Strukturiertes Programmieren - strukturelles Denken

Mieczysław A. Kłopotek,
Institut für Grundlagen der Informatik der Polnischen Akademie der Wissenschaften,
ul. Ordona 21, 01-237 Warszawa, Polen , klopotek@ipipan.waw.pl

Abstract: In dem Beitrag wird auf die Rolle hingewiesen, die in der Ausbildung der Programmierer an der TU Dresden die Idee der Strukturierten Programmierung hatte. Aus der heutigen Sicht ist das Bestehen auf das Trennen des Programmierungsstils vom Programmierungswerkzeug (Programmierungssprache) als besonders wichtig anzusehen. Auch das Bestreben nach der Integration von Programmentwicklungsmethode und Darstellungsweise des Endprodukts gilt immer noch als moderne Denkweise.

1 Einleitung

Die Programmierung wird oftmals mit der Kunst verglichen. Mit Recht, denn das Entwickeln eines komplizierten Softwareproduktes benötigt jedes Mal einzigartige Ideen und bleibt wohl noch lange eine Herausforderung. Aber auch ein Künstler muss lernen, seine Werkzeuge gut im Griff zu haben, um diese einzigartigen Vorstellungen materialisieren zu können. Und dennoch dürfen die Werkzeuge seine Denk- und Arbeitsweise nicht einschränken, sondern im Gegenteil, eben seine Arbeitsweise sollte ein Ansporn für das Schaffen neuer Werkzeuge sein. Dies galt auch bei der Lehre an der TU Dresden, wo man strukturiertes Programmieren lehrte, ohne dass strukturierte Programmiersprachen verfügbar waren.

In den Jahren 1978-83 war ich Student und dann anschließend Aspirant an der Sektion 08 Informationsverarbeitung der Technischen Universität Dresden. Zu dieser Zeit, also Ende der 70er, Anfang der 80 Jahre des vorigen Jahrhunderts war die Zeit, wo die ersten sog. Höheren Programmiersprachen bereits entwickelt waren - man nenne nur Fortran (50-ger, IBM), Lisp (1958, John McCarthy) C (1972, Dennis Ritchie), Pascal (1970, Niklaus Wirth, Anfang 80er auch an der Technischen Hochschule Karl-Marx-Stadt implementiert), PL/1 (seit 60er Jahre), Algol 60, Algol 68 (das auch an der TU Dresden eine Implementation hatte). Und dennoch galt Assembler (der damaligen ESER-Rechner-Serie, aber auch von Z-80-Mikroprozessoren) als die Grundlage für die praktische Ausbildung eines Programmierers.

Aus der heutigen Sicht scheint es darum bemerkenswert, wie man es damals geschafft hat, uns die noch heute als modern geltenden Software-Entwicklungs-Regeln wie Abstraktion, Dekomposition und Wiederverwendung beizubringen.

Nachfolgend möchte ich kurz die Idee des Strukturierten Programmierens erläutern (Abschnitt 2), um dann ihre Umsetzung in der Lehre (Abschnitt 3), Forschung (Abschnitt 4) an der TU zu erörtern. Ich erlaube mir auch ein paar Worte für und gegen die Struktu-

rierte Programmierung. Insbesondere werde ich darauf hinweisen, wie wir als Studenten den Wert der strukturierten Programmierung sahen (Abschnitt 5). Ich werde kurz einen Beweis für die Strukturierbarkeit eines jeden Programms vorführen (Abschnitt 6) und auf die Schwierigkeiten hinweisen, die man mit Parallelprogrammen dabei hat (Abschnitt 7). Schließlich werde ich kurz darauf eingehen, wie die Idee der Strukturierten Programmierung auf die Darstellung von Wahrscheinlichkeitsverteilungsfunktionen angewandt werden kann (Abschnitt 8).

Insbesondere wurde die strukturierte (oder strukturelle) Programmierung Gegenstand des Ausbildungsprozesses.

2 Die Idee der Strukturierten Programmierung

Das strukturierte Programmieren verkörperte gleichzeitig die Art und Weise, wie das fertige Programm auszusehen hatte, wie es dokumentiert wurde, aber auch wie der Entwicklungsprozess der Software voranschreiten sollte.

Auf der einen Seite bestand also ein strukturiertes Programm aus den elementaren Anweisungen, Prozeduren (oder Funktionen) und Strukturen wie Sequenzen, Alternativen, Zyklen und (auch) parallelen Programmzweigen, die ineinander geschachtelt waren (siehe Abb. 1). .

Auf der anderen Seite war damit direkt eine (top-down) Programmentwicklungsmethode verbunden, die darin bestand, dass die anfangs informal gefasste Aufgabenstellung durch solche Strukturen (also Sequenzen, Alternativen, Zyklen, Parallelzweige, Einschließen in eine Prozedur) verfeinert wurde.

Es wurde dabei von uns abverlangt, dass jedes so verfeinerte Programmstück als auch die Hauptteile des Zielprogramms in den Begriffen von Eingangs-, Ausgangs- und transitiver Information (Parametern), sowie durch eine Kurzfassung der Aufgabe des Programmstücks dokumentiert wurde. Man hat dabei gefordert, dass z.B. in einer Prozedur nicht nur ihre Parameter, sondern auch jegliche globalen Variablen mitbeschrieben wurden, die gelesen oder verändert wurden. Dies betraf nicht nur den Entwicklungsprozess, sondern auch das fertige Programm.

