" umzuändern, ein stärkeres Wort, das Behauptung und Überzeugung ausdrückt. Formal ist dieser Unterschied natürlich unwichtig, aber er könnte sich psychologisch auf die Einstellung des Softwareentwicklers auswirken.

Die Definition der Programmiersprache Eiffel ist vergleichsweise präzise, aber Definitionen der verschiedenen Sprachelemente, Anweisungen usw. in mathematischer Form (z.B. durch Angabe von Vor- und Nachbedingungen für Systemprozeduren wie die automatischen Create-Routinen u.ä.) würden die Korrektheitsbeweissführung nicht nur erleichtern, sondern sie von der Interpretation natursprachlicher Texte und Intuition unabhängig machen. Sie würde eine mathematisch präzise Schnittstellenspezifikation zwischen der auf das Anwendungsprogramm bezogenen Korrektheitsbeweissführung einerseits und der Konstruktion des Eiffelübersetzungs- bzw. -ausführungssystems andererseits darstellen. Dadurch könnten einige mögliche Fehlerquellen eliminiert werden. Eine solche Idee ist nicht neu; einen Ansatz dazu liefert z.B. [Hoare; 1973].

Nach Möglichkeit sollte der Entwickler eines in Eiffel geschriebenen Programms, das einer Korrektheitsbeweissführung unterzogen werden sollte, auf die Verwendung der Ausnahmebehandlungsmechanismen verzichten. Die Gründe für diesen Vorschlag wurden bereits in Abschnitt 7.3.2 (siehe die letzten zwei Absätze über **rescue** und **retry**) und im zweiten Absatz des Abschnitts 7.3.4 (siehe Seite 152) erläutert.

[Yau] schlägt die Verlagerung von Programmsegmenten für die Wiederherstellung der Wahrheit der Klasseninvariante aus den Routinen (z.B. aus den **rescue**-Abschnitten) in den **invariant**-Abschnitt des Klassentexts (d.h. zur Klasseninvariante hin) vor. Jedes solche "fix"-Segment wird nach diesem Vorschlag dem Teil der Klasseninvariante, dessen Wahrheit dadurch wiederhergestellt wird, unmittelbar untergliedert und zugeordnet. Dadurch soll die Übersichtlichkeit und Verständlichkeit des Programmtexts erhöht und die Wiederholung gleicher Programmsegmente in mehreren **rescue**-Abschnitten und in verschiedenen Routinen eliminiert werden. Ferner umfaßt dieser Vorschlag **preference**-Abschnitte, die Bedingungen angeben, die möglichst wahr sein sollen, aber nicht müssen. Diesen Sollbedingungen werden Prioritäten zugeordnet.

Die wichtigsten Verbesserungsvorschläge oben können vielleicht mit der Forderung zusammengefaßt werden, *Verifikation* statt Testen zu betonen und hervorzuheben.

8. Schlußfolgerungen

8.1 Erreichtes

Die in dieser Arbeit zusammengestellte mathematische, theoretische Grundlage für die praktische Softwareentwicklung entspricht den Grundlagen der klassischen Ingenieurwissenschaften. Sie unterstützt sowohl das Konstruieren als auch das Verifizieren von Programmen. Durch die gekonnte und konsequente Anwendung dieser Grundlage läßt sich ein für die bisherige Softwareentwicklungspraxis atypisch hoher Grad an Entwurfsfehlerfreiheit erreichen. Diese Grundlage kann auf formale, halbformale oder informale Weise angewendet werden.

Diese Grundlage für die Softwareentwicklung basiert auf relativ wenigen und verhältnismäßig einfachen mathematischen Konzepten und Kenntnissen, hauptsächlich Mengen, Funktionen und logischer (Boolescher) Algebra, und orientiert sich an praktischen Belangen. Alternative Notationsformen für verschiedene Anwenderkreise mit unterschiedlichen Kenntnissen und Orientierungen können eingesetzt (und untereinander ausgetauscht) werden. Die aus praktischer Sicht wesentlichen Aspekte von z.B. Definitionsbereichen der verschiedenen Programmanweisungen, Rechengenauigkeit, nicht terminierenden Prozessen usw. werden ausdrücklich berücksichtigt.

Die Anforderungen der hier vorgeschlagenen Vorgehensweise bei der Softwareentwicklung liefern Leitlinien für die Aufstellung von Schnittstellenspezifikationen und unterstützen damit auch diese Phase des Softwareentwicklungsprozesses.

Die hier vorgestellte Vorgehensweise bildet eine in sich vollständige und abgeschlossene Einheit, die jedoch für Ergänzungen und Erweiterungen offen ist, z.B. hinsichtlich der Form und des Inhalts der zugelassenen mathematischen Ausdrücke.

Die konsequente Anwendung der Beweisregeln führt zu einer straffen Systematisierung und Mechanisierung der Korrektheitsbeweissführung, zu einer relativ leicht verständlichen und nachprüfaren Darstellung des Beweises und zu einer übersichtlichen Dokumentation. Dadurch wird die Beweisführung u.a. unabhängig von Interpretation, Intuition und der damit verbundenen Problematik. Komplexe und umfangreiche Beweisaufgaben werden systematisch und konsequent in einfache und kleine Teilaufgaben zerlegt.

Zusammenfassend ist hier ein praxisgerechter Programmkalkül für eine wirklich ingenieurmäßige Praxis der Softwareentwicklung — sowohl für die Konstruktion als auch für die Verifikation — geschaffen worden.

8.2 Als offen Erkanntes

Die hier zusammengestellte Grundlage ist ausreichend vollständig entwickelt, so daß sie mit gutem Erfolg in der Praxis angewendet werden kann. Einiges deutet darauf hin, daß ihre Anwendung bei der Programm*konstruktion* zu Zeiteinsparung führen und daß der mit der selektiven Anwendung zur Korrektheitsbeweissführung verbundene Aufwand für kritische Software gerechtfertigt werden kann. Diese Behauptungen sind jedoch strittig und können nicht als wissenschaftlich gesichert angesehen werden; ausreichende und geeignete

Untersuchungen sind nicht durchgeführt worden, siehe Kapitel 6. Insbesondere in Bezug auf die Programmverifikation mit Hilfe von VDM wird über einen sehr hohen Aufwand berichtet. Auch hinsichtlich der Notwendigkeit maschineller Unterstützung für die praktikable Korrektheitsbeweissführung liegen entgegengesetzte Meinungsäußerungen vor.

Angewendet auf ein bestimmtes Programm löst die hier beschriebene mathematisch rigorose Vorgehensweise nicht alle Aspekte des Problems der Korrektheit eines Anwendungssystems. Außer den behandelten Fragestellungen (Übereinstimmung eines Programms mit einer mathematisch präzisen Spezifikation und ggf. Übereinstimmung einer mathematisch präzisen Spezifikation mit einer anderen) muß man sich mit zusätzlichen potentiellen Fehlerquellen auseinandersetzen. Die Spezifikation selbst muß validiert werden, d.h. die Eignung der mathematisch präzisen Spezifikation als Lösungsansatz für das nur informal definierte Anwendungsproblem muß bestätigt werden. Die Korrektheit des Übersetzers und des Programmiersprachensystems (z.B. Laufzeitroutinen) ist Gegenstand einer getrennten Untersuchung, bei der die hier vorgestellte Grundlage sowie andere mathematische Methoden angewendet werden können. Ebenfalls sind Fragen der Korrektheit anderer Anwendungs- und Systemsoftware (z.B. des Betriebssystems) zu klären. Auch die Korrektheit der Hardware, die die fragliche Software ausführen soll, muß sichergestellt werden. Dabei sind sowohl die Entwurfsfehlerfreiheit als auch die Betriebsfähigkeit zu berücksichtigen.

8.3 Handlungsbedarf

Der für das Erlernen dieser Vorgehensweise zur Softwarekonstruktion und -verifikation erforderliche Aufwand ist etwa vergleichbar mit dem entsprechenden Lernaufwand für die mathematische Grundlage der klassischen Ingenieurwissenschaften. U.a. wegen dieses Lernaufwands ist es wahrscheinlich unrealistisch zu erwarten, daß sich dieser Stoff unter bereits in der Praxis stehenden Softwareentwicklern weit verbreitet. Lediglich wo ein besonderer Bedarf vorhanden ist, kann mit der entsprechenden Managementunterstützung für die nötige Investition in die Personalqualifizierung gerechnet werden.

Wegen des zunehmenden Stellenwerts von Software in allen technologischen Bereichen wird es früher oder später unumgänglich sein, Entwurfsfehler auch in Software in Griff zu bekommen. Die in dieser Arbeit vorgestellte Grundlage für die Softwareentwicklung ist eine versprechende Möglichkeit dazu. Wie oben angeführt ist jedoch derzeit nicht mit ihrer unmittelbaren und verbreiteten Aufnahme in der Softwareentwicklungspraxis zu rechnen. Vielmehr sind die Hochschulen gefordert, für die Übertragung dieser Kenntnisse in die Softwareentwicklungspraxis zu sorgen, z.B. durch ein konsequentes und ausreichend breites Angebot an ingenieurwissenschaftlich orientierten Lehrveranstaltungen für zukünftige Softwareentwickler. "Ingenieurwissenschaftlich" bedeutet hier, daß sie gleichzeitig mathematisch und theoretisch gut fundiert sowie auf die Belange der Anwendungspraxis gerichtet sein sollen. Die hier vorgestellte Grundlage erfüllt nach Meinung des Verfassers dieses Kriterium und eignet sich gut für eine derartige Präsentation. Die Entwicklungsgeschichten der traditionellen Ingenieurfächern deuten darauf hin, daß die Einführung einer mathematischen Basis für die Softwareentwicklung in die Praxis nicht unmittelbar, sondern über die tertiäre Ausbildung eher zu erwarten ist. Gleichzeitig können die traditionellen Ingenieurwissenschaften als Vorbilder für diese Einführung und für die Rolle der Hochschulen dabei dienen.

Das Problem der Softwarefehler ist in erster Linie ein technisches Problem mit technischen Ursachen. Führungs- und organisatorische Maßnahmen können das Problem weder lösen noch beseitigen, sondern bestenfalls seine Auswirkungen nur etwas lindern oder den Einsatz technischer Lösungen fördern. Der Versuch, das technische Problem nur oder überwiegend mit Management- und organisatorischen Maßnahmen zu lösen, lenkt die Aufmerksamkeit von versprechenderen Lösungsansätzen ab, stellt eine unwirtschaftliche Verwendung wertvoller Ressourcen dar und ist deshalb kontraproduktiv. Solche Maßnahmen sind keine Alternative, kein Ersatz für eine wie hier vorgestellte technische Lösung des Problems der fehlerhaften Software.

Anhang 1. Beispiele unterschiedlicher Spezifikationsstufen

A1.1 Kernreaktorüberwachung

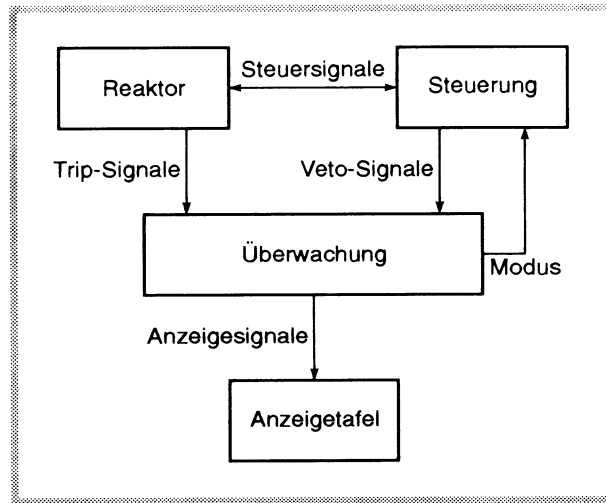
Dieses Beispiel und die VDM-Formulierungen der Spezifikationen entstammen [Bloomfield; 1986 Sept. und 1987] und [Fields]. Es zeigt zwei VDM-Spezifikationen für die fragliche Programmfunktion, eine abstrakte und eine konkrete, die mit Hilfe des Werkzeugs "mural" (siehe auch Abschnitt 2.1.7) aufgestellt und verifiziert wurden. Im Abschnitt A1.1.3 wird eine andere auf einer für das Anwendungsfachgebiet traditionelleren Denkweise basierende Vorgehensweise zur Erarbeitung einer Spezifikation aufgezeigt.

A1.1.1 Allgemeine Beschreibung der Überwachungsaufgabe

Während des Betriebs des Kernreaktors werden mehrere Größen (z.B. Temperatur, Druck, Kühlmittelfluß und Neutronenfluß) an verschiedenen Stellen der Anlage gemessen und an den Überwachungs- und Steuerungsrechner übermittelt. Diese Eingangsdaten werden vom Rechnersystem ausgewertet. Die Ergebnisse der Berechnungen steuern sowohl gewisse Sicherheitsfunktionen als auch eine Anzeigetafel.

Für jede Meßgröße kommt zum Überwachungsuntersystem ein "Trip"-Signal, das angibt, ob die jeweilige Meßgröße im normalen ("OK") oder abnormalen ("TRIP") Bereich liegt, sowie ein "Veto"-Signal, das angibt, ob das entsprechende Trip-Signal außer Acht zu lassen ist, z.B. weil es vom Steuerungsuntersystem bereits angemessen berücksichtigt worden ist. Ausgabevariablen des fraglichen Überwachungsunterprogramms sind der Überwachungsmodus und die Soll-Zustände der Anzeigelampen (ein oder aus).

Das folgende Diagramm nach [Fields; S. 280] veranschaulicht das gesamte System:



Kernreaktorsystemübersicht

Der Überwachungsmodus ist entweder normal oder abnormal ("OK" bzw. "TRIP"). Falls irgendein Trip-Signal aktiv ist, ohne daß das entsprechende Veto-Signal gesetzt ist, soll der Überwachungsmodus auf "TRIP" gesetzt werden. Wenn ein Trip-Signal aktiv ist, soll die entsprechende Anzeigelampe eingeschaltet werden. Alle Ausgabevariablen dieses Unterprogramms sollen einrasten, d.h., nach der Einschaltung im eingeschalteten Zustand bleiben. Ein eventuelles Rücksetzen wird nicht von diesem, sondern ggf. von anderen Unterprogrammen vorgenommen werden.

Das Überwachungsunterprogramm wird vom Hauptprogramm wiederholt nach kurzen Zeitintervallen aufgerufen, um die Werte der Ausgabevariablen zu aktualisieren.

A1.1.2 VDM-Spezifikationen unterschiedlicher Abstraktionsgrade

[Fields] gibt für das zu konstruierende Unterprogramm (genannt "watchdog") die folgende abstrakte Spezifikation an. Sie wird nachstehend in der Originalform wiedergegeben und ist somit nicht konsistent mit der in anderen Teilen dieser Arbeit verwendeten Schreibweise.

```

Signal = {1, ..., max}
Mode = {Trip, Ok}
State :: tr : Signal-set           [Überwachungszustand]
        vt : Signal-set
        md : Mode
        ind : Signal-set
WDstate = State                   [Zustand von "watchdog"]
  
```

```

where
inv-WDstate ≜ (s-ind(s) ⊆ s-tr(s))           [Zustandsinvariante]
              ∧ (s-vt(s) ⊆ s-tr(s))         [Anforderung an ein anderes Untersystem]

watch
ext wr s : WDstate
post ((s-md(s) = Trip) ⇔                    [Nachbedingung]
      ((¬(s-tr(s̄) ⊆ s-vt(s̄))) ∨ s-md(s̄) = Trip)) ∧
      s-ind(s) = s-ind(s̄) ∪ s-tr(s̄) ∧
      s-tr(s) = s-tr(s̄) ∧
      s-vt(s) = s-vt(s̄))
  
```

Für die konkrete Spezifikation schreibt [Fields]:

```

Mode = {Trip, Ok}
State2 :: tr : B*
         vt : B*
         md : Mode
         ind : B*
WDstate2 = State2
where
inv-WDstate2(mk-State2(tr, vt, md, ind)) ≜ [Zustandsinvariante]
len tr = max ∧
len vt = max ∧
len ind = max ∧
ptwImpliedBy(tr, ind) ∧                    ["pointwise implied by"]
ptwImpliedBy(tr, vt)

watch2
ext wr s : WDstate2
post ((s-md(s) = Trip) ⇔                    [Nachbedingung]
      (sigNotVetoed(s) ∨ s-md(s̄) = Trip)) ∧
      indsCorrect(s̄, s) ∧
      restUnchanged(s̄, s))
  
```

Die Hilfsfunktionen `ptwImpliedBy`, `sigNotVetoed`, `indsCorrect` und `restUnchanged` werden wie folgt definiert:

```

ptwImpliedBy : B* × B* → B
ptwImpliedBy(a, b) ≜ ∀ i : N1 · (i ∈ inds a ∧ i ∈ inds b) ⇒ (b[i] ⇒ a[i])

sigNotVetoed : WDstate2 → B
sigNotVetoed(s) ≜ ∃ i : N1 · (i ∈ inds s-ind(s) ∧ s-tr(s)[i] ∧ ¬(s-vt(s)[i]))

indsCorrect : WDstate2 × WDstate2 → B
indsCorrect(s, ss) ≜ ∀ i : N1 · (i ∈ (inds s-ind(s) ∩ inds s-ind(ss))) ⇒
  s-ind(ss)[i] = (s-ind(s)[i] ∨ s-tr(s)[i])

restUnchanged : WDstate2 × WDstate2 → B
restUnchanged(s, ss) ≜ s-tr(s) = s-tr(ss) ∧ s-vt(s) = s-vt(ss)
  
```

[Fields; S. 290] berichtet, daß ein erfahrener Benutzer etwa zwei Wochen brauchen würde, diese Spezifikationen zu verifizieren und zu validieren. ("Formally verifying speci-

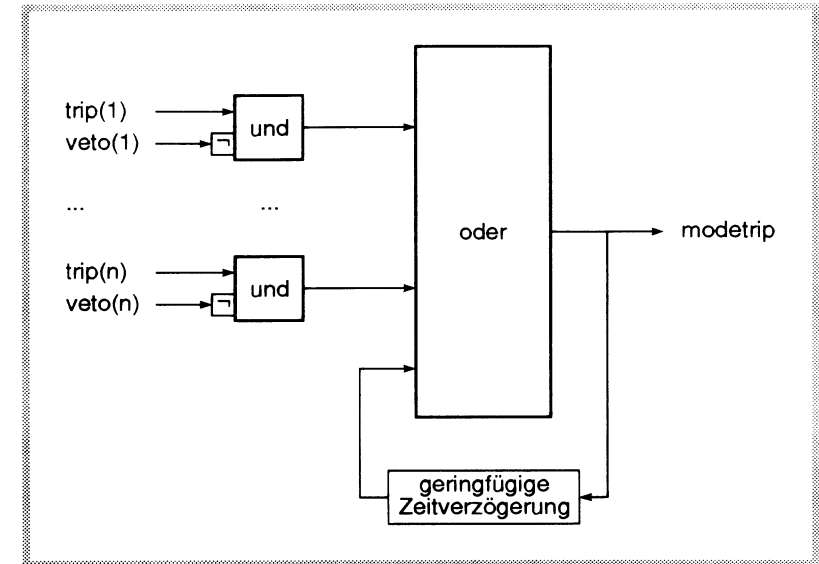
fications and developments is *extremely* tedious and time consuming — the task of verifying and validating an example like the one presented here may take an experienced user two weeks. This is not a problem specific to *mural*, but is more to do with the level of detail at which one is forced to work when doing “fully formal” development. Of course, when the only support system available is a pencil and paper, doing formal proofs on a large scale would be practically impossible rather than merely difficult.”) Im fraglichen Beispiel gibt [Fields; S. 282] acht Validierungsbedingungen an. Angesichts dieser Aussage über den Aufwand und der Einfachheit dieses Beispiels sind die Skepsis und die ablehnende Haltung mancher Softwarepraktiker formalen Methoden gegenüber (siehe Abschnitt 2.2.1) durchaus verständlich.

Die oben verwendete mathematische Sprache gehört nicht zum Erfahrungsbereich typischer in der industriellen Praxis tätiger Techniker und Ingenieure. Mengentheoretische Betrachtungen, Prädikatenlogik, Folgen usw. sind ihnen zwar mehr oder weniger bekannt, aber nicht geläufig. Die mathematischen Modelle, die solche Praktiker regelmäßig verwenden, basieren auf anderen mathematischen Konzepten und Objekten.

Das Ziel der oben geschilderten Vorgehensweise — mit einer möglichst abstrakten, wenig einschränkenden Spezifikation anzufangen — entspricht oft nicht dem Ziel potentieller Vertragspartner oder eines Systemkonstruktors — mit möglichst geringem Aufwand eine ausreichend detaillierte und eindeutige Schnittstellenspezifikation zu vereinbaren bzw. festzulegen, wobei häufig bestimmte Teile des gesamten Systems bereits konstruiert worden sind, wofür Spezifikationen in konkreter Form deshalb vorliegen. In der Praxis ist das Ziel meist nicht eine möglichst abstrakte, sondern eine möglichst *natürliche* Spezifikation; natürlich hinsichtlich der Anwendung und hinsichtlich der Erfahrung und der Ausbildung aller Beteiligten: Anwendungsingenieure, Klienten und Softwareentwickler. Die Spezifikation soll die Kommunikation zwischen diesen Parteien erleichtern und die technische Basis für den Vertrag bzw. Auftrag bilden.

A1.1.3 Anwenderorientierte Vorgehensweise zur Erarbeitung der Spezifikation

Ein auf dem Gebiet der elektronischen Steuerung und Instrumentation erfahrener Ingenieur würde unmittelbar nach dem Lesen der allgemeinen Beschreibung der Überwachungsaufgabe (siehe Abschnitt A1.1.1) wahrscheinlich an das folgende logische Diagramm (bzw. eine genormte Version davon) für ein elektronisches Netzwerk, das den Überwachungsmodus ermitteln soll, denken:



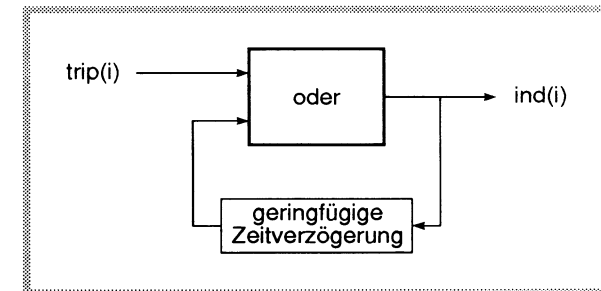
Schaltbild für die Ermittlung des Überwachungsmodus

Ein Boolescher Ausdruck, der dieses Netzwerk beschreibt, lässt sich vom Diagramm unmittelbar ableiten:

$$\text{modetrip} = (\text{modetrip}' \text{ oder } \bigvee_{i=1}^n (\text{trip}(i) \text{ und nicht veto}(i)))$$

wo n die Anzahl der Signalleitungen ist.

Das folgende Diagramm für die Ermittlung des Soll-Zustands jeder Anzeigelampe bietet sich an:

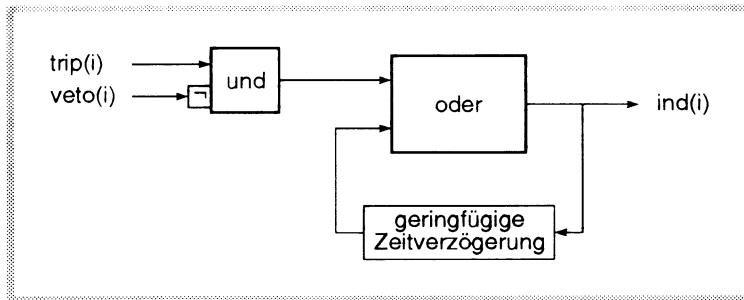


Ermittlung des Soll-Zustands einer Anzeigelampe

Der entsprechende, alle Anzeigelampen berücksichtigende Ausdruck der Booleschen Algebra ist:

$$\bigwedge_{i=1}^n (\text{ind}(i) = (\text{ind}(i)' \text{ oder } \text{trip}(i)))$$

Dabei würde voraussichtlich die Frage auftreten, ob nicht das Veto-Signal auch hier berücksichtigt werden sollte, in welchem Fall das Diagramm wie folgt aussehen würde:



Alternative Ermittlung des Soll-Zustands einer Anzeigelampe

Nach Absprache mit den zuständigen Ingenieuren würde man sich auf die eine oder die andere Alternative einigen. In diesem Fall soll die erste Möglichkeit gelten.

Damit ist die vollständige Nachbedingung für das fragliche Überwachungsunterprogramm

$$\text{modetrip} = (\text{modetrip}' \text{ oder } \prod_{i=1}^n (\text{trip}(i) \text{ und nicht veto}(i)))$$

$$\text{und } \prod_{i=1}^n (\text{ind}(i) = (\text{ind}(i)' \text{ oder trip}(i)))$$

Eine strikte Vorbedingung muß sicherstellen, daß alle Variablen auf geeignete Weise deklariert und initialisiert sind. Eine mögliche strikte Vorbedingung wäre z.B.:

$$n \in \mathbb{Z} \text{ und } 0 \leq n \text{ und Menge. "modetrip" } = \mathbb{B}$$

$$\text{und } \prod_{i=1}^n (\text{Menge. "trip}(i)" } = \mathbb{B} \text{ und}$$

$$\text{Menge. "veto}(i)" } = \mathbb{B} \text{ und}$$

$$\text{Menge. "ind}(i)" } = \mathbb{B})$$

oder allgemeiner

$$n \in \mathbb{Z} \text{ und } 0 \leq n \text{ und } \mathbb{B} \subseteq \text{Menge. "modetrip" und modetrip} \in \mathbb{B}$$

$$\text{und } \prod_{i=1}^n (\mathbb{B} \subseteq \text{Menge. "trip}(i)" und trip}(i) \in \mathbb{B} \text{ und}$$

$$\mathbb{B} \subseteq \text{Menge. "veto}(i)" und veto}(i) \in \mathbb{B} \text{ und}$$

$$\mathbb{B} \subseteq \text{Menge. "ind}(i)" und ind}(i) \in \mathbb{B})$$

Ferner würde man eine obere Schranke für n in die Vorbedingung aufnehmen. Eine solche Schranke würde sich sowohl nach der Anwendung (der beabsichtigten Anzahl der Signalleitungen) als auch nach dem vorgesehenen Rechnersystem (der Programmiersprache und der Hardware) richten.

Die Spezifikation des Überwachungsunterprogramms besteht aus einer der strikten Vorbedingungen oben, der Nachbedingung und der Forderung, daß es nur die Werte der Variablen modetrip und $\text{ind}(i)$ (für $i=1, \dots, n$) verändert.

A1.2 Kellerspeicher

Eine abstrakte Spezifikation muß nicht wie im Abschnitt A1.1.2 oben in der Form einer Vor- und Nachbedingung angegeben werden. Besonders wenn ein System von Funktionen bzw. Unterprogrammen spezifiziert werden soll, liegt es oft nahe, die geforderten Eigenschaften in der Form von Beziehungen zwischen den Funktionen anzugeben. Das Beispiel des Kellerspeichers zeigt eine derartige Vorgehensweise anschaulich. Siehe z.B. [Jones; 1986], [Meyer; 1988] und [Parnas; 1989 Oct.].

A1.2.1 Abstrakte Spezifikationen eines Kellerspeichers (Funktionensystem)

[Meyer; 1988, Abschnitt 4.7.5] schreibt die folgende abstrakte Spezifikation für ein System von Funktionen, das einen Kellerspeicher verwirklicht:

TYPES

STACK [X]

FUNCTIONS

empty: STACK [X] \rightarrow BOOLEAN

new: \rightarrow STACK [X] [Konstante, Element aus STACK [X]]

push: $X \times$ STACK [X] \rightarrow STACK [X]

pop: STACK [X] \leftrightarrow STACK [X]

[partielle Funktion]

top: STACK [X] \leftrightarrow X

[partielle Funktion]

PRECONDITIONS

pre pop(s: STACK [X]) = (not empty(s))

pre top(s: STACK [X]) = (not empty(s))

AXIOMS

For all $x: X, s: \text{STACK [X]}$:

empty(new())

not empty(push(x, s))

pop(push(x, s)) = s

top(push(x, s)) = x

X und STACK [X] sind hier nicht näher festgelegte Mengen. Die Vorbedingungen dienen als Einschränkungen der Definitionsbereiche der Funktionen pop und top; die Funktionswerte pop(new()) und top(new()) sind nicht definiert.

A1.2.2 Abstrakte Spezifikation eines Systems parameterloser Unterprogramme

Eine Spezifikation für ein System von Unterprogrammen, die ohne formale Parameterübergabe aufgerufen werden, kann auf der Basis der funktionenbezogenen Spezifikation (siehe Abschnitt A1.2.1 oben) aufgestellt werden. Voraussetzung dafür ist eine Verbindung zwischen den Datenumgebungen des Unterprogrammzusammenhangs und den Funktions- und Variablenwerten des Funktionenzusammenhangs; eine solche Verbindung drückt die Auslegung (beabsichtigte Bedeutung) des Unterprogrammzusammenhangs aus.

Eine solche Auslegung kann als eine Sammlung von Funktionen auf \mathbf{D} nach den jeweiligen Mengen der funktionenbezogenen Spezifikation (hier BOOLEAN, X und STACK [X]) ausgedrückt werden:

$$\begin{aligned} \text{Aempty: } \mathbf{D} &\rightarrow \text{BOOLEAN} && \text{[Wert der Funktion empty]} \\ \text{As: } \mathbf{D} &\rightarrow \text{STACK [X]} && \text{[STACK-Wert der relevanten Argumente und Funktionen]} \\ \text{Ax: } \mathbf{D} &\rightarrow X && \text{[X-Wert der relevanten Argumente und Funktionen]} \end{aligned}$$

Als Funktionen auf \mathbf{D} können diese Auslegungsfunktionen als Programmausdrücke aufgefaßt werden. Eine typische Implementierung würde eine Variable, eine Gruppe von Variablen oder ein Datenobjekt für jede Auslegungsfunktion vorsehen; jedes Unterprogramm würde den jeweiligen Funktionswert berechnen und ihn der entsprechenden Ergebnisvariable zuordnen.

Sind die Auslegungsfunktionen surjektiv, kann von einer vollständigen Implementierung gesprochen werden.

Soll jede Kellerspeicherfunktion von einem eigenen Unterprogramm (z.B. UPempty, UPnew, UPpush, UPpop und UPTop) berechnet werden und sollen gemeinsame Argument- und Funktionswertvariablen verwendet werden, können unter Berücksichtigung der Vorbedingungen (siehe "pre" in Abschnitt A1.2.1) die Forderungen, daß die Funktionswerte richtig berechnet werden, wie folgt geschrieben werden:

$$(\mathbf{A} \ d : d \in \mathbf{D} \wedge \text{As.deSTACK [X]} : \text{Aempty.((UPempty; UPnew).d)} = \text{empty.(As.d)}) \quad [1]$$

$$(\mathbf{A} \ d : d \in \mathbf{D} : \text{As.(UPnew.d)} = \text{new}) \quad [2]$$

$$(\mathbf{A} \ d : d \in \mathbf{D} \wedge \text{Ax.deX} \wedge \text{As.deSTACK [X]} : \text{As.(UPpush.d)} = \text{push.(Ax.d, As.d)}) \quad [3]$$

$$(\mathbf{A} \ d : d \in \mathbf{D} \wedge \text{As.deSTACK [X]} \wedge \neg \text{empty.(As.d)} : \text{As.(UPpop.d)} = \text{pop.(As.d)}) \quad [4]$$

$$(\mathbf{A} \ d : d \in \mathbf{D} \wedge \text{As.deSTACK [X]} \wedge \neg \text{empty.(As.d)} : \text{Ax.(UPTop.d)} = \text{top.(As.d)}) \quad [5]$$

Unter diesen Voraussetzungen können die Axiome der funktionenbezogenen Spezifikation (siehe Abschnitt A1.2.1) in die folgenden Bedingungen umformuliert werden:

$$(\mathbf{A} \ d : d \in \mathbf{D} : \text{Aempty.((UPnew; UPempty).d)}) \quad [6]$$

$$(\mathbf{A} \ d : d \in \mathbf{D} \wedge \text{Ax.deX} \wedge \text{As.deSTACK [X]} : \neg \text{Aempty.((UPpush; UPempty).d)}) \quad [7]$$

$$(\mathbf{A} \ d : d \in \mathbf{D} \wedge \text{Ax.deX} \wedge \text{As.deSTACK [X]} : \text{As.((UPpush; UPPop).d)} = \text{As.d}) \quad [8]$$

$$(\mathbf{A} \ d : d \in \mathbf{D} \wedge \text{Ax.deX} \wedge \text{As.deSTACK [X]} : \text{Ax.((UPpush; UPTop).d)} = \text{Ax.d}) \quad [9]$$

Ein Unterprogrammsystem, das die Forderungen [1] bis [9] oben erfüllt, erfüllt (ggf. mit Einschränkung, siehe unten) die in Abschnitt A1.2.1 aufgeführte Spezifikation, d.h., die vier Axiome der Spezifikation des Abschnitts A1.2.1 folgen aus den Bedingungen [1]

bis [9]. Als Beispiel dieses für manche intuitiv bzw. offensichtlich wahren Satzes wird im folgenden bewiesen, daß das vierte Axiom aus den Bedingungen [3], [5] und [9] folgt.

Der folgende Satz bezieht sich nur auf die vom Unterprogrammsystem dargestellten Kombinationen von Elementen aus X und STACK [X]; diese Kombinationen sind genau die Paare (x, s) aus der Bildmenge (Ax, As). \mathbf{D} , die eine Teilmenge von X \times STACK [X] ist ((Ax, As). $\mathbf{D} \subseteq X \times \text{STACK [X]}$). Sind die Auslegungsfunktionen Ax und As surjektiv (siehe oben), dann gilt, daß Ax. $\mathbf{D} = X$ bzw. As. $\mathbf{D} = \text{STACK [X]}$. Ist die kombinierte Auslegungsfunktion (Ax, As) surjektiv, dann gilt, daß (Ax, As). $\mathbf{D} = X \times \text{STACK [X]}$.

Satz: (A x, s : (x, s) \in (Ax, As). \mathbf{D} : top.(push.(x, s)) = x)

Beweis: Sei (x, s) aus (Ax, As). \mathbf{D} . Für jedes $d \in \mathbf{D}$ mit Ax.d = x und As.d = s gilt:

$$\begin{aligned} & \text{Ax.((UPpush; UPTop).d)} = \text{Ax.d} && \text{[siehe [9] oben]} \\ = & \text{Ax.(UPTop.(UPpush.d))} = x && \text{[Definition einer Folge von Anweisungen]} \\ = & \text{top.(As.(UPpush.d))} = x && [3, 5] \\ = & \text{top.(push.(Ax.d, As.d))} = x && [3] \\ = & \text{top.(push.(x, s))} = x \quad \blacksquare \end{aligned}$$

Die anderen Axiome der Spezifikation des Abschnitts A1.2.1 können auf die gleiche Weise bewiesen werden.

A1.2.3 Spezifikation durch Vor- und Nachbedingungen

Die Bedingungen [1] bis [9] im Abschnitt A1.2.2 oben können aufgrund der in Abschnitt 3.1 vorgestellten Definitionen in der Form von Vor- und Nachbedingungen geschrieben werden:

$$\{\text{AseSTACK [X]} \text{ UPempty } \{\text{Aempty} = \text{empty.(As')}\} \text{ strikt} \quad [1]$$

$$\{\text{wahr}\} \text{ UPnew } \{\text{As} = \text{new}\} \text{ strikt} \quad [2]$$

$$\{\text{Ax} \in X \wedge \text{AseSTACK [X]}\} \text{ UPpush } \{\text{As} = \text{push.(Ax', As')}\} \text{ strikt} \quad [3]$$

$$\{\text{AseSTACK [X]} \wedge \neg \text{empty.As}\} \text{ UPPop } \{\text{As} = \text{pop.(As')}\} \text{ strikt} \quad [4]$$

$$\{\text{AseSTACK [X]} \wedge \neg \text{empty.As}\} \text{ UPTop } \{\text{Ax} = \text{top.(As')}\} \text{ strikt} \quad [5]$$

$$\{\text{wahr}\} \text{ UPnew; UPempty } \{\text{Aempty}\} \text{ strikt} \quad [6]$$

$$\{\text{Ax} \in X \wedge \text{AseSTACK [X]}\} \text{ UPpush; UPempty } \{\neg \text{Aempty}\} \text{ strikt} \quad [7]$$

$$\{\text{Ax} \in X \wedge \text{AseSTACK [X]}\} \text{ UPpush; UPPop } \{\text{As} = \text{As'}\} \text{ strikt} \quad [8]$$

$$\{\text{Ax} \in X \wedge \text{AseSTACK [X]}\} \text{ UPpush; UPTop } \{\text{Ax} = \text{Ax'}\} \text{ strikt} \quad [9]$$

Dabei stellen die Bedingungen [1] bis [5] eine Zuordnung der Funktionen der Spezifikation des Abschnitts A1.2.1 zu den Unterprogrammen dar; sie stellen vor allem sicher, daß gewisse Auswirkungen der Ausführung der Unterprogramme bestimmte funktionale Beziehungen zu Eigenschaften der ursprünglichen Datenumgebungen aufweisen. Die Bedingungen [6] bis [9] stellen die vier Axiome dar.

A1.2.4 Konkrete Spezifikationsstufe 1 eines Unterprogrammsystems

Die Menge (der Typ) STACK [X] ist in den bisherigen Spezifikationsstufen nicht näher festgelegt worden. Eine in einer typischen Programmiersprache geschriebene Implementierung setzt eine detailliertere Angabe dieser Menge bzw. einer programmtechnischen Abbildung davon voraus.

Es liegt nahe, eine Variable des Typs STACK [X] durch eine endlich lange Folge von Elementen aus X zu verwirklichen. Entsprechend wird die Auslegungsfunktion As als die Zusammensetzung zweier Funktionen Af und Afs festgelegt:

$$\begin{aligned} \text{Af: } \mathbf{D} &\rightarrow X^* \\ \text{Afs: } X^* &\rightarrow \text{STACK [X]} \\ \text{As} &\triangleq \text{Afs} \circ \text{Af} \end{aligned}$$

Af kann als eine Variable in der Implementierung dieser Spezifikationsstufe aufgefaßt werden. Die Funktion Afs stellt die Auslegung dieser Variable im Zusammenhang der höheren Spezifikationsstufe dar.

Definiert man Afs derart, daß

$$\begin{aligned} \text{Afs: } [] &\rightarrow \text{new} && [\text{Afs bildet die leere Folge in new ab.}] \\ \text{Afs: } f\&[x] &\rightarrow \text{push.}(x, \text{Afs.}f), && \text{für alle } f \in X^* \text{ und } x \in X \end{aligned}$$

dann wird die Spezifikation (Bedingungen [1] bis [9]) des Abschnitts A1.2.3 erfüllt, falls

$$\begin{aligned} \{AfeX^*\} \text{UPempty} \{Aempty=(Af'=[])\} &\text{ strikt} && [1] \\ \{\text{wahr}\} \text{UPnew} \{Af=[]\} &\text{ strikt} && [2] \\ \{Ax \in X \wedge AfeX^*\} \text{UPpush} \{(Af' \&[Ax'])=Af\} &\text{ strikt} && [3] \\ \{AfeX^* \wedge Af\neq []\} \text{UPpop} \{(\exists x : x \in X : Af\&[x]=Af')\} &\text{ strikt} && [4] \\ \{AfeX^* \wedge Af\neq []\} \text{UPtop} \{(\exists f : feX^* : f\&[Ax]=Af')\} &\text{ strikt} && [5] \end{aligned}$$

Werden in der Zielprogrammiersprache Folgen als Datenarten unterstützt, kann von dieser Spezifikation ausgehend der Programmcode unmittelbar konstruiert werden. Sonst muß Af als ein Ausdruck betrachtet und die entsprechende Funktion festgelegt werden.

A1.2.5 Konkrete Spezifikationsstufe 2 eines Unterprogrammsystems

Eine Folge kann u.a. durch Feldelemente dargestellt werden, z.B.:

$$\text{Af} \triangleq \&_{i=1}^n [x(i)]$$

Hier wird die Folge Af in der Datenumgebung durch die Variablen n, x(1), ... x(n) dargestellt. Als Auslegungsfunktion betrachtet gibt Af an, wie diese Variablen als eine Folge zu interpretieren sind. Dabei ist der Feldname x willkürlich gewählt; er kann durch jeden anderen Namen ersetzt werden, der zu keinem Namenskonflikt führt.

Schreibt man die Spezifikation des Abschnitts A1.2.4 entsprechend um, erhält man die verfeinerte Spezifikation für das fragliche Unterprogrammsystem:

$$\begin{aligned} \{n \in \mathbf{Z} \text{ und } 0 \leq n \text{ und } \&_{i=1}^n x(i) \in X\} \text{UPempty} \{Aempty=(n'=0)\} &\text{ strikt} && [1] \\ \{\text{wahr}\} \text{UPnew} \{n=0\} &\text{ strikt} && [2] \end{aligned}$$

$$\begin{aligned} \{Ax \in X \text{ und } n \in \mathbf{Z} \text{ und } 0 \leq n \text{ und } \&_{i=1}^n x(i) \in X\} \\ \text{UPpush} \{n=n'+1 \text{ und } \&_{i=1}^{n-1} x(i)=x(i)' \text{ und } x(n)=Ax'\} &\text{ strikt} && [3] \end{aligned}$$

$$\begin{aligned} \{n \in \mathbf{Z} \text{ und } 0 < n \text{ und } \&_{i=1}^n x(i) \in X\} \\ \text{UPpop} \{n=n'-1 \text{ und } \&_{i=1}^n x(i)=x(i)'\} &\text{ strikt} && [4] \end{aligned}$$

$$\begin{aligned} \{n \in \mathbf{Z} \text{ und } 0 < n \text{ und } \&_{i=1}^n x(i) \in X\} \\ \text{UPtop} \{Ax=x(n)'\} &\text{ strikt} && [5] \end{aligned}$$

Wird in der Zielprogrammiersprache die Menge X nicht implementiert, dann müssen auch die Variable Ax und die Feldelemente x(.) dieser Spezifikationsstufe auf ähnliche Weise wie hier Af konkreter dargestellt werden.

Soll aus Implementierungsgründen die Größe des Felds x begrenzt werden, dann wird zunächst die zusätzliche Bedingung (nmax ∈ Z und n ≤ nmax) allen Vor- und Nachbedingungen oben hinzugefügt. Nach Vereinfachung der fraglichen Ausdrücke können die Spezifikationen wie folgt geschrieben werden:

$$\begin{aligned} \{n \in \mathbf{Z} \text{ und } nmax \in \mathbf{Z} \text{ und } 0 \leq n \leq nmax \text{ und } \&_{i=1}^n x(i) \in X\} \\ \text{UPempty} \{Aempty=(n'=0)\} &\text{ strikt} && [1a] \end{aligned}$$

$$\{\text{wahr}\} \text{UPnew} \{n=0\} \text{ strikt} \quad [2a]$$

$$\begin{aligned} \{Ax \in X \text{ und } n \in \mathbf{Z} \text{ und } nmax \in \mathbf{Z} \text{ und } 0 \leq n < nmax \text{ und } \&_{i=1}^n x(i) \in X\} \\ \text{UPpush} \{n=n'+1 \text{ und } \&_{i=1}^{n-1} x(i)=x(i)' \text{ und } x(n)=Ax'\} &\text{ strikt} && [3a] \end{aligned}$$

$$\begin{aligned} \{n \in \mathbf{Z} \text{ und } nmax \in \mathbf{Z} \text{ und } 0 < n \leq nmax \text{ und } \&_{i=1}^n x(i) \in X\} \\ \text{UPpop} \{n=n'-1 \text{ und } \&_{i=1}^n x(i)=x(i)'\} &\text{ strikt} && [4a] \end{aligned}$$

$$\begin{aligned} \{n \in \mathbf{Z} \text{ und } nmax \in \mathbf{Z} \text{ und } 0 < n \leq nmax \text{ und } \&_{i=1}^n x(i) \in X\} \\ \text{UPtop} \{Ax=x(n)'\} &\text{ strikt} && [5a] \end{aligned}$$

Wegen der stärkeren Bedingung n < nmax in der Vorbedingung bezüglich des Unterprogramms UPpush liegt es nahe, ein zusätzliches Unterprogramm mit Ergebnisvariable zu spezifizieren, womit festgestellt werden kann, ob diese zusätzliche Bedingung erfüllt ist oder nicht:

$$\begin{aligned} \{n \in \mathbf{Z} \text{ und } nmax \in \mathbf{Z} \text{ und } 0 \leq n \leq nmax \text{ und } \&_{i=1}^n x(i) \in X\} \\ \text{UPfull} \{Afull=(n' \geq nmax)\} &\text{ strikt} && [6a] \end{aligned}$$

Die oben aufgeführten Vorbedingungen stellen die vollständige Korrektheit der jeweiligen Unterprogramme nur dann sicher, wenn darin Variablen neue Werte durch declare-Anweisungen — nicht durch Zuweisungen — zugeordnet werden. Es liegt deshalb nahe, alle Vorbedingungen oben durch eine Bedingung zu stärken, die sicherstellt, daß alle eventuell benötigten Variablen auf geeignete Weise bereits vereinbart sind, z.B.

$$nmax \in \mathbf{Z} \text{ und } \{0, \dots, nmax\} \subseteq \text{Menge. "n" und } \&_{i=1}^{nmax} X \subseteq \text{Menge. "x(i)"}'$$

sowie ggf. darüber hinaus

und $B \subseteq \text{Menge}$. "Aempty" und $B \subseteq \text{Menge}$. "Afull" und $X \subseteq \text{Menge}$. "Ax"

Diese zusätzliche Bedingung wäre die Nachbedingung einer Initialisierung, die die fraglichen Variablen deklariert. Eine erweiterte Version des Unterprogramms UPnew könnte eine solche Initialisierung durchführen. Bei einer objektorientierten Implementierung würde die create-Routine der entsprechenden Klasse diese Aufgabe übernehmen.

Wegen der Stärkung der Vorbedingungen ist es nicht mehr gewährleistet, daß die Spezifikationen [1] bis [5] im vollen Umfang erfüllt werden. Effektiv ist die Auslegungsfunktion Af auf einen kleineren Definitionsbereich eingeschränkt worden mit dem Ergebnis, daß sie nicht mehr surjektiv ist. (Folgen mit mehr als nmax Gliedern werden nicht dargestellt.) Dadurch wird die zusammengesetzte Funktion As ebenfalls nicht mehr surjektiv. (Entsprechende Elemente aus STACK [X] werden nicht dargestellt.) Die durch [1a] bis [6a] spezifizierte Implementierung ist also keine vollständige Implementierung der allgemeineren Spezifikationen [1] bis [5]. Jede zulässige *Ausführung* eines durch [1a] bis [6a] spezifizierten Unterprogramms wird die entsprechende Anforderung der allgemeineren Spezifikation erfüllen, aber die eingeschränkte Spezifikation läßt nicht alle von der allgemeineren Spezifikation erlaubten Ausführungen zu.

Anhang 2. Dokumentationsbeispiel mit Korrektheitsbeweis für den Programmteil "Aufteilen"

Inhaltsübersicht

1. Schnittstellenspezifikation und Korrektheitsaussage
 - 1.1 Allgemeine Beschreibung der Wirkung des Programmteils "Aufteilen"
 - 1.2 Vorbedingung
 - 1.3 Nachbedingung
 - 1.4 Durch die Ausführung des Programmteils verursachte Veränderungen
 - 1.5 Korrektheitsaussage
2. Programmcode, Zwischenbedingungen und Schleifeninvarianten
 - 2.1 Programmcode mit Zwischenbedingungen
 - 2.2 Definitionen der Zwischenbedingungen und Schleifeninvarianten
 - 2.3 Diagrammatische Darstellung der Schleifeninvarianten
3. Zerlegung der zu beweisenden Korrektheitsaussage
 - 3.1 Untergeordnete Korrektheitsaussagen
 - 3.2 Beweiszerlegungsstruktur
4. Vollständige Korrektheit
 - 4.1 Terminierung der ersten while-Schleife
 - 4.2 Terminierung der zweiten while-Schleife
 - 4.3 Terminierung der repeat-Schleife
 - 4.3.1 Schleifenvariante und Terminierungsbedingung
 - 4.3.2 Korrektheitsaussage 2 für die Terminierung
 - 4.3.3 Zwischenbedingungen für den Terminierungsbeweis
 - 4.3.4 Zerlegung der Korrektheitsaussage 2
 - 4.3.5 Beweiszerlegungsstruktur für die Korrektheitsaussage 2
 - 4.4 Ausführung der einzelnen Anweisungen

Anlage 1. Beweise der nicht weiter zerlegten Korrektheitsaussagen

1. Schnittstellenspezifikation und Korrektheitsaussage

1.1 Allgemeine Beschreibung der Wirkung des Programmteils "Aufteilen"

Der Programmteil "Aufteilen" permutiert die Werte der Feldvariablen $a(i_1), \dots, a(i_r)$ und teilt sie in drei Bereiche ein. Nach der Ausführung des Programmteils werden die Werte der im unteren Bereich befindlichen Feldvariablen kleiner oder gleich $a(i)$ und diejenigen

im oberen Bereich größer oder gleich a(i) sein. Der mittlere Bereich wird nur aus a(i) bestehen.

Die Eingabevariablen sind il, ir und die Feldvariablen a(il), ... a(ir). Die Ergebnisvariablen sind i und die Feldvariablen a(il), ... a(ir).

1.2 Vorbedingung

Die gewöhnliche Vorbedingung V ist

$$V \triangleq \text{il} \in Z \text{ und } \text{ir} \in Z \text{ und } \text{il} \leq \text{ir} \quad [\text{zulässige Wertebereiche von il und ir}]$$

Eine strikte Vorbedingung Vst ist wie folgt, wobei M eine beliebige nicht leere linear geordnete Menge ist. (Die Bezeichnung M kommt im Programmteil Aufteilen nicht vor.)

$$Vst \triangleq V \text{ und } \bigwedge_{k=\text{il}}^{\text{ir}} (\text{Menge. "a(k)"} = M) \quad [\text{a(il), ... a(ir) deklariert, Wertebereich linear geordnet}]$$

Ggf. müssen zusätzliche zielsprachenspezifische Bedingungen erfüllt werden.

1.3 Nachbedingung

$$P \triangleq \text{il} \in Z \text{ und } \text{ir} \in Z \text{ und } \text{ie} \in Z \text{ und } \text{il} \leq \text{ir} \quad [\text{Wertebereiche}]$$

$$\text{und } \bigwedge_{k=\text{il}}^{i-1} a(k) \leq a(i) \quad [\text{unterer Bereich } \leq]$$

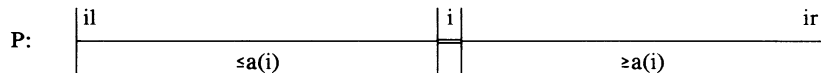
$$\text{und } \bigwedge_{k=i+1}^{\text{ir}} a(k) \geq a(i) \quad [\text{oberer Bereich } \geq]$$

$$\text{und } \&_{k=\text{il}}^{\text{ir}} [a(k)] \text{ Perm } A' \quad [\text{Werte in a(.) ursprünglich in a(.)}]$$

wobei A' als die Folge der ursprünglichen Werte der Feldvariablen a(k) definiert wird:

$$A' \triangleq \&_{k=\text{il}}^{\text{ir}} [a'(k)]$$

In diagrammatischer Form ist die Nachbedingung wie folgt:



1.4 Durch die Ausführung des Programmteils verursachte Veränderungen

Die Ausführung des Programmteils deklariert die Variable i und verändert die Werte der Feldvariablen a(il), ... a(ir). Sonst sind die vor- und nachherigen Datenumgebungen gleich. D.h., für jede Datenumgebung deVst gilt, daß Aufteilen.d=[(i, Z, .)]&d bis auf die Werte von a(il), ... a(ir).

1.5 Korrektheitsaussage

Die Korrektheitsaussage 1 lautet: {V} Aufteilen {P} bzw. {V} call Aufteilen {P}.

2. Programmcode, Zwischenbedingungen und Schleifeninvarianten

2.1 Programmcode mit Zwischenbedingungen

```

{V und  $\bigwedge_{k=\text{il}}^{\text{ir}} a(k) = a'(k)$ }
declare (i, Z, il); declare (j, Z, ir); declare (g, M, a(i));
{IR}
repeat
  {IR}
  while a(j) ≥ g und i < j do j := j - 1 invariant {IR} variant {j - i} endwhile;
  {IR und (i = j oder a(j) < g)}
  a(i) := a(j);
  {IW2 und (i = j oder a(i) < g)}
  while a(i) ≤ g und i < j do i := i + 1 invariant {IW2} variant {j - i} endwhile;
  {IW2 und (i = j oder a(i) > g)}
  a(j) := a(i)
invariant {IR} variant {j - i}
until i = j endrepeat;
{C}
a(i) := g;
{P}
release g; release j
{P}
    
```

Dieser Algorithmus [Grams; pers. Komm.], [Schauer; S. 148] vermeidet die Vertauschoperation, wovon andere vergleichbare Algorithmen [Foley], [Dijkstra; 1976, Chapter 14] Gebrauch machen.

2.2 Definitionen der Zwischenbedingungen und Schleifeninvarianten

$$IR \triangleq \text{il} \in Z \text{ und } \text{ir} \in Z \text{ und } \text{ie} \in Z \text{ und } \text{je} \in Z \text{ und } \text{il} \leq \text{ir} \leq \text{je} \leq \text{ir}$$

$$\text{und } \bigwedge_{k=\text{il}}^{i-1} a(k) \leq g \text{ und } \bigwedge_{k=j+1}^{\text{ir}} a(k) \geq g$$

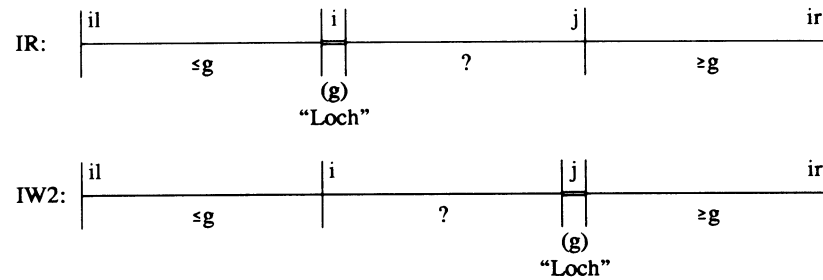
$$\text{und } \&_{k=\text{il}}^{i-1} [a(k)] \ \& \ [g] \ \&_{k=i+1}^{\text{ir}} [a(k)] \ \text{Perm } A'$$

$IW2 \triangleq$ $il \in Z$ und $ir \in Z$ und $i \in Z$ und $j \in Z$ und $ils \leq j \leq ir$
 und $\bigwedge_{k=il}^{i-1} a(k) \leq g$ und $\bigwedge_{k=j+1}^{ir} a(k) \geq g$
 und $\&_{k=il}^{j-1} [a(k)] \& [g] \&_{k=j+1}^{ir} [a(k)]$ Perm A'

(Bis auf j statt i in der jeweils dritten Zeile sind IR und IW2 gleich.)

$C \triangleq$ $il \in Z$ und $ir \in Z$ und $i \in Z$ und $ils \leq ir$
 und $\bigwedge_{k=il}^{i-1} a(k) \leq g$ und $\bigwedge_{k=i+1}^{ir} a(k) \geq g$
 und $\&_{k=il}^{i-1} [a(k)] \& [g] \&_{k=i+1}^{ir} [a(k)]$ Perm A'

2.3 Diagrammatische Darstellung der Schleifeninvarianten



3. Zerlegung der zu beweisenden Korrektheitsaussage

3.1 Untergeordnete Korrektheitsaussagen

- Zu beweisen: KA 1

KA 1 ($\{V \bigwedge_{k=il}^{ir} a(k) = a'(k)\}$ Aufteilen {P}, siehe Abschnitte 1.5 und 2.1) gilt (F1), falls

KA 1.1: $\{V \bigwedge_{k=il}^{ir} a(k) = a'(k)\}$
 declare (i, Z, il); declare (j, Z, ir); declare (g, M, a(i)) {IR}

KA 1.2: {IR} repeat ... invariant {IR} until i=j endrepeat {C}

KA 1.3: {C} a(i)=g {P}

g.o.: {P} release g; release j {P} [g,j kommen in P nicht vor]

- Noch zu beweisen: 1.1, 1.2, 1.3

3. Zerlegung der zu beweisenden Korrektheitsaussage

KA 1.1 gilt (F1, Z1 und Z2), falls

KA 1.4: $V \bigwedge_{k=il}^{ir} a(k) = a'(k) \Rightarrow ((IR_{a(i)}^g)_{ir}^j)_{il}^i$

- Noch zu beweisen: 1.2, 1.3, 1.4*

KA 1.2 gilt (R2), falls

KA 1.5: {IR} Schleifenkern der repeat-Schleife {IR}

KA 1.6: IR und i=j \Rightarrow C

- Noch zu beweisen: 1.3, 1.4*, 1.5, 1.6*

KA 1.5 gilt (F1), falls

KA 1.7: {IR} while a(j) \geq g und i<j do j:=j-1 invariant {IR} endwhile
 {IR und (i=j oder a(j)<g)}

KA 1.8: {IR und (i=j oder a(j)<g)} a(i):=a(j) {IW2 und (i=j oder a(i)<g)}

KA 1.9: {IW2 und (i=j oder a(i)<g)}
 while a(i) \leq g und i<j do i:=i+1 invariant {IW2} endwhile
 {IW2 und (i=j oder a(i)>g)}

KA 1.10: {IW2 und (i=j oder a(i)>g)} a(j):=a(i) {IR}

- Noch zu beweisen: 1.3, 1.4*, 1.6*, 1.7, 1.8, 1.9, 1.10

KA 1.7 gilt (W2), falls

KA 1.11: {IR und a(j) \geq g und i<j} j:=j-1 {IR}

KA 1.12: IR und nicht (a(j) \geq g und i<j) \Rightarrow IR und (i=j oder a(j)<g)

- Noch zu beweisen: 1.3, 1.4*, 1.6*, 1.8, 1.9, 1.10, 1.11, 1.12*

KA 1.11 gilt (Z2), falls

KA 1.13: IR und a(j) \geq g und i<j \Rightarrow IR $_{j-1}^j$

- Noch zu beweisen: 1.3, 1.4*, 1.6*, 1.8, 1.9, 1.10, 1.12*, 1.13*

KA 1.8 gilt (Z2), falls

KA 1.14: IR und (i=j oder a(j)<g) \Rightarrow [IW2 und (i=j oder a(i)<g)] $_{a(j)}^{a(i)}$

- Noch zu beweisen: 1.3, 1.4*, 1.6*, 1.9, 1.10, 1.12*, 1.13*, 1.14*

KA 1.9 gilt (W2, B1), falls

g.o.: IW2 und (i=j oder a(i)<g) \Rightarrow IW2

KA 1.15: {IW2 und a(i) \leq g und i<j} i:=i+1 {IW2}

KA 1.16: IW2 und nicht (a(i) \leq g und i<j) \Rightarrow IW2 und (i=j oder a(i)>g)

- Noch zu beweisen: 1.3, 1.4*, 1.6*, 1.10, 1.12*, 1.13*, 1.14*, 1.15, 1.16*

KA 1.15 gilt (Z2), falls

KA 1.17: IW2 und a(i) \leq g und i<j \Rightarrow IW2 $_{i+1}^i$

- Noch zu beweisen: 1.3, 1.4*, 1.6*, 1.10, 1.12*, 1.13*, 1.14*, 1.16*, 1.17*

KA 1.10 gilt (Z2), falls

$$\text{KA 1.18: IW2 und } (i=j \text{ oder } a(i)>g) \Rightarrow \text{IR}_{a(i)}^{a(i)}$$

• Noch zu beweisen: 1.3, 1.4*, 1.6*, 1.12*, 1.13*, 1.14*, 1.16*, 1.17*, 1.18*

KA 1.3 gilt (Z2), falls

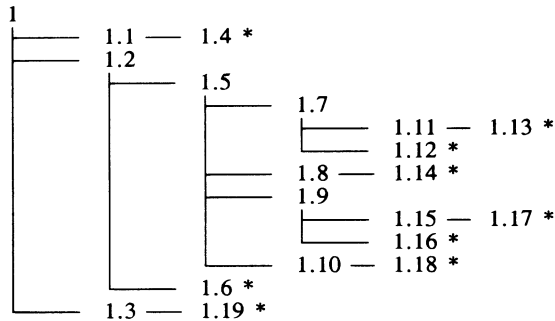
$$\text{KA 1.19: } C \Rightarrow P_g^{a(i)}$$

• Noch zu beweisen: 1.4*, 1.6*, 1.12*, 1.13*, 1.14*, 1.16*, 1.17*, 1.18*, 1.19*

* Boolescher Ausdruck (in der Anlage 1 verifiziert)

g.o. = gilt offensichtlich

3.2 Beweiserlegungsstruktur



* Die nicht weiter zerlegten Korrektheitsaussagen werden in der Anlage 1 bewiesen.

4. Vollständige Korrektheit

4.1 Terminierung der ersten while-Schleife

Im Kern der ersten while-Schleife wird j um 1 verringert. Die while-Bedingung enthält eine untere Schranke für j (i). Daraus folgt, daß die Schleife terminiert.

4.2 Terminierung der zweiten while-Schleife

Im Kern der zweiten while-Schleife wird i um 1 erhöht. Die while-Bedingung enthält eine obere Schranke für i (j). Daraus folgt, daß die Schleife terminiert.

4.3 Terminierung der repeat-Schleife

4.3.1 Schleifenvariante und Terminierungsbedingung

Die Anzahl $(j-i)$ der Feldvariablen im ?Bereich (siehe das Diagramm für die Schleifenvariante IR im Abschnitt 2.3 oben) ist eine geeignete Schleifenvariante für die repeat-Schleife. Der Wert dieses Ausdrucks ist nicht negativ (siehe die Schleifenvariante IR im Abschnitt 2.2). Falls der Wert dieses Ausdrucks positiv ist, verringert ihn der Schleifenkern um mindestens 1. Wenn $(j-i)=0$, terminiert die Schleife (siehe die until-Bedingung).

Die Nachbedingung

$$T \triangleq 0=j-i \text{ oder } 0<j-i \leq j'-i'-1$$

bezüglich des Schleifenkerns stellt sicher, daß die Schleife terminiert. Dabei sind i' und j' die Werte der Variablen i bzw. j vor der jeweiligen Ausführung des Schleifenkerns.

4.3.2 Korrektheitsaussage 2 für die Terminierung

Die folgende Korrektheitsaussage 2 stellt also sicher, daß die repeat-Schleife terminiert:

$$\text{KA 2} \triangleq \{ \text{IR und } i=i' \text{ und } j=j' \} \text{ Schleifenkern der repeat-Schleife } \{ T \}$$

4.3.3 Zwischenbedingungen für den Terminierungsbeweis

```
{IR und i=i' und j=j'}
  while a(j)≥g und i<j do j:=j-1 invariant {IW1T} endwhile;
{IW1T und (i=j oder a(j)<g)}
  a(i):=a(j);
{IW1T und (i=j oder a(i)<g)}
  while a(i)≤g und i<j do i:=i+1 invariant {IW2T} endwhile;
{T}
  a(j):=a(i)
{T}
```

wobei IW1T und IW2T wie folgt definiert werden:

$$\text{IW1T} \triangleq i \in \mathbb{Z} \text{ und } j \in \mathbb{Z} \text{ und } 0 \leq j-i \leq j'-i'$$

$$\text{IW2T} \triangleq \text{IW1T und } (0=j-i \text{ oder } j-i \leq j'-i'-1 \text{ oder } a(i)<g)$$

4.3.4 Zerlegung der Korrektheitsaussage 2

• Zu beweisen: KA 2

KA 2 (siehe Abschnitte 4.3.2 und 4.3.3) gilt (F1), falls

KA 2.1: $\{IR \text{ und } i=i' \text{ und } j=j'\}$
 $\text{while } a(j) \geq g \text{ und } i < j \text{ do } j:=j-1 \text{ invariant } \{IW1T\} \text{ endwhile}$
 $\{IW1T \text{ und } (i=j \text{ oder } a(j) < g)\}$
 g.o.: $\{IW1T \text{ und } (i=j \text{ oder } a(j) < g)\} a(i):=a(j)$
 $\{IW1T \text{ und } (i=j \text{ oder } a(i) < g)\}$
 [Z1, [a(.) kommt in IW1T nicht vor]]

KA 2.2: $\{IW1T \text{ und } (i=j \text{ oder } a(i) < g)\}$
 $\text{while } a(i) \leq g \text{ und } i < j \text{ do } i:=i+1 \text{ invariant } \{IW2T\} \text{ endwhile } \{T\}$
 g.o.: $\{T\} a(j):=a(i) \{T\}$ [a(.) kommt in T nicht vor]]

• Noch zu beweisen: 2.1, 2.2

KA 2.1 gilt (W2, B1), falls

KA 2.3: $IR \text{ und } i=i' \text{ und } j=j' \Rightarrow IW1T$

KA 2.4: $\{IW1T \text{ und } a(j) \geq g \text{ und } i < j\} j:=j-1 \{IW1T\}$

KA 2.5: $IW1T \text{ und nicht } (a(j) \geq g \text{ und } i < j) \Rightarrow IW1T \text{ und } (i=j \text{ oder } a(j) < g)$

• Noch zu beweisen: 2.2, 2.3*, 2.4, 2.5*

KA 2.4 gilt (Z2), falls

KA 2.6: $IW1T \text{ und } a(j) \geq g \text{ und } i < j \Rightarrow IW1T_{j-1}^j$

• Noch zu beweisen: 2.2, 2.3*, 2.5*, 2.6*

KA 2.2 gilt (W2, B1), falls

KA 2.7: $IW1T \text{ und } (i=j \text{ oder } a(i) < g) \Rightarrow IW2T$

KA 2.8: $\{IW2T \text{ und } a(i) \leq g \text{ und } i < j\} i:=i+1 \{IW2T\}$

KA 2.9: $IW2T \text{ und nicht } (a(i) \leq g \text{ und } i < j) \Rightarrow T$

• Noch zu beweisen: 2.3*, 2.5*, 2.6*, 2.7*, 2.8, 2.9*

KA 2.8 gilt (Z2), falls

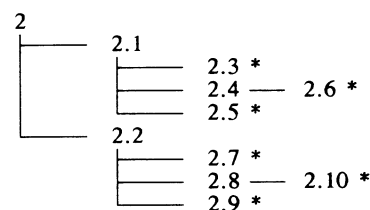
KA 2.10: $IW2T \text{ und } a(i) \leq g \text{ und } i < j \Rightarrow IW2T_{i+1}^i$

• Noch zu beweisen: 2.3*, 2.5*, 2.6*, 2.7*, 2.9*, 2.10*

* Boolescher Ausdruck (in der Anlage 1 verifiziert)

g.o. = gilt offensichtlich

4.3.5 Beweiserlegungsstruktur für die Korrektheitsaussage 2



* Die nicht weiter zerlegten Korrektheitsaussagen werden in der Anlage 1 bewiesen.

4.4 Ausführung der einzelnen Anweisungen

Die strikte Vorbedingung Vst (die auch eine Programminvariante ist), die im Programmteil enthaltenen declare-Anweisungen sowie die an den verschiedenen Stellen geltenden Zwischenbedingungen (siehe Abschnitt 2.1) stellen sicher, daß vor jeder Anweisung bzw. vor jeder Auswertung einer Schleifenbedingung die angesprochenen Programmvariablen deklariert sind und Werte im jeweils erforderlichen Bereich aufweisen. Deswegen wird jede Anweisung mit einem definierten Ergebnis ausgeführt werden.

Die im Programmteil und im Korrektheitsbeweis angesprochene Menge Z darf eine endliche Teilmenge der ganzen Zahlen sein, solange sie alle Ganzzahlen zwischen il und ir einschließlich umfaßt (vgl. die Bedingungen IR, IW2, C und P).