3 Strukturierte Programmierung in der Lehre

Um uns die entsprechende strukturelle Disziplin beizubringen, wurde vom Lehrkörper eine künstliche strukturelle graphische (zweidimensionale) Programmiersprache - PAP („Programmablaufplan“) - entwickelt, die damals noch nirgends implementiert[1] und unsere erste Programmiersprache war, in der wir mit einem Bleistift und Radiergummi unsere Programme im ersten Semester geschrieben haben.

[1] ihre Implementation wurde noch zu meiner Studienzeit Gegenstand einer Dissertation von Dr. Czaja

Von außen her war PAP eigentlich eine Art der noch heute gängigen Flowcharts. Es gab allerdings einen wichtigen Unterschied zu den Flowcharts - die Syntax und Semantik war nach strikten Regeln vorgeschrieben und wurde im Laufe des Semesters erweitert, so dass typische Probleme des Assemblers als auch der höheren Programmiersprachen, und nicht zuletzt (in weiteren Semestern) der Parallelprogrammierung erörtert werden konnten. In dieser Sprache konnte man schließlich nicht nur einfache Datentypen behandeln, sondern auch Strukturen, Zeiger, Listen, dynamische Speicherverwaltung usw.

Das strukturierte Programmieren musste dann von den Studenten in die Praxis umgesetzt werden. Bevor die Assemblerprogramme oder später PL/1-Programme geschrieben wurden, musste jeder den entsprechenden PAP fertig stellen und anhand dessen dann die Implementation durchführen.

4 Die Strukturierte Programmierung in der Forschung

An der Sektion 08 blieb das strukturierte Programmieren nicht in der Welt der Ideen. Wie schon erwähnt, wurde die zweidimensionale PAP-Sprache, mit der Differenzierung der zugelassenen Konstrukte implementiert. Somit konnten die Studenten ihre ersten Programme auch an einem Rechner testen.

Die Idee der strukturellen Programmentwicklung wiederum wurde in Form eines Werkzeugs (geschrieben in Lisp, an dem auch ich als Hilfsassistent arbeitete) zur schrittweise Programmentwicklung materialisert. In einem praxisbezogenen Projekt hat man das Werkzeug auf Entscheidungstabellenentwicklung zugeschnitten, die von der Eisenbahn in Form von Schaltwerken realisiert werden sollten.

5 Der Wert der Strukturierten Programmierung

An dieser Stelle sei daran erinnert, dass der Begriff der strukturierten Programmierung von E. Dijkstra um 1968 [Dij68, Dij82] mit der Zielstellung vorgeschlagen wurde, durch die Berücksichtigung verschiedener methodischer Ansätze die Programmzuverlässigkeit und Wartbarkeit zu verbessern. Dies startete eine lang andauernde Diskussion, ob man auch jedes Programm in der strukturierten Form schreiben kann bzw. soll.

Durch Arbeiten von Bohm und Jacopini[BJ66], Ashcroft und Manna [AM71] sowie Kosaraju [Kos74] und schließlich Mills [Mil75] wurde der Beweis erbracht, dass jedes (nichtparallele) Programm in ein strukturiertes (nichtparalleles) Programm mechanisch umgewandelt werden kann.

Trotzdem enthielten die zuvor als auch danach entwickelte Programmiersprachen unstrukturierte Elemente, die sogenannten GoTo Konstrukte. Zum Beispiel können wir in C schreiben: „Hier: ... goto Hier;“. In FORTRAN 77 gibt es „GOTO Zeilennummer“. In PASCAL kann man schreiben: „If (x<0) then goto Sprung; Writeln(„Positive Zahl„); Sprung;“. ALGOL 60 erlaubt zu schreiben: „if x>q then goto STOP else if x>w-2 then

goto S;“. Demzufolge glaubten und glauben weiterhin viele nicht an Strukturiertes Programmieren.

Die Diskussion um Strukturiertes Programmieren ging an der TU Dresden nicht vorbei. In der Lehre hat man zwar den Studenten die Beweise für „Strukturierbarkeit“ nicht beigebracht, aber statt dessen anhand von konkreten Beispielen die Argumente der Befürworter der „Misch-Masch-Programmierung“ zurückgewiesen. Man behauptete damals, dass diese Art der Programmierung große Einsparungen an Ausführungszeiten und an Programmlänge erlauben würde. Dies widerlegten Studien, die zeigten, dass nur erfahrene Programmierer entsprechende Kürzungen erzielen würden, und dass durch den Einsatz von automatischen Optimierungswerkzeugen für wohl strukturierte Programme eine vergleichbare Zielcodeeffizienz erreicht werden kann, wobei die Programmentwicklungszeit als auch die Fehlerrate entschieden gesunken werden kann.