Anlage 1. Beweise der nicht weiter zerlegten Korrektheitsaussagen

• Für KA 1 noch zu beweisen: 1.4*, 1.6*, 1.12*, 1.13*, 1.14*, 1.16*, 1.17*, 1.18*, 1.19* (siehe Ende des Abschnitts 3.1)

KA 1.4: $((IR_{a(i)}^g)_{ir}^j)_{il}^i$

$$\begin{aligned}
 &= il \in Z \text{ und } ir \in Z \text{ und } il \leq ir \text{ und } ir \leq ir \text{ und } il \leq ir \text{ und } ir \leq ir \\
 &\text{und } \bigwedge_{k=il}^{il-1} a(k) \leq a(il) \text{ und } \bigwedge_{k=ir+1}^{ir} a(k) \geq a(ir) \quad [\text{leere und-Reihen}] \\
 &\text{und } \&_{k=il}^{il-1} [a(k)] \& [a(il)] \&_{k=il+1}^{ir} [a(k)] \text{ Perm } A' \\
 &= il \in Z \text{ und } ir \in Z \text{ und } il \leq ir \quad [\text{Definition von } A'] \\
 &\text{und } \&_{k=il}^{ir} [a(k)] \text{ Perm } \&_{k=il}^{ir} [a'(k)] \\
 &\leftarrow \\
 &il \in Z \text{ und } ir \in Z \text{ und } il \leq ir \text{ und } \bigwedge_{k=il}^{ir} a(k) = a'(k) \\
 &=
 \end{aligned}$$

$$\forall \text{ und } \underset{k=i}{\text{ir}} a(k) = a'(k) \blacksquare$$

KA 1.6: IR und $i=j$

$$\begin{aligned} &= \\ & \text{ileZ und ireZ und iεZ und jεZ und } i \leq i = j \leq i \text{r} \\ & \text{und } \underset{k=i}{\text{ir}} a(k) \leq g \text{ und } \underset{k=j+1}{\text{ir}} a(k) \geq g \\ & \text{und } \&_{k=i}^{i-1} [a(k)] \& [g] \&_{k=i+1}^{\text{ir}} [a(k)] \text{ Perm A}' \\ \Rightarrow & \\ & \text{ileZ und ireZ und iεZ und } i \leq i \leq i \text{r} \\ & \text{und } \underset{k=i}{\text{ir}} a(k) \leq g \text{ und } \underset{k=i+1}{\text{ir}} a(k) \geq g \\ & \text{und } \&_{k=i}^{i-1} [a(k)] \& [g] \&_{k=i+1}^{\text{ir}} [a(k)] \text{ Perm A}' \\ &= \\ & \text{C } \blacksquare \end{aligned}$$

KA 1.12: IR und nicht $(a(j) \geq g \text{ und } i < j)$

$$\begin{aligned} &= \quad [i \leq j \text{ ist ein Term in IR, siehe Definition von IR}] \\ & \text{IR und } i \leq j \text{ und } (a(j) < g \text{ oder } i \geq j) \\ &= \\ & \text{IR und } (i=j \text{ oder } a(j) < g) \blacksquare \end{aligned}$$

KA 1.13: IR_{j-1}^j

$$\begin{aligned} &= \\ & \text{ileZ und ireZ und iεZ und } j-1 \text{εZ und } i \leq i \leq j-1 \leq i \text{r} \\ & \text{und } \underset{k=i}{\text{ir}} a(k) \leq g \text{ und } \underset{k=j}{\text{ir}} a(k) \geq g \\ & \text{und } \&_{k=i}^{i-1} [a(k)] \& [g] \&_{k=i+1}^{\text{ir}} [a(k)] \text{ Perm A}' \\ \Leftarrow & \\ & \text{ileZ und ireZ und iεZ und jεZ und } i \leq i < j \leq i \text{r} \\ & \text{und } \underset{k=i}{\text{ir}} a(k) \leq g \text{ und } a(j) \geq g \text{ und } \underset{k=j+1}{\text{ir}} a(k) \geq g \\ & \text{und } \&_{k=i}^{i-1} [a(k)] \& [g] \&_{k=i+1}^{\text{ir}} [a(k)] \text{ Perm A}' \\ &= \\ & \text{IR und } a(j) \geq g \text{ und } i < j \blacksquare \end{aligned}$$

KA 1.14: IW2 und $(i=j \text{ oder } a(i) < g)$

$$\begin{aligned} &= \quad [i \leq j \text{ ist ein Term in IW2, siehe Definition von IW2}] \\ & \text{IW2 und } (i=j \text{ oder } i < j \text{ und } a(i) < g) \\ &= \end{aligned}$$

$$\begin{aligned} & \text{ileZ und ireZ und iεZ und jεZ und } i \leq i \leq j \leq i \text{r} \\ & \text{und } \underset{k=i}{\text{ir}} a(k) \leq g \text{ und } \underset{k=j+1}{\text{ir}} a(k) \geq g \\ & \text{und } [i=j \text{ und } \&_{k=i}^{j-1} [a(k)] \& [g] \&_{k=j+1}^{\text{ir}} [a(k)] \text{ Perm A}' \\ & \quad \text{oder } i < j \text{ und } a(i) < g \text{ und } \&_{k=i}^{j-1} [a(k)] \& [g] \&_{k=j+1}^{\text{ir}} [a(k)] \text{ Perm A}'] \\ &= \\ & \text{ileZ und ireZ und iεZ und jεZ und } i \leq i \leq j \leq i \text{r} \\ & \text{und } \underset{k=i}{\text{ir}} a(k) \leq g \text{ und } \underset{k=j+1}{\text{ir}} a(k) \geq g \\ & \text{und } [i=j \text{ und } \&_{k=i}^{i-1} [a(k)] \& [g] \&_{k=i+1}^{\text{ir}} [a(k)] \text{ Perm A}' \\ & \quad \text{oder } i < j \text{ und } a(i) < g \\ & \quad \text{und } \&_{k=i}^{i-1} [a(k)] \& [a(i)] \&_{k=i+1}^{j-1} [a(k)] \& [g] \&_{k=j+1}^{\text{ir}} [a(k)] \\ & \quad \text{Perm A}'] \end{aligned}$$

Deshalb gilt, daß

$$\begin{aligned} & [\text{IW2 und } (i=j \text{ oder } a(i) < g)]^{a(i)}_{a(j)} \\ &= \\ & \text{ileZ und ireZ und iεZ und jεZ und } i \leq i \leq j \leq i \text{r} \\ & \text{und } \underset{k=i}{\text{ir}} a(k) \leq g \text{ und } \underset{k=j+1}{\text{ir}} a(k) \geq g \\ & \text{und } [i=j \text{ und } \&_{k=i}^{i-1} [a(k)] \& [g] \&_{k=i+1}^{\text{ir}} [a(k)] \text{ Perm A}' \\ & \quad \text{oder } i < j \text{ und } a(j) < g \\ & \quad \text{und } \&_{k=i}^{i-1} [a(k)] \& [a(j)] \&_{k=i+1}^{j-1} [a(k)] \& [g] \&_{k=j+1}^{\text{ir}} [a(k)] \\ & \quad \text{Perm A}'] \\ &= \\ & \text{ileZ und ireZ und iεZ und jεZ und } i \leq i \leq j \leq i \text{r} \\ & \text{und } \underset{k=i}{\text{ir}} a(k) \leq g \text{ und } \underset{k=j+1}{\text{ir}} a(k) \geq g \\ & \text{und } [i=j \text{ und } \&_{k=i}^{i-1} [a(k)] \& [g] \&_{k=i+1}^{\text{ir}} [a(k)] \text{ Perm A}' \\ & \quad \text{oder } i < j \text{ und } a(j) < g \\ & \quad \text{und } \&_{k=i}^{i-1} [a(k)] \& [g] \&_{k=i+1}^{\text{ir}} [a(k)] \text{ Perm A}'] \\ &= \\ & \text{ileZ und ireZ und iεZ und jεZ und } i \leq i \leq j \leq i \text{r} \\ & \text{und } \underset{k=i}{\text{ir}} a(k) \leq g \text{ und } \underset{k=j+1}{\text{ir}} a(k) \geq g \\ & \text{und } \&_{k=i}^{i-1} [a(k)] \& [g] \&_{k=i+1}^{\text{ir}} [a(k)] \text{ Perm A}' \\ & \text{und } (i=j \text{ oder } i < j \text{ und } a(j) < g) \end{aligned}$$

=
 IR und (i=j oder i<j und a(j)<g)
 = [i≤j ist ein Term in IR, siehe Definition von IR]
 IR und (i=j oder a(j)<g) ■

KA 1.16: IW2 und nicht (a(i)≤g und i<j)

=
 IW2 und (i≥j oder a(i)>g)
 = [i≤j ist ein Term in IW2, siehe Definition von IW2]
 IW2 und (i=j oder a(i)>g) ■

KA 1.17: IW2_{i+1}ⁱ

=
 ileZ und ireZ und i+1eZ und jeZ und ilsi+1sjsir
 und $\underset{k=i}{\text{und}}^{i-1} a(k) \leq g$ und $\underset{k=j+1}{\text{und}}^{ir} a(k) \geq g$
 und $\&_{k=i}^{j-1} [a(k)] \& [g] \&_{k=j+1}^{ir} [a(k)]$ Perm A'
 ⇐
 ileZ und ireZ und ieZ und jeZ und ilsi<jsir
 und $\underset{k=i}{\text{und}}^{i-1} a(k) \leq g$ und a(i)≤g und $\underset{k=j+1}{\text{und}}^{ir} a(k) \geq g$
 und $\&_{k=i}^{j-1} [a(k)] \& [g] \&_{k=j+1}^{ir} [a(k)]$ Perm A'
 =
 IW2 und a(i)≤g und i<j ■

KA 1.18: IR

=
 ileZ und ireZ und ieZ und jeZ und ilsi≤jsir
 und $\underset{k=i}{\text{und}}^{i-1} a(k) \leq g$ und $\underset{k=j+1}{\text{und}}^{ir} a(k) \geq g$
 und $\&_{k=i}^{i-1} [a(k)] \& [g] \&_{k=i+1}^{ir} [a(k)]$ Perm A'
 =
 ileZ und ireZ und ieZ und jeZ und ilsi≤jsir
 und $\underset{k=i}{\text{und}}^{i-1} a(k) \leq g$ und $\underset{k=j+1}{\text{und}}^{ir} a(k) \geq g$
 und [i=j und $\&_{k=i}^{j-1} [a(k)] \& [g] \&_{k=j+1}^{ir} [a(k)]$ Perm A'
 oder
 i<j und $\&_{k=i}^{i-1} [a(k)] \& [g] \&_{k=i+1}^{j-1} [a(k)] \& [a(j)] \&_{k=j+1}^{ir} [a(k)]$
 Perm A']

Deshalb gilt, daß

IR_{a(i)}^{a(j)}
 =

ileZ und ireZ und ieZ und jeZ und ilsi≤jsir
 und $\underset{k=i}{\text{und}}^{i-1} a(k) \leq g$ und $\underset{k=j+1}{\text{und}}^{ir} a(k) \geq g$
 und [i=j und $\&_{k=i}^{j-1} [a(k)] \& [g] \&_{k=j+1}^{ir} [a(k)]$ Perm A'
 oder
 i<j und $\&_{k=i}^{i-1} [a(k)] \& [g] \&_{k=i+1}^{j-1} [a(k)] \& [a(i)] \&_{k=j+1}^{ir} [a(k)]$
 Perm A']

=
 ileZ und ireZ und ieZ und jeZ und ilsi≤jsir
 und $\underset{k=i}{\text{und}}^{i-1} a(k) \leq g$ und $\underset{k=j+1}{\text{und}}^{ir} a(k) \geq g$
 und [i=j und $\&_{k=i}^{j-1} [a(k)] \& [g] \&_{k=j+1}^{ir} [a(k)]$ Perm A'
 oder
 i<j und $\&_{k=i}^{j-1} [a(k)] \& [g] \&_{k=j+1}^{ir} [a(k)]$ Perm A']

=
 ileZ und ireZ und ieZ und jeZ und ilsi≤jsir
 und $\underset{k=i}{\text{und}}^{i-1} a(k) \leq g$ und $\underset{k=j+1}{\text{und}}^{ir} a(k) \geq g$
 und $\&_{k=i}^{j-1} [a(k)] \& [g] \&_{k=j+1}^{ir} [a(k)]$ Perm A'
 =
 IW2
 ⇐
 IW2 und (i=j oder a(i)>g) ■

KA 1.19: P

=
 ileZ und ireZ und ieZ und ilsi≤jsir
 und $\underset{k=i}{\text{und}}^{i-1} a(k) \leq a(i)$ und $\underset{k=i+1}{\text{und}}^{ir} a(k) \geq a(i)$
 und $\&_{k=i}^{i-1} [a(k)] \& [a(i)] \&_{k=i+1}^{ir} [a(k)]$ Perm A'

Deshalb gilt, daß

P_g^{a(i)}
 =
 ileZ und ireZ und ieZ und ilsi≤jsir
 und $\underset{k=i}{\text{und}}^{i-1} a(k) \leq g$ und $\underset{k=i+1}{\text{und}}^{ir} a(k) \geq g$
 und $\&_{k=i}^{i-1} [a(k)] \& [g] \&_{k=i+1}^{ir} [a(k)]$ Perm A'
 =
 C ■

• Für KA 2 noch zu beweisen: 2.3*, 2.5*, 2.6*, 2.7*, 2.9*, 2.10* (siehe Ende des Abschnitts 4.3.4)

KA 2.3: IW1T

$$\begin{aligned}
 &= \\
 &= i \in \mathbb{Z} \text{ und } j \in \mathbb{Z} \text{ und } 0 \leq j - i \leq j' - i' \\
 &= i \in \mathbb{Z} \text{ und } j \in \mathbb{Z} \text{ und } i \leq j \text{ und } j - i \leq j' - i' \\
 &\Leftarrow \\
 &= IR \text{ und } i = i' \text{ und } j = j' \quad \blacksquare
 \end{aligned}$$

KA 2.5: IW1T und nicht $(a(j) \geq g \text{ und } i < j)$

$$\begin{aligned}
 &= \\
 &= IW1T \text{ und } (i \geq j \text{ oder } a(j) < g) \\
 &= IW1T \text{ und } (i = j \text{ oder } a(j) < g) \quad \blacksquare \quad [0 \leq j - i (= i \leq j) \text{ ist ein Term in IW1T}]
 \end{aligned}$$

KA 2.6: $IW1T_{j-1}^i$

$$\begin{aligned}
 &= \\
 &= i \in \mathbb{Z} \text{ und } j - 1 \in \mathbb{Z} \text{ und } 0 \leq j - 1 - i \leq j' - i' \\
 &\Leftarrow \\
 &= i \in \mathbb{Z} \text{ und } j \in \mathbb{Z} \text{ und } i < j \text{ und } j - i \leq j' - i' \\
 &\Leftarrow \\
 &= IW1T \text{ und } i < j \text{ und } a(j) \geq g \quad \blacksquare
 \end{aligned}$$

KA 2.7: IW1T und $(i = j \text{ oder } a(i) < g)$

$$\begin{aligned}
 &\Rightarrow \\
 &= IW1T \text{ und } (0 = j - i \text{ oder } j - i \leq j' - i' - 1 \text{ oder } a(i) < g) \\
 &= \\
 &= IW2T \quad \blacksquare
 \end{aligned}$$

KA 2.9: IW2T und nicht $(a(i) \leq g \text{ und } i < j)$

$$\begin{aligned}
 &= \\
 &= IW2T \text{ und } (a(i) > g \text{ oder } i \geq j) \\
 &= \\
 &\Rightarrow IW1T \text{ und } (0 = j - i \text{ oder } j - i \leq j' - i' - 1 \text{ oder } a(i) < g) \text{ und } (a(i) > g \text{ oder } 0 \geq j - i) \\
 &= (0 = j - i \text{ oder } 0 \leq j - i \leq j' - i' - 1 \text{ oder } a(i) < g) \text{ und } (0 = j - i \text{ oder } a(i) > g) \quad [0 \leq j - i \text{ ist ein Term in IW1T}] \\
 &= \\
 &\Rightarrow 0 = j - i \text{ oder } 0 \leq j - i \leq j' - i' - 1 \text{ und } a(i) > g \\
 &\Rightarrow 0 = j - i \text{ oder } 0 \leq j - i \leq j' - i' - 1 \\
 &= \\
 &= 0 = j - i \text{ oder } 0 < j - i \leq j' - i' - 1 \\
 &= \\
 &= T \quad \blacksquare
 \end{aligned}$$

KA 2.10: $IW2T_{i+1}^i$

$$\begin{aligned}
 &= \\
 &= i + 1 \in \mathbb{Z} \text{ und } j \in \mathbb{Z} \text{ und } 0 \leq j - i - 1 \leq j' - i' \text{ und } (0 = j - i - 1 \text{ oder } j - i - 1 \leq j' - i' - 1 \text{ oder } a(i + 1) < g)
 \end{aligned}$$

$$\begin{aligned}
 &\Leftarrow \\
 &= i \in \mathbb{Z} \text{ und } j \in \mathbb{Z} \text{ und } i < j \text{ und } j - i \leq j' - i' \text{ und } j - i - 1 \leq j' - i' - 1 \\
 &= \\
 &= i \in \mathbb{Z} \text{ und } j \in \mathbb{Z} \text{ und } i < j \text{ und } j - i \leq j' - i' \\
 &\Leftarrow \quad \text{[siehe Definition von IW1T]} \\
 &= IW1T \text{ und } i < j \\
 &\Leftarrow \quad \text{[siehe Definition von IW2T]} \\
 &= IW2T \text{ und } i < j \text{ und } a(i) \leq g \quad \blacksquare
 \end{aligned}$$

Ende der Beweise

Anhang 3. Einige Fehlerschranken für Gleitkommaarithmetik

Bei der Korrektheitsbeweissführung muß man berücksichtigen, daß gleitkommaarithmetische Operationen nicht alle Eigenschaften der entsprechenden Operationen der klassischen Mathematik aufweisen. Z.B. ist die Gleitkommaaddition nicht assoziativ, Rundung führt zu Ungenauigkeiten usw.

Gleitkommaarithmetische Effekte können auf verschiedene Weise in Korrektheitsbeweisen berücksichtigt werden. Vielleicht am einfachsten ist es, in den Booleschen Ausdrücken der Korrektheitsbeweise die im Programm vorkommenden gleitkommaarithmetischen Operationen erscheinen zu lassen und außerhalb des Programmkorrektheitsbeweises selbst Aussagen über die Genauigkeit errechneter Ergebnisse — d.h. Aussagen über die größtmöglichen Abweichungen zwischen Ergebnissen der Gleitkommaarithmetik und den entsprechenden mathematisch idealen Werten — zu treffen.

In diesem Anhang werden einige dafür geeignete Fehlerschranken abgeleitet. Bestimmte Annahmen über die fraglichen gleitkommaarithmetischen Operationen bilden die Basis für die Ableitung dieser Fehlerschranken. Typische — jedoch nicht unbedingt alle — (korrekte) Gleitkommaarithmetik-Implementierungen erfüllen diese Annahmen.

A3.1 Definition und Axiome eines Gleitkommaarithmetiksystems

Die verschiedenen Implementierungen von Gleitkommaarithmetiksystemen weisen zum Teil wesentlich unterschiedliche Eigenschaften auf. Manche enthalten bekannte und offensichtliche Fehler [Du Croz]. U.a. deshalb wird der Einsatz der Gleitkommaarithmetik in kritischen Anwendungen nicht allgemein und ohne Widerspruch akzeptiert; siehe z.B. [Ministry of Defence (U.K.)] (insbesondere den früheren Entwurf von 1989 May) und [Wichmann; 1992 Feb.].

Alle Gleitkommaarithmetik-Implementierungen basieren jedoch auf dem gleichen Prinzip und weisen deshalb bestimmte gemeinsame Eigenschaften auf. Für Gleitkommaarithmetik gibt es Normen, z.B. [IEEE; 1987 Feb.] und [IEEE; 1987 October 5]. Die folgende Betrachtung eines Gleitkommaarithmetiksystems widerspiegelt die Eigenschaften typischer Implementierungen bzw. die Anstrengungen ihrer Konstrukteure [Goldberg], [Hoffmann; 1990 April-Juli, Abschnitt 9.8, nach einer nicht mehr feststellbaren Quelle], [IEEE; 1987 Feb.] und [IEEE; 1987 October 5].

Mathematisch kann eine Gleitkommaarithmetik-Implementierung als ein aus den folgenden Bestandteilen bestehendes System betrachtet werden:

- G**: eine endliche Menge (die Menge der Gleitkommazahlen)
- \oplus : eine partielle Funktion von $G \times G$ nach G (Gleitkommaaddition)
- \ominus : eine partielle Funktion von $G \times G$ nach G (Gleitkommasubtraktion)
- \otimes : eine partielle Funktion von $G \times G$ nach G (Gleitkommamultiplikation)
- \oslash : eine partielle Funktion von $G \times G$ nach G (Gleitkommadivision)

Reelle Zahlen werden durch Elemente von G dargestellt:

- Rg**: eine nicht leere Teilmenge von \mathbf{R} , gewöhnlich ein auf beiden Seiten beschränktes Intervall von \mathbf{R}
- g**: eine vollständige Funktion von \mathbf{Rg} nach G (die Darstellungsfunktion)

Sind im fraglichen Gleitkommaarithmetiksystem nur "normalisierte" Zahlen vorgesehen, hat die Menge G die folgende spezifische Struktur:

- R**: eine Ganzzahl größer als 1 (die Basis des Zahlensystems)
 - Man**: eine nicht leere endliche Teilmenge des Intervalls $[1, R)$ von \mathbf{R} (die Menge der Mantissenwerte)
 - emin**: eine Ganzzahl
 - emax**: eine Ganzzahl, wobei $emin < emax$
 - Exp**: die Menge aller Ganzzahlen zwischen $emin$ und $emax$ einschließlich (die Menge der Exponenten)
- $$G \triangleq (\cup s, m, e : se\{0, 1\} \wedge m \in \mathbf{Man} \wedge e \in \mathbf{Exp} : \{(-1)^s \cdot m \cdot R^e\}) \cup \{0\} \quad (1.1)$$

Der kleinste und der größte Wert aus G werden mit $gmin$ bzw. $gmax$ bezeichnet:

$$gmin \triangleq (\min a : a \in G : a)$$

$$gmax \triangleq (\max a : a \in G : a)$$

Es wird unterstellt, daß die Funktion g jedes Element aus G in sich selbst abbildet, d.h., daß die Funktion g beschränkt auf G die Identitätsfunktion ist:

$$G \subset \mathbf{Rg} \wedge (A a : a \in G : g.a = a) \quad (1.2)$$

Ferner wird unterstellt, daß g eine Zahl aus $\mathbf{Rg} - G$ in einen der nächst umliegenden Werte aus G abbildet ("rundet"):

$$(A x : x \in \mathbf{Rg} \wedge x < gmin : g.x = gmin) \wedge$$

$$(A x : x \in \mathbf{Rg} \wedge gmin \leq x \leq gmax :$$

$$(\max a : a \in G \wedge a \leq x : a) = g.x \vee g.x = (\min b : b \in G \wedge x \leq b : b)) \wedge$$

$$(A x : x \in \mathbf{Rg} \wedge gmax < x : g.x = gmax) \quad (1.3)$$

Die Annahme (1.3) oben ist schwächer als die naheliegende Annahme, die Funktion g sei monoton nicht absteigend:

$$(A x, y : x \in \mathbf{Rg} \wedge y \in \mathbf{Rg} \wedge x \leq y : g.x \leq g.y) \quad [\text{alternative Annahme, } \Rightarrow (1.3)]$$

Der Eigenschaft (1.3) äquivalent ist eine bedingte Form der Monotonie: $x \leq y \Rightarrow g.x \leq g.y$, falls ein Element aus G zwischen x und y liegt. Formal,

$$(A a, x, y : a \in G \wedge x \in \mathbf{Rg} \wedge y \in \mathbf{Rg} \wedge x \leq a \leq y : g.x \leq g.y) \quad [\text{bedingt monoton, } = (1.3)]$$

Für die Ableitung von Fehlerschranken ist der größte Abstand δ zwischen benachbarten Elementen aus \mathbf{Man} bzw. zwischen dem größten Wert aus \mathbf{Man} und R von Interesse, wobei unterstellt wird, daß der Wert 1 in der Menge \mathbf{Man} enthalten ist.

$$1 \in \mathbf{Man} \quad (1.4)$$

$$\delta \triangleq (\max m : m \in \mathbf{Man} : \text{nachf.}(m, \mathbf{Man} \cup \{R\}) - m) \quad (1.5)$$

Dabei ist nachf. die Nachfolgerfunktion: $\text{nachf.}(x, M) \triangleq (\min y : y \in M \wedge x < y : y)$.

In implementierten Gleitkommaarithmetiksystemen sind solche Abstände gleich groß; $\delta = R^{-n}$, wo n die Anzahl von Ziffern im Bruchteil der Mantisse ist.

Über die Eigenschaften der einzelnen gleitkommaarithmetischen Operationen wird angenommen, daß das genaue Ergebnis zuerst effektiv berechnet und anschließend auf einen Wert aus G abgebildet (gerundet) wird. Eingangswerte für die Berechnung sind Gleitkommazahlen (Elemente aus G). D.h.:

$$(A \ a, b : a \in G \wedge b \in G : a \oplus b = g.(a+b)) \quad (1.6)$$

$$(A \ a, b : a \in G \wedge b \in G : a \ominus b = g.(a-b)) \quad (1.7)$$

$$(A \ a, b : a \in G \wedge b \in G : a \otimes b = g.(a \times b)) \quad (1.8)$$

$$(A \ a, b : a \in G \wedge b \in G : a \oslash b = g.(a/b)) \quad (1.9)$$

Man merke, der Wert der Gleitkommaaddition $a \oplus b$ ist nur dann definiert, wenn das Ergebnis der idealen Addition $a+b$ ein Element aus dem Definitionsbereich von g , d.h. aus Rg , ist. Entsprechendes gilt für die anderen gleitkommaarithmetischen Operationen.

Aus (1.6) bzw. (1.8) folgt, daß \oplus und \otimes kommutativ sind. Aus (1.1), (1.6) und (1.7) folgt, daß $a \oplus b = a \oplus (-b)$. (Aus (1.1) folgt, daß $b \in G \Rightarrow -b \in G$.) Das entsprechende für \oslash gilt nicht, da $b \in G$ nicht sicherstellt, daß $(1/b) \in G$.

A3.2 Genauigkeit der Gleitkommadarstellung

Für jedes $e \in \text{Exp}$ (d.h. $e_{\min} \leq e \leq e_{\max}$) und jedes $x \in Rg$ mit $R^e \leq |x| \leq R^{e+1}$ gilt, daß

$$\begin{aligned} & |g.x - x| \leq \delta R^e \\ \Rightarrow & |g.x - x| \leq \delta |x| \end{aligned}$$

Der letzte Ausdruck oben hängt nicht von e ab. Deshalb gilt, daß

$$(A \ x : x \in Rg \wedge R^{e_{\min}} \leq |x| \leq R^{e_{\max}+1} : |g.x - x| \leq \delta |x|) \quad (2.1)$$

Das Intervall $[-R^{e_{\min}}, R^{e_{\min}}]$ von Rg enthält nur drei Elemente aus G ($-R^{e_{\min}}$, 0 und $R^{e_{\min}}$), woraus folgt, daß

$$(A \ x : x \in Rg \wedge 0 \leq |x| \leq R^{e_{\min}} : |g.x - x| \leq R^{e_{\min}}) \quad (2.2)$$

Aus (2.1) und (2.2) folgen weniger scharfe Aussagen über die durch die Gleitkommadarstellung verursachte Ungenauigkeit:

$$(A \ x : x \in Rg \wedge 0 \leq |x| \leq R^{e_{\max}+1} : |g.x - x| \leq \max.(R^{e_{\min}}, \delta |x|)) \quad (2.3)$$

$$(A \ x : x \in Rg \wedge 0 \leq |x| \leq R^{e_{\max}+1} : |g.x - x| \leq R^{e_{\min}} + \delta |x|) \quad (2.4)$$

Falls die Funktion g ihr Argument zum nächsten Wert aus G , d.h. zum ggf. näheren der zwei umliegenden Elemente aus G , rundet und falls Rg hinreichend beschränkt ist, dann halbieren sich die oben angegebenen Fehlerschranken. Eine dafür hinreichende Beschränkung ist es, daß $Rg \subseteq [-(R-\delta/2)R^{e_{\max}}, (R-\delta/2)R^{e_{\max}}]$.

Aus $R^e \leq |x|$ folgt, daß $R^e \leq |g.x|$ für alle $x \in Rg$. Daraus folgt wiederum, daß man $\delta |x|$ durch $\delta |g.x|$ in den oben angegebenen Fehlerschranken ersetzen darf:

$$(A \ x : x \in Rg \wedge 0 \leq |x| \leq R^{e_{\max}+1} : |g.x - x| \leq \max.(R^{e_{\min}}, \delta |g.x|)) \quad (2.5)$$

$$(A \ x : x \in Rg \wedge 0 \leq |x| \leq R^{e_{\max}+1} : |g.x - x| \leq R^{e_{\min}} + \delta |g.x|) \quad (2.6)$$

Meistens ist Rg entweder das Intervall $[-R^{e_{\max}+1}, R^{e_{\max}+1}]$ oder eine Teilmenge davon, in welchen Fällen die Bedingung $0 \leq |x| \leq R^{e_{\max}+1}$ in den Quantifizierungsbereichsbedingungen überflüssig ist.

A3.3 Genauigkeit der Gleitkommaaddition

Aus dem vorhergehenden lassen sich obere Schranken für die Ungenauigkeit des Ergebnisses einer Gleitkommaaddition ableiten:

$$(A \ a, b : a \in G \wedge b \in G \wedge (a+b) \in Rg \wedge 0 \leq |a+b| \leq R^{e_{\max}+1} :$$

$$|(a \oplus b) - (a+b)| \leq \max.(R^{e_{\min}}, \delta |a+b|) \wedge \quad (3.1)$$

$$|(a \otimes b) - (a+b)| \leq \max.(R^{e_{\min}}, \delta |a \otimes b|) \wedge \quad (3.2)$$

$$|(a \oplus b) - (a+b)| \leq \max.(R^{e_{\min}}, \delta(|a|+|b|)) \wedge \quad (3.3)$$

$$|(a \otimes b) - (a+b)| \leq \max.(R^{e_{\min}}, 2\delta|a|, 2\delta|b|) \wedge \quad (3.4)$$

$$|(a \otimes b) - (a+b)| \leq R^{e_{\min}} + \delta |a+b| \quad (3.5)$$

Interessiert man sich für die Summe der reellen Größen x und y , die durch die Gleitkommazahlen a bzw. b maschinell dargestellt werden, sind die folgenden Fehlerschranken, die aus der Dreiecksungleichung, (3.1) und (3.5) folgen, nützlich:

$$(A \ a, b, x, y : a \in G \wedge b \in G \wedge (a+b) \in Rg \wedge 0 \leq |a+b| \leq R^{e_{\max}+1} \wedge x \in R \wedge y \in R :$$

$$|(a \oplus b) - (x+y)| \leq \max.(R^{e_{\min}}, \delta |a+b|) + |a-x| + |b-y| \wedge \quad (3.6)$$

$$|(a \otimes b) - (x+y)| \leq R^{e_{\min}} + \delta |x+y| + (1+\delta)(|a-x|+|b-y|) \quad (3.7)$$

Für die Summenreihe für Gleitkommaaddition wird die Klammerung von links nach rechts definiert, d.h.

$$\bigoplus_{i=1}^m x(i) \triangleq (\dots ((x(1) \oplus x(2)) \oplus x(3)) \oplus \dots) \oplus x(m)$$

Falls $Rg \subseteq [-R^{e_{\max}+1}, R^{e_{\max}+1}]$ (siehe den letzten Absatz im Abschnitt A3.2 oben), falls m eine Ganzzahl mit $m \geq 1$ ist, falls alle $x(i) \in G$ und falls der Wert der Gleitkommasummenreihe definiert ist, dann gelten die folgenden Fehlerschranken:

$$|\bigoplus_{i=1}^m x(i) - \sum_{i=1}^m x(i)| \leq \sum_{i=2}^m \max.(R^{e_{\min}}, \delta |\bigoplus_{j=1}^i x(j)|) \quad (3.8)$$

$$|\bigoplus_{i=1}^m x(i) - \sum_{i=1}^m x(i)| \leq \sum_{i=2}^m (1+\delta)^{m-i} (R^{e_{\min}} + \delta |\sum_{j=1}^i x(j)|) \quad (3.9)$$

$$|\bigoplus_{i=1}^m x(i) - \sum_{i=1}^m x(i)| \leq (m-1) (1+\delta)^{m-2} (R^{e_{\min}} + \delta \sum_{j=1}^m |x(j)|) \quad (3.10)$$

Die Ungleichung (3.10) ist weniger scharf als (3.8) und (3.9), aber wegen der einfacheren Form ggf. für die praktische Anwendung nützlich.

A3.4 Genauigkeit der Gleitkommamultiplikation

Ähnlich wie bei der Gleitkommaaddition lassen sich Fehlerschranken für die Gleitkommamultiplikation ableiten:

$$(A \ a, b : a \in \mathbf{G} \wedge b \in \mathbf{G} \wedge (a \times b) \in \mathbf{Rg} \wedge 0 \leq |a \times b| \leq R^{e_{\max}+1} :$$

$$|(a \otimes b) - (a \times b)| \leq \max.(R^{e_{\min}}, \delta |a \times b|) \wedge \quad (4.1)$$

$$|(a \otimes b) - (a \times b)| \leq \max.(R^{e_{\min}}, \delta |a \otimes b|) \wedge \quad (4.2)$$

$$|(a \otimes b) - (a \times b)| \leq \max.(R^{e_{\min}}, \delta |a| \times |b|) \wedge \quad (4.3)$$

$$|(a \otimes b) - (a \times b)| \leq R^{e_{\min}} + \delta |a| \times |b| \quad (4.4)$$

Interessiert man sich für das Produkt der reellen Größen x und y , die durch die Gleitkommazahlen a bzw. b maschinell dargestellt werden, sind die folgenden Fehlerschranken nützlich:

$$(A \ a, b, x, y : a \in \mathbf{G} \wedge b \in \mathbf{G} \wedge (a \times b) \in \mathbf{Rg} \wedge 0 \leq |a \times b| \leq R^{e_{\max}+1} \wedge x \in \mathbf{R} \wedge y \in \mathbf{R} :$$

$$|(a \otimes b) - (x \times y)| \leq \max.(R^{e_{\min}}, \delta |a \times b|) + |a \times b - x \times y| \wedge \quad (4.5)$$

$$|(a \otimes b) - (x \times y)| \leq \max.(R^{e_{\min}}, \delta |a \otimes b|) + |a \times b - x \times y| \wedge \quad (4.6)$$

$$|(a \otimes b) - (x \times y)| \leq \max.(R^{e_{\min}}, \delta (|x \times y| + |a \times b - x \times y|)) + |a \times b - x \times y| \wedge \quad (4.7)$$

$$|(a \otimes b) - (x \times y)| \leq R^{e_{\min}} + \delta |x \times y| + (1 + \delta) |a \times b - x \times y| \quad (4.8)$$

In Verbindung damit ist die Dreiecksungleichung

$$|a \times b - x \times y| \leq |x \times (b - y)| + |y \times (a - x)| + |(a - x) \times (b - y)|$$

nützlich.

Entsprechendes gilt auch für die Gleitkommadivision \oplus .

A3.5 Rundungseigenarten

Bei der Analyse von gleitkommaarithmetischen Operationen ist Vorsicht bei zunächst wenig einschränkend erscheinenden Annahmen geboten. Manche Rundungsverfahren, auch solche, die ausdrücklich in den ANSI/IEEE Normen 754-1985 [IEEE; 1987 Feb.] und 854-1987 [IEEE; 1987 October 5] vorgesehen sind, stoßen gegen einige naheliegende Annahmen.

Ein Beispiel ist die Annahme, daß $g \cdot -x = -g \cdot x$. Eine derartige Darstellungsfunktion schließt Runden nach oben (in positive Richtung) sowie nach unten (in negative Richtung) aus; sie läßt nur Runden in Richtung 0 oder von 0 weg zu.

Ein zweites Beispiel ist die Annahme, daß $|g \cdot x| \leq g \cdot |x|$ für alle $x \in \mathbf{Rg}$. Zusammen mit der Annahme, g sei monoton nicht absteigend, folgt daraus die Dreiecksungleichung bezüglich der Gleitkommaaddition ($|a \otimes b| \leq |a| \otimes |b|$). Runden nach unten (in negative Richtung) stoßt gegen diese Annahme; es gibt Gegenbeispiele gegen $|g \cdot x| \leq g \cdot |x|$ sowie gegen die Dreiecksungleichung, wenn das Gleitkommaarithmetiksystem nach unten rundet.

Anhang 4. Korrektheit eines rekursiven Unterprogramms

In diesem Anhang wird als Beispiel für die auf rekursive Unterprogramme angewandte Korrektheitsbeweismethode die Korrektheit einer Version des bekannten, von Hoare konzipierten "Quicksort"-Algorithmus bewiesen. Siehe [Hoare; 1989, p. 19-30] (Nachdruck eines 1962 veröffentlichten Artikels), [Foley] und [Baber; 1987, Abschnitt 6.4]. Im untenstehenden Beweis handelt es sich um die vollständige Korrektheit, also auch um die Terminierung der Rekursion. Darüber hinaus enthält der Korrektheitsatz eine obere Schranke für die maximale Länge einer während der rekursiven Ausführung des Unterprogramms entstehenden Datenumgebung, die als Maßstab für die maximale Speicherbelegung bzw. für den Speicherbedarf angesehen werden kann. Der Beweis sieht eine endliche Menge für die Zahlendarstellung im Programm vor. Entsprechend schließt die strikte Vorbedingung Überlauf bei der Auswertung numerischer Ausdrücke aus.

Im untenstehenden Beweis wird sowohl von den bereits erläuterten Beweistechniken der (nicht rekursiven) Programmkorrektheitsbeweismethode als auch von den üblichen Beweistechniken der Mathematik für einen Beweis durch Induktion Gebrauch gemacht. Zusätzliche Besonderheiten treten nicht auf.

Gegeben sei das rekursive Unterprogramm Sort:

```

if il < ig
then  call Einteilen
      declare (ig, Ze, gl-1); call Sort; release ig
      declare (il, Ze, gr+1); call Sort; release il
      release gl; release gr
endif

```

wobei \mathbf{Ze} eine nicht leere Menge aufeinanderfolgender Ganzzahlen ist. \mathbf{Ze} darf — und in der Regel wird — eine endliche Menge sein.

Der then-Teil des Unterprogramms oben könnte z.B. vom folgenden Programmsegment mit Aufrufen mit formaler Parameterübergabe abgeleitet worden sein, wobei gl und gr in Sort lokal vereinbarte Variablen sind.

```

call Einteilen(il, gl, gr, ig)           [gl und gr sind Ergebnisvariablen]
call Sort(il, gl-1)
call Sort(gr+1, ig)

```

Dabei sind die überflüssigen Vereinbarungen $\text{declare}(il, \mathbf{Ze}, il)$ und $\text{declare}(ig, \mathbf{Ze}, ig)$ sowie die dazugehörigen release -Anweisungen weggelassen worden.

Unterprogramm Einteilen: Gegeben sei auch das (nicht rekursive) Unterprogramm Einteilen (vgl. Anhang 2 sowie das Programmsegment "Dutch National Flag" in [Dijkstra; 1976, Chapter 14]) mit der strikten Vorbedingung

$$\text{Veint} \triangleq \text{il} \in \mathbf{Ze} \text{ und } \text{ige} \in \mathbf{Ze} \text{ und } \text{il} \leq \text{ig} \text{ und } \bigcup_{i=\text{il}}^{\text{ig}} \text{Menge. "x(i)} = \mathbf{M}$$

wo \mathbf{M} eine nicht leere linear geordnete Menge ist, und der Nachbedingung

Peint \triangleq $gl \in Ze$ und $gr \in Ze$ und $il \leq gl \leq gr \leq ig$
 und $\underset{i=il}{\text{und}}^{gl-1} x(i) < x(gl)$
 und $\underset{i=gl}{\text{und}}^{gr} x(i) = x(gl)$
 und $\underset{i=gr+1}{\text{und}}^{ig} x(i) > x(gl)$
 und $((\&_{i=il}^{ig} [x(i)]) \text{ Perm } (\&_{i=il}^{ig} [x'(i)]))$

Ferner gilt für alle $d \in Veint$, daß $(call \text{ Einteilen}).d = [(gl, Ze, .), (gr, Ze, .)] \& d$ bis auf die Werte von $x(il), \dots, x(ig)$.

Während der Ausführung von Einteilen wird noch eine Programmvariable vereinbart (deklariert) und vor dem Rücksprung gelöscht, d.h., während der Ausführung von Einteilen wird entsprechend viel zusätzlicher Speicherplatz benötigt.

Korrektheitssatz für Sort: Sei

$V_{\text{sort}} \triangleq il-1 \in Ze$ und $ig+1 \in Ze$ und $0 \leq ig-il+1$ [$\{il-1, \dots, ig+1\} \subseteq Ze$]
 und $\underset{i=il}{\text{und}}^{ig}$ Menge. "x(i)" = M
 $P_{\text{sort}} \triangleq \underset{i=il}{\text{und}}^{ig-1} x(i) \leq x(i+1)$
 und $((\&_{i=il}^{ig} [x(i)]) \text{ Perm } (\&_{i=il}^{ig} [x'(i)]))$

wo M eine nicht leere linear geordnete Menge ist. Dann gilt, daß

{Vsort} call Sort {Psort} strikt [KS1]
 (A d : $d \in V_{\text{sort}}$: $(call \text{ Sort}).d = d$ bis auf die Werte von $x(il), \dots, x(ig)$) [KS2]

sowie, daß die maximale Länge einer während der Ausführung von call Sort auf die Datenumgebung d (während der Anwendung der Funktion (call Sort) auf d) entstehenden Datenumgebung höchstens

$Länge.d + 3 * \max((ig-il).d, 0)$ [KS3]

beträgt, wo Länge.d die Anzahl der Programmvariablen in der Datenumgebung d ist. (Länge ist eine Funktion von \mathbf{D} nach \mathbf{N}_0).

Man merke, die Werte von il und ig dürfen die Extremwerte der Menge Ze nicht annehmen, siehe die Vorbedingung Vsort. Diese Einschränkung schließt Überlauf bei der Auswertung der in den declare-Anweisungen vorkommenden Ausdrücke aus.

Beweis des Korrektheitssatzes: Der Korrektheitssatz für Sort wird durch Induktion über den anfänglichen Wert des Ausdrucks (ig-il) bewiesen. (Die Anzahl der zu sortierenden Elemente beträgt ig-il+1.) Die Basis für die Induktion bildet der Fall ig-il ≤ 0. Im Induktionsschritt wird angenommen, daß der Korrektheitssatz für alle Werte von ig-il mit ig-il < k gilt (wobei 0 < k), und bewiesen, daß er für ig-il = k gilt. Der Beweis wird in Beweise für die Induktionsbasis und den Induktionsschritt sowie für KS1, KS2 und KS3 aufgeteilt. In den einfacheren Fällen wird der Beweis nur angedeutet bzw. nur eine Skizze davon angegeben.

KS1 wird gemäß der Beweisregel IFS gelten, falls

{Vsort und $il < ig$ } [in "Induktionsschritt, KS1" unten bewiesen]
 call Einteilen
 declare (ig, Ze, gl-1); call Sort; release ig
 declare (il, Ze, gr+1); call Sort; release il
 release gl; release gr
 {Psort} strikt
 {Vsort und $il \leq ig$ } [in "Basis, KS1" unten bewiesen]
 Null-Anweisung
 {Psort} strikt
 $V_{\text{sort}} \Rightarrow (il < ig) \in \{\text{falsch}, \text{wahr}\}$

Aus der Vorbedingung Vsort und den angegebenen Eigenschaften von Ze folgt, daß $il \in Ze$ und $ig \in Ze$. Es wird unterstellt, daß diese Bedingung sicherstellt, daß der Boolesche Wert der if-Bedingung ($il < ig$) vom ausführenden Programmiersprachensystem ermittelt wird.

Beweis, Basis, KS1: Weil $ig-il \leq 0$ gilt (Induktionsbasis), ist "call Sort" äquivalent zum (leeren) else-Teil der if-Anweisung, also zur Null-Anweisung, für die jede Vorbedingung strikt ist (vgl. die Beweisregel NS). Die und-Reihe in der Nachbedingung ist leer. Die Werte der Feldvariablen x(.) werden nicht verändert, so daß der "Perm"-Term in der Nachbedingung wahr ist. Damit ist die gesamte Nachbedingung — und wiederum auch KS1 — wahr.

Beweis, Basis, KS2: $(call \text{ Sort}).d = null.d \triangleq d$. (Siehe oben.)

Beweis, Basis, KS3: Während der Ausführung der leeren else-Teil der if-Anweisung wird keine neue Programmvariable deklariert, weshalb die (maximale) Länge einer währenddessen entstehenden Datenumgebung unverändert Länge.d bleibt. Wegen $ig-il \leq 0$ gilt, daß $Länge.d = Länge.d + 3 * \max((ig-il).d, 0)$; also gilt KS3.

Beweis, Induktionsschritt, KS1: Weil $0 < ig-il$ gilt (siehe Bemerkungen zum Induktionsschritt im Absatz "Beweis des Korrektheitssatzes" oben), ist "call Sort" äquivalent zum then-Teil der if-Anweisung. Die dementsprechend zu beweisende Aussage wurde im Abschnitt "Beweis des Korrektheitssatzes" oben bereits angegeben. Die verschiedenen Zwischenbedingungen zeigt das folgende Beweisschema, das auch die in der Nachbedingung sowie in den Zwischenbedingungen enthaltenen Bezüge auf die ursprünglichen Werte bestimmter Variablen ausdrücklich berücksichtigt. Dabei ist jede Vorbedingung strikt, wie unten bewiesen wird. Der Term $ig-il < ig'-il'$ in den mit * markierten Zwischenbedingungen ist wegen der Annahme im Induktionsschritt erforderlich ($k = ig'-il'$). Die Wahrheit dieses Terms bestätigt, daß die untergeordneten (rekursiven) Aufrufe auf Sort mit echt kleineren Werten von ig-il als beim aktuellen Aufruf erfolgen. Dadurch wird mathematisch für die Terminierung der Rekursion (Erreichung des Basisfalls) gesorgt.

{Vsort und $il < ig$ und $il = il'$ und $ig = ig'$ und $\underset{i=il}{\text{und}}^{ig} x(i) = x'(i)$ }
 call Einteilen
 {Peint und Vsort und $il = il'$ und $ig = ig'$ }
 declare (ig, Ze, gl-1)
 {Peint $\underset{ig'}{ig}$ und Vsort $\underset{ig'}{ig}$ und $il = il'$ und $ig = gl-1$ und $ig-il < ig'-il'$ } [*]
 call Sort
 {Psa und Vsort $\underset{ig'}{ig}$ und $il = il'$ und $ig = gl-1$ }

```

release ig
{Psa und Vsortigig, und il=il' und ig=ig'}
  declare (il, Ze, gr+1)
  {Psa und Vsortilil, und il=gr+1 und ig=ig' und ig-il<ig'-il'}      [*]
  call Sort
  {Psortilil, und glεZe und grεZe und il=gr+1 und ig=ig'}
  release il
  {Psortilil, und glεZe und grεZe und il=il' und ig=ig'}
  release gl; release gr
  {Psort und il=il' und ig=ig'} strikt

```

wobei

$$\begin{aligned}
\text{Psa} \triangleq & \text{gl}\epsilon\text{Ze und gr}\epsilon\text{Ze und il}'\leq\text{gl}\leq\text{gr}\leq\text{ig}' \\
& \text{und}_{i=\text{il}}^{\text{gl}-1} x(i)\leq x(i+1) \\
& \text{und}_{i=\text{gl}}^{\text{gr}} x(i)=x(\text{gl}) \\
& \text{und}_{i=\text{gr}+1}^{\text{ig}'} x(i)>x(\text{gl}) \\
& \text{und } ((\&_{i=\text{il}}^{\text{ig}'} [x(i)]) \text{ Perm } (\&_{i=\text{il}}^{\text{ig}'} [x'(i)]))
\end{aligned}$$

Der Korrektheitsatzteil KS1 wird gemäß den Beweisregeln FS, DS, RS und DC3 gelten und die Voraussetzung für die Induktionsannahme wird eingehalten, falls die folgenden Aussagen gelten:

```

{Vsort und il<ig und il=il' und ig=ig' und  $\text{und}_{i=\text{il}}^{\text{ig}} x(i)=x'(i)$ }      [KS1.1]
  call Einteilen
  {Peint und Vsort und il=il' und ig=ig'} strikt

{Peint und Vsort und il=il' und ig=ig'}      [KS1.2]
  declare (ig, Ze, gl-1)
  {Peintigig, und Vsortigig, und il=il' und ig=gl-1}

Peint und Vsort und il=il' und ig=ig' ⇒ gl-1εZe      [KS1.3]

(Peintigig, und Vsortigig, und il=il' und ig=gl-1)      [KS1.4]
= (Peintigig, und Vsortigig, und il=il' und ig=gl-1 und ig-il<ig'-il')

{Peintigig, und Vsortigig, und il=il' und ig=gl-1 und ig-il<ig'-il'}      [KS1.5]
  call Sort
  {Psa und Vsortigig, und il=il' und ig=gl-1} strikt

```

```

{Psa und Vsortigig, und il=il' und ig=gl-1}      [KS1.6]
  release ig
  {Psa und Vsortigig, und il=il'}

Psa und Vsortigig, und il=il' und ig=gl-1 ⇒ Menge.“ig” ≠ ∅      [KS1.7]

{Peint und Vsort und il=il' und ig=ig'}      [KS1.8]
  declare (ig, Ze, gl-1)
  call Sort
  release ig
  {ig=ig'}

{Psa und Vsortigig, und il=il' und ig=ig'}      [KS1.9]
  declare (il, Ze, gr+1)
  {Psa und Vsortilil, und il=gr+1 und ig=ig'}

Psa und Vsortigig, und il=il' und ig=ig' ⇒ gr+1εZe      [KS1.10]

(Psa und Vsortilil, und il=gr+1 und ig=ig')      [KS1.11]
= (Psa und Vsortilil, und il=gr+1 und ig=ig' und ig-il<ig'-il')

{Psa und Vsortilil, und il=gr+1 und ig=ig' und ig-il<ig'-il'}      [KS1.12]
  call Sort
  {Psortilil, und glεZe und grεZe und il=gr+1 und ig=ig'} strikt

{Psortilil, und glεZe und grεZe und il=gr+1 und ig=ig'}      [KS1.13]
  release il
  {Psortilil, und glεZe und grεZe und ig=ig'}

Psortilil, und glεZe und grεZe und il=gr+1 und ig=ig'      [KS1.14]
⇒ Menge.“il” ≠ ∅

{Psa und Vsortigig, und il=il' und ig=ig'}      [KS1.15]
  declare (il, Ze, gr+1)
  call Sort
  release il
  {il=il'}

{Psortilil, und glεZe und grεZe und il=il' und ig=ig'}      [KS1.16]
  release gl; release gr
  {Psort und il=il' und ig=ig'}

```

Psort_{il}^{ig} , **und** gleZe **und** greZe **und** $il=il'$ **und** $ig=ig'$ [KS1.17]
 \Rightarrow Menge."gl" $\neq \emptyset$ **und** Menge."gr" $\neq \emptyset$

Die Beweise für viele der Aussagen oben sind ggf. nach Anwendung der jeweils geeigneten Beweisregel einfache bis sogar triviale Übungen in der Booleschen Algebra, z.B. KS1.2, KS1.3, KS1.4, KS1.6, KS1.7, KS1.9, KS1.10, KS1.11, KS1.13, KS1.14, KS1.16 und KS1.17. Übrig bleiben KS1.1, KS1.5, KS1.8, KS1.12 und KS1.15.

KS1.8 kann durch unmittelbare Anwendung der Definitionen der declare- und release-Anweisungen sowie der Induktionsannahme KS2 für den Aufruf auf Sort bewiesen werden. Danach gilt, daß (declare (ig, Ze, gl-1); call Sort; release ig).d=d bis auf die Werte gewisser Feldvariablen x(.). Die Ausführung dieser Folge von Anweisungen verändert also den Wert der Programmvariable ig nicht. Damit gilt KS1.8. KS1.15 wird auf die gleiche Weise bewiesen. Vgl. den Beweis für den Induktionsschritt, KS2 unten. Zu beweisen bleiben nur noch KS1.1, KS1.5 und KS1.12.

KS1.1 wird gemäß Beweisregel U3 gelten, falls die folgenden Aussagen gelten:

{Vsort **und** $il=il'$ **und** $ig=ig'$ } [KS1.1.1]

call Einteilen

{Vsort **und** $il=il'$ **und** $ig=ig'$ }

Vsort **und** $il < ig$ **und** $il=il'$ **und** $ig=ig'$ **und** $\text{und}_{i=il}^{ig} x(i)=x'(i)$ [KS1.1.2]

\Rightarrow Veint $\text{und}_{i=il}^{ig} x(i)=x'(i)$

{Veint **und** $\text{und}_{i=il}^{ig} x(i)=x'(i)$ } call Einteilen {Point} strikt [KS1.1.3]

KS1.1.3 gilt nach der Spezifikation von Einteilen. KS1.1.1 gilt, weil die Ausführung von Einteilen die Wahrheit der Bedingung {Vsort **und** $il=il'$ **und** $ig=ig'$ } nicht beeinflusst. Einteilen kann zwar die Werte gewisser Feldvariablen x(.) verändern, jedoch nicht die ihnen zugeordneten Mengen, siehe die Spezifikation von Einteilen sowie die Definition der Bedingung Vsort. KS1.1.2 läßt sich auf einfache Weise durch Boolesche Algebra zeigen, wobei die spezifizierte Eigenschaft von Ze gebraucht wird, daß Ze eine Menge aufeinanderfolgender Ganzzahlen ist.

Zu beweisen bleiben nur noch KS1.5 und KS1.12.

Der Beweis von KS1.5, in dem es sich um den ersten untergeordneten Aufruf auf Sort handelt, basiert auf der Beweisregel U3. Die spezifizierte Nachbedingung von Sort bezieht sich auf die ursprünglichen (d.h. gleich vor dem fraglichen untergeordneten Aufruf auf Sort herrschenden) Werte der Feldvariablen x(il), ... x(ig). Diese Werte werden unten durch das Zeichen " gekennzeichnet, da das Zeichen ' bereits für die am Anfang des übergeordneten Aufrufs auf Sort geltenden Variablenwerte verwendet wird. Damit ist die zu beweisende Aussage wie folgt:

{Point $_{ig}^{ig}$, **und** Vsort $_{ig}^{ig}$, **und** $il=il'$ **und** $ig=gl-1$

$\text{und}_{i=il}^{ig} x(i)=x''(i)$ **und** $ig-il < ig'-il'$ }

call Sort

{Psa **und** Vsort $_{ig}^{ig}$, **und** $il=il'$ **und** $ig=gl-1$ } strikt

Gemäß den Beweisregeln U3 und B1 wird diese Aussage gelten, falls

Point $_{ig}^{ig}$, **und** Vsort $_{ig}^{ig}$, **und** $il=il'$ **und** $ig=gl-1$ [KS1.5.1]

und $\text{und}_{i=il}^{ig} x(i)=x''(i)$ **und** $ig-il < ig'-il'$

\Rightarrow BS1

{BS1} call Sort {BS1} [KS1.5.2]

Point $_{ig}^{ig}$, **und** Vsort $_{ig}^{ig}$, **und** $il=il'$ **und** $ig=gl-1$ [KS1.5.3]

und $\text{und}_{i=il}^{ig} x(i)=x''(i)$ **und** $ig-il < ig'-il'$

\Rightarrow Vsort **und** $\text{und}_{i=il}^{ig} x(i)=x''(i)$ **und** $ig-il < ig'-il'$

{Vsort **und** $\text{und}_{i=il}^{ig} x(i)=x''(i)$ **und** $ig-il < ig'-il'$ } [KS1.5.4]

call Sort

{**und** $\text{und}_{i=il}^{ig-1} x(i) \leq x(i+1)$ **und** (($\&_{i=il}^{ig} [x(i)]$) Perm ($\&_{i=il}^{ig} [x''(i)]$))} strikt

BS1 **und** $\text{und}_{i=il}^{ig-1} x(i) \leq x(i+1)$ [KS1.5.5]

und (($\&_{i=il}^{ig} [x(i)]$) Perm ($\&_{i=il}^{ig} [x''(i)]$))

\Rightarrow Psa **und** Vsort $_{ig}^{ig}$, **und** $il=il'$ **und** $ig=gl-1$

wobei BS1 aus Termen besteht, deren Wahrheit von der Ausführung von Sort nicht beeinflusst wird:

BS1 \triangleq Vsort $_{ig}^{ig}$, **und** $il=il'$ **und** $ig=gl-1$

und gleZe **und** greZe **und** $il' \leq gl \leq gr \leq ig'$

und $\text{und}_{i=il'}^{gl-1} x''(i) < x(gl)$

und $\text{und}_{i=gl}^{gr} x(i) = x(gl)$

und $\text{und}_{i=gr+1}^{ig'} x(i) > x(gl)$

und (($\&_{i=il'}^{gl-1} [x''(i)]$) $\&_{i=gl}^{ig'}$ [$x(i)$] Perm ($\&_{i=il'}^{ig'} [x''(i)]$))

KS1.5.4 gilt nach der Spezifikation von Sort (KS1). KS1.5.2 gilt, weil die Ausführung von Sort die Wahrheit der Bedingung BS1 nicht beeinflusst. Die in BS1 vorkommenden Bezüge auf Werte von Feldvariablen x(.) weisen Indexwerte außerhalb des Bereichs il, ... ig auf, siehe die Spezifikation von Sort (KS2). Die anderen Aussagen KS1.5.1, KS1.5.3 und KS1.5.5 können algebraisch verifiziert werden. Damit gilt KS1.5.

Zu beweisen bleibt nur noch die Aussage KS1.12, die auf die gleiche Weise wie KS1.5 oben bewiesen wird. Nach Berücksichtigung der in der Nachbedingung von Sort enthaltenen Bezüge auf die ursprünglichen Werte von Feldvariablen x(.) ist die zu beweisende Aussage wie folgt:

$\{Psa \text{ und } Vsort_{il}^{il}, \text{ und } il=gr+1 \text{ und } ig=ig'\}$
 $\text{und}_{i=il}^{ig} x(i)=x''(i) \text{ und } ig-il < ig'-il'\}$
 call Sort
 $\{Psort_{il}^{il}, \text{ und } gl\epsilon Ze \text{ und } gr\epsilon Ze \text{ und } il=gr+1 \text{ und } ig=ig'\}$ strikt

Gemäß den Beweisregeln U3 und B1 wird diese Aussage gelten, falls

$Psa \text{ und } Vsort_{il}^{il}, \text{ und } il=gr+1 \text{ und } ig=ig'$ [KS1.12.1]

$\text{und}_{i=il}^{ig} x(i)=x''(i) \text{ und } ig-il < ig'-il'$
 $\Rightarrow BS2$

$\{BS2\}$ call Sort $\{BS2\}$ [KS1.12.2]

$Psa \text{ und } Vsort_{il}^{il}, \text{ und } il=gr+1 \text{ und } ig=ig'$ [KS1.12.3]

$\text{und}_{i=il}^{ig} x(i)=x''(i) \text{ und } ig-il < ig'-il'$
 $\Rightarrow Vsort \text{ und}_{i=il}^{ig} x(i)=x''(i) \text{ und } ig-il < ig'-il'$

$\{Vsort \text{ und}_{i=il}^{ig} x(i)=x''(i) \text{ und } ig-il < ig'-il'\}$ [KS1.12.4]

call Sort
 $\{\text{und}_{i=il}^{ig-1} x(i) \leq x(i+1) \text{ und } ((\&_{i=il}^{ig} [x(i)]) \text{ Perm } (\&_{i=il}^{ig} [x''(i)]))\}$ strikt

$BS2 \text{ und}_{i=il}^{ig-1} x(i) \leq x(i+1)$ [KS1.12.5]

$\text{und } ((\&_{i=il}^{ig} [x(i)]) \text{ Perm } (\&_{i=il}^{ig} [x''(i)]))$

$\Rightarrow Psort_{il}^{il}, \text{ und } gl\epsilon Ze \text{ und } gr\epsilon Ze \text{ und } il=gr+1 \text{ und } ig=ig'$

wobei BS2 aus Termen besteht, deren Wahrheit von der Ausführung von Sort nicht beeinflußt wird:

$BS2 \triangleq Vsort_{il}^{il}, \text{ und } il=gr+1 \text{ und } ig=ig'$
 $\text{und } gl\epsilon Ze \text{ und } gr\epsilon Ze \text{ und } il' \leq gl \leq gr \leq ig'$

$\text{und}_{i=il}^{gl-1} x(i) \leq x(i+1)$

$\text{und}_{i=gl}^{gr} x(i) = x(gl)$

$\text{und}_{i=gr+1}^{ig'} x''(i) > x(gr)$

$\text{und } ((\&_{i=il}^{gr} [x(i)] \&_{i=gr+1}^{ig'} [x''(i)]) \text{ Perm } (\&_{i=il}^{ig'} [x''(i)]))$

KS1.12.4 gilt nach der Spezifikation von Sort (KS1). KS1.12.2 gilt, weil die Ausführung von Sort die Wahrheit der Bedingung BS2 nicht beeinflußt. Die in BS2 vorkommenden Bezüge auf Werte von Feldvariablen $x(\cdot)$ weisen Indexwerte außerhalb des Bereichs il, \dots, ig auf, siehe die Spezifikation von Sort (KS2). Die anderen Aussagen KS1.12.1, KS1.12.3 und KS1.12.5 können algebraisch verifiziert werden. Damit gelten KS1.12 und KS1.

Beweis, Induktionsschritt, KS2: Aus den Definitionen der einzelnen Programmanweisungen, der Spezifikation von Einteilen sowie der Induktionsannahme über Sort folgt, daß die Datenumgebungen an den verschiedenen Stellen des Unterprogramms die folgenden Strukturen bzw. Eigenschaften aufweisen. Die Datenumgebung, auf die das Unterprogramm angewendet wird, wird unten mit "d" bezeichnet. Alle untenstehenden Angaben über die Datenumgebungen (außer der ersten) gelten mit der Einschränkung "bis auf die Werte von $x(il.d), \dots, x(ig.d)$ ".

d

call Einteilen

$[(gl, Ze, \cdot), (gr, Ze, \cdot)] \& d$

declare $(ig, Ze, gl-1)$

$[(ig, Ze, \cdot), (gl, Ze, \cdot), (gr, Ze, \cdot)] \& d$

call Sort [Induktionsannahme, KS2]

$[(ig, Ze, \cdot), (gl, Ze, \cdot), (gr, Ze, \cdot)] \& d$

release ig

$[(gl, Ze, \cdot), (gr, Ze, \cdot)] \& d$

declare $(il, Ze, gr+1)$

$[(il, Ze, \cdot), (gl, Ze, \cdot), (gr, Ze, \cdot)] \& d$

call Sort [Induktionsannahme, KS2]

$[(il, Ze, \cdot), (gl, Ze, \cdot), (gr, Ze, \cdot)] \& d$

release il

$[(gl, Ze, \cdot), (gr, Ze, \cdot)] \& d$

release gl; release gr

d

Die Ergebnisdatenumgebung ist also gleich der anfänglichen Datenumgebung d bis auf die Werte von $x(il), \dots, x(ig)$. Damit gilt KS2 auch für den aktuellen (übergeordneten) Aufruf auf Sort.

Weil die zwei untergeordneten Aufrufe auf Sort mit Werten von il und ig erfolgen, die von $il.d (=il')$ und $ig.d (=ig')$ abweichen, muß man zeigen, daß die Indexwerte der ggf. veränderten Feldvariablen innerhalb des Bereichs il', \dots, ig' liegen. Aus den Vorbedingungen für die untergeordneten Aufrufe auf Sort läßt sich in beiden Fällen zeigen, daß entweder $il' \leq il \leq ig \leq ig'$ oder $ig = il - 1$. Im ersten Falle werden alle von Sort veränderten Feldwerte im Indexbereich il', \dots, ig' liegen. Im zweiten Falle ist der fragliche Bereich leer und Sort wird keinen Wert verändern.

Beweis, Induktionsschritt, KS3: Man nenne die anfängliche Datenumgebung d; sie enthält Länge.d Programmvariablen. Während der Ausführung von "call Einteilen" entstehen Datenumgebungen mit höchstens Länge.d+3 Programmvariablen, siehe die Spezifikation von Einteilen. Nach Ausführung des Unterprogramms Einteilen ist die Datenumgebung Länge.d+2 Programmvariablen lang, d.h. Länge.((call Einteilen).d) = Länge.d+2. Gleich vor der Ausführung des ersten Aufrufs auf Sort ist die (unten d1 genannte) Datenumgebung Länge.d+3 Programmvariablen lang, d.h. Länge.d1=Länge.d+3. (Vgl. den Abschnitt "Induktionsschritt, KS2" oben.)

Man bezeichne die maximale Länge einer während der Ausführung des ersten Aufrufs auf Sort entstehenden Datenumgebung mit ML1. Gemäß Induktionsannahme gilt, daß

$$ML1 \leq \text{Länge.d1} + 3 * \max((ig-il).d1, 0)$$

Aber in $d1$ gilt, daß $ig-il < ig'-il'$ (siehe die Vorbedingung in KS1.5 oben). D.h.

$$\begin{aligned}
 & (ig-il).d1 < (ig-il).d && [0 < (ig-il).d \text{ im Induktionsschritt}] \\
 = & \max((ig-il).d1, 0) < (ig-il).d && [0 < (ig-il).d \text{ im Induktionsschritt}] \\
 = & \max((ig-il).d1, 0) < \max((ig-il).d, 0) && [\text{alle Werte Ganzzahlen}] \\
 = & \max((ig-il).d1, 0) \leq \max((ig-il).d, 0) - 1 \\
 = & 3 * \max((ig-il).d1, 0) \leq 3 * \max((ig-il).d, 0) - 3 && [\text{Länge}.d1 = \text{Länge}.d + 3, \text{ siehe oben}] \\
 = & \text{Länge}.d1 + 3 * \max((ig-il).d1, 0) \leq \text{Länge}.d + 3 * \max((ig-il).d, 0) \\
 \Rightarrow & ML1 \leq \text{Länge}.d + 3 * \max((ig-il).d, 0) && [\text{siehe Ungleichung für ML1 oben}]
 \end{aligned}$$

Die maximale Länge einer während der Ausführung des zweiten Aufrufs auf Sort entstehenden Datenumgebung kann auf die gleiche Weise ermittelt werden; sie hat die gleiche obere Schranke wie ML1 oben.

Eine obere Schranke für die maximale Länge einer während der Ausführung des aktuellen Aufrufs auf Sort entstehenden Datenumgebung ist deshalb der größere Wert von $\text{Länge}.d + 3$ (für die Ausführung von Einteilen) und $\text{Länge}.d + 3 * \max((ig-il).d, 0)$ (für die Ausführung eines untergeordneten Aufrufs auf Sort). Weil $0 < (ig-il).d$ gilt, woraus $1 \leq (ig-il).d$ folgt, ist der zweite Ausdruck immer mindestens so groß wie der erste.

Die in KS3 angegebene obere Schranke für die maximale Länge einer während der Ausführung eines Aufrufs auf Sort entstehenden Datenumgebung gilt also auch für den übergeordneten Aufruf auf Sort.

Damit ist der Korrektheitsatz für das rekursive Unterprogramm Sort bewiesen.

Literatur- und Quellenhinweise

In den Literaturhinweisen werden die folgenden Abkürzungen verwendet:

ACM = Association for Computing Machinery

BCS = The British Computer Society

CACM = *Communications of the ACM*

EWD = semi-publicly distributed series of articles by Prof. Dr. Edsger W. Dijkstra

IEEE = Institute of Electrical and Electronics Engineers

IEEE TSE = *IEEE Transactions on Software Engineering*

JACM = *Journal of the ACM*

LNCS = Lecture Notes in Computer Science

SCSC Newsletter = *Safety Systems, The Safety-Critical Systems Club Newsletter*

TOPLAS = *ACM Transactions on Programming Languages and Systems*

Literaturhinweise

- Abrial, J.-R.; "On constructing large software systems", in van Leeuwen, Jan (ed.); *Algorithms, Software, Architecture: Information Processing 92 — Proceedings of the IFIP 12th World Computer Congress, Madrid, Spain, 7-11 September 1992, Volume 1*, p. 103-112, North-Holland, Amsterdam, 1992.
- Acklin, Charlotte; Esser, Manuel; *Die elektronischen Eierköpfe — Macht und Macher künstlicher Intelligenz*, Drehbuch für eine Fernsehsendung, Westdeutscher Rundfunk — Fernsehen — Wirtschafts- und Sozialpolitik, Köln, 1990 Juli 5.
- Aho, Alfred V.; Hopcroft, John E.; Ullman, Jeffrey D.; *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Massachusetts, 1974.
- Alagić, Suad; Arbib, Michael A.; *The Design of Well-Structured and Correct Programs*, Springer-Verlag, New York, 1978.
- Alencar, Antonio J.; Goguen, Joseph A.; "OOZE: An Object Oriented Z Environment", in America, Pierre (ed.); *ECOOP '91, European Conference on Object-Oriented Programming, Geneva, Switzerland, July 1991, Proceedings*, LNCS 512, p. 180-199, Springer-Verlag, Berlin, 1991.
- Allen, Robert; Garland, David; "A Formal Approach to Software Architectures", in van Leeuwen, Jan (ed.); *Algorithms, Software, Architecture: Information Processing 92 — Proceedings of the IFIP 12th World Computer Congress, Madrid, Spain, 7-11 September 1992, Volume 1*, p. 134-141, North-Holland, Amsterdam, 1992.
- America, Pierre; "Formal techniques for parallel object-oriented languages", in Baeten, J.C.M.; Groote, J.F. (eds.); *CONCUR '91. 2nd International Conference on Concurrency Theory Proceedings, Amsterdam, Netherlands, 26-29 Aug. 1991*, p. 1-17, Springer-Verlag, Berlin, 1991.
- America, Pierre H.M.; de Boer, Frank S.; *A Proof Theory for a Sequential Version of POOL*, Report CS-R9118, Centre for Mathematics and Computer Science (CWI), Amsterdam, 1991 March.