Als Studenten konnten wir den Beweis der Richtigkeit der These anhand eines Ereignisses erleben, dessen Kenntnis glücklicherweise dem Lehrkörper erspart wurde. Einer der Mitkommilitonen hatte ein Praktikum an einer anderen Sektion, wo eine Bibliothek von Steuerprogrammen entwickelt wurde. Die Grundbibliothek war da, mit einer ziemlich ordentlichen Dokumentation, und seine Aufgabe war es, zusammengesetzte Steuerelemente über dieser Bibliothek zu entwickeln. Die Schwierigkeit bestand darin, dass zwar die Ein- und Ausgangsparameter ordentlich beschrieben wurden, aber die Prozeduren hatten Seiteneffekte, die nicht beschrieben waren. Diese waren für die Einzelanwendung zwar unwichtig, aber als Komponenten waren sie nicht geeignet. Kurzerhand wickelte der Student die Bibliothekprozeduren in kurze Programme ein, die durch Speicherung auf dem Stack (Stapelspeicher) die Seiteneffekte eliminierten. Danach war die eigentliche Aufgabenstellung einfach mit der strukturierten Programmierung zu lösen. Dies gefiel den Auftraggebern nicht, denn nach ihrer Meinung war die Lösung ineffizient (besonders bei Mikrorechnern war damals jedes Byte sehr kostbar). Man wollte ihm vorführen, wie man es richtig anpacken soll. Aber man schaffte esnicht einmal, die einfachste zusammengesetzte Komponente aus der originalen Bibliothek zu fertigen. Man hat sich also mit der strukturierten Programmierung abgefunden. Für uns Studenten war es ein Nachweis, dass das strukturierte Programmieren etwas an sich hat.

Also galt der Grundgedanke, dass die fortschreitenden automatischen Optimierungstechniken für die optimale Codierung verantwortlich sein sollten, weil der Programmierer sich eher auf das korrekte Lösen der gestellten Aufgaben konzentrieren sollte. Das war ein wichtiger Gedanke in vielerlei Hinsicht. Erstens kam da ein wichtiges Prinzip des sich entfaltenden Softwareengineering zum Vorschein: die Werkzeuge sollten dazu dienen, den Menschen zu unterstützen und nicht vice versa. Zweitens machte der Ansatz klar, dass die Fähigkeit und der Stil des Programmierens gar nicht vom verfügbaren Werkzeug (der Programmiersprache) abhängen muss. Drittens war es eine wichtige Einsicht, dass man auf einem viel höheren Niveau als auf dem der Programmiersprache denken muss, um ein Problem richtig zu lösen.

Nicht zuletzt war aber für uns Studenten die Frage eine Herausforderung, ob man auch jedes einzelne Programm in der strukturierten Form aufschreiben kann. Mich faszinierte diese theoretische Frage, und da man uns den Beweis nicht vorgeführt hatte, baute ich eines Tages meinen eigenen kompletten Beweis auf, den ich allerdings nie veröffentlichte, also

auch mit den anderen nie verglichen habe. Stattdessen entwickelte ich ein Makrosystem, welches erlaubte, Assemblerprogramme strukturell zu schreiben, ohne Sprungmarken zu verwenden.

6 Strukturierbarkeit eines Programms

Nun wollen wir aber zu der Frage der Darstellbarkeit eines jeden GoTo-Programms in der Form eines strukturierten Programms, eines PAP, kommen. Ein PAP bestand aus (1) einem START-Knoten, (2) einem ENDE-Knoten, (3) mehreren Ausführungsknoten („einfachen" Anweisungen), (4) Verzweigungsknoten, (5) Zusammenführungsknoten, (6) Parallelanstoß knoten und (7) Paralleltermin ierungsknoten. Das Interessante an einem strukturierten Programm ist das Verschachtelungsprinzip: wenn man eine Sequenz, oder eine Alternative, oder einen Zyklus, oder einen Parallelanstoß nimmt, welche(r) nur aus einfachen Anweisungen besteht, (siehe die Knotenstrukturen der Form aus der Abbildung 1), und diese(n) durch eine einfache Anweisung ersetzt (zu einer einfachen Instrruktion reduziert), so erhält man wieder ein wohlstrukturiertes Programm. Wenn man diese Operation fortfährt, kommt man schließlich zu einem Programm, das nur aus einer einfachen Anweisung besteht. Dies war der Grundgedanke beim Aufbau des Beweises für die Möglichkeit, jedes Programm in eine strukturierte Form umzuwandeln. Es wurde immer ein Stück genommen, welches in eine strukturierte Form gebracht und danach durch einfache Anweisung ersetzt wird. Die Programmstruktur soll sich dabei nach einem Kriterium vereinfachen (z.B. die Anzahl der Zusammenführungen der alternativen Zweige wird verkleinert), so dass der Abschluss des Verfahrens gewährleistet wird.

Das Ziel der Umwandlung eines beliebigen Programmgraphen in ein strukturiertes Programm besteht darin, den Graphen so zu verändern, dass das resultierende Programm genau dieselbe Operation ausführt, aber wohlstrukturiert ist.

Zuerst wollen wir uns den Programmen zuwenden, deren Programmgraphen keine Zyklen und keine Parallelanstöße enthalten, dafür aber Verzweigungen, Zusammenführungen und sequenzielle Anweisungen (siehe Abb. 2). Man kann im Graphen immer den frühesten Zusammenführungsknoten J1 finden. D.h. dass man auf keinem Wege vom Startpunkt zu diesem Zusammenführungsknoten keinen anderen Zusammenführungsknoten findet (siehe 2). Zu ihm gehört immer ein Verzweigungsknoten T1, von dem aus auf zwei verschiedenen Wegen J1 erreicht werden kann und von dessen Nachfolgerverzweigungsknoten dies nicht mehr möglich ist. T3 sei jetzt der späteste Verzweigungsknoten auf dem Weg von T1 zu J1. Um jetzt zu einem strukturierten Programm zu gelangen, verlagern wir diesen Verzweigungspunkt hinter J1. Zu diesem Zweck wird eine zusätzliche logische Variable Z eingeführt, die mit false initiiert wird. Die Zusatzvariable Z wird direkt vor T3 auf den logischen Wert von T3 gesetzt und soll ihn über J1 hinaustragen. Anstelle von T3 setzen wir ein Test des Z-Wertes, wobei hier der Zweig true bis vor J1 geführt wird. Direkt hinter J1 wird wieder der Wert von Z getestet, um jetzt die Abzweigungen von T3 in die originale Richtung zu führen. Jetzt ist T3 aus dem T1-J1-Pfad entfernt und die erste Z-Alternative kann durch eine einfache Anweisung ersetzt werden, denn sie genügt der Forderung der Strukturierten Programmierung und braucht nicht mehr betrachtet zu werden. Auf die Art

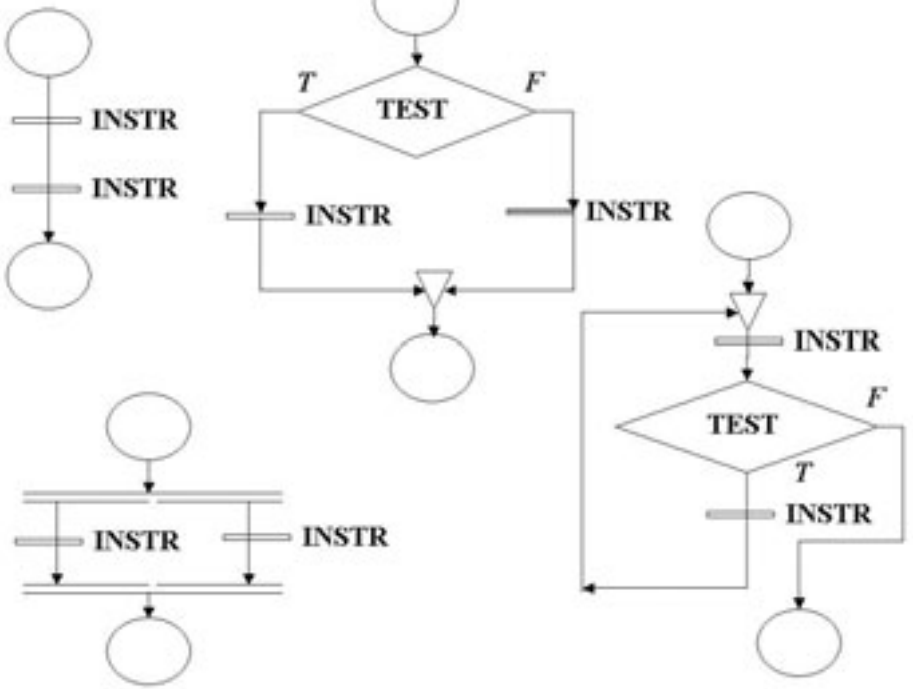

Abbildung 1: Strukturelle Elemente eines Programms

und Weise können alle Verzweigungen zwischen T1 und J1 entfernt werden Danach kann man die T1-J1-Alternative durch eine einfache Anweisung ersetzen und dieses Verfahren wiederholt durchführen, bis alle unstrukturierten Fragmente korrigiert worden sind.

Es sei an dieser Stelle bereits darauf hingewiesen, dass man durch Einführung der Variablen Z die Datenstruktur „kompliziert“, um die Programmstruktur zu vereinfachen.

Des Weiteren sei an der Stelle betont, dass man dasselbe Verfahren anwenden kann, wenn es sich um ein Programm handeln würde, welches keine Zyklen und keine Verzweigungen, dafür aber Parallelanstöße und Parallelterminierungen und sequenzielle Anweisungen enthält. Die eingeführten Hilfsvariablen haben als Aufgabe, in die richtigen Parallelzweige zu springen. Eine Verallgemeinerung auf azyklische Programmgraphen liegt auf der Hand. Man betrachte die Paare Verzweigung/Zusammenführung und Parallelanstoß/Parallelterminierung und dabei jeweils die früheste Terminierung oder Zusammenführung. Es muss jedoch gewährleistet sein, dass auf jedem Pfad vom Anfangsknoten des Graphen die Anzahl der Parallelterminierungen die Anzahl der Parallelanstöße nicht übersteigt.