- American Radio Relay League; *The Radio Amateur's Handbook*, American Radio Relay League, West Hartford, Conn., 1962.
- Anderson, Robert B.; *Proving Programs Correct*, John Wiley & Sons, New York, 1979.
- Anderson, Stuart; Cleland, George; "Adopting Mathematically-based Methods for Safety-Critical Systems Production", *SCSC Newsletter*, Vol. 1, No. 2, p. 6, 1992 Jan.
- Anderson, Tom; Barrett, Peter A.; Halliwell, Dave N.; Moulding, Michael R.; "Software Fault Tolerance: An Evaluation", *IEEE TSE*, Vol. SE-11, No. 12, p. 1502-1510, 1985 Dec.
- Andrews, Gregory R.; *Concurrent Programming: Principles and Practice*, Benjamin/Cummings, Redwood City, California, 1991.
- Apt, Krzysztof R.; "Ten Years of Hoare's Logic: A Survey — Part I", *TOPLAS*, Vol. 3, No. 4, p. 431-483, 1981 Oct.
- Apt, Krzysztof R.; Olderog, Ernst-Rüdiger; *Verification of Sequential and Concurrent Programs*, Springer-Verlag, New York, 1991.
- Arbib, Michael A.; *Theories of Abstract Automata*, Prentice-Hall, Englewood Cliffs, N.J., 1969.
- Arbib, Michael A.; Alagić, Suad; "Proof Rules for Gotos", *Acta Informatica*, Vol. 11, p. 139-148, 1979.
- Austin, Stephen; Parkin, Graeme I.; *Formal Methods: A Survey*, Division of Information Technology and Computing, National Physical Laboratory, Teddington, Middlesex, U.K., 1993 March 31.
- Avizienis, Algirdas; "The N-Version Approach to Fault-Tolerant Software", *IEEE TSE*, Vol. SE-11, No. 12, p. 1491-1501, 1985 Dec.
- Ayres, D.M.; "Documenting the software lifecycle using formal methods", in Ross, M.; Brebbia, C.A.; Staples, G.; Stapleton, J. (eds.); *Software Quality Management*, (Proceedings of the SQM '93 Conference, Southampton), p. 647-661, Computational Mechanics Publications, Southampton, 1993.
- Baber, Robert L.; vertrauliche firmen-interne Arbeitspapiere und Berichte für mehrere Beratungsklienten, 1978-1992.
- Baber, Robert L.; "Software Reflected: the Land of Moc", *Systems, Objectives, Solutions (SOS)*, Vol. 1, No. 3, p. 105-117, North-Holland, Amsterdam, 1981 Aug.
- Baber, Robert L.; *Software Reflected: The Socially Responsible Programming of Our Computers*, North-Holland, Amsterdam, 1982.
- Baber, Robert L.; "De ontwikkeling van programmatuur gisteren, vandaag en morgen: Wetenschap of broddelwerk?", (Dutch translation by Timman, T.E.), *1 & 1 Kwartaalreeks over Informatie en Informatiebeleid*, No. 4, p. 60-70, 1983.
- Baber, Robert L.; "Der Kommentar: Informatikausbildung in der Bundesrepublik Deutschland: Der zukünftige Preis des gegenwärtigen Nichthandelns", *Informatik-Spektrum*, Band 6, Heft 1, S. 36, 1983 Feb.
- Baber, Robert L.; "Software Development: Science or Patchwork?", *CWI Newsletter*, No. 2, p. 18-34, published by the Centre for Mathematics and Computer Science, Amsterdam, 1984 March.
- Baber, Robert L.; "Software Development: Science, Craft or Racket?", *Data Processing*, Vol. 26, No. 10, p. 7-10, 1984 Dec.
- Baber, Robert L.; *Die Konstruktion fehlerfreier Software*, Unterlagen für Seminarteilnehmer, mehrere Ausgaben in deutscher, englischer und chinesischer Sprache, 1985-1993.

- Baber, Robert L.; "Softwareentwicklung gestern, heute und morgen — Wissenschaft oder Flickwerk?", Berichte der Arbeitsgruppe Mathematisierung, Interdisziplinäre Arbeitsgruppe Mathematisierung (IAGM), Gesamthochschule Kassel, Heft 5, 1985 Feb.
- Baber, Robert L.; "Software-Entwicklung: von 'Stricken und Flickern' zu einer Ingenieurwissenschaft", *Vortrag*, Fachhochschule Fulda, 1985 Nov. 5.
- Baber, Robert L.; *Software-reflexionen: Ideen und Konzepte für die Praxis*, Springer-Verlag, Berlin, 1986.
- Baber, Robert L.; *The Spine of Software: Designing Provably Correct Software — Theory and Practice*, John Wiley & Sons, Chichester, 1987.
- Baber, Robert L.; "Software + Wartung = Widerspruch oder beliebte, bequeme Mythen entblößt", in Wix, Barbara; Balzert, Helmut (Hrsg.); *Softwarewartung* (Band 2 in der Reihe Angewandte Informatik), BI-Wissenschaftsverlag, Mannheim, S. 105-122, 1988.
- Baber, Robert L.; "Verlässlichkeit und Fehlertoleranz aus der Sicht eines Software-Ingenieurs", *Informationstechnik* *it*, 30. Jahrgang, Heft 3, S. 209-218, 1988.
- Baber, Robert L.; "Methoden zur Sicherung der Korrektheit von Computerprogrammen", Vortrag, Jahrestreffen der Freunde des Fachbereichs Angewandte Informatik und Mathematik der Fachhochschule Fulda e.V., 1988 Mai 28.
- Baber, Robert L.; "Methoden zur Sicherung der Korrektheit von Computerprogrammen", Vortrag, Kolloquium der Fakultät für Elektrotechnik, Universität Karlsruhe, 1988 Nov. 24.
- Baber, Robert L.; "'Software engineering' vs. software engineering", *IEEE Computer*, Vol. 22, No. 5, Open Channel column, p. 81, 1989 May. Relevant letter to the editor and author's reply in *IEEE Computer*, Vol. 22, No. 9, p. 8, 1989 Sept.
- Baber, Robert L.; *Fehlerfreie Programmierung für den Software-Zauberlehrling*, R. Oldenbourg Verlag, München, 1990.
- Baber, Robert L.; "Neue Ideen und Konzepte für die Erstellung nachprüfbar fehlerfreier Software", in Meckelburg, Hans-Jürgen; Jansen, Herbert (eds.); *Entwicklung und Prüfung sicherheitsbezogener Systeme: Software- und Systemaspekte*, p. 139-155, VDE-Verlag, Berlin, 1990.
- Baber, Robert L.; "Entwurf prüfbarer und korrekter Software", Vortrag, Technische Hochschule Darmstadt, Praktische Informatik, 1990 Jan. 25.
- Baber, Robert L.; "Neue Ideen und Konzepte für eine fehlerfreie Software-Erstellung — Was Software-Unternehmer und -Entwickler beachten sollten", Vortrag, 3. Berliner Software-Unternehmer-Gespräch, inhaltliche Konzeption Diebold Deutschland, Frankfurt/Main, 1990 Feb. 9.
- Baber, Robert L.; "Some Personal Views of Software Design", Special Lecture at the International Summer School on Programming and Mathematical Method, Marktoberdorf, Germany, 1990 Aug. 3.
- Baber, Robert L.; *Error-free Software: Know-how and Know-why of Program Correctness*, John Wiley & Sons, Chichester, 1991.
- Baber, Robert L.; "Epilogue: Future Developments", in McDermid, John A. (ed.); *Software Engineer's Reference Book*, p. 63/1-15, Butterworth-Heinemann, Oxford, 1991.
- Baber, Robert L.; "Portrait of a (Software) Engineer", *Journal of Systems and Software*, Vol. 15, p. 91-100, 1991.
- Baber, Robert L.; "Software Development Tomorrow: Possible Future Worlds", *The Software Practitioner*, Vol. 1, No. 3-4, p. 12-13, 1991 May.

- Baber, Robert L.; "Konzepte für die Erstellung möglichst fehlerfreier Software in der Vergangenheit und Zukunft", *HMD — Theorie und Praxis der Wirtschaftsinformatik*, Heft 163, S. 3-16, 1992 Jan.
- Baber, Robert L.; "Erstellung fehlerfreier Software", Vortrag, GI Regionalgruppe Rhein-Main, Darmstadt-Wixhausen, 1992 Jan. 21.
- Baber, Robert L.; "Programmkorrektheitsbeweise: Theorie und Praxis, Konstruktion und Verifikation", Vortrag, VDI-GIS-ATZ Tagung "Zuverlässigkeit und Qualität von Software — Zufall oder bestimmbar?", Hamburg, 1993 März 25.
- Baber, Robert L.; "Proofs of Correctness", in Marciniak, John J. (ed.); *Encyclopedia of Software Engineering*, p. 925-930, John Wiley & Sons, New York, 1994.
- Baber, Robert L.; "The Engineering of Engineering Software", *Boundary Elements Communications*, Vol. 5, No. 1, p. 6-11, 1994 Jan.
- Babin, Gilbert; Lustman, François; Shoval, Peretz; "Specification and Design of Transactions in Information Systems: A Formal Approach", *IEEE TSE*, Vol. 17, No. 8, p. 814-829, 1991 Aug.
- Back, R.J.R.; "A Calculus of Refinements for Program Derivations", *Acta Informatica*, Vol. 25, p. 593-624, 1988.
- Backhouse, Roland C.; *Program Construction and Verification*, Prentice-Hall International, Englewood Cliffs, N. J., 1986.
- Balzert, Helmut; *Die Entwicklung von Software-Systemen — Prinzipien, Methoden, Sprachen, Werkzeuge*, B.I.-Wissenschaftsverlag, Mannheim, 1982.
- Barroca, L.M.; McDermid, J.A.; "Formal Methods: Use and Relevance for the Development of Safety-Critical Systems", *The Computer Journal*, Vol. 35, No. 6, p. 579-599, 1992 Dec.
- Bartsch, M.; "Produkthaftung für Softwaretools", in *STAK '92, Software Technik in Automation und Kommunikation, Projektierungs- und Entwicklungswerkzeuge, Tagung Karlsruhe, 18. und 19. Februar 1992*, VDI Berichte 937, S. 115-120, VDI Verlag, 1992.
- Basili, Victor R.; "The Experience Factory and its Relationship to Other Improvement Paradigms", in Sommerville, Ian; Paul, Manfred (eds.); *Software Engineering — ESEC '93, 4th European Software Conference, Garmisch-Partenkirchen, Germany, September 1993, Proceedings*, LNCS 717, p. 68-83, Springer-Verlag, Berlin, 1993.
- Bauer, Friedrich L.; Wössner, Hans; *Algorithmische Sprache und Programmentwicklung*, Springer-Verlag, Berlin, 1984.
- Bauer, Friedrich Ludwig; Möller, Bernhard; Partsch, Helmut; Pepper, Peter; "Formal Program Construction by Transformations — Computer-Aided, Intuition-Guided Programming", *IEEE TSE*, Vol. 15, No. 2, p. 165-180, 1989 Feb.
- BCS; *Code of Practice*, Handbook No. 6, BCS, London, 1983.
- BCS; *Code of Conduct*, Handbook No. 5, BCS, London, 1983 Oct.
- Becker, G.; Camarinopoulos, L.; "A Bayesian Estimation Method for the Failure Rate of a Possibly Correct Program", *IEEE TSE*, Vol. 16, No. 11, Concise Papers column, p. 1307-1310, 1990 Nov.
- Beckman, Frank S.; *Mathematical Foundations of Programming*, Addison-Wesley, Reading, Massachusetts, 1980.
- Beierle, Christoph; Olthoff, Walter; Voss, Angelika; "Qualitätssicherung durch Programmverifikation und algebraische Methoden in der Softwareentwicklung", *Informatik-Spektrum*, Band 11, Heft 6, S. 292-302, 1988 Dez.

- Belli, F.; Jędrzejowicz, P.; "Ein Ansatz zur Zuverlässigkeits-Optimierung fehlertoleranter Software", *Informationstechnik*, 29. Jahrgang, Heft 2, S. 61-68, 1987.
- Bennett, Paul; Correspondence to the editor, *SCSC Newsletter*, Vol. 2, No. 2, p. 8, 1993 Jan.
- Berlioux, Pierre; Bizard, Philippe; *Algorithms: The Construction, Proof, and Analysis of Programs*, John Wiley & Sons, Chichester, 1986.
- Berry, Daniel M.; *Formal Specification and Verification of Concurrent Programs: SEI Curriculum Module SEI-CM-27-1.0*, Software Engineering Institute, Carnegie Mellon University, (Internet ftp files cm27.part1.ps and cm27.part2.ps in directory /pub/education at ftp.sei.cmu.edu), 1993 Feb.
- Berzins, Valdis; Luqi; *Software Engineering with Abstractions*, Addison-Wesley, Reading, Massachusetts, 1991.
- Bertziss, Alfs; *Formal Specification of Software: SEI Curriculum Module SEI-CM-8-1.0*, Software Engineering Institute, Carnegie Mellon University, (Internet ftp file cm8.ps in directory /pub/education at ftp.sei.cmu.edu), 1987 Oct.
- Bertziss, Alfs T.; Ardis, Mark A.; *Formal Verification of Programs: SEI Curriculum Module SEI-CM-20-1.0*, Software Engineering Institute, Carnegie Mellon University, (Internet ftp file cm20.ps in directory /pub/education at ftp.sei.cmu.edu), 1988 Dec.
- Bevier, William R.; "Kit: A Study in Operating System Verification", *IEEE TSE*, Vol. 15, No. 11, p. 1382-1396, 1989 Nov.
- Bhansali, P.V.; "Survey of software safety standards shows diversity", *IEEE Computer*, Vol. 26, No. 1, p. 88-89, 1993 Jan.
- Bijlsma, A.; "Semantics of Quasi-Boolean Expressions", in Feijen, W.H.J.; van Gasteren, A.J.M.; Gries, D.; Misra, J. (eds.); *Beauty Is Our Business — A Birthday Salute to Edsger W. Dijkstra*, p. 27-35, Springer-Verlag, New York, 1990.
- Bjørner, D.; Hoare, C.A.R.; Langmaack, H. (eds.); *VDM '90, VDM and Z — Formal Methods in Software Development, Third International Symposium of VDM Europe, Kiel, FRG, April 17-21, 1990, Proceedings*, LNCS 428, Springer-Verlag, Berlin, 1990.
- Bjørner, Dines; Langmaack, Hans; Hoare, C.A.R. (eds.); *Provably Correct Systems*, final report of the Esprit Basic Research Action project no. 3104: "ProCoS: Provably Correct Systems", Department of Computer Science, Technical University of Denmark, Lyngby, 1993.
- Blikle, Andrzej J.; "On the Development of Correct Specified Programs", *IEEE TSE*, Vol. SE-7, No. 5, p. 519-527, 1981 Sept.
- Blikle, A.; position statement for panel session P3.2, "Tough Nuts of Computer Science", in Mason, R.E.A. (ed.); *Information Processing 83: panel discussions*, IFIP 9th World Computer Congress, Paris, France, 1983 Sept. 22, p. 26, North-Holland, Amsterdam, 1983.
- Blikle, Andrzej; Tarlecki, Andrzej; "Naive Denotational Semantics", in Mason, R.E.A. (ed.); *Information Processing 83 — Proceedings of the IFIP 9th World Computer Congress, Paris, France, September 19-23, 1983*, p. 345-355, North-Holland, Amsterdam, 1983.
- Blikle, Andrzej; "Why Denotational?", in Janicki, Ryszard; Koczkodaj, Waldemar W. (eds.); *Computing and Information, Proceedings of the International Conference on Computing and Information, ICCI '89, Toronto, Canada, 23-27 May, 1989*, p. 3-14, North-Holland, Amsterdam, 1989.

- Bloom, Stephen L.; Ésik, Zoltán; "Floyd-Hoare Logic in Iteration Theories", *JACM*, Vol. 38, No. 4, p. 887-934, 1991 Oct.
- Bloomfield, Robin E.; Froome, Peter K.D.; "The Application of Formal Methods to the Assessment of High Integrity Software", *IEEE TSE*, Vol. SE-12, No. 9, Correspondence column, p. 988-993, 1986 Sept.
- Bloomfield, R.E.; Ehrenberger, W.D.; *Licensing issues associated with the use of computers in the nuclear industry*, EUR11147en, Commission of the European Communities, Directorate-General Telecommunications, Information Industries and Innovation, Luxembourg, in particular p. 183+200, 1987.
- Blyth, David; Boldyreff, Cornelia; Ruggles, Clive; Tetteh-Lartey, Nik; "The Case for Formal Methods in Standards", *IEEE Software*, Vol. 7, No. 5, p. 65-67, 1990 Sept.
- Boehm, Barry W.; Papaccio, Philip N.; "Understanding and Controlling Software Costs", *IEEE TSE*, Vol. 14, No. 10, p. 1462-1477, 1988 Oct.
- Boehm, Hans-Juergen; "Side Effects and Aliasing Can Have Simple Axiomatic Descriptions", *TOPLAS*, Vol. 7, No. 4, p. 637-655, 1985 Oct.
- Böhme, G. (Hrsg.); *Prüfungsaufgaben Informatik*, Springer-Verlag, Berlin, 1984.
- Boiten, E.A.; Partsch, H.A.; Tuijnman, D.; Völker, N.; "How to Produce Correct Software — An Introduction to Formal Specification and Program Development by Transformations", *The Computer Journal*, Vol. 35, No. 6, p. 547-554, 1992 Dec.
- Bonsiepen, Lena; Coy, Wolfgang; "Eine Curriculardebatte", *Informatik-Spektrum*, Band 15, Heft 6, S. 323-325, 1992 Dez.
- Boom, H.J.; "A Weaker Precondition for Loops", *TOPLAS*, Vol. 4, No. 4, p. 668-677, 1982 Oct.
- Borer, J.R.; "Software for emergency shut down systems", in Ross, M.; Brebbia, C.A.; Staples, G.; Stapleton, J. (eds.); *Software Quality Management*, (Proceedings of the SQM '93 Conference, Southampton), p. 457-472, Computational Mechanics Publications, Southampton, 1993.
- Borgenstam, Curt; Sandström, Anders; *Why Wasa Capsized*, Wasa Studies 13, Statens sjöhistoriska museum, Stockholm, keine Jahresangabe (1984?).
- Bosco, P.G.; Ferrari, L.; Giovannetti, E.; Moiso, C.; "An overview of ALCOHOL: an Applicative Logic Concurrent Object-oriented Higher-Order Language", in van Leeuwen, Jan (ed.); *Algorithms, Software, Architecture: Information Processing 92 — Proceedings of the IFIP 12th World Computer Congress, Madrid, Spain, 7-11 September 1992, Volume I*, p. 142-148, North-Holland, Amsterdam, 1992.
- Bourdoncle, Françoise; "Assertion-based debugging of imperative programs by abstract interpretation", in Sommerville, Ian; Paul, Manfred (eds.); *Software Engineering — ESEC '93, 4th European Software Conference, Garmisch-Partenkirchen, Germany, September 1993, Proceedings*, LNCS 717, p. 501-516, Springer-Verlag, Berlin, 1993.
- Boyer, Robert S.; Moore, J. Strother (eds.); *The Correctness Problem in Computer Science*, Academic Press, London, 1981.
- Bradley, Peter; Shackelton, Linda; Stavridou, Victoria; "The Practical Application of Formal Methods to High Integrity Systems — The SafeFM Project", in Redmill, Felix; Anderson, Tom (eds.); *Directions in Safety-critical Systems, Proceedings of the First Safety-critical Systems Symposium, Bristol, 9-11 February 1993*, p. 168-176, Springer-Verlag, London, 1993.
- Brady, J.M.; *The Theory of Computer Science: A Programming Approach*, Chapman and Hall, London, 1977.

- Brainerd, Walter S.; Landweber, Lawrence H.; *Theory of Computation*, John Wiley & Sons, New York, 1974.
- Breu, Ruth; Breu, Michael; "Ein Konzept der Modul — und Typvererbung", in Hoffmann, Hans-Jürgen (Hrsg.); *Eiffel, Fachtagung des German Chapter of the ACM e.V. in Zusammenarbeit mit der Gesellschaft für Informatik e.V., FA 2.1, am 25. und 26. Mai 1992 in Darmstadt*, Band 35, S. 23-37, B.G. Teubner, Stuttgart, 1992.
- Britcher, Robert N.; "Using Inspections to Investigate Program Correctness: A Proposal", *IEEE Computer*, Vol. 21, No. 11, p. 38-44, 1988 Nov.
- Britcher, Robert N.; "Cards, Couriers, and the Race to Correctness", *Journal of Systems and Software*, Vol. 17, p. 281-284, 1992.
- Bronstein, I.N.; Semendjajew, K.A.; *Taschenbuch der Mathematik*, 25. Auflage, B.G. Teubner, Stuttgart, 1991. *Ergänzende Kapitel*, 6. Auflage, Verlag Harri Deutsch, Thun und Frankfurt/Main, 1991.
- Brooks, Jr., Frederick P.; *The Mythical Man-Month*, Addison-Wesley, Reading, Massachusetts, 1979.
- Brooks, Jr., Frederick P.; "No Silver Bullet — Essence and Accidents of Software Engineering", in Kugler, H.-J. (ed.); *Information Processing 86 — Proceedings of the IFIP 10th World Computer Congress, Dublin, Ireland, September 1-5, 1986*, p. 1069-1076, North-Holland, Amsterdam, 1986.
- Brown, David B.; Maghsoodloo, Saeed; Deason, William H.; "A Cost Model for Determining the Optimal Number of Software Test Cases", *IEEE TSE*, Vol. 15, No. 2, Concise Papers column, p. 218-221, 1989 Feb.
- Broy, Manfred; Pepper, Peter; "Program Development as a Formal Activity", *IEEE TSE*, Vol. SE-7, No. 1, p. 14-22, 1981 Jan.
- Broy, Manfred; "Applicative Real-Time Programming", in Mason, R.E.A. (ed.); *Information Processing 83 — Proceedings of the IFIP 9th World Computer Congress, Paris, France, September 19-23, 1983*, p. 259-264, North-Holland, Amsterdam, 1983.
- Broy, Manfred; "Formal treatment of concurrency and time", in McDermid, John A. (ed.); *Software Engineer's Reference Book*, p. 23/1-19, Butterworth-Heinemann, Oxford, 1991.
- Broy, Manfred (ed.); *Programming and Mathematical Method*, Springer-Verlag, Berlin, 1992.
- Broy, Manfred; "Functional Specification of Time Sensitive Communicating Systems", in Broy, Manfred (ed.); *Programming and Mathematical Method*, p. 325-367, Springer-Verlag, Berlin, 1992.
- Broy, M.; Bibel, W.; Jähnichen, S.; Kreowski, H.-J.; Siekmann, J.; Vogt, F.; "Sicherheit, Zuverlässigkeit und Korrektheit von Software", *Informatik-Spektrum*, Band 16, Heft 4, S. 227-228, 1993 Aug.
- Broy, Manfred; Wirsing, Martin; "Korrekte Software: Vom Experiment zur Anwendung", in Reichel, Horst (Hrsg.); *Informatik — Wirtschaft — Gesellschaft, 23. GI-Jahrestagung, Dresden, 27. September - 1. Oktober 1993*, p. 29-43, Springer-Verlag, Berlin, 1993.
- Bryant, Randal E.; "Formal Verification: A Slow, but Certain Evolution", in van Leeuwen, Jan (ed.); *Algorithms, Software, Architecture: Information Processing 92 — Proceedings of the IFIP 12th World Computer Congress, Madrid, Spain, 7-11 September 1992, Volume I*, p. 712, North-Holland, Amsterdam, 1992.

- Buchheim, Gisela; Sonnemann, Rolf; *Lebensbilder von Ingenieurwissenschaftlern — Eine Sammlung von Biographien aus zwei Jahrhunderten*, Birkhäuser, Basel, 1989.
- Bustard, David W.; Winstanley, Adam C.; "Making Changes to Formal Specifications: Requirements and an Example", in Sommerville, Ian; Paul, Manfred (eds.); *Software Engineering — ESEC '93, 4th European Software Conference, Garmisch-Partenkirchen, Germany, September 1993, Proceedings*, LNCS 717, p. 115-126, Springer-Verlag, Berlin, 1993.
- Buth, Bettina; "PAMELA — Ein Ansatz zur computergestützten Softwareverifikation in industriellen Anwendungen", in Meckelburg, Hans-Jürgen; Jansen, Herbert (eds.); *Entwicklung und Prüfung sicherheitsbezogener Systeme: Software- und Systemaspekte*, S. 157-171, VDE-Verlag, Berlin, 1990.
- Butler, Ricky W.; Finelli, George B.; "The Infeasibility of Quantifying the Reliability of Life-Critical Real-Time Software", *IEEE TSE*, Vol. 19, No. 1, p. 3-12, 1993 Jan.
- Buxton, J.N.; Randell, B. (eds.); *Software Engineering Techniques, Report on a conference sponsored by the NATO Science Committee, Rome, Italy, 27th to 31th October 1969*, Scientific Affairs Division, NATO, Brussels, 1970 April.
- BWB Bundesamt für Wehrtechnik und Beschaffung; *Software-Entwicklungsstandard für DV-Anteile in Wehrmaterial, Vorgehensmodell, Kurzbeschreibung*, Bundesamt für Wehrtechnik und Beschaffung, FE VI 2, Koblenz, 1989 Juni.
- Cantin, Guylaine; *The Use of Function-Tables to Specify Complex Algorithms*, CRL Report No. 236, Communications Research Laboratory, Faculty of Engineering, McMaster University, Hamilton, Ontario, Canada, 1991 Sept.
- Card, David N.; McGarry, Frank E.; Page, Gerald T.; "Evaluating Software Engineering Technologies", *IEEE TSE*, Vol. SE-13, No. 7, p. 845-851, 1987 July.
- Carré, Bernard; "Program analysis and verification", in Sennett, Chris T. (ed.); *High-integrity Software*, p. 176-197, Pitman, London, 1989.
- Casais, Eduardo; Lewerentz, Claus; Lindner, Thomas; Weber, Franz; *Formal Methods and Object-Orientation, Tutorial at TOOLS Europe 1993*, Forschungszentrum Informatik, Karlsruhe, 1993.
- Chalin, Patrice; Grogono Peter; "Z Specification of an Object Manager", in Bjørner, D.; Hoare, C.A.R.; Langmaack, H. (eds.); *VDM '90, VDM and Z — Formal Methods in Software Development, Third International Symposium of VDM Europe, Kiel, FRG, April 17-21, 1990, Proceedings*, LNCS 428, p. 41-71, Springer-Verlag, Berlin, 1990.
- Chedghey, Chris; Kearney, Seamus; Kugler, Hans-Jürgen; "Using VDM in an Object-Oriented Development Method for Ada Software", in Bjørner, D.; Jones, C.B.; Mac an Airchinnigh, M.; Neuhold, E.J. (eds.); *VDM '87, VDM — A Formal Method at Work, VDM-Europe Symposium 1987, Brussels, Belgium, March 23-26, 1987, Proceedings*, LNCS 252, p. 63-76, Springer-Verlag, Berlin, 1987.
- Cristian, Flaviu; "Correct and Robust Programs", *IEEE TSE*, Vol. SE-10, No. 2, p. 163-174, 1984 March.
- Claus, Volker; *Stochastische Automaten*, B.G. Teubner, Stuttgart, 1971.
- Cobb, Richard H.; Mills, Harlan D.; "Engineering Software under Statistical Quality Control", *IEEE Software*, Vol. 7, No. 6, p. 44-54, 1990 Nov. Relevant letter to the editor and authors' reply in *IEEE Software*, Vol. 8, No. 2, p. 8, 1991 March.
- Cohen, Edward; *Programming in the 1990's: An Introduction to the Calculation of Programs*, Springer-Verlag, New York, 1990.

- Coleman, Derek; Hayes, Fiona; Bear, Stephen; "Introducing Objectcharts or How to Use Statecharts in Object-Oriented Design", *IEEE TSE*, Vol. 18, No. 1, p. 9-18, 1992 Jan.
- Cooke, J.; "Formal methods — Mathematics, Theory, Recipes or what?", *The Computer Journal*, Vol. 35, No. 5, p. 419-423, 1992 Oct.
- Cooper, D.C.; "Program Correctness, Proof of", in Ralston, Anthony; Meek, Chester L. (eds.); *Encyclopedia of Computer Science*, p. 1160-1163, Petrocelli/Charter, New York, 1976.
- Counsell, S.; "Software reliability issues in CCS and CSP", in Ross, M.; Brebbia, C.A.; Staples, G.; Stapleton, J. (eds.); *Software Quality Management*, (Proceedings of the SQM '93 Conference, Southampton), p. 663-679, Computational Mechanics Publications, Southampton, 1993.
- Cox, Brad J.; "Planning the Software Industrial Revolution", *IEEE Software*, Vol. 7, No. 6, p. 25-33, 1990 Nov.
- Craigen, Dan; "Strengths and Weaknesses of Program Verification Systems", *Proceedings of the (First) European Software Engineering Conference*, p. 396-404, 1987.
- Craigen, Dan; "Verification environments", in McDermid, John A. (ed.); *Software Engineer's Reference Book*, p. 26/1-14, Butterworth-Heinemann, Oxford, 1991.
- Craigen, Dan; Gerhart, Susan; Ralston, Ted; *An International Survey of Industrial Applications of Formal Methods, Volume 1 Purpose, Approach, Analysis, and Conclusions; Volume 2 Case Studies*, NISTGCR 93/626, U.S. Department of Commerce, Technology Administration, National Institute of Standards and Technology, Computer Systems Laboratory, Gaithersburg, MD, 1993 March.
- Culik, K.; "On formal and informal proofs for program correctness", *ACM SIGPLAN Notices*, Vol. 18, No. 1, p. 23-28, 1983 Jan.
- Cunningham, R.J.; Kramer, J.; "An Exercise in Program Design Using SIMULA Class Invariants", *Software — Practice and Experience*, Vol. 8, No. 3, p. 355-369, 1978 May-June.
- Currit, P. Allen; Dyer, Michael; Mills, Harlan D.; "Certifying the Reliability of Software", *IEEE TSE*, Vol. SE-12, No. 1, p. 3-11, 1986 Jan.
- Cusack, Elspeth; "Inheritance In Object Oriented Z", in America, Pierre (ed.); *ECOOP '91, European Conference on Object-Oriented Programming, Geneva, Switzerland, July 1991, Proceedings*, LNCS 512, p. 167-179, Springer-Verlag, Berlin, 1991.
- Cusack, Elspeth; von Bochmann, Gregor; "Formal Object-Oriented Methods in Communication Standards", *ACM OOPS Messenger*, Vol. 3, No. 2, p. 7-8, 1992 April.
- Dahl, O.-J.; Dijkstra, E.W.; Hoare, C.A.R.; *Structured Programming*, Academic Press, London, 1972.
- Dahl, Ole-Johan; "Object Orientation and Formal Techniques", in Bjørner, D.; Hoare, C.A.R.; Langmaack, H. (eds.); *VDM '90, VDM and Z — Formal Methods in Software Development, Third International Symposium of VDM Europe, Kiel, FRG, April 17-21, 1990, Proceedings*, LNCS 428, p. 1-11, Springer-Verlag, Berlin, 1990.
- Dal Cin, Mario; Skriptum für die Vorlesung "Einführung in die Parallelprogrammierung", Fachbereich Informatik, Johann Wolfgang Goethe-Universität, Frankfurt/Main, Wintersemester 1985-1986.
- Dal Cin, Mario; *Grundlagen der systemnahen Programmierung*, B.G. Teubner, Stuttgart, 1988.

- Dannenberg, Roger B.; Ernst, George W.; "Formal Program Verification Using Symbolic Execution", *IEEE TSE*, Vol. SE-8, No. 1, p. 43-52, 1982 Jan.
- Darringer, John A.; King, James C.; *Applications of Symbolic Execution to Program Testing*, Research Report RC 6965 (#29464), Computer Sciences Department, IBM Thomas J. Watson Research Center, Yorktown Heights, 1977 Dec. 1. Also published in *IEEE Computer*, Vol. 11, No. 4, p. 51-60, 1978 April.
- Davis, Paul I.; "Certification of safety-critical software by licensed software engineers", *IEEE Computer*, Vol. 25, No. 12, p. 72-73, 1992 Dec.
- de Bakker, J.W.; *Mathematical Theory of Program Correctness*, Prentice-Hall International, Englewood Cliffs, N.J., 1980.
- de Boer, Frank S.; *A Proof System for the Language POOL*, in de Bakker, J.W.; de Rooter, W.P.; Rozenberg, G. (eds.); *Foundations of Object-Oriented Languages — REX School/Workshop, Noordwijkerhout, The Netherlands, May 28-June 1, 1990, Proceedings*, LNCS 489, p. 124-150, Springer-Verlag, Berlin, 1990.
- de Boer, F.S.; *A Proof Theory for the Language POOL*, Report CS-R9117, Centre for Mathematics and Computer Science (CWI), Amsterdam, 1991 March.
- De Millo, Richard A.; Lipton, Richard J.; Perlis, Alan J.; "Social Processes and Proofs of Theorems and Programs", *CACM*, Vol. 22, No. 5, p. 271-280, 1979 May.
- Denvir, Tim; *Introduction to Discrete Mathematics for Software Engineering*, Macmillan Education, Basingstoke, 1986.
- Deutsche Gesellschaft für Qualität (DGQ); Informationstechnische Gesellschaft im VDE (ITG) (Hrsg.); *Methoden und Verfahren der Software-Qualitätssicherung*, Deutsche Gesellschaft für Qualität, Frankfurt am Main, 1992.
- Di Giovanni, R.; Iachini, P.L.; "HOOD and Z for the Development of Complex Software Systems", in Bjørner, D.; Hoare, C.A.R.; Langmaack, H. (eds.); *VDM '90, VDM and Z — Formal Methods in Software Development, Third International Symposium of VDM Europe, Kiel, FRG, April 17-21, 1990, Proceedings*, LNCS 428, p. 262-289, Springer-Verlag, Berlin, 1990.
- Dijkstra, E.W.; "A Constructive Approach to the Problem of Program Correctness", *BIT Nord. Tidskr. Inform.*, Vol. 8, p. 174-186, 1968.
- Dijkstra, E.W.; "Goto Statement Considered Harmful", Letter to the Editor, *CACM*, Vol. 11, p. 147-148, and reply, p. 538-541, 1968.
- Dijkstra, E.W.; "The Structure of the 'THE' Multiprogramming System", *CACM*, Vol. 11, p. 341-346, 1968.
- Dijkstra, Edsger W.; "Guarded Commands, Nondeterminacy and Formal Derivation of Programs", *CACM*, Vol. 18, No. 8, p. 453-457, 1975 Aug.
- Dijkstra, Edsger W.; *A Discipline of Programming*, Prentice-Hall, Inc., Englewood Cliffs, N.J., 1976.
- Dijkstra, Edsger W.; "Correctness Concerns and, among other Things, Why They Are Resented", in Gries, David (ed.); *Programming Methodology*, Springer-Verlag, New York, 1978.
- Dijkstra, Edsger W.; *Selected Writings on Computing: A Personal Perspective*, Springer-Verlag, New York, 1982.
- Dijkstra, Edsger W.; Feijen, W.H.J.; *Methodik des Programmierens*, Addison-Wesley Verlag (Deutschland), 1985.
- Dijkstra, Edsger W.; *Science fiction and science reality in computing*, *EWD 952*, 1986.
- Dijkstra, Edsger W.; "By way of introduction", *EWD 1041*, 1989 Feb. 12.