Und jetzt zum komplizierteren Fall der Zyklen, wobei die Parallelanstöße zunächst unberücksichtigt werden (siehe Abb.3). Identifizieren wir einen Zyklus, wobei der Anfang des Zyklus eine Zusammenführung ist und die letzte diejenige Anweisung, von der aus zur ersten Anweisung des Zyklus übergegangen wird. Zuerst entfernen wir alle in den Zyklus eingehenden Pfeile (alle Zusammenführungspunkte außer dem ersten des Zyklus). Wir eliminieren zuerst den frühesten unerwünschten Eintrittspunkt. Eine Hilfsvariable Z wird eingeführt, die direkt vor dem Zyklus zu true initiiert wird und auf dem zu beseitigenden Pfeil zu false. Jetzt verlegen wir den unerwünschten Eintrittspunkt vor den Zyklus. Durch

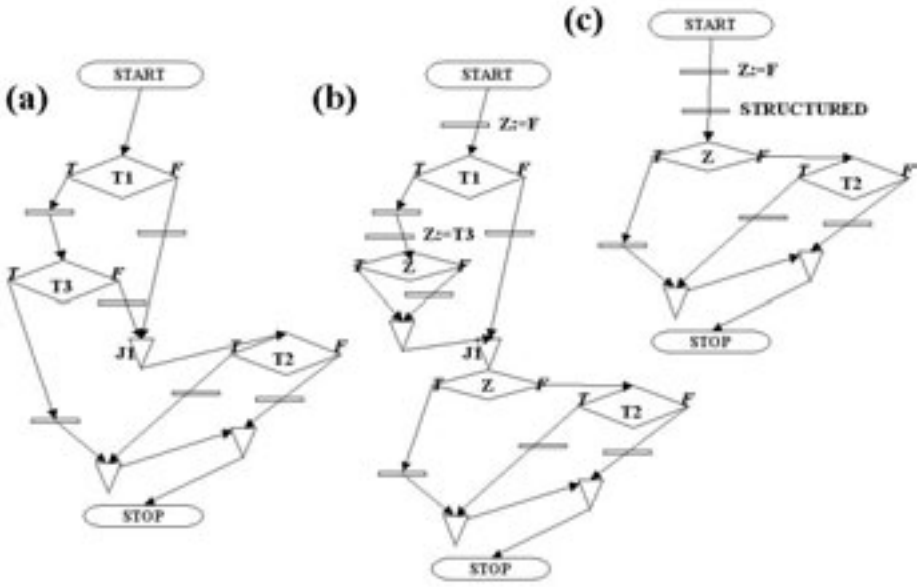

Abbildung 2: Behandlung zyklenfreier Programme

einen Test auf Z direkt nach dem Eintritt in den Zyklus verhindern wir die Ausführung der einfachen Anweisungen bis zu dem einstigen unerwünschten Eintrittspunkt. Auf diese Weise werden alle unerwünschten Eintrittspunkte entfernt. Ähnlich verlegen wir alle Austrittsverzweigungen bis zum letzten Austrittspunkt aus dem Zyklus.

Danach der Zyklus ist „sauber“ und kann durch eine einfache Anweisung ersetzt werden. In diesem Prozess wird die Anzahl der Zusammenführungen um 1 reduziert, so dass das Entfernen aller Zyklen garantiert ist. Jeder Graph lässt sich in einen zyklenfreien Graphen überführen, indem man einige Bögen (Pfeile) entfernt. Wir behandeln zuerst den Zyklus, der durch Rückerstellung des Bogens entsteht, der als „letzter“ unter den entfernten Bögen eingeht. Unerwünschte Eintrittspunkte sind die Kanten, die man in dem azyklischen Graphen vom Startpunkt auf dem Wege zum Endpunkt des Zyklus nicht durchquert, die aber in einen Knoten eingehen, der sich auf einem solchen Wege durchqueren lässt. Unerwünschte Austrittspunkte sind die Kanten, die man in dem azyklischen Graphen vom Startpunkt auf dem Wege zum Endpunkt des Zyklus nicht durchquert, die aber aus einen Knoten ausgehen, der sich auf einem solchen Wege durchqueren lässt.

Diese Vorangehensweise lässt sich auf Parallelanstöße und Parallelterminierungen ausdehnen, allerdings mit gewissen Einschränkungen. Erstens können bei der Erstellung eines azyklischen Graphen nur Bögen entfernt werden, welche in Zusammenführungen eingehen (und nicht etwa Terminrationen). Zweitens muss sich auf die Weise auch ein azyklischer Graph schaffen lassen. Drittens muss in dem azyklischen Graphen gewährleistet sein, dass auf jedem Pfad vom Anfangsknoten des Graphen die Anzahl der Parallelterminierungen die Anzahl der Parallelanstöße nicht übersteigt. Die Gründe dafür sind trivial: in einem wohlstrukturierten PAP kann man unendlich viele parallele Prozesse aus einer endlichen Struktur nicht generieren, und darum lassen sich nur solche Programmgraphen in eine gute Struktur umwandeln, die solche Prozesse nicht generieren können.

Damit sind wir also in der Lage, jedes Programm in ein strukturiertes Programm zu überführen. Allerdings ist darauf zu achten, dass man zusätzliche Variablen einsetzen muss, was als eine Konsequenz die Verlagerung der Programmstruktur auf die Datenstruktur bedeutet. Demzufolge ist eine so formal abgeleitete Struktur des Programms nicht unbedingt besser lesbar als das Original. Der Beweis hat hier eher die Bedeutung einer Anschauung, dass es keinen zwingenden Grund gibt, „GoTos“ einzuführen.