- Dijkstra, Edsger W.; Zitat in einem Pressebericht über die 17th Annual ACM Computer Science Conference in *IEEE Software*, Vol. 6, No. 3, p. 97, 1989 May.
- Dijkstra, Edsger W.; "On the Cruelty of Really Teaching Computing Science", in Denning, Peter J. (ed.); "A Debate on Teaching Computing Science", *CACM*, Vol. 32, No. 12, p. 1397-1414, 1989 Dec.
- Dijkstra, Edsger W. (ed.); *Formal Development of Programs and Proofs*, Addison-Wesley, Reading, Massachusetts, 1990.
- Dijkstra, Edsger W.; "A logician's anomaly or: Leibniz vindicated", *EWD 1084*, 1990 Sept. 23.
- Dijkstra, Edsger W.; "On the Economy of doing Mathematics", *EWD 1130*, 1992 June 26.
- Dijkstra, Edsger W.; "A somewhat open letter to Cathleen Synge Morawetz", *EWD 1145*, 1992 Nov. 22.
- Dillon, Laura K.; "Verifying General Safety Properties of Ada Tasking Programs", *IEEE TSE*, Vol. 16, No. 1, p. 51-63, 1990 Jan.
- DKE Deutsche Elektrotechnische Kommission im DIN und VDE (DKE); *Software für Rechner im Sicherheitssystem von Kernkraftwerken*, Deutsche Norm DIN IEC 880, Beuth Verlag, Berlin, 1987 Aug.
- DKE Deutsche Elektrotechnische Kommission im DIN und VDE (DKE); *Grundsätze für Rechner in Systemen mit Sicherheitsaufgaben*, Vornorm DIN V VDE 0801/01.90, VDE-Verlag, Berlin, 1990 Jan.
- Downs, Ed; Clare, Peter; Coe, Ian; *Structured Systems Analysis and Design Method*, Prentice Hall, New York, 1988.
- Dromey, R.G.; "Derivation of Sorting Algorithms from a Specification", *The Computer Journal*, p. 512-518, Vol. 30, No. 6, 1987.
- Dromey, R. Geoff; "Systematic Program Development", *IEEE TSE*, Vol. 14, No. 1, p. 12-29, 1988 Jan.
- Dromey, Geoff; *Program Derivation: The Development of Programs from Specifications*, Addison-Wesley, Sydney, 1989.
- Dröschel, Wolfgang; "Wie entwickle ich Software?", *wt wehrtechnik*, 8/89, S. 33-35, 1989.
- Du Croz, J.; "FPV — a floating point validation package", in Ince, Darrel (ed.); *Software Quality and Reliability*, p. 71-79, Chapman & Hall, London, 1991.
- Duke, David; Duke, Roger; "Towards a Semantics for Object-Z", in Bjørner, D.; Hoare, C.A.R.; Langmaack, H. (eds.); *VDM '90, VDM and Z — Formal Methods in Software Development, Third International Symposium of VDM Europe, Kiel, FRG, April 17-21, 1990, Proceedings*, LNCS 428, p. 244-261, Springer-Verlag, Berlin, 1990.
- Dunlop, Douglas D.; Basili, Victor R.; "A Heuristic for Deriving Loop Functions", *IEEE TSE*, Vol. SE-10, No. 3, p. 275-285, 1984 May.
- Dunn, William R.; Corliss, Lloyd D.; "Software Safety: A User's Practical Perspective", *Proceedings Annual Reliability and Maintainability Symposium 1990*, p. 430-435, IEEE, New York, 1990 Jan.
- Dyer, Michael; *The Cleanroom Approach to Quality Software Development*, John Wiley & Sons, New York, 1992.
- Eckhardt, Dave E.; Caglayan, Alper K.; Knight, John C.; Lee, Larry D.; McAllister, David F.; Vouk, Mladen A.; Kelly, John P.J.; "An Experimental Evaluation of Soft-

- ware Redundancy as a Strategy For Improving Reliability", *IEEE TSE*, Vol. 17, No. 7, p. 692-702, 1991 July.
- Ehrenberger, Wolfgang; "Kein Programm ohne Fehler! Über die Erstellung zuverlässiger und sicherer Software", *Computer Magazin*, Nr. 11, p. 52-54, 1985.
- Ehrenberger, Wolfgang D.; "Understanding Programs versus Understanding Program Proofs", draft, 1993.
- Ehrig, Hartmut; Kiermeier, Klaus-Dieter; Kreowski, Hans-Jörg; Kühnel, Wolfgang; *Universal Theory of Automata — A Categorical Approach*, B.G. Teubner, Stuttgart, 1974.
- Emonts, Jörg; "Formale Spezifikation und Verifikation für 'sichere Software', Tools für die Zuverlässigkeit auch in Ausnahmesituationen", *Computerwoche*, Nr. 5, S. 30, 1991 Feb. 1.
- Endres, Albert; *Analyse und Verifikation von Programmen: Systematische Verfahren und Untersuchungen zur Erstellung fehlerfreier Software*, R. Oldenbourg Verlag, München, 1977.
- Enfield, Ronald L.; "The Limits of Software Reliability", *Technology Review*, p. 36-40+42-43, 1987 April.
- Engineering Council (U.K.); *Engineers and Risk Issues: Code of Professional Practice*, 1992 Oct.
- Ernst, George W.; Navlakha, Jainendra K.; Ogden, William F.; "Verification of Programs with Procedure-Type Parameters", *Acta Informatica*, Vol. 18, p. 149-169, 1982.
- Ershov, A.P.; position statement for panel session P3.2, "Tough Nuts of Computer Science", in Mason, R.E.A. (ed.); *Information Processing 83: panel discussions*, IFIP 9th World Computer Congress, Paris, France, 1983 Sept. 22, p. 26, North-Holland, Amsterdam, 1983.
- Erskine, Neil Stuart; *The Usefulness of the Trace Assertion Method for Specifying Device Module Interfaces*, CRL Report No. 258, Communications Research Laboratory, Faculty of Engineering, McMaster University, Hamilton, Ontario, Canada, 1992 Aug.
- Esprit; *Esprit Synopses of Information Processing Systems, ESPRIT II Projects and Exploratory Actions*, Vol. 4, Directorate General XIII, Commission of the European Communities, Brussels, 1990 Sept.
- Esprit; *Esprit Synopses of Basic Research*, Vol. 8, Directorate General XIII, Commission of the European Communities, Brussels, 1991 Jan.
- Europäische Gemeinschaften — Kommission; *Kriterien für die Bewertung der Sicherheit von Systemen der Informationstechnik (ITSEC)*, Amt für amtliche Veröffentlichungen der Europäischen Gemeinschaften, Luxemburg, 1991 Juni.
- Evans, Ralph A.; "ISO 9000 — Get Real!", *IEEE Transactions on Reliability*, Vol. 42, No. 1, Editorial column, p. 1, 1993 March.
- Fachausschuß 7.4 der GI; "Zur Berufssituation der Informatiker 1991 — Ergebnisse der Mitgliederbefragung der GI 1991/92", *Informatik-Spektrum*, Band 15, S. 335-351, 1992.
- Fachbereich 2 der GI; "Zur Aus- und Weiterbildung im Bereich der ingenieurmäßigen System- und Programmentwicklung", *Informatik-Spektrum*, Band 16, Heft 1, S. 31-33, 1993.
- Fagan, Michael E.; "Advances in Software Inspections", *IEEE TSE*, Vol. SE-12, No. 7, p. 744-751, 1986 July.

- Feder, Christiane; *Ausnahmebehandlung in objektorientierten Programmiersprachen*, Informatik-Fachberichte 235, Springer-Verlag, Berlin, 1990.
- Fehr, E.; "Funktionale Programmierung", *Informatik-Spektrum*, Band 5, Heft 3, S. 194-196, 1982 Sept.
- Feijen, W.H.J.; van Gasteren, A.J.M.; Gries, D.; Misra, J. (eds.); *Beauty Is Our Business — A Birthday Salute to Edsger W. Dijkstra*, Springer-Verlag, New York, 1990.
- Feijen, W.H.J.; "Phase Synchronization for Two Machines", in Broy, Manfred (ed.); *Programming and Mathematical Method*, p. 27-31, Springer-Verlag, Berlin, 1992.
- Feijen, W.H.J.; "The Lexicographic Minimum of a Cyclic Array", in Broy, Manfred (ed.); *Programming and Mathematical Method*, p. 33-42, Springer-Verlag, Berlin, 1992.
- Fejer, Peter A.; Simovici, Dan A.; *Mathematical Foundations of Computer Science, Volume 1: Sets, Relations, and Induction*, Springer-Verlag, New York, 1991.
- Feldtkeller, Richard; *Einführung in die Vierpoltheorie*, S. Hirzel Verlag, Stuttgart, 1962 und andere Auflagen.
- Fetzer, James H.; "Program Verification: the Very Idea", *CACM*, Vol. 31, No. 9, p. 1048-1063, 1988 Sept.
- Fields, Bob; Elvang-Göransson, Morten; "A VDM Case Study in mural", *IEEE TSE*, Vol. 18, No. 4, p. 279-295, 1992 April.
- Filman, Robert E.; Friedman, Daniel P.; *Coordinated Computing: Tools and Techniques for Distributed Software*, McGraw-Hill, New York, 1984.
- Fischbach, Rainer; "Programming by Contract — Erfüllt Eiffel das Ideal?", in Hoffmann, Hans-Jürgen (Hrsg.); *Eiffel, Fachtagung des German Chapter of the ACM e.V. in Zusammenarbeit mit der Gesellschaft für Informatik e.V., FA 2.1, am 25. und 26. Mai 1992 in Darmstadt*, Band 35, S. 55-68, B.G. Teubner, Stuttgart, 1992.
- Fitting, Melvin; *First-Order Logic and Automated Theorem Proving*, Springer-Verlag, New York, 1990.
- Fitzgerald, Karen; "Vulnerability exposed in AT&T's 9-hour glitch", *IEEE Institute*, Vol. 14, No. 3, p. 1, 1990 March.
- Floyd, Robert W.; "Assigning Meanings to Programs", *Proceedings of the Symposium of Applied Mathematics*, Vol. 19, p. 19-32, American Mathematical Society, Providence, R.I., 1967.
- Foley, M.; Hoare, C.A.R.; "Proof of a recursive program: Quicksort", *The Computer Journal*, Vol. 14, No. 4, p. 391-395, 1971 Nov.
- Föppl, August; *Vorlesungen über Technische Mechanik*, mehrere Bände und Auflagen, R. Oldenbourg Verlag, München, 1940-1943.
- Foremski, Tom; "US industry aims to block bill on consumer rights", *Computing*, 1985 Nov. 28.
- Foremski, Tom; Garrett, Alex; "UK, US software suppliers face liability threat", *Computing*, p. 16, 1986 Jan. 9.
- Föbmeier, Reinhard; "Enkonduko al la problemoj de programaj-produktado", *Aktoj de SUS (Sanmarinaj Universitataj Sesioj)*, Akademio Internacia de la Sciencoj (AIS) San Marino, (in Esperanto) LTV Leuchtturm Verlag, Alsbach/Bergstraße, Deutschland, Vol. I, Kajero L1, 1990.
- Föbmeier, Reinhard; "Enkonduko al la problemoj de la programaja ingenieriko", *Acta Sanmarinensia Academiae Scientiarum Internationalis*, Akademio Internacia de la Sciencoj (AIS) San Marino, (in Esperanto) Vol. 3/1.1, 1993.

- Fóthi, Ákos; "Unu matematika modelo por programado", in Koutny, Ilona (red.); *Teoriaj kaj Praktikaj Problemoj de la Programado, prelegoj de Interkomputo, Budapest 1982*, p. 60-71, (in Esperanto) Neumann János Számítógéptudományi Társaság, 1982.
- France, Robert B.; "Semantically Extended Data Flow Diagrams: A Formal Specification Tool", *IEEE TSE*, Vol. 18, No. 4, p. 329-346, 1992 April.
- Francez, Nissim; *Program Verification*, Addison-Wesley, Wokingham, England, 1992.
- Frank, Nathaniel H.; *Introduction to Electricity and Optics*, McGraw-Hill, New York, 1950.
- Fraser, Martin D.; Kumar, Kuldeep; Vaishnavi, Vijay K.; "Informal and Formal Requirements Specification Languages: Bridging the Gap", *IEEE TSE*, Vol. 17, No. 5, p. 454-466, 1991 May.
- Frühauf, Karol; "ISO 9000-3 im Spiegel der IEEE-Normenreihe für Software Engineering", *atp — Automatisierungstechnische Praxis*, 33. Jahrgang, Heft 12, S. 626-632, 1991.
- Futschek, Gerald; *Programmentwicklung und Verifikation*, Springer-Verlag, Wien, New York, 1989.
- Gallier, Jean H.; *Logic for Computer Science: Foundations of Automatic Theorem Proving*, Harper & Row, New York, 1986.
- Gannon, John D.; Hamlet, Richard G.; Mills, Harlan D.; "Theory of Modules", *IEEE TSE*, Vol. SE-13, No. 7, p. 820-829, 1987 July.
- Gardener, Stewart; "Software in Safety-Critical Systems", *SCSC Newsletter*, Vol. 2, No. 1, p. 5-6, 1992 Sept.
- Gaudel, Marie-Claude; "Algebraic specifications", in McDermid, John A. (ed.); *Software Engineer's Reference Book*, p. 22/1-10, Butterworth-Heinemann, Oxford, 1991.
- Gerhart, Susan; "Formal methodists warn of software disasters", *IEEE Software*, Vol. 6, No. 6, p. 77+83, 1989 Nov.
- Gerhart, Susan L.; "Applications of Formal Methods: Developing Virtuoso Software", *IEEE Software*, Vol. 7, No. 5, p. 7-10, 1990 Sept.
- Glass, Robert L.; "A Critical Analysis of the Cleanroom Technique", with rebuttal by Mills, Harlan D.; *The Software Practitioner*, Vol. 1, No. 5, p. 1-2+4+7, 1991 Sept.-Oct.
- Glass, Robert L.; "Case Studies of Formal Methods: What is Their Value? Will We Be Forced to Use Them?", *The Software Practitioner*, p. 9, 1992 Jan.
- Goguen, Joseph A.; Meseguer, José; *Semantics of Computation: Initial Algebra Semantics and Programming Language Paradigms*, draft, 1986.
- Goguen, Joseph A.; "One, None, a Hundred Thousand Specification Languages", in Kugler, H.-J. (ed.); *Information Processing 86 — Proceedings of the IFIP 10th World Computer Congress, Dublin, Ireland, September 1-5, 1986*, p. 995-1003, North-Holland, Amsterdam, 1986.
- Goguen, Joseph; Moriconi, Mark; "Formalization in Programming Environments", *IEEE Computer*, Vol. 20, No. 11, p. 55-64, 1987 Nov.
- Goguen, Joseph A.; Winkler, Timothy; *Introducing OBJ3*, Technical Report SRI-CSL-88-9, Computer Science Laboratory, SRI International, Menlo Park, California, 1988 Aug.
- Goldberg, David; "What Every Computer Scientist Should Know About Floating-Point Arithmetic", *ACM Computing Surveys*, Vol. 23., No. 1, p. 5-48, 1991 March.

- Goldson, Doug; Reeves, Steve; Bornat, Richard; "A Review of Several Programs for the Teaching of Logic", *The Computer Journal*, Vol. 36, No. 4, p. 373-386, 1993.
- Goossens, K.G.W.; "Integrating Hardware Description Languages and Proof Systems", in *Poster Session — 12th World Computer Congress, IFIP Congress '92*, p. 74, IFIP, Madrid, 1992 Sept. 7-11.
- Grams, Timm; "Diversitäre Programmierung: Kein Allheilmittel", *Informationstechnik* *it*, 28. Jahrgang, Heft 4, S. 196-203, 1986.
- Grams, Timm; "Biased Programming Faults — How to Overcome Them?", in Belli, F.; Görke, W. (Hrsg.); *Fault-Tolerant Computing Systems, 3rd International GI/ITG/GMA Conference, Bremerhaven, 9-11 September 1987, Proceedings*, p. 13-23, Springer-Verlag, Berlin, 1987.
- Grams, Timm; "Thinking Traps in Programming — A Systematic Collection of Examples", in Ehrenberger, W.D. (ed.); *Safety of Computer Control Systems 1988 (SAFE-COMP '88), Safety Related Computers in an Expanding Market, Proceedings of the IFAC Symposium, Fulda, FRG, 9-11 Nov. 1988*, p. 95-100, Pergamon Press, Oxford, 1988.
- Grams, Timm; "Ursachen häufiger Programmierfehler", *HMD Handbuch der Modernen Datenverarbeitung*, Heft 144, S. 46-57, 1988 Nov.
- Grams, Timm; *Denkfallen und Programmierfehler*, Springer-Verlag, Berlin, 1990.
- Grams, Timm; "Software-Zuverlässigkeit, gibt es das?", *Informationstechnik* *it*, 32. Jahrgang, Heft 2, S. 125-134, 1990.
- Grams, Timm; "Denkfallen beim objektorientierten Programmieren", *Informationstechnik* *it*, 34. Jahrgang, Heft 2, S. 102-112, 1992.
- Grams, Timm; "Täuschwörter im Software Engineering", *Informatik-Spektrum*, Band 16, Heft 3, S. 165-166, 1993 Juni.
- Greif, Irene; Meyer, Albert R.; "Specifying the Semantics of while Programs: A Tutorial and Critique of a Paper by Hoare and Lauer", *TOPLAS*, Vol. 3, No. 4, p. 484-507, 1981 Oct.
- Gries, David; "An Illustration of Current Ideas on the Derivation of Correctness Proofs and Correct Programs", *IEEE TSE*, Vol. SE-2, No. 4, p. 238-244, 1976 Dec.
- Gries, David (ed.); *Programming Methodology*, Springer-Verlag, New York, 1978.
- Gries, David; "Educating the Programmer: Notation, Proofs and the Development of Programs", in Lavington, Simon (ed.); *Information Processing 80 — Proceedings of IFIP Congress 80, Tokyo, Japan, October 6-9, 1980, Melbourne, Australia, October 14-17, 1980*, p. 935-944, North-Holland, Amsterdam, 1980.
- Gries, David; Levin, Gary; "Assignment and Procedure Call Proof Rules", *TOPLAS*, Vol. 2, No. 4, p. 564-579, 1980 Oct.
- Gries, David; *The Science of Programming*, Springer-Verlag, New York, 1981.
- Gries, David; "Calculation and Discrimination: A More Effective Curriculum", *CACM*, Vol. 34, No. 3, p. 45-55, 1991 March.
- Gries, David; "Lectures on Data Refinement", in Broy, Manfred (ed.); *Programming and Mathematical Method*, p. 213-244, Springer-Verlag, Berlin, 1992.
- Gruman, Galen; "Software safety focus of new British standard", *IEEE Software*, Vol. 6, No. 3, p. 95-96, 1989 May.
- Gruman, Galen; "ICSE assesses the state of software engineering", *IEEE Software*, Vol. 6, No. 4, p. 109-111, 1989 July.

- Gruman, Galen; "IFIP participants debate programming approaches", *IEEE Software*, Vol. 6, No. 6, p. 75-76, 1989 Nov.
- Gruman, Galen; "Major changes in federal software policy urged", *IEEE Software*, Vol. 6, No. 6, p. 78-80, 1989 Nov.
- Gruman, Galen (ed.); "Lessons from Patriot missile software effort", *IEEE Software*, Vol. 8, No. 3, p. 105+108, 1991 May.
- Guaspari, David; Marceau, Carla; Polak, Wolfgang; "Formal Verification of Ada Programs", *IEEE TSE*, Vol. 16, No. 9, p. 1058-1075, 1990 Sept.
- Guillemin, Ernst A.; *Introductory Circuit Theory*, John Wiley & Sons, New York, 1955.
- Gutttag, John; Horning, J.J.; "Formal Specification As a Design Tool", in *Conference Record of the Seventh Annual ACM Symposium on Principles of Programming Languages, Las Vegas, Nevada, January 28-30, 1980*, p. 251-261, ACM, New York, 1980.
- Halbwachs, Nicolas; Lagnier, Fabienne; Ratel, Christophe; "Programming and Verifying Real-Time Systems by Means of the Synchronous Data-Flow Language LUSTRE", *IEEE TSE*, Vol. 18, No. 9, p. 785-793, 1992 Sept.
- Hall, A.; "Using Z as a Specification Calculus for Object-Oriented Systems", in Bjørner, D.; Hoare, C.A.R.; Langmaack, H. (eds.); *VDM '90, VDM and Z — Formal Methods in Software Development, Third International Symposium of VDM Europe, Kiel, FRG, April 17-21, 1990, Proceedings*, LNCS 428, p. 290-318, Springer-Verlag, Berlin, 1990.
- Hall, Anthony; "Seven Myths of Formal Methods", *IEEE Software*, Vol. 7, No. 5, p. 11-19, 1990 Sept.
- Hamer, Ute; Hörcher, Hans-Martin; Peleska, Jan; *Safer Software — an Introduction into Formal Software Engineering*, internal report, DST Deutsche System-Technik GmbH, Kiel, 1993 May 3.
- Hamming, R.W.; book review of Fejer, Peter A.; Simovici, Dan A.; *Mathematical Foundations of Computer Science, Volume 1*, in *IEEE Computer*, Vol. 25, No. 1, p. 134-135, 1992 Jan. Related letter to the editor and reviewer's reply in *IEEE Computer*, Vol. 25, No. 4, p. 4, 1992 April.
- Hantler, Sidney L.; King, James C.; "An Introduction to Proving the Correctness of Programs", *ACM Computing Surveys*, Vol. 8, No. 3, p. 331-353, 1976 Sept.
- Harrison, Michael A.; *Introduction to Switching and Automata Theory*, McGraw-Hill, New York, 1965.
- Harry, Andrew; *Introduction to formal methods*, internal report, National Physical Laboratory, Teddington, Middlesex, U.K., publication forthcoming.
- Hauff, Harald; Schröder, Klaus-Werner; Ullmann, Markus; *Formales Software-Engineering am Beispiel der formalen Spezifikationsprache VSE-SL*, interner Bericht des Bundesamts für Sicherheit in der Informationstechnik und Vortrag im Seminar "Zuverlässige Software durch Formales Software-Engineering", 1993 Mai 10.
- Hayes, Brian; "The Discovery of Debugging", *The Sciences*, p. 10-13, 1993 July-August.
- He, Xudong; Ding, Yingjia; "A Temporal Logic Approach for Analyzing Safety Properties of Predicate Transition Nets", in van Leeuwen, Jan (ed.); *Algorithms, Software, Architecture: Information Processing 92 — Proceedings of the IFIP 12th World Computer Congress, Madrid, Spain, 7-11 September 1992, Volume 1*, p. 127-133, North-Holland, Amsterdam, 1992.

- Hegner, Eric C.R.; *The Logic of Programming*, Prentice/Hall International, Englewood Cliffs, N.J., 1984.
- Heisel, Maritta; Weber-Wulff, Debora; "Korrekte Software: Nur eine Illusion?", in Reichel, Horst (Hrsg.); *Informatik — Wirtschaft — Gesellschaft, 23. GI-Jahrestagung, Dresden, 27. September - 1. Oktober 1993*, S. 240, Springer-Verlag, Berlin, 1993.
- Henhapl, Wolfgang; Letschert, Thomas; "VDM — Vienna Development Method", *Informationstechnik ü*, 29. Jahrgang, Heft 4, S. 202-210, 1987.
- Herzog, Otthein; Reisig, Wolfgang; Valk, Rüdiger; "Petri-Netze: ein Abriß ihrer Grundlagen und Anwendungen", *Informatik-Spektrum*, Band 7, Heft 1, S. 20-27, 1984 Feb.
- Hessischer Landtag; "Gesetz über die Errichtung einer Ingenieurkammer und über die Berufsordnung der Beratenden Ingenieure in Hessen (Ingenieurkammergesetz — IngKammG) Vom 30. September 1986", S. 281-286, *Gesetz- und Verordnungsblatt für das Land Hessen, Teil I*, 1986 Nr. 22, 1986 Oktober 7.
- Hessischer Landtag; "Gesetz zum Schutze der Berufsbezeichnung "Ingenieur" (Ingenieurgesetz — IngG) Vom 15. Juli 1970", S. 407-408, *Gesetz- und Verordnungsblatt für das Land Hessen*, 1970 Juli 21.
- Hevner, Alan R.; Becker, Shirley A.; Pedowitz, Lenard B.; "Integrated CASE for Cleanroom Development", *IEEE Software*, Vol. 9, No. 2, p. 69-76, 1992 March.
- Higman, Bryan; *A Comparative Study of Programming Languages*, Macdonald and Jane's, London, 1977.
- Hoare, C.A.R.; "An Axiomatic Basis for Computer Programming", *CACM*, Vol. 12, No. 10, p. 576-580+583, 1969 Oct.
- Hoare, C.A.R.; "Proof of Correctness of Data Representations", *Acta Informatica*, Vol. 1, p. 271-281, 1972.
- Hoare, C.A.R.; Wirth, N.; "An Axiomatic Definition of the Programming Language PASCAL", *Acta Informatica*, Vol. 2, p. 335-355, 1973.
- Hoare, C.A.R.; "Monitors: An operating system structuring concept", *CACM*, Vol. 17, No. 10, p. 549-557, 1974 Oct.
- Hoare, C.A.R.; "The Engineering of Software: A Startling Contradiction", in Gries, David (ed.); *Programming Methodology*, p. 37-41, Springer-Verlag, New York, 1978.
- Hoare, C.A.R.; *A Calculus of Total Correctness for Communicating Processes*, Technical Monograph PRG-23, Oxford University Computing Laboratory, 1981 April.
- Hoare, C.A.R.; *Programming is an Engineering Profession*, Technical Monograph PRG-27, Oxford University Computing Laboratory, 1982 May.
- Hoare, C.A.R.; "Programming: Sorcery or Science?", *IEEE Software*, Vol. 1, No. 2, p. 5-16, 1984 April.
- Hoare, C.A.R.; *Communicating Sequential Processes*, Prentice/Hall International, Englewood Cliffs, N.J., 1985.
- Hoare, C.A.R.; Hayes, I.J.; Jifeng, He; Morgan, C.C.; Roscoe, A.W.; Sanders, J.W.; Sorensen, I.H.; Spivey, J.M.; Sufrin, B.A.; "Laws of Programming", *CACM*, Vol. 30, No. 8, p. 672-686, 1987 Aug. Corrigenda, *CACM*, Vol. 30, No. 9, p. 770, 1987 Sept.
- Hoare, C.A.R.; "An Overview of Some Formal Methods for Program Design", *IEEE Computer*, Vol. 20, No. 9, p. 85-91, 1987 Sept.
- Hoare, C.A.R.; Jones, C.B. (ed.); *Essays in Computing Science*, Prentice Hall, New York, 1989.

- Hoare, C.A.R.; Hayes, I.J.; Jifeng, He; Morgan, C.C., Roscoe, A.W.; Sanders, J.W.; Sorenson, I.H.; Spivey, J.M.; Sufirin, B.A.; "Laws of Programming", in Broy, Manfred (ed.); *Programming and Mathematical Method*, p. 95-122, Springer-Verlag, Berlin, 1992.
- Hoare, C.A.R.; Jifeng, He; "Refinement Algebra Proves Correctness of Compilation", in Broy, Manfred (ed.); *Programming and Mathematical Method*, p. 245-269, Springer-Verlag, Berlin, 1992.
- Hoare, C.A.R.; Page, I.; "Hardware and Software: The Closing Gap", in Gutknecht, Jürg (ed.); *Programming Languages and System Architectures, International Conference, Zurich, Switzerland, March 2-4, 1994, Proceedings*, LNCS 782, p. 49-68, Springer-Verlag, Berlin, 1994.
- Hoffmann, Hans-Jürgen; *Grundzüge der Informatik II, Begleittext zu einer Vorlesung im SS 1990*, PU1R5/90, Fachgebiet Programmiersprachen und Übersetzer, Fachbereich Informatik, Technische Hochschule Darmstadt, 1990 April-Juli.
- Hoffmann, Hans-Jürgen; Fuhr, N.; Snelting, G.; *Prüfungsklausur Vordiplom Informatik A, SS 1990*, Fachbereich Informatik, Technische Hochschule Darmstadt, 1990 Okt. 1.
- Hogg, John; "Object-Oriented Formal Methods", *ACM OOPS Messenger*, Vol. 3, No. 2, p. 6, 1992 April.
- Hogg, John; Lea, Doug; Wills, Alan; deChampeaux, Dennis; Holt, Richard; "The Geneva Convention On The Treatment of Object Aliasing", *ACM OOPS Messenger*, Vol. 3, No. 2, p. 11-16, 1992 April.
- Hohlfeld, Bernhard; *Zur Verifikation von modular zerlegten Programmen*, Dissertation (Dr.rer.nat.), Fachbereich Informatik, Universität Kaiserslautern, 1988 Feb. 5.
- Hohlfeld, Bernhard; Struckmann, Werner; *Einführung in die Programmverifikation*, Reihe Informatik, Band 88, B.I.-Wissenschaftsverlag, Mannheim, 1992.
- Holt, Richard; deChampeaux, Dennis; "A Framework for Using Formal Methods In Object-Oriented Software Development", *ACM OOPS Messenger*, Vol. 3, No. 2, p. 9-10, 1992 April.
- Huang, J.C.; "A New Verification Rule and Its Applications", *IEEE TSE*, Vol. SE-6, No. 5, p. 480-484, 1980 Sept.
- IEC/SC 65A, Working Group 9; *Software for computers in the application of industrial safety-related systems (draft standard)*, International Electrotechnical Commission Committee Document Number SC 65A/WG 9/39, 1989 March.
- IEC/SC 65A, Working Group 9; *Software for computers in the application of industrial safety-related systems*, International Electrotechnical Commission Committee Draft Reference number 65A(Secretariat)122, 1991 Nov.
- IEEE; *Software Engineering Standards*, IEEE, New York, 1987.
- IEEE; *IEEE Standard for Binary Floating-Point Arithmetic*, ANSI/IEEE Std 754-1985, IEEE, New York, 1985, in *ACM SIGPLAN Notices*, Vol. 22, No. 2, p. 9-25, 1987 Feb.
- IEEE; *IEEE Standard for Radix-Independent Floating-Point Arithmetic*, ANSI/IEEE Std 854-1987, IEEE, New York, 1987 October 5.
- IEEE; *Software Engineering Standards: Third Edition*, IEEE, New York, 1989 May 15.
- IEEE; "IEEE Code of Ethics", *IEEE Institute*, 1990 Oct.
- Illingworth, Valerie (ed.); *Dictionary of Computing*, Third Edition, Oxford University Press, Oxford, 1991.

- Ince, Darrel; "Software requirements analysis, formal methods and software prototyping", in Ince, Darrel (ed.); *Software Quality and Reliability*, p. 96-101, Chapman & Hall, London, 1991.
- Ince, D.C.; "Arrays and Pointers Considered Harmful", *ACM SIGPLAN Notices*, Vol. 27, No. 1, p. 99-104, 1992 Jan.
- Jacobs, Dean; Gries, David; "General Correctness: A Unification of Partial and Total Correctness", *Acta Informatica*, Vol. 22, Fasc. 1, p. 67-83, 1985 April.
- James, Robert C.; Beckenbach, Edwin F. (eds.); *Mathematics Dictionary*, James & James third multilingual edition, Van Nostrand, Princeton, New Jersey, 1968.
- Janicki, Ryszard; *Towards a Formal Semantics of Tables*, CRL Report No. 264, Communications Research Laboratory, Faculty of Engineering, McMaster University, Hamilton, Ontario, Canada, 1993 Sept.
- Jeng, Jun-Jang; Cheng, Betty H.C.; "Using Formal Methods to Construct a Software Component Library", in Sommerville, Ian; Paul, Manfred (eds.); *Software Engineering - ESEC '93, 4th European Software Conference, Garmisch-Partenkirchen, Germany, September 1993, Proceedings*, LNCS 717, p. 397-417, Springer-Verlag, Berlin, 1993.
- Jensen, Kathleen; Wirth, Niklaus; *Pascal User Manual and Report*, Second Corrected Reprint of the Second Edition, Springer-Verlag, New York, 1978.
- Jesty, P.H.; Buckley, T.F.; West, M.M.; "The development of safe advanced road transport telematic software", *Microprocessors and Microsystems*, Vol. 17, No. 1, p. 37-46, 1993 Jan.-Feb.
- Jones, Cliff B.; *Software Development: A Rigorous Approach*, Prentice/Hall International, Englewood Cliffs, N.J., 1980.
- Jones, Cliff B.; *Systematic Software Development using VDM*, Prentice/Hall International, Englewood Cliffs, N.J., 1986.
- Joyce, Ed; "Software Bugs: A Matter of Life and Liability", *Datamation*, p. 88-92, 1987 May 15.
- Joyce, Edward J.; "Is Error-Free Software Achievable?", *Datamation*, p. 53+56, 1989 Feb. 15.
- Juliff, Peter; Buchbesprechung über [Baber; *Error-free Software*, 1991], *The Australian Computer Journal*, 1993 Jan. 18.
- Jüllig, Richard K.; "Applying Formal Software Synthesis", *IEEE Software*, Vol. 10, No. 3, p. 11-22, 1993 May.
- Kaldewaj, Anne; *Programming: The Derivation of Algorithms*, Prentice Hall, New York, 1990.
- Kaps, Carola; "Stundenlang kein Klingeln", *Frankfurter Allgemeine Zeitung*, S. 12, 1991 Juni 28.
- Käufel, Thomas; "The Simplifier of the Program Verifier 'Tatzelwurm'", paper presented at the Österreichische Artificial Intelligence Tagung, Vienna, 1985 Sept.
- Käufel, Thomas; *Simplification and Decision of Systems of Linear Inequalities over the Integers*, interner Bericht des Instituts für Logik, Komplexität und Deduktionssysteme, Universität Karlsruhe, Fakultät für Informatik, 1988 Sept.
- Käufel, Thomas; *Program Verifier "Tatzelwurm": A Sketch*, internes Arbeitspapier des Instituts für Logik, Komplexität und Deduktionssysteme, Universität Karlsruhe, undatiert, ca. 1989.

- Kearney, P.; Staples, J.; Abbas, A.; "Functional Verification of Hard Real-Time Programs", in van Leeuwen, Jan (ed.); *Algorithms, Software, Architecture: Information Processing 92 — Proceedings of the IFIP 12th World Computer Congress, Madrid, Spain, 7-11 September 1992, Volume 1*, p. 113-119, North-Holland, Amsterdam, 1992.
- Keenan, Joseph H.; *Thermodynamics*, John Wiley & Sons, New York, 1953.
- Keil-Slawik, Reinhard; "Die Gestaltung des Unsichtbaren", *Computer Magazin*, S. 39-41, 1988 Juli-Aug.
- Kemmerer, Richard A.; "Testing Formal Specifications and the INATEST System", *GI-SE Softwaretechnik-Trends, Mitteilungen der Fachgruppe 'Software Engineering'*, S. 108-115, 1986 Juni.
- Kemmerer, Richard A.; "Integrating Formal Methods into the Development Process", *IEEE Software*, Vol. 7, No. 5, p. 37-50, 1990 Sept.
- Kershaw, John; "The special problems of military systems", *Microprocessors and Microsystems*, Vol. 17, No. 1, p. 25-30, 1993 Jan.-Feb.
- Kfoury, A.J.; Moll, Robert N.; Arbib, Michael A.; *A Programming Approach to Computability*, Springer-Verlag, New York, 1982.
- Khanna, S.; "Logic Programming for Software Verification and Testing", *The Computer Journal*, Vol. 34, No. 4, p. 350-357, 1991.
- Kimm, Reinhold; Koch, Wilfried; Simonsmeier, Werner; Tontsch, Friedrich; *Einführung in Software Engineering*, Walter de Gruyter, Berlin, 1979.
- Kley, Adolf; "Dynamische Systeme und kommunizierende Prozesse — eine Analogiebehandlung", *Informatik-Spektrum*, Band 9, Heft 1, S. 29-38, 1986 Feb.
- Kline, Morris; *Mathematics: The Loss of Certainty*, Oxford University Press, New York, 1980.
- Klotz, Karlhorst; "Was Prüfsiegel garantieren: Drum prüfe, wer sich ewig bindet", *Chip*, Nr. 6, S. 152-154, 1993 Juni.
- Klusmeier, Stefan; "Ohne Fehl und Tadel: Formale Methoden helfen Software-Fehler vermeiden", *Frankfurter Allgemeine Zeitung*, S. B8, 1991 Okt. 21.
- Knight, John C.; Leveson, Nancy G.; "An Experimental Evaluation of the Assumption of Independence in Multiversion Programming", *IEEE TSE*, Vol. SE-12, No. 1, p. 96-109, 1986 Jan.
- Knuth, Donald E.; *The Art of Computer Programming, Volume 2, Seminumerical Algorithms*, Addison-Wesley, Reading, Massachusetts, 1969.
- Knuth, Donald E.; *The Art of Computer Programming, Volume 3, Sorting and Searching*, Addison-Wesley, Reading, Massachusetts, 1973.
- Knuth, Donald E.; *The Art of Computer Programming, Volume 1, Fundamental Algorithms*, Addison-Wesley, Reading, Massachusetts, second edition, 1978.
- Koomen, C.J.; "Towards a Scientific Discipline for System Engineering", Colloquium "Van Specificatie tot Implementatie", Centrum voor Wiskunde en Informatica (CWI), Amsterdam, 1983 Oct. 7.
- Kramer, J.; Cunningham, R.J.; "Invariants for Specifications", *Proceedings of the 4th International Conference on Software Engineering, Munich, West Germany, 17-19 September 1979*, p. 183-193, IEEE, New York, 1979.
- Kreitz, Christoph; Tzeras, Kostas; Theissen, W.; Misch, C.; *Klausur WS 93/94, Grundzüge der Informatik I*, Fachbereich Informatik, Technische Hochschule Darmstadt, 1993-94.

- Kumar, R.; Kropf, T.; Schneider, K.; "Bridging the gap between Hardware Verification and Design", in *Poster Session — 12th World Computer Congress, IFIP Congress '92*, p. 76, IFIP, Madrid, 1992 Sept. 7-11.
- Küpfmüller, Karl; *Einführung in die theoretische Elektrotechnik*, Springer-Verlag, Berlin, 1968.
- Lachenmayer, Peter; Schmitz, Lothar; "A Framework and Tools for Rigorously Documenting Smalltalk Programs", draft, 1991, of an unpublished contribution to the Workshop "Formal Methods", ECOOP '91, European Conference on Object-Oriented Programming, Geneva, Switzerland, (siehe auch Schmitz, Lothar), 1991 July.
- Lange, Danny B.; "A formal approach to hypertext using post-prototype formal specification", in Bjørner, D.; Hoare, C.A.R.; Langmaack, H. (eds.); *VDM '90, VDM and Z — Formal Methods in Software Development, Third International Symposium of VDM Europe, Kiel, FRG, April 17-21, 1990, Proceedings*, LNCS 428, p. 99-121, Springer-Verlag, Berlin, 1990.
- Laprie, J.-C.; "Dependability: a unifying concept for reliable, safe, secure computing", in van Leeuwen, Jan (ed.); *Algorithms, Software, Architecture: Information Processing 92 — Proceedings of the IFIP 12th World Computer Congress, Madrid, Spain, 7-11 September 1992, Volume 1*, p. 585-593, North-Holland, Amsterdam, 1992.
- Leavens, Gary T.; "Modular Specification and Verification of Object-Oriented Programs", *IEEE Software*, Vol. 8, No. 4, p. 72-80, 1991 July.
- Lehman, M.M.; Belady, L.A. (eds.); *Program Evolution — Processes of Software Change*, Academic Press, London, 1985.
- Lehman, M.M.; "Uncertainty in Computer Application and its Control Through the Engineering of Software", draft, 1989 March 28.
- Lehman, M.M.; "Software Engineering, the Software Process and Their Support", Research Report 91/1, Department of Computing, Imperial College of Science and Technology, London, 1991 Jan. 31.
- Lenders, Patrick M.; "Distributed Computing with Single Read-Single Write Variables", *IEEE TSE*, Vol. 15, No. 5, p. 569-574, 1989 May.
- Leung, Wu Hon; Ramamoorthy, C.V.; "An Approach to Formal Specification of Control Modules", *IEEE TSE*, Vol. SE-6, No. 5, p. 485-489, 1980 Sept.
- Leveson, Nancy G.; *Software Safety: Why, What, and How*, Technical Report 86-04, Information and Computer Science, University of California, Irvine, 1986 Feb.
- Leveson, Nancy G.; "Software Fault Tolerance in Safety-Critical Applications", draft, undated, ca. 1987.
- Leveson, Nancy G.; "The Challenge of Building Process-Control Software", *IEEE Software*, Vol. 7, No. 6, p. 55-62, 1990 Nov.
- Leveson, Nancy G.; Turner, Clark S.; "An Investigation of the Therac-25 Accidents", *IEEE Computer*, Vol. 26, No. 7, p. 18-41 (see also summary on p. 7), 1993 July.
- Levi, Giorgio; "New Research Directions in Logic Specification Languages", in Kugler, H.-J. (ed.); *Information Processing 86 — Proceedings of the IFIP 10th World Computer Congress, Dublin, Ireland, September 1-5, 1986*, p. 1005-1008, North-Holland, Amsterdam, 1986.
- Lewis, Harry R.; Papadimitriou, Christos H.; *Elements of the Theory of Computation*, Prentice-Hall, Englewood Cliffs, N.J., 1981.
- Lewis, Ted. G.; Oman, Paul; "The Challenge of Software Development", *IEEE Software*, Vol. 7, No. 6, p. 9-12, 1990 Nov.

- Lieberherr, K.; Holland, I.; Riel, A.; "Object-Oriented Programming: An Objective Sense of Style", in Meyrowitz, Norman (ed.); *ACM SIGPLAN 3rd Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 88)*, San Diego, CA, USA, 25-30 Sept. 1988, Special Issue of *SIGPLAN-Notices*, Vol. 23, No. 11, p. 323-334, 1988 Nov.
- Lin, Herbert; "The Software for Star Wars: An Achilles Heel?", *Technology Review*, Vol. 88, No. 5, p. 16-18, 1985 July.
- Lindeberg, Johan F.; "The Swedish State Railways' Experience with n-Version Programmed Systems", in Redmill, Felix; Anderson, Tom (eds.); *Directions in Safety-critical Systems, Proceedings of the First Safety-critical Systems Symposium, Bristol, 9-11 February 1993*, p. 36-42, Springer-Verlag, London, 1993.
- Linger, Richard C.; Mills, Harlan D.; Witt, Bernard I.; *Structured Programming: Theory and Practice*, Addison-Wesley, Reading, Massachusetts, 1979.
- Littlewood, Bev; Miller, Douglas R.; "Conceptual Modeling of Coincident Failures in Multiversion Software", *IEEE TSE*, Vol. 15, No. 12, p. 1596-1614, 1989 Dec.
- Littlewood, Bev; "Software reliability modelling", in McDermid, John A. (ed.); *Software Engineer's Reference Book*, p. 31/1-12, Butterworth-Heinemann, Oxford, 1991.
- Littlewood, Bev; Strigini, Lorenzo; "The Risks of Software", *Scientific American*, p. 38-43, 1992 Nov.
- Littlewood, Bev; "The Need for Evidence from Disparate Sources to Evaluate Software Safety", in Redmill, Felix; Anderson, Tom (eds.); *Directions in Safety-critical Systems, Proceedings of the First Safety-critical Systems Symposium, Bristol, 9-11 February 1993*, p. 217-231, Springer-Verlag, London, 1993.
- Loeckx, Jacques; Sieber, Kurt; *The Foundations of Program Verification*, B.G. Teubner, Stuttgart, und John Wiley & Sons, Chichester, 1984.
- Loomes, Martin; "Logics and proofs of correctness", in McDermid, John A. (ed.); *Software Engineer's Reference Book*, p. 2/1-16, Butterworth-Heinemann, Oxford, 1991.
- Lucas, Peter; "VDM: Origins, Hopes, and Achievements", in Bjørner, D.; Jones, C.B.; Mac, an Airchinnigh, M.; Neuhold, E.J. (eds.); *VDM '87, VDM — A Formal Method at Work, VDM-Europe Symposium 1987, Brussels, Belgium, March 23-26, 1987, Proceedings*, LNCS 252, p. 1-18, Springer-Verlag, Berlin, 1987.
- Macro, Allen; Buxton, John; *The Craft of Software Engineering*, Addison-Wesley, Wokingham, England, 1987.
- Mahony, Brendan P.; Hayes, Ian J.; "A Case-Study in Timed Refinement: A Mine Pump", *IEEE TSE*, Vol. 18, No. 9, p. 817-826, 1992 Sept.
- Maibaum, T.S.E.; Veloso, Paulo A.S.; Sadler, M.R.; "A Theory of Abstract Data Types for Program Development: Bridging the Gap?", in Ehrig, Hartmut; Floyd, Christiane; Nivat, Maurice; Thatcher, James (eds.); *Formal Methods and Software Development — Proceedings of the International Joint Conference on Theory and Practice of Software Development (TAPSOFT)*, Berlin, March 25-29, 1985, Volume 2: *Colloquium on Software Engineering (CSE)*, LNCS 186, p. 214-230, Springer-Verlag, Berlin, 1985.
- Manna, Zohar; *Mathematical Theory of Computation*, McGraw-Hill Kogakusha, Tokyo, 1974.
- Manna, Zohar; Waldinger, Richard; "Fundamentals of Deductive Program Synthesis", *IEEE TSE*, Vol. 18, No. 8, p. 674-704, 1992 Aug.

- Mark, William; Tyler, Sherman W.; McGuire, James; Schlossberg, Jon L.; "Commitment-Based Software Development", *IEEE TSE*, Vol. 18, No. 10, p. 870-885, 1992 Oct.
- Martin, Alain J.; "A General Proof Rule for Procedures in Predicate Transformer Semantics", *Acta Informatica*, Vol. 20, p. 301-313, 1983.
- Martin, Alain J.; "Tomorrow's Digital Hardware will be Asynchronous and Verified", in van Leeuwen, Jan (ed.); *Algorithms, Software, Architecture: Information Processing 92 — Proceedings of the IFIP 12th World Computer Congress, Madrid, Spain, 7-11 September 1992, Volume I*, p. 684-695, North-Holland, Amsterdam, 1992.
- Martin, James; *System Design from Provably Correct Constructs*, Prentice-Hall, Englewood Cliffs, N.J., 1985.
- McCarthy, J.; "Towards a Mathematical Science of Computation", in *Proceedings of the IFIP Congress '62*, p. 21-28, North-Holland, Amsterdam, 1962.
- McCarthy, John; "A Basis for a Mathematical Theory of Computation", in *Computer programming and formal systems*, p. 33-70, North-Holland, Amsterdam, 1963.
- McCarthy, J.; position statement for panel session P3.2, "Tough Nuts of Computer Science", in Mason, R.E.A. (ed.); *Information Processing 83: panel discussions*, IFIP 9th World Computer Congress, Paris, France, 1983 Sept. 22, p. 26, North-Holland, Amsterdam, 1983.
- McDermid, John A. (ed.); *Software Engineer's Reference Book*, Butterworth-Heinemann, Oxford, 1991.
- McGowan, Clement L.; Kelly, John R.; *Top-Down Structured Programming Techniques*, Petrocelli/Charter, New York, 1975.
- Meertens, Lambert; "Algorithmics: An Algebraic Approach to Algorithm Specification and Construction", Syllabus Colloquium "Van Specificatie tot Implementatie", Centrum voor Wiskunde en Informatica (CWI), Amsterdam, 1983 Okt. 21.
- Mellor, Peter; "10-9 and all that ... The non-certification of flight-critical software", *SCSC Newsletter*, Vol. 3, No. 1, p. 4-5, 1993 Sept.
- Meriam, J.L.; *Mechanics — Part I, Statics*, John Wiley & Sons, New York, 1953.
- Meriam, J.L.; *Mechanics — Part II, Dynamics*, John Wiley & Sons, New York, 1954.
- Meyer, Bertrand; *Object-oriented Software Construction*, Prentice-Hall, New York, 1988.
- Meyer, Bertrand; *Programming as Contracting*, Version 2, Report TR-EI-12/CO, Interactive Software Engineering Inc., 1988 March 5.
- Meyer, Bertrand; *Introduction to the Theory of Programming Languages*, Prentice Hall, New York, 1991.
- Meyer, Bertrand; *Eiffel: The Language*, Prentice Hall, New York, 1992.
- Meyer, Bertrand; "Applying 'Design by Contract'", *IEEE Computer*, Vol. 25, No. 10, p. 40-51, 1992 Oct.
- Mili, Ali; "The Bottom Up Analysis of While Statements: Strongest Invariant Functions", in Mason, R.E.A. (ed.); *Information Processing 83 — Proceedings of the IFIP 9th World Computer Congress, Paris, France, September 19-23, 1983*, p. 339-343, North-Holland, Amsterdam, 1983.
- Mili, Ali; Desharnais, Jules; Gagné, Jean-Raymond; "Strongest Invariant Functions: Their Use in the Systematic Analysis of While Statements", *Acta Informatica*, Vol. 22, p. 47-66, 1985.

- Mili, Ali; Desharnais, Jules; Gagné, Jean Raymond; "Formal Models of Stepwise Refinement of Programs", *ACM Computing Surveys*, Vol. 18, No. 3, p. 231-276, 1986 Sept.
- Mills, Harlan D.; "The New Math of Computer Programming", *CACM*, Vol. 18, No. 1, p. 43-48, 1975 Jan.
- Mills, Harlan D.; "How to Write Correct Programs and Know It", *SIGPLAN Notices*, Vol. 10, No. 6, p. 363-370, 1975 June.
- Mills, Harlan D.; "Function Semantics for Sequential Programs", *Information Processing 80 — Proceedings of the IFIP 8th World Computer Congress 1980*, p. 241-250, North-Holland, Amsterdam, 1980.
- Mills, Harlan D.; "How to Make Exceptional Performance Dependable and Manageable in Software Engineering", in *Proceedings COMPSAC 80, The IEEE Computer Society's Fourth International Computer Software and Applications Conference, October 27-31, 1980*, p. 19-23, IEEE, New York, 1980.
- Mills, Harlan D.; Linger, Richard C.; "Data Structured Programming: Program Design without Arrays and Pointers", *IEEE TSE*, Vol. SE-12, No. 2, p. 192-197, 1986 Feb.
- Mills, Harlan D.; "Structured Programming: Retrospect and Prospect", *IEEE Software*, Vol. 3, No. 6, p. 58-66, 1986 Nov.
- Mills, Harlan D.; Dyer, Michael; Linger, Richard C.; "Cleanroom Software Engineering", *IEEE Software*, Vol. 4, No. 5, p. 19-25, 1987 Sept.
- Mills, Harlan D.; Basili, Victor R.; Gannon, John D.; Hamlet, Richard G.; "Mathematical Principles for a First Course in Software Engineering", *IEEE TSE*, Vol. 15, No. 5, p. 550-559, 1989 May.
- Mills, Harlan D.; "Engineering Software under Statistical Quality Control", Vortrag, *Spektrum 90 Kongreß für Software-Qualitätssicherung*, DECCollege, München, 1990 Dez. 5-7.
- Mills, Harlan D.; "A Rebuttal of 'A Critical Analysis of the Cleanroom Technique'", *The Software Practitioner*, Vol. 1, No. 5, p. 1-2+4+7, 1991 Sept.-Oct.
- Milner, Robin; "Process Constructors and Interpretations", in Kugler, H.-J. (ed.); *Information Processing 86 — Proceedings of the IFIP 10th World Computer Congress, Dublin, Ireland, September 1-5, 1986*, p. 507-514, North-Holland, Amsterdam, 1986.
- Milner, Robin; *Communication and Concurrency*, Prentice Hall, New York, 1989.
- Ministry of Defence (U.K.); *Interim Defence Standard 00-55, The Procurement of Safety Critical Software in Defence Equipment, Part 1: Requirements and Part 2: Guidance*, Ministry of Defence, Directorate of Standardization, Glasgow, 1991 April 5. Also in Wichmann, Brian A. (ed.); *Software in Safety-related Systems*, John Wiley & Sons, Chichester, 1992. See also earlier draft: *Interim Defence Standard 00-55 (Draft), Requirements for the Procurement of Safety Critical Software in Defence Equipment*, 1989 May 9.
- Minkowitz, Cydney; Henderson, Peter; "A formal description of object-oriented programming using VDM", in Bjørner, D.; Jones, C.B.; Mac an Airchinnigh, M.; Neuhold, E.J. (eds.); *VDM '87, VDM — A Formal Method at Work, VDM-Europe Symposium 1987, Brussels, Belgium, March 23-26, 1987, Proceedings*, LNCS 252, p. 237-259, Springer-Verlag, Berlin, 1987.
- Minsky, Marvin; *Computation: Finite and Infinite State Machines*, Prentice-Hall, Englewood Cliffs, N.J., 1967.

- Miriyala, Kanth; Harandi, Mehdi T.; "Automatic Derivation of Formal Software Specifications From Informal Descriptions", *IEEE TSE*, Vol. 17, No. 10, p. 1126-1142, 1991 Oct.
- Monahan, Brian; Shaw, Roger; "Model-based specifications", in McDermid, John A. (ed.); *Software Engineer's Reference Book*, p. 21/1-37, Butterworth-Heinemann, Oxford, 1991.
- Morgan, Carroll; *Programming from Specifications*, Prentice Hall, New York, 1990.
- Moriconi, Mark; Winkler, Timothy C.; "Approximate Reasoning About the Semantic Effects of Program Changes", *IEEE TSE*, Vol. 16, No. 9, p. 980-992, 1990 Sept.
- Müller-Scholz, Wolfgang; Schönbeck, Knut; "Software — das weit unterschätzte Risiko", *Capital*, Nr. 6/90, S. 206-208+210, 1990.
- Musa, John D.; Everett, William W.; "Software-Reliability Engineering: Technology for the 1990s", *IEEE Software*, Vol. 7, No. 6, p. 36-43, 1990 Nov.
- Musgrave, G.; "Formal Methods For Design", in van Leeuwen, Jan (ed.); *Algorithms, Software, Architecture: Information Processing 92 — Proceedings of the IFIP 12th World Computer Congress, Madrid, Spain, 7-11 September 1992, Volume 1*, p. 710, North-Holland, Amsterdam, 1992.
- Myers, Glenford J.; *Reliable Software through Composite Design*, Van Nostrand Reinhold, New York, 1975.
- Myers, Glenford J.; *The Art of Software Testing*, John Wiley & Sons, New York, 1979.
- Nahin, Paul J.; *Oliver Heaviside: Sage in Solitude*. IEEE Press, New York, 1988.
- Nakajima, Reiji; "Formal Development Methods — Their impacts on programmers and software houses", unpublished position paper distributed at panel session P2.3 of the IFIP 9th World Computer Congress, Paris, France, 1983 Sept. 23.
- Nakajo, Takeshi; Kume, Hitoshi; "A Case History Analysis of Software Error Cause-Effect Relationships", *IEEE TSE*, Vol. 17, No. 8, p. 830-838, 1991 Aug.
- Naumann, David A.; "On the Essence of Oberon", in Gutknecht, Jürg (ed.); *Programming Languages and System Architectures, International Conference, Zurich, Switzerland, March 2-4, 1994, Proceedings*, LNCS 782, p. 313-327, Springer-Verlag, Berlin, 1994.
- Naur, Peter (ed.); *Revised Report on the Algorithmic Language Algol 60*, Regnecentralen, Copenhagen, 1962.
- Naur, Peter; Randell, Brian (eds.); *Software Engineering, Report on a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7th to 11th October 1968*, Scientific Affairs Division, NATO, Brussels, 1969 Jan.
- Naur, Peter; "Formalization in Program Development", *BIT*, Vol. 22, p. 437-453, 1982.
- Naur, P.; "Intuition in Software Development", in Ehrig, Hartmut; Floyd, Christiane; Nivat, Maurice; Thatcher, James (eds.); *Formal Methods and Software Development — Proceedings of the International Joint Conference on Theory and Practice of Software Development (TAPSOFT), Berlin, March 25-29, 1985, Volume 2: Colloquium on Software Engineering (CSE)*, LNCS 186, p. 60-79, Springer-Verlag, Berlin, 1985.
- Neumann, Peter G. (moderator); "Risks to the Public in Computers and Related Systems", regular column in *ACM SIGSOFT Software Engineering Notes*, 1994 and earlier.
- Newton, Jr., George C.; Gould, Leonard A.; Kaiser, James F.; *Analytical Design of Linear Feedback Controls*, John Wiley & Sons, New York, 1957.

- Nievergelt, J.; "Schulbeispiele zur Rekursion", *Informatik-Spektrum*, Kolumne "Over-flow", Bd. 13, S. 106-108, 1990.
- Normenausschuß Informationsverarbeitungssysteme (NI) im DIN Deutsches Institut für Normung e.V.; *Mathematische Zeichen und Symbole der Schaltalgebra DIN 66000*, Beuth Verlag, Berlin, 1985 Nov.
- Norris, Mark; Rigby, Peter; *Software Engineering Explained*, John Wiley & Sons, Chichester, 1992.
- Olderog, Ernst-Rüdiger; "Systematic Derivation of Communicating Programs", in Broy, Manfred (ed.); *Programming and Mathematical Method*, p. 369-407, Springer-Verlag, Berlin, 1992.
- Oppelt, Winfried; *Kleines Handbuch technischer Regelvorgänge*, Verlag Chemie, Weinheim/Bergstr., 1972.
- Ott, Hans Jürgen; "Das "Ingenieurgemäße" am Software Engineering", *GI-SE-RE Softwaretechnik-Trends, Mitteilungen der Fachgruppen 'Software-Engineering' und 'Requirements-Engineering'*, Band 14, Heft 1, S. 31-37, 1994 Feb.
- Owicki, Susan; Gries, David; "An Axiomatic Proof Technique for Parallel Programs I", *Acta Informatica*, Vol. 6, p. 319-340, 1976.
- Palermo, Christopher J.; "Software Engineering Malpractice and Its Avoidance", in *Proceedings, Third International Symposium on Software Reliability Engineering, October 7-10, 1992, Research Triangle Park, North Carolina*, IEEE Cat. No. 92TH0486-1, p. 41-50, IEEE Computer Society Press, Los Alamitos, CA, 1992.
- Parnas, David Lorge; Clements, Paul C.; Weiss, David M.; "The Modular Structure of Complex Systems", *IEEE TSE*, Vol. SE-11, No. 3, p. 259-266, 1985 March.
- Parnas, David Lorge; *Software Aspects of Strategic Defense Systems*, Report DCS-47-IR, Department of Computer Science, University of Victoria, B.C., Canada, 1985 July.
- Parnas, D.L.; Weiss, D.M.; *Active Design Reviews: Principles and Practices*, NRL Report 8927, Naval Research Laboratory, Washington, D.C., 1985 Nov. 18.
- Parnas, David Lorge; van Schouwen, A. John; Kwan, Shu Po; *Evaluation Standards for Safety Critical Software*, Technical Report 88-220, Department of Computing and Information Science, Queen's University, Kingston, Ontario, Canada, 1988 May.
- Parnas, David Lorge; "Wann können wir Software trauen? Die Benutzung von Software in sicherheitskritischen Anwendungen", Vortrag, *Spektrum 88 Kongreß für Software-Qualitätssicherung*, DECollege, München, 1988 Nov. 9-11.
- Parnas, David Lorge; Wang, Yabo; *The Trace Assertion Method of Module Interface Specification*, Technical Report 89-261, Department of Computing and Information Science, Queen's University, Kingston, Ontario, Canada, 1989 Oct.
- Parnas, David Lorge; "Education for Computing Professionals", *IEEE Computer*, Vol. 23, No. 1, p. 17-22, 1990 Jan.
- Parnas, David Lorge; *Documentation of Communications Services and Protocols*, Technical Report 90-272, Department of Computing and Information Science, Queen's University, Kingston, Ontario, Canada, 1990 Feb.
- Parnas, David Lorge; Madey, Jan; *Functional Documentation for Computer Systems Engineering*, Technical Report 90-287, Telecommunications Research Institute of Ontario (TRIO), Department of Computing and Information Science, Queen's University, Kingston, Ontario, Canada, 1990 Sept.
- Parnas, David Lorge; Asmis, G.J.K.; Madey, Jan; *Assessment of Safety-Critical Software*, Technical Report 90-295, Telecommunications Research Institute of Ontario (TRIO),

- Department of Computing and Information Science, Queen's University, Kingston, Ontario, Canada, 1990 Dec.
- Parnas, David Lorge; "Qualitätssicherung für sicherheitskritische Software", Vortrag, *Spektrum 90 Kongreß für Software-Qualitätssicherung*, DECollege, München, 1990 Dez. 5-7.
- Parnas, David Lorge; "Functional Specifications for Old (and New) Software", Telecommunications Research Institute of Ontario, Queen's University at Kingston, Ontario, Canada, undated, 1990 or later.
- Parnas, David Lorge; Madey, Jan; *Functional Documentation for Computer Systems Engineering (Version 2)*, CRL Report No. 237, Communications Research Laboratory, Faculty of Engineering, McMaster University, Hamilton, Ontario, Canada, 1991 Sept.
- Parnas, David Lorge; *Predicate Logic for Software Engineering*, CRL Report No. 241, Communications Research Laboratory, Faculty of Engineering, McMaster University, Hamilton, Ontario, Canada, 1992 Feb.
- Parnas, David Lorge; *Tabular Representation of Relations*, CRL Report No. 260, Communications Research Laboratory, Faculty of Engineering, McMaster University, Hamilton, Ontario, Canada, 1992 Oct.
- Parnas, David Lorge; "Predicate Logic for Software Engineering", *IEEE TSE*, Vol. 19, No. 9, p. 856-862, 1993 Sept.
- Partsch, Helmut A.; *Specification and Transformation of Programs: A Formal Approach to Software Development*, Springer-Verlag, Berlin, 1990.
- Pedersen, Jan Storbak; *Software Development Using VDM: SEI Curriculum Module SEI-CM-16-1.1*, Software Engineering Institute, Carnegie Mellon University, (Internet ftp file cm16.ps in directory /pub/education at ftp.sei.cmu.edu), 1989 Dec.
- Peleska, Jan; "Design and verification of fault tolerant systems with CSP", *Distributed Computing*, Vol. 5, p. 95-106, 1991.
- Peleska, Jan; "Software-Sicherheit — Herausforderung der 90er Jahre", *ComputerMagazin*, S. 40-41, 1991 März.
- Peleska, Jan; *Formale Spezifikation generischer ITSEC-Funktionalitätsklassen*, interner Bericht, DST Deutsche System-Technik GmbH, Kiel, 1993 Feb. 8.
- Peleska, Jan; Hörcher, Hans-Martin; *Structured Methods, Z and DST-Fuzz — Provable Correctness for safety-critical Software Systems*, internal report, DST Deutsche System-Technik GmbH, Kiel, 1993 April 13.
- Perry, Dewayne E.; Stieg, Carol S.; "Software Faults in Evolving a Large, Real-Time System: a Case Study", in Sommerville, Ian; Paul, Manfred (eds.); *Software Engineering — ESEC '93, 4th European Software Conference, Garmisch-Partenkirchen, Germany, September 1993, Proceedings*, LNCS 717, p. 48-67, Springer-Verlag, Berlin, 1993.
- Peterson, J.L.; *Petri Net Theory and the Modeling of Systems*, Prentice-Hall, Englewood Cliffs, NJ, 1981.
- Petrone, Giovanna; Petrone, Luigi; "Program Development and Documentation by Informal Transformations and Derivations", in Ehrig, Hartmut; Floyd, Christiane; Nivat, Maurice; Thatcher, James (eds.); *Formal Methods and Software Development — Proceedings of the International Joint Conference on Theory and Practice of Software Development (TAPSOFT), Berlin, March 25-29, 1985, Volume 2: Colloquium on Software Engineering (CSE)*, LNCS 186, p. 231-245, Springer-Verlag, Berlin, 1985.

- Place, Patrick R.H.; Kang, Kyo C.; *Safety-Critical Software: Status Report and Annotated Bibliography*, Technical Report CMU/SEI-92-TR-5 ESC-TR-93-182, Carnegie Mellon University, Software Engineering Institute, Pittsburgh, Pennsylvania, 1993 June.
- Plenderleith, William T.; *Software Engineering for Signal-Processing Systems: A Case Study*, Technical Report 90-291, Telecommunications Research Institute of Ontario (TRIO), Department of Computing and Information Science, Queen's University, Kingston, Ontario, Canada, 1990 Nov.
- Pnueli, A.; "Specification and Development of Reactive Systems", in Kugler, H.-J. (ed.); *Information Processing 86 — Proceedings of the IFIP 10th World Computer Congress, Dublin, Ireland, September 1-5, 1986*, p. 845-858, North-Holland, Amsterdam, 1986.
- Pnueli, Amir; Rosner, Roni; "A Framework for the Synthesis of Reactive Modules", in Vogt, F.H. (ed.); *CONCURRENCY 88 — International Conference on Concurrency, Hamburg, FRG, October 18-19, 1988, Proceedings*, LNCS 335, p. 4-17, Springer-Verlag, Berlin, 1988.
- Poore, J.H.; Mills, Harlan D.; Mutchler, David; "Planning and Certifying Software System Reliability", *IEEE Software*, Vol. 10, No. 1, p. 88-99, 1993 Jan.
- Potocki de Montalk, J.P.; "Computer software in civil aircraft", *Microprocessors and Microsystems*, Vol. 17, No. 1, p. 17-23, 1993 Jan.-Feb.
- Potter, Ben; Sinclair, Jane; Till, David; *An Introduction to Formal Specification and Z*, Prentice Hall, New York, 1991.
- Pratt, Terrence W.; "Control Computations and the Design of Loop Control Structures", *IEEE TSE*, Vol. SE-4, No. 2, p. 81-89, 1978 March.
- Redmill, Felix; Anderson, Tom (eds.); *Directions in Safety-critical Systems, Proceedings of the First Safety-critical Systems Symposium, Bristol, 9-11 February 1993*, Springer-Verlag, London, 1993.
- Reisig, Wolfgang; *Petri Nets: An Introduction*, Springer-Verlag, Berlin, 1985.
- Reisig, Wolfgang; *Petrinetze: Eine Einführung*, Springer-Verlag, Berlin, 1986.
- Reynolds, John C.; "Reasoning About Arrays", *CACM*, Vol. 22, No. 5, p. 290-299, 1979 May.
- Riebisch, Matthias; "Halbformale Beschreibung von Softwarekomponenten zum Zweck ihrer Wiederverwendung", *GI-SE Softwaretechnik-Trends, Mitteilungen der Fachgruppe 'Software-Engineering'*, Band 12, Heft 1, S. 30-41, 1992 Feb.
- Risse, Thomas; *On the symbolical evaluation of the reliability of systems whose structure function is given by any Boolean expression in its components*, Interner Bericht 4/86, Fachbereich Informatik, Johann Wolfgang Goethe-Universität, Frankfurt/Main, 1986 June 11.
- Robinson, J.; Merad, S. Menani; "Developing an Environment for Computer-based Automotive Suspension and Steering Systems", in Redmill, Felix; Anderson, Tom (eds.); *Directions in Safety-critical Systems, Proceedings of the First Safety-critical Systems Symposium, Bristol, 9-11 February 1993*, p. 150-167, Springer-Verlag, London, 1993.
- Rolland, F.D.; *Programming with VDM*, Macmillan, London, 1992.
- Ross, M.; Brebbia, C.A.; Staples, G.; Stapleton, J. (eds.); *Software Quality Management*, (Proceedings of the SQM '93 Conference, Southampton), Computational Mechanics Publications, Southampton, 1993.
- Royden, H.L.; *Real Analysis*, 2nd edition, Collier-Macmillan, London, 1970.
- RTCA SC-167/EUROCAE WG-12; *Software Considerations in Airborne Systems and Equipment Certification*, RTCA Inc., Washington, D.C., 1992 Dec. 1.