Nebenbei bemerkt ist die Beweisführung auch ein Hinweis darauf, wie man hierarchisch Sprungmarken für Anweisungen automatisch vergeben kann, so dass man sich auch in der Assemblerprogrammierung nicht besonders anstrengen muss, wenn man ein wohlstrukturiertes Programm in Code umsetzen will. Dies lag dem von mir erwähnten Makrosystem für Assembler zugrunde, in dem gänzlich auf Sprungmarken der Programmierer verzichten werden konnte, weil diese von den Makros automatisch vergeben wurden. Hierzu ist vorerst zu vermerken, dass „wilde Springe“ auch in höheren Programmiersprachen (lineares Rückgrat plus GoTos) Verwendung von Sprungmarken verlangen. Die strukturierte Programmierung aber kann in höheren Sprachen ohne Sprungmarken auskommen. Man verdankt dies der besonderen Graphstruktur, die sich auf eine besondere Weise linearisieren lässt.

7 Die Strukturierte Programmierung und Parallelverarbeitung

Die wohl wichtigste Eigenschaft des strukturierten Programmierens war, dass man sich anhand der Beschreibung eines Programmstücks sofort ein Bild über seine Einsetzbarkeit in einem größeren Programm machen konnte. Und wenn man sich ein algorithmisches Bild des Codes im Programmentwurf machen wollte konnte man das anhand der Unterkomponenten machen, ohne den Code zu durchwühlen. Hierfür war die Disziplin der Beschreibung der Ein- und Ausgangsvariablen von einer vorrangigen Rolle: Falls ein Zustand der Eingangsgrößen vorlag, konnte man den Ausgangszustand der Ausgangsgrößen aus der Beschreibung ablesen.

Problematisch war allerdings die Konstruktion der parallelen Verarbeitung. Man musste auf die interne Disziplin zurückgreifen, dass in den parallelen Zweigen nie dieselben Variablen verwendet wurden, bzw. dass man die Verwendung von gemeinsamen Variablen richtig bedenkt. Natürlich wurden die Synchronisationswerkzeuge, die auch heute angewandt werden, wie Semaphore, kritische Regionen oder Monitore auch gelehrt. Dies führte in die Welt der „Deadlocks“, die wieder durch „Struktur“ (Reihenfolge) der Ressourcenanforderungen (obgleich nicht effizient) gelöst werden konnten, was Gegenstand meiner Forschung wurde [Kło84, HK86].

Es muss hier noch ergänzt werden, dass die Parallelität der Prozesse nicht nur die Konkurrenz um Ressourcen, sondern auch Kooperation verlangen kann. Bei der vorhergehenden Diskussion über die Umwandlung in die strukturierte Form könnte man synchrone Kooperation in Form von Parallelterminationen mit anschließendem Parallelanstoß darstellen, während für die asynchrone Kooperation die Zwischenschaltung eines Prozesses ausreicht, der in einem Prozess angestoßen und in einem anderen zusammengeführt wird.

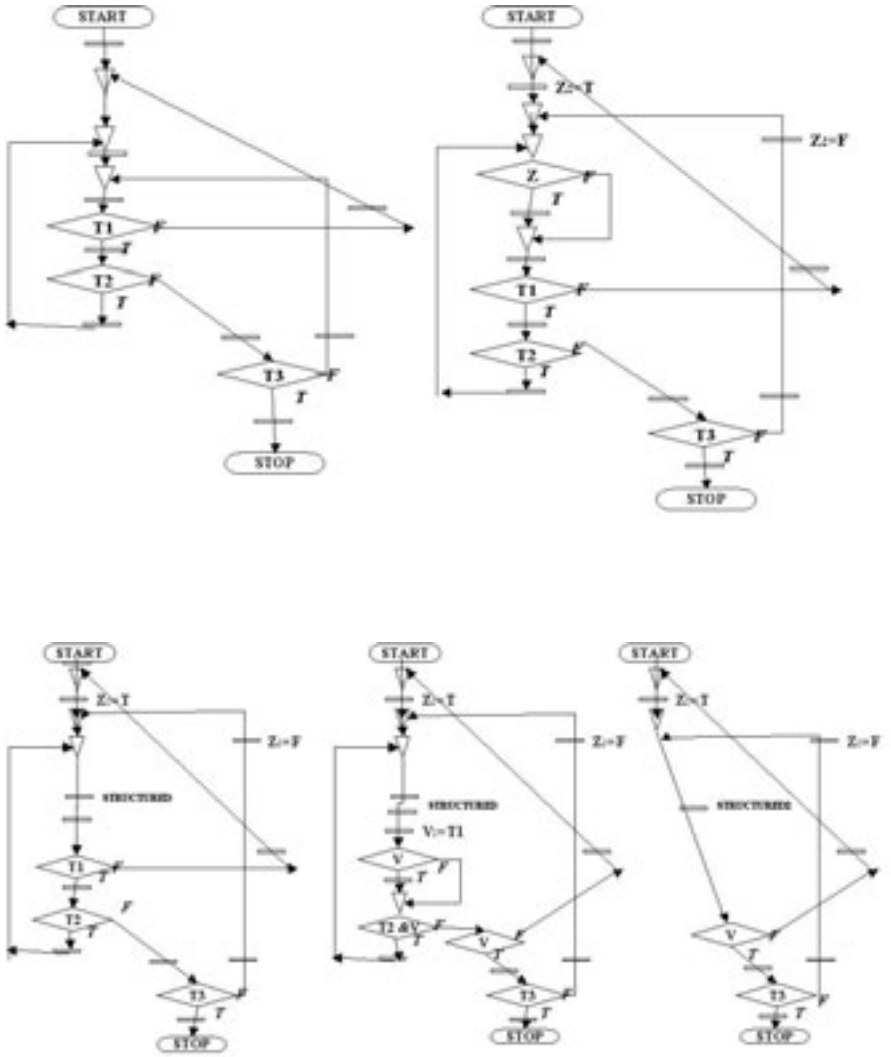

Abbildung 3: Behandlung der Zyklen

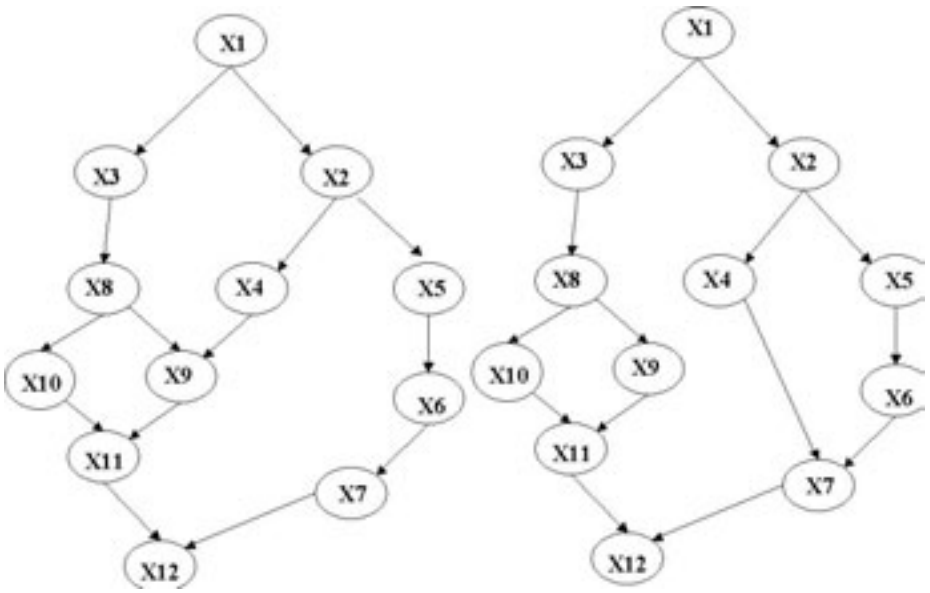

Abbildung 4: Allgemeine (links) und strukturierte (rechts) Bayesche Netzwerke

Natürlich ist der PAP-Parallelismus heute nicht die einzige und auch damals gekannte Form der Parallelverarbeitung. Ein Bestandteil der Lehre waren auch die Petri-Netze in verschiedenen Formen. Algorithmen mit massiver Parallelität (single instruction stream, multiple data stream) wurden auch in der Lehre angeschnitten.

8 Die Strukturierte Programmierung und andere Graphstrukturen

Überraschenderweise griff ich nach Jahren auf den Gedanken des Strukturierbarkeitsbeweises zurück, als ich an der Inferenz in den sogenannten Bayesschen Netzwerken arbeitete.

Die Bayesschen Netzwerke [Jen96] bieten eine kompakte Darstellung einer Wahrscheinlichkeitsverteilungsfunktion in mehreren Variablen, indem die relative Unabhängigkeit der Variablen ausgenutzt wird. .

Eine beliebige Wahrscheinlichkeitsverteilungsfunktion in diskreten Variablen kann man als

$$P(x_1, \ldots, x_n) = \prod_{i=1,\ldots,n} P(x_i | x_1, \ldots, x_{i-1})) \tag{1}$$

darstellen.

Falls wir wissen, dass X_i von manchen Variablen von den $X_1, \ldots, X_{i-1}$ (statistisch) nicht

direkt abhängt, vereinfacht sich die Struktur zu:

$$P(x_1, \ldots, x_n) = \prod_{i=1,\ldots,n} P(x_i|\pi(x_i)) \tag{2}$$

wo $\pi(X_i)$ den Satz der „Eltern“ der Variable X_i darstellen, d.h. der Variablen, von denen X_i direkt abhängt. Symbolisch wird diese Formel in der graphischen Form eines Orientierten Azyklischen Graphen (dag) dargestellt (Abb.4). Die Knoten stehen hier für Variablen und die Kanten verbinden die Eltern mit der von ihnen direkt abhängenden Variable.

Das Bayessche Netzwerk stellt zwar eine Wahrscheinlichkeitsverteilungsfunktion kompakt dar, aber trotzdem ist die Inferenz schwierig. Nur die graphischen Strukturen eines Baumes oder Polybaumes erlauben eine effiziente Inferenz.

Ich überlegte, welche Voraussetzungen nötig sind, um die Klasse der Bayesischen Netzwerke, für die die effiziente Inferenz relativ einfach ist, über die Baum- und Polybaumstrukturen hinaus auszudehnen. Bekanntlich muss man ein Bayessches Netzwerk in den sogenannten Markovschen Baum umwandeln, in dem die Inferenz einfach ist, aber die Umwandlung selbst ist NP-komplex (mit Ausnahme eben der Bäume und Polybäume). Das Charakteristische an der Inferenz im Markovschen Baum ist die „Reduktion“ der Baumstruktur im Laufe der Vermittlung der sogenannten Nachrichten. Das erinnerte an die Eigenschaft der wohlstrukturierten Programme.