- Rushby, John M.; von Henke, Friedrich; "Formal Verification of Algorithms for Critical Systems", *IEEE TSE*, Vol. 19, No. 1, p. 13-23, 1993 Jan.
- Ryder, John D.; Fink, Donald G.; *Engineers & Electrons: A Century of Electrical Progress*, IEEE Press, New York, 1984.
- Sankar, Sriram; Mandal, Manas; "Concurrent Runtime Monitoring of Formally Specified Programs", *IEEE Computer*, Vol. 26, No. 3, p. 32-41, 1993 March.
- Sarkar, D.; De Sarkar, S.C.; "Some Inference Rules for Integer Arithmetic for Verification of Flowchart Programs on Integers", *IEEE TSE*, Vol. 15, No. 1, p. 1-9, 1989 Jan.
- Sarkar, D.; De Sarkar, S.C.; "A Set of Inference Rules for Quantified Formula Handling and Array Handling in Verification of Programs Over Integers", *IEEE TSE*, Vol. 15, No. 11, p. 1368-1381, 1989 Nov.
- Schäfer, Torsten; *Revision von Software-Anwendungsmodellen auf der Basis eines objekt-orientierten Datenmodells*, Dissertation (Dr.rer.nat.), Fachbereich Mathematik, Universität Marburg, 1992 Dez. 10.
- Schauer, Helmut; *PASCAL für Anfänger*, R. Oldenbourg Verlag, Wien, 1979.
- Schelle, Heinz; Molzberger, Peter (Hrsg.); *Psychologische Aspekte der Software-Entwicklung*, R. Oldenbourg Verlag, München, 1983.
- Scherlis, William L.; Scott, Dana S.; "First Steps towards Inferential Programming", in Mason, R.E.A. (ed.); *Information Processing 83 — Proceedings of the IFIP 9th World Computer Congress, Paris, France, September 19-23, 1983*, p. 199-212, North-Holland, Amsterdam, 1983.
- Schlüter, P.; Behdjati, A.; Fleischer, P.; Bagdon, S.; "Objektorientierte Software-Entwicklung: Konzepte und Terminologie", *GI-SE Softwaretechnik-Trends, Mitteilungen der Fachgruppe 'Software-Engineering'*, Band 10, Heft 2, S. 22-44, 1990 Okt.
- Schmitz, Lothar; "Zur Spezifikation objekt-orientierter Programme", *GI-SE Softwaretechnik-Trends, Mitteilungen der Fachgruppe 'Software-Engineering'*, Band 12, Heft 4, S. 60-73, 1992.
- Schneider, Hans J.; "Programmverifikation", *Computer Magazin*, Nr. 4, p. 50-51+55+57, 1986.
- Schnorr, C.P.; *Rekursive Funktionen und ihre Komplexität*, B.G. Teubner, Stuttgart, 1974.
- Scholefield, David; *The Formal Development Of Real-Time Systems: A Review*, University of York, Department of Computer Science, 1992 Feb. 28.
- Schrade, Bruno; "Petrinetze als Programmiersprache", *GI-SE Softwaretechnik-Trends, Mitteilungen der Fachgruppe 'Software-Engineering'*, Band 12, Heft 1, S. 42-52, 1992 Feb.
- Schröten, M.B.; Inggs, M.R.; "Development of a Fail-Safe Data Transmission System for use in Life-Critical Applications", in Górski, Janusz (ed.); *SAFECOMP '93, The 12th International Conference on Computer Safety, Reliability and Security, Poznań-Kiekrz, Poland, 27-29 October 1993*, p. 379-388, Springer-Verlag, London, 1993.
- Schulz, Arno; *Methoden des Softwareentwurfs und Strukturierte Programmierung*, Walter de Gruyter, Berlin, 1978.
- Selby, Richard W.; Basili, Victor R.; Baker, F. Terry; "Cleanroom Software Development: An Empirical Evaluation", *IEEE TSE*, Vol. SE-13, No. 9, p. 1027-1037, 1987 Sept.
- Sennett, Chris T. (ed.); *High-integrity Software*, Pitman, London, 1989.

- Shaw, Mary; "Prospects for an Engineering Discipline of Software", *IEEE Software*, Vol. 7, No. 6, p. 15-24, 1990 Nov.
- Smith, Douglas R.; "KIDS: A Semiautomatic Program Development System", *IEEE TSE*, Vol. 16, No. 9, p. 1024-1043, 1990 Sept.
- Sneed, Harry M.; "Der automatische Test digitaler Computerprogramme gegen eine formale Spezifikation", *Technische Zuverlässigkeit, 13. Fachtagung, Nürnberg, 21-22 Mai 1985*, p. 106-132, Nachrichten-Technische-Gesellschaft (NTG), VDE Verlag, Berlin, 1985.
- Snyder, Alan; "The Essence of Objects: Concepts and Terms", *IEEE Software*, Vol. 10, No. 1, p. 31-42, 1993 Jan.
- Sommerville, Ian; *Software Engineering*, second edition, Addison-Wesley, Wokingham, England, 1985.
- Sommerville, Ian; Paul, Manfred (eds.); *Software Engineering — ESEC '93, 4th European Software Conference, Garmisch-Partenkirchen, Germany, September 1993, Proceedings*, LNCS 717, Springer-Verlag, Berlin, 1993.
- Spaniol, Otto; *Arithmetik in Rechenanlagen*, B.G. Teubner, Stuttgart, 1976.
- Spivey, J.M.; *The Z Notation: A Reference Manual*, Prentice Hall, New York, 1989.
- Spivey, J. Michael; "Specifying a Real-Time Kernel", *IEEE Software*, Vol. 7, No. 5, p. 21-28, 1990 Sept.
- Stroustrup, Bjarne; "Yet More Bjarne", *.EXE Magazine*, Vol. 6, Issue 9, p. 30-36, 1992 March.
- Thomas, Martyn; *Should we trust computers?*, The BCS/UNISYS Annual Lecture 1988, BCS, London, 1988 July 4.
- Thomas, Martyn; "The industrial use of formal methods", *Microprocessors and Microsystems*, Vol. 17, No. 1, p. 31-36, 1993 Jan.-Feb.
- Thomas, Muffy; "Case Study: The story of the Therac-25 in LOTOS", *High Integrity Systems*, Vol. 1, No. 1, p. 3-15, 1994.
- Thomas, Muffy; "A proof of incorrectness using the LP theorem prover: the editing problem in the Therac-25", *High Integrity Systems*, Vol. 1, No. 1, p. 35-48, 1994.
- Thuau, Ghislaine; Berkane, Bachir; "Correctness Verification of Synchronous Sequential Circuits", in *Poster Session — 12th World Computer Congress, IFIP Congress '92*, p. 75, IFIP, Madrid, 1992 Sept. 7-11.
- Tierney, Margaret; "Potential Difficulties in Managing Safety-critical Computing Projects: a Sociological View", in Redmill, Felix; Anderson, Tom (eds.); *Directions in Safety-critical Systems, Proceedings of the First Safety-critical Systems Symposium, Bristol, 9-11 February 1993*, p. 43-64, Springer-Verlag, London, 1993.
- Toetenel, Hans; van Katwijk, Jan; Plat, Nico; "Structured Analysis — Formal Design, using Stream and Object Oriented Formal Specification", in Moriconi, Mark (ed.); *Proceedings of the ACM SIGSOFT International Workshop on Formal Methods in Software Development, Napa, CA, USA, 9-11 May 1990, Software Engineering Notes*, Vol. 15, No. 4, p. 118-127, 1990 Sept.
- Trachtenberg, Martin; "Why Failure Rates Observe Zipf's Law in Operational Software", *IEEE Transactions on Reliability*, Vol. 41, No. 3, p. 386-389, 1992 Sept.
- Trubow, George B.; "How liable are you for your software?", *IEEE Software*, Vol. 8, No. 4, Law Review column, p. 94-95+101, 1991 July.

- Turski, Władysław M.; "Software Engineering — Some Principles and Problems", in Gries, David (ed.); *Programming Methodology*, p. 29-36, Springer-Verlag, New York, 1978.
- Turski, Władysław M.; "A View of Current Concerns in Software Engineering", in Wössner, Hans (Hrsg.); *Programmiersprachen und Programmwicklung — 7. Fachtagung, veranstaltet vom Fachausschuß 2 der GI, München, 9./10. März 1982*, Informatik-Fachberichte 53, Springer-Verlag, Berlin, 1982.
- Turski, W.M.; "On Programming by Iterations", *IEEE TSE*, Vol. SE-10, No. 2, p. 175-178, 1984 March.
- Turski, Władysław M.; "And No Philosopher's Stone, either", in Kugler, H.-J. (ed.); *Information Processing 86 — Proceedings of the IFIP 10th World Computer Congress, Dublin, Ireland, September 1-5, 1986*, p. 1077-1080, North-Holland, Amsterdam, 1986.
- Turski, Władysław M.; Maibaum, Thomas S.E.; *The Specification of Computer Programs*, Addison-Wesley, Wokingham, 1987.
- Turski, Władysław M.; "On Starvation and Some Related Issues", draft, 1990.
- Turski, Władysław M.; "Prescribing Behaviours", draft, 1990.
- Várkonyi, Zsolt; "Programada Teknologio, Programar-krizo", *Internacia Komputado*, (in Esperanto) n-ro 4, p. 2-9, 1984.
- VDI/VDE-Gesellschaft Meß- und Automatisierungstechnik, Fachausschuß Sicherheitstechnik der Automatisierungssysteme, Unterausschuß Begriffe für sicherheitsbezogene Software; *Sicherheitstechnische Begriffe für Automatisierungssysteme, Zuverlässigkeit und Sicherheit komplexer Systeme (Begriffe)*, VDI/VDE 3542 Blatt 4 Entwurf, VDI/VDE-Richtlinien, 1993 Juli.
- VDI-Gemeinschaftsausschuß Industrielle Systemtechnik (VDI-GIS) (Hrsg.); *Software-Zuverlässigkeit — Grundlagen, Konstruktive Maßnahmen, Nachweisverfahren*, VDI-Verlag, Düsseldorf, 1993.
- Verity, John W.; "Bridging the Software Gap", *Datamation*, p. 84, 86-88, 1985 Feb. 15.
- Vickers, Steven; "Formal implementation", in McDermid, John A. (ed.); *Software Engineer's Reference Book*, p. 25/1-15, Butterworth-Heinemann, Oxford, 1991.
- Vitruvius, Marcus Pollio; *De architectura libri decem* (ca. 25 B.C.), Latin text and English translation by Granger, Frank (1931, 1955) in *Vitruvius: On Architecture*, Volumes 1 and 2, The Loeb Classical Library, Heinemann, London. Deutsche Übersetzung (1796) von Rode, August in *Vitruv: Baukunst*, Bände 1 und 2, Artemis Verlag für Architektur, Zürich, München, 1987.
- Voges, Udo; "Fehlertoleranz gegenüber Entwurfsfehlern", Entwurf eines Fachartikels, Kernforschungszentrum Karlsruhe, 1988 Jan. 27.
- vom Hövel, R.; Krebs, H.; Clendining, M.; "Safety in Software for Manned Mission Control: A Classification Methodology", lecture given at the First International Symposium on Ground Data Systems for Spacecraft Control, ESA, Darmstadt, 1990 June 28-29.
- Wallmüller, Ernest; *Software-Qualitätssicherung in der Praxis*, Carl Hanser Verlag, München, 1990.
- Wand, Mitchell; "A New Incompleteness Result for Hoare's System", *JACM*, Vol. 25, No. 1, p. 168-175, 1978 Jan.
- Wand, Mitchell; *Induction, Recursion, and Programming*, North-Holland, New York, 1980.

- Wang, Yabo; *Formal and Abstract Software Module Specifications — A Survey*, Technical Report 91-307, Computing and Information Science, Queen's University, Kingston, Ontario, Canada, 1991 May.
- Waters, Richard C.; "Automatic Analysis of the Logical Structure of Programs", revised version of Ph.D. dissertation, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1978 Dec.
- Waters, Richard C.; "A Method for Automatically Analyzing Programs", *Proceedings of the Seventh International Joint Conference on Artificial Intelligence IJCAI 1979*, p. 935-941, 1979.
- Waters, Richard C.; "A Method for Analyzing Loop Programs", *IEEE TSE*, Vol. SE-5, No. 3, p. 237-247, 1979 May.
- Webb, J.T.; *Static Analysis using MALPAS: An Introduction and simple examples*, internal report, Rex, Thompson & Partners Ltd., Farnham, England, 1988 Dec. 18.
- Webb, J.T.; "The role of verification and validation tools in the production of critical software", in Ince, Darrel (ed.); *Software Quality and Reliability*, p. 33-41, Chapman & Hall, London, 1991.
- Weber, Herbert; *Die Software-Krise und ihre Macher*, Springer-Verlag, Berlin, 1992.
- Weigele, Martin; "Von der Idee zum Computerprogramm: Formale Verfahren der Software-Entwicklung / Erfolgreiche Kooperation zwischen Theorie und Praxis in England", *Frankfurter Allgemeine Zeitung*, 1990 Feb. 21.
- Weigele, Martin; Peleska, Jan; "Die Anwender verschlafen den Trend zu 'Safer Software'", *Computerwoche*, Nr. 10, S. 18-20, 1991 März 8.
- Weiser, Mark; "Source Code", *IEEE Computer*, Vol. 20, No. 11, p. 66-73, 1987 Nov. Relevant letter to the editor and author's reply in *IEEE Computer*, Vol. 21, No. 4, p. 8+11, 1988 April.
- Weizenbaum, Joseph; *Computer Power and Human Reason: From Judgment to Calculation*, W.H. Freeman, San Francisco, 1976. Deutsche Übersetzung: *Die Macht der Computer und die Ohnmacht der Vernunft*, Suhrkamp Taschenbuch Wissenschaft 274, 1978.
- Wendt, S. "Defizite im Software Engineering", *Informatik-Spektrum*, Band 16, Heft 1, S. 34-38, 1993.
- Wexler, John; *Concurrent Programming in Occam 2*, Ellis Horwood, John Wiley & Sons, Chichester, 1989.
- Whiddett, Dick; *Concurrent programming for software engineers*, Ellis Horwood, John Wiley & Sons, Chichester, 1987.
- Wichmann, Brian A. (ed.); *Software in Safety-related Systems*, John Wiley & Sons, Chichester, 1992.
- Wichmann, B.A.; "A Note on the Use of Floating Point in Critical Systems", *The Computer Journal*, Vol. 35, No. 1, p. 41-44, 1992 Feb.
- Wieczorek, M.J.; "Structured specification of real time distributed systems using topological locative temporal logic", in Ross, M.; Brebbia, C.A.; Staples, G.; Stapleton, J. (eds.); *Software Quality Management*, (Proceedings of the SQM '93 Conference, Southampton), p. 681-698, Computational Mechanics Publications, Southampton, 1993.

- Wilbur, Steve R.; "Networks and distributed systems", in McDermid, John A. (ed.); *Software Engineer's Reference Book*, p. 53/1-25, Butterworth-Heinemann, Oxford, 1991.
- Wilde, Nicholas P.; "A WYSIWYC (What You See Is What You Compute) Spreadsheet", in *Proceedings, 1993 IEEE Symposium on Visual Languages, August 24-27, 1993, Bergen, Norway*, p. 72-76, IEEE Computer Society Press, Los Alamitos, CA, 1993.
- Wildes, Karl L.; Lindgren, Nilo A.; *A Century of Electrical Engineering and Computer Science at MIT, 1882-1982*, MIT Press, Cambridge, Massachusetts, 1985.
- Willmer, Heidemarie; Balzert, Helmut; *Fallstudie einer industriellen Software-Entwicklung — Definition, Entwurf, Implementierung, Abnahme, Qualitätssicherung*, B.I.-Wissenschaftsverlag, Mannheim, 1984.
- Willson, Reg G.; Krogh, Bruce H.; "Petri Net Tools for the Specification and Analysis of Discrete Controllers", *IEEE TSE*, Vol. 16, No. 1, p. 39-50, 1990 Jan.
- Wing, Jeannette M.; Gong, Chun; "Experience with the Larch Prover", in Moriconi, Mark (ed.); *Proceedings of the ACM SIGSOFT International Workshop on Formal Methods in Software Development, Napa, CA, USA, 9-11 May 1990, Software Engineering Notes*, Vol.15, No.4, p. 140-143, 1990 Sept.
- Wing, Jeannette M.; "Using Larch to Specify Avalon/C++ Objects", *IEEE TSE*, Vol. 16, No. 9, p. 1076-1088, 1990 Sept.
- Wing, Jeannette M.; "A Specifier's Introduction to Formal Methods", *IEEE Computer*, Vol. 23, No. 9, p. 8-22+24, 1990 Sept.
- Winkler, J.F.H.; Nievergelt, J.; "Wie soll die Fakultätsfunktion programmiert werden?", *Informatik-Spektrum*, Kolumne "Overflow", Bd. 12, S. 220-221, 1989.
- Winkler, Jürgen F.H.; "Object-CHILL — Eine objektorientierte Erweiterung von CHILL", in Zorn, Werner (Hrsg.); *Tagungsband TOOL 90, 1. Int. Fachmesse und Kongreß für Software- und Datenbank-Management*, S. 272-280, MESAGO Messe & Kongreß GmbH, 1990. Auch in Diskussionsveranstaltung des ITG FA 4.2/3 "System- und Anwendersoftware" und des Forschungsinstitutes der Deutschen Bundespost TELEKOM zum Thema "Objekt-orientierte Ansätze in der Telekommunikation", Darmstadt, 1991 Feb. 6.
- Wirth, Niklaus; *Systematisches Programmieren: Eine Einführung*, B.G. Teubner, Stuttgart, 1972.
- Wirth, Niklaus; *Algorithms + Data Structures = Programs*, Prentice-Hall, Englewood Cliffs, N.J., 1976.
- Wirth, Niklaus; "Die Dinge beim Namen nennen", *Chip Inside: Erstes European Software Festival, München*, S. 60-64, Vogel Verlag, Würzburg, 1991.
- Wirth, Niklaus; "Gedanken zur Software-Explosion", *Informatik-Spektrum*, Band 17, Heft 1, p. 5-10, 1994 Feb.
- Witt, Jan; "Weakest Precondition", *Informatik-Spektrum*, Band 5, Heft 1, S. 48-50, 1982 Feb.
- Witt, Jan; "Die Weitsicht des Software-Entwicklers", in Ackermann; Ulich (Hrsg.); *Software-Ergonomie '91, Gemeinsame Fachtagung des German Chapter of the ACM, der Gesellschaft für Informatik (GI) und der Schweizer Informatiker Gesellschaft SI vom 18. bis 20. März 1991 in Zürich*, Band 33, S. 25-35, B.G. Teubner, Stuttgart, 1991.
- Wix, Barbara; unveröffentlichte Auswertung einer Befragung der Teilnehmer am "Spektrum 86 — Kongreß für Software-Maintainability", Digital Equipment (DEC) College, München, 1986 Nov. 5-7.

- Wloka, Josef; *Funktionalanalysis und Anwendungen*, Walter de Gruyter, Berlin, 1971.
- Wood, William G.; "Application of Formal Methods to System and Software Specification", in Moriconi, Mark (ed.); *Proceedings of the ACM SIGSOFT International Workshop on Formal Methods in Software Development, Napa, CA, USA, 9-11 May 1990*, *Software Engineering Notes*, Vol.15, No.4, p. 144-146, 1990 Sept.
- Wulf, William A.; Shaw, Mary; Hilfinger, Paul N.; Flon, Lawrence; *Fundamental Structures of Computer Science*, Addison-Wesley, Reading, Massachusetts, 1981.
- Yau, S.S.; Gore, J.V.; "Paradigm Lost: Discovering the Intersection of Assertion-Guided Program Construction and Constraint-Driven Reevaluation", *The Computer Journal*, Vol. 36, No. 2, p. 137-142, 1993.
- Yourdon, Edward; *Techniques of Program Structure and Design*, Prentice-Hall, Englewood Cliffs, N.J., 1975.
- Yourdon, Edward; *Decline and Fall of the American Programmer*, Yourdon Press, Englewood Cliffs, N.J., 1992.
- Zedan, H.; "On the Analysis of **occam** Real-time Distributed Computations", *Microprocessing and Microprogramming*, Vol. 24, p. 491-500, 1988.
- Zelkowitz, Marvin V.; "A Functional Correctness Model of Program Verification", *IEEE Computer*, Vol. 23, No. 11, p. 30-39, 1990 Nov.
- Zhou Shengzong; Tang Zhisong; *Designing and Verifying Distributed Programs Using Temporal Logic*, Technical Report AS-IS-85-4, XYZ Report No. 4, Software Institute, Academia Sinica, Beijing, China, 1985 Dec. 22.
- Zhou Shengzong; *Verifying Liveness Properties of Protocols*, Technical Report AS-IS-86-5-4, XYZ Report No. 11, Software Institute, Academia Sinica, Beijing, China, 1986 June 19.
- Zhu, H.; "Convincing introduction on proof of program correctness", Buchbesprechung über [Baber; *Error-free Software*, 1991], *Information and Software Technology*, 1992 June.
- Zima, Hans; *Betriebssysteme: Parallele Prozesse*, Bibliographisches Institut, Mannheim, 1980.
- Zucker, J.I.; *Normal and Inverted Function Tables*, CRL Report No. 265, Communications Research Laboratory, Faculty of Engineering, McMaster University, Hamilton, Ontario, Canada, 1993 Dec.
- ; *The Programming Language Ada Reference Manual*, LNCS 106, Springer-Verlag, Berlin, 1981.
- ; "'Vertrauensverlust' beim Bildschirmtext-Dienst", *Frankfurter Allgemeine Zeitung*, S. 1+12+16, 1984 Nov. 22.
- ; Special Issues on Software Reliability, Parts I and II: *IEEE TSE*, Vol. SE-11, No. 12, 1985 Dec. and Vol. SE-12, No. 1, 1986 Jan.
- ; "Lethal dose", *IEEE Spectrum*, Faults & Failures column, p. 16, 1987 Dec.
- ; "Prozeß um Deckeneinsturz in Hallenbad: Bewährungsstrafen", *Frankfurter Allgemeine Zeitung*, 1987 Dez. 4.
- ; "Softwarefehler kostet US-Airline 50 Millionen", *Computerwoche*, 1988 Sept. 23.
- ; *Eiffel: An introduction*, Version 2.2, Report TR-EI-3/GI, Interactive Software Engineering Inc., 1989.
- ; "Softwarebug legte Reisebuchungssystem lahm", *Computerwoche*, S. 68, 1989 Juni 2.
- ; *Malpas Management Guide (International)*, Issue No. 2, Report RTP/020/17/2, RTP Software Ltd., Farnham, England, 1989 Sept.

- ; "Wirth: 'Irgendwann einmal geht es nicht mehr weiter'", Interview der Woche, *Computerwoche*, S. 7, 10 und 12, 1989 Nov. 24.
- ; "Formal Verification: Is It Practical for Real-World Design? — A D&T Roundtable", *IEEE Design & Test of Computers*, p. 50-58, 1989 Dec.
- ; "Die programmierte Katastrophe", *Der Spiegel*, Nr. 42/1990, S. 80+83+86+88+91+93, 1990.
- ; Special Issue on Formal Methods in Software Engineering: *IEEE TSE*, Vol. 16, No. 9, 1990 Sept.
- ; *TickIT Guide to Software Quality Management System Construction and Certification Using EN29001*, BCS and the U.K. Department of Trade and Industry, 1990 Sept. 30.
- ; "Sichere SW: DST stellt ein Konzept vor", *Computerwoche*, 1991 Mai 17.
- ; *Predictably Dependable Computing Systems (PDCS) Second Year Report, Volume 3*, Report of Esprit BRA Project 3092, Department of Computing Science, University of Newcastle upon Tyne, England, 1991 May.
- ; "Patriot's software failure let missile hit troops", *IEEE Software*, Vol. 8, No. 4, p. 97, 1991 July.
- ; *Safety-Related Systems, A professional brief for the engineer*, Institution of Electrical Engineers, London, 1992.
- ; Special Issue, Formal Methods, Part 1, *The Computer Journal*, Vol. 35, No. 5, 1992 Oct.
- ; "Tandem bug throws banks into time warp", *Computing*, p. 3, 1992 Nov. 5.
- ; "An unlikely couple", *Computing*, p. 36, 1992 Nov. 26.
- ; Special Issue, Formal Methods, Part 2, *The Computer Journal*, Vol. 35, No. 6, 1992 Dec.
- ; Special Issue on Software for Critical Systems, *IEEE TSE*, Vol. 19, No. 1, 1993 Jan.
- ; *Using Z und DST-fuzz — An Introduction*, internal report, DST Deutsche System-Technik GmbH, Kiel, 1993 Feb.
- ; "Formal education", *Computing*, p. 24, 1993 May 6.

Persönliche Kommunikation

Anregungen und Ideen, die ich durch persönliche Kommunikation, während Diskussionen usw. mit den folgenden Personen und Personengruppen gewonnen habe, haben zur Entwicklung und Ausarbeitung des Inhalts dieser Dissertation beigetragen. Besonders die in solchen Diskussionen gestellten Fragen haben meinen Bemühungen, zu einer praxisgerechteren Gestaltung dieses Stoffs beizutragen, eine Zielrichtung gegeben.

Prof. Dr. Manfred Broy	Dr. Peter Kearney
Prof. Dr. Bernard Cohen	Prof. Dr. M.M. Lehman
Dr. J.F. Coxhead	Prof. Dipl. Ing. Peter Lucas
Prof. Dr. Mario Dal Cin	Mr. Allen Macro
Mr. Tim Denvir	Dr. Keith Mander
Drs. Willem Dijkhuis	Prof. Dr. John A. McDermid
Prof. Dr. Edsger W. Dijkstra	Prof. Dr. Andrew D. McGettrick
Prof. Dr. Wolfgang D. Ehrenberger	Mr. Basil W. Osborne
Prof. Dr. Wim H.J. Feijen	Prof. Dr. David Lorge Parnas
Prof. Dr. Timm Grams	Dr. Jan Peleska
Mr. John Harvey	Prof. Dr. Brian Randell
Prof. Dr. Wolfgang Henhapl	Dr. David Rudd
Prof. Dr. Charles Antony Richard Hoare	Prof. Dr.-Ing. Henner Schneider
Prof. Dr.-Ing. Hans-Jürgen Hoffmann	Mr. Mike B. Schröner
Herr Oberstudienrat Kagerhuber	Prof. Dr. Władysław M. Turski

meine Beratungsklienten

Mitglieder des GI-Fachausschusses 7.4 "Arbeitsmarkt und Berufssituation"

Mitglieder der DGQ/ITG/VDI-Arbeitsgruppe 4.1 "Software-Zuverlässigkeit"

Mitglieder des VDI/VDE-GMA-Fachausschusses 4.4 "Sicherheitstechnik der Automatisierungssysteme"

Studenten, die meine Lehrveranstaltungen belegt haben

Teilnehmer an meinen Seminaren

Teilnehmer an verschiedenen Kongressen, Konferenzen u.ä.

Zuhörer meiner Vorträge

Lebenslauf

Robert L. Baber ist 1937 in Los Angeles, California, U.S.A., geboren. Er besuchte Schulen in Modesto, California; Seattle, Washington; Anchorage, Alaska; San Antonio, Texas und Roswell, New Mexico. 1954 bis 1959 studierte er Elektrotechnik am Massachusetts Institute of Technology, Cambridge, Massachusetts, und schloß dieses Studium mit den akademischen Graden Bachelor of Science und Master of Science ab. Dieser Studiengang schloß Praktika bei der International Business Machines Corporation in Poughkeepsie und Binghamton, New York, ein. Herr Baber war in der ersten Gruppe von M.I.T.-Studenten, die im Rahmen des "Cooperative Course in Electrical Engineering" ihre Praktika bei der IBM Corporation absolvierten.

Anschließend studierte Robert L. Baber Kerntechnik und Betriebswirtschaft. 1962 schloß er sein Studium am Massachusetts Institute of Technology mit dem Master of Science in industrial management ab.

Seine Wehrdienstpflicht leistete Robert L. Baber 1962 bis 1964 als Offizier im Signal Corps der U.S. Army in Fort Gordon, Georgia; Washington, D.C. und Fort Ritchie, Maryland, wo er mit dem Einsatz und der Pflege von Softwaresystemen für die militärische Führung beschäftigt war.

Robert L. Baber zog 1964 nach Deutschland um, wo er bis 1966 bei Control Data in der technischen Unterstützung des Vertriebs arbeitete. 1966 bis 1970 war er als Berater bei Diebold S.A. in Frankfurt/Main im Diebold-Forschungs-Programm, das sich mit verschiedenen Aspekten der Anwendung von EDV-Systemen im betriebswirtschaftlichen Bereich befaßte, tätig.

1970 bis 1975 war Herr Baber Senior Consultant bei der Unternehmensberatungsgesellschaft Harbridge House Europe in Frankfurt/Main. Seit 1975 ist er als selbständiger Unternehmensberater freiberuflich tätig. Seine Beratungsprojekte haben verschiedene betriebswirtschaftliche Themen zum Gegenstand, wobei der Schwerpunkt auf der Konzipierung, Planung und Realisierung von Informationssystemen für betriebswirtschaftliche und technische Anwendungen liegt.

Robert L. Baber hat ein dreitägiges Seminar über die Konstruktion beweisbar korrekter Software konzipiert und es mehrmals in Deutschland, den Niederlanden, Dänemark und England unterrichtet. Auf Einladung des Instituts für Software der Academia Sinica hat er das Seminar 1987 in Beijing, China, gehalten.

Mitte der 1970er Jahre studierte Robert L. Baber nebenberuflich und auf Teilzeitbasis Wahrscheinlichkeitstheorie am Fachbereich Mathematik der Johann Wolfgang Goethe-Universität in Frankfurt/Main.

Robert L. Baber hat die drei Bücher *Software Reflected: The Socially Responsible Programming of Our Computers* (1982), *The Spine of Software: Designing Provably Correct Software — Theory and Practice* (1987) und *Fehlerfreie Programmierung für den Software-Zauberlehrling* (1990) verfaßt. Von *Software Reflected* sind deutsche und polnische Übersetzungen erschienen. Die von Herrn Baber ins Englische übersetzte Ausgabe von *Fehlerfreie Programmierung für den Software-Zauberlehrling* wurde 1991 veröffentlicht. Eine russische Übersetzung davon ist in Vorbereitung.

Herr Baber ist in *International Who's Who in Engineering*, *International Who's Who of Intellectuals*, *Men of Achievement* und anderen biographischen Nachschlagewerken aufgeführt.

Als Lehrbeauftragter im Fachbereich Informatik der Johann Wolfgang Goethe-Universität in Frankfurt/Main hält Herr Baber die von ihm konzipierte Vorlesung "Einführung in die Konstruktion fehlerfreier Software für praktische Anwendungen" sowie das Seminar "Projektstudien zur Konstruktion fehlerfreier Software".

Robert L. Baber ist Chartered Engineer und Fellow of the British Computer Society (BCS), Senior Member of the Institute of Electrical and Electronics Engineers (IEEE) und dessen Computer Society sowie Mitglied der Gesellschaft für Informatik. Er war Kogründer und erster Chairman des Computer Chapters der Deutschen Sektion des IEEE und ist jetzt Vice Chairman der Deutschen Sektion des IEEE. Ferner ist er Mitglied des Vereins Deutscher Ingenieure (VDI) und des Verbands Deutscher Elektrotechniker (VDE) sowie Europäischer Ingenieur in der Fédération Européenne d'Associations Nationales d'Ingénieurs (FEANI). Er wirkt in der DGQ/ITG/VDI-Arbeitsgruppe 4.1 "Software-Zuverlässigkeit", im VDI/VDE-GMA Fachausschuß 4.4 "Sicherheitstechnik der Automatisierungssysteme" sowie im GI Fachausschuß 7.4 "Arbeitsmarkt und Berufssituation", insbesondere in dessen CEPIS-Arbeitskreis, mit.

**Zur praktischen Anwendbarkeit
mathematisch rigoroser Methoden
zum Sicherstellen der Korrektheit
von sequentiellen Computerprogrammen**

an den
Fachbereich Informatik
der
Technischen Hochschule Darmstadt
zum Erlangen des akademischen Grades
Doktor-Ingenieur (Dr.-Ing.)
eingereichte Dissertation

von
Robert Laurence Baber
geboren in Los Angeles, California, U.S.A.
Master of Science in Electrical Engineering
und
Master of Science in Industrial Management,
Massachusetts Institute of Technology

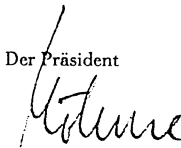
Erstreferent: Prof. Dr.-Ing. Hans-Jürgen Hoffmann
Korreferent: Prof. Dr. David Lorge Parnas
eingereicht am: 1994 August 4
Prüfungstermin: 1994 September 30
Erschienen in Darmstadt, 1994

URKUNDE

Während der Amtszeit
des Präsidenten Professor Dr. phil. Helmut Böhme
und des Dekans Professor Dr. rer. nat. Christoph Walther
verleiht der Fachbereich Informatik
durch diese Urkunde
HERRN ROBERT LAURENCE BABER, S.B. S.M./MIT
geboren am 11. Dezember 1937 in Los Angeles (USA)
DEN AKADEMISCHEN GRAD
EINES DOKTOR-INGENIEURS (DR.-ING.)
nachdem er in ordnungsgemäßem Promotionsverfahren unter Mitwirkung
der Referenten Professor Dr.-Ing. H.-J. Hoffmann
und Professor D. L. Parnas, Ph. D. durch seine Dissertation
»Zur praktischen Anwendbarkeit mathematisch rigoroser Methoden
zum Sicherstellen der Korrektheit von sequentiellen Computerprogrammen«
und durch die mündliche Prüfung
seine wissenschaftliche Befähigung erwiesen hat.
Das Gesamturteil lautet: »sehr gut bestanden«

Darmstadt, den 30. September 1994

Der Präsident



Der Dekan