So definierte ich strukturierte Bayesische Netzwerke nach dem Vorbild der strukturierten Programme [Kło03b], und bewies (nach dem Vorbild der strukturierten Programmierung), dass sich jedes Bayesische Netzwerk in einer strukturierten Form darstellen lässt [Kło04] Ich entwickelte auch Methoden, wie man solche strukturierte Bayesschen Netzwerke aus Daten gewinnt [Kło03a].

9 Abschließende Bemerkungen

Am Ende muss man noch einmal auf den wichtigen Unterschied zwischen der automatisch aus einem beliebigen Graphen erzeugten strukturierten Form des Programms oder Bayesschen Netzwerkes und der menschlischen Entwicklung des wohlstrukturierten Graphen hinweisen. Die Komplexität der inneren Struktur wird nämlich von dem Automaten aus der Programm- oder Netzwerkstruktur auf die Datenstrukturen verschoben. Durch die automatische Strukturierung wird also das Programm nicht übersichtlicher.

Aber das Komplexitätsphänomen ist auch bei menschlich erstellten Programmen ein Problem. Dies war bereits damals sichtbar, als man in das einheitliche Konzept der strukturierten Programmierung auch die Parallelverarbeitung integrieren wollte und die Synchronisation außer dem allgemeinen Rahmen fiel. Dieses Problem bleibt übrigens bis heute ungelöst. So bleibt also immer noch die wichtige Frage für die Forscher, wie man durchschaubare Strukturen sowohl für die Programme als auch für die Daten schaffen kann, in denen der Arbeitsaufwand des Programmierers in Grenzen gehalten wird. Einer der Gutachter dieses Beitrages hat darauf hingewiesen, dass Software-Objekte und

-Komponenten als die Mittel angesehen werden können, parallele Programmstrukturen zu finden, die den Arbeitsaufwand eines Programmierers bei der Entwicklung von parallelen Programmstrukturen in Grenzen halten. Ich stimme dieser Meinung zu, allerdings mit dem Vorbehalt, der dem bekannten Buch von S. Wolfram *The new kind of science* entnommen werden kann, dass die von dem Menschen beherrschten konzeptuellen Strukturen sehr klein im Vergleich zu den möglichen sind, wodurch auch das vom Menschen Schaffbare beschränkt ist. Es müssen neue strukturierte Formen auf ihre Brauchbarkeit und Beherrschbarkeit experimentell überprüft werden, wenn der technische Fortschritt voranschreiten soll. Und dies gilt auch für die Programmierung. Und dennoch (oder gerade deswegen) wird das Prinzip der Zusammensetzung, welches der Strukturierten Programmierung zugrunde liegt, noch lange ein wichtiger Bestandteil des Softwarengineering als auch der Forschung und Lehre bleiben.

Literatur

[AM71] Edward A. Ashcroft und Zohar Manna. The Translation of 'Go To' Programs to 'While' Programs. IFIP Congress (1) 1971: 250-255, 1971.

[BJ66] C. Bohm und G. Jacopini. Flow Diagrams, Turing Machines, and Languages with Only Two Formation Rules. Communications of the ACM 9:5, p. 266, May 1966., 1966.

[Dij68] E. Dijkstra. goto Statement Considered Harmful. Communications of the ACM 11:3, p. 147, March 1968., 1968.

[Dij82] E. Dijkstra. Stepwise Program Construction. Printed in Selected Writings on Computing: A Personal Perspective, Springer-Verlag, 1982, p. 2., 1982.

[HK86] O. Herrlich und M. A. Kłopotek. Hierarchische Betriebsmittelvergabe. Rechentechnik/Datenverarbeitung 23(1986)10, pp. 30-33., 1986.

[Jen96] J. Jensen. An Introduction to Bayesian Networks, Springer Verlag, 1996.

[Kło84] M. A. Kłopotek. Ein Beitrag zur statischen Verklemmungsvermeidung in Systemen nichtbinärer Betriebsmittel. Dissertationsarbeit, Technische Universität Dresden, Sektion Informationsverarbeitung, 1984.

[Kło03a] M. A. Kłopotek. Reasoning and Learning in Extended Structured Bayesian Networks. Fundamenta Informaticae. 58(2)2003, pp.105-137, 2003.

[Kło03b] M. A. Kłopotek. Reasoning in Structured Bayesian Networks IN Rutkowski, L., Kacprzyk, J., (Eds.) *Neural Networks and Soft Computing*. Sixth International Conference on Neural Network and Soft Computing, Zakopane, Poland, June 11-15, 2002, Advances in Soft Computing. Springer-Verlag, ISBN 3-7908-0005-8, pp. 418-423, 2003.

[Kło04] M. A. Kłopotek. Impact of Structuring on Bayesian Network Learning and Reasoning in:P. Grzegorzewski, M. Krawczak, S. Zadrozny eds.: Soft Computing Tools, Techniques and Applications. Akademicka Oficyna Wydawnicza EXIT Warszawa. pp.77-118, 2004.

[Kos74] S R Kosaraju. Analysis of structured programs J Comptr Syst Sct 9, 3 (1974), 232-254, 1974.

[Mil75] H. D. Mills. The New Math of Computer Programming. Com.ACM Jan. 1975, vol.18, Number 1, pp.43, 1975.