Baumstrukturierung von Funktionsprozeduren

Eine Funktionsprozedur ist ein ergebnisorientiertes Programm. Falls wir das Programm in Form eines Petrinetzes darstellen, stehen die Anweisungen in den Transitionen. Nun müssen wir der Frage nachgehen, wie die Struktur des Petrinetzes in Textform in den Programmiersprachen zu formulieren ist. Die Struktur des Petrinetzes besteht ja darin, dass einerseits Transitionen mit Stellen und andererseits Stellen mit Transitionen über Pfeile verbunden sind.

Heute gibt es den Begriff der wohlstrukturiereten Programme, die man gegen die wildstrukturierten oder chaotisch strukturierten Programme klar abgrenzen kann. In der Anfangszeit des Programmierens, die erst in der zweiten Hälfte der 60-iger Jahre endete, kannte man diese Unterscheidung noch nicht. Als ich im Jahre 1960 programmieren lernte, war es noch selbstverständlich, dass man ein imperatives Programm dadurch entwickelte, dass man sich von Transition zu Transition bewegte, wie dies in Bild 17 an einem Beispiel gezeigt ist. Jede bereits eingeführte Stelle im Petrinetz wurde als zulässiges Ziel für später hinzukommende Transitionsausgangspfeile angesehen.

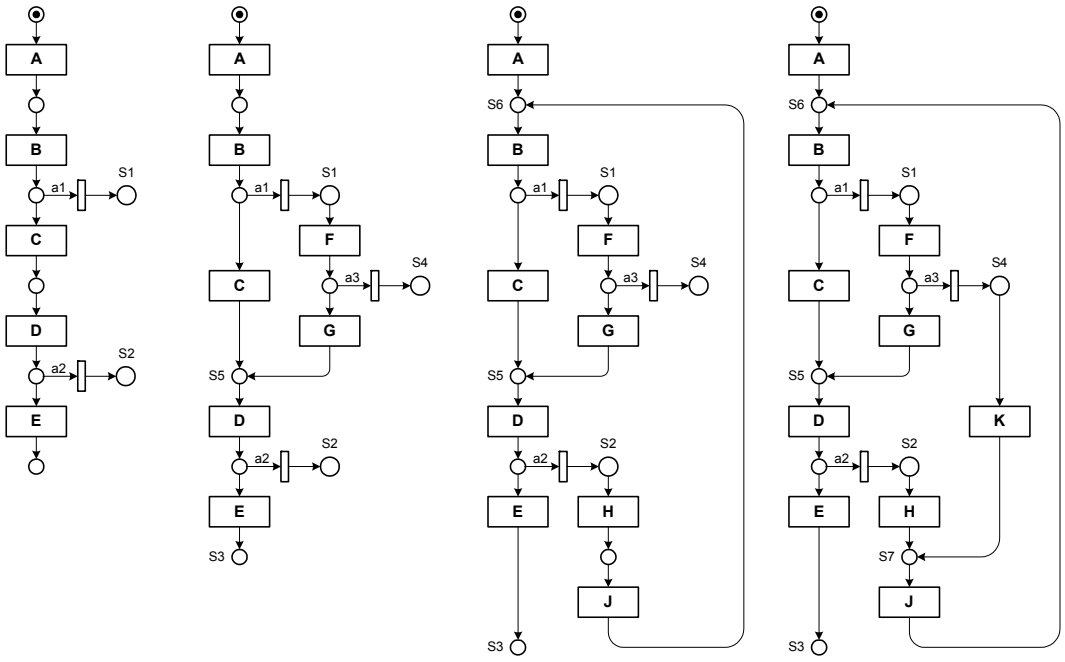


Bild 17 Beispiel einer schrittweisen Programmentwicklung ohne Bemühen um Wohlstrukturiertheit

Wenn man dann ein so entstandenes Programm in eine programmiersprachliche Textform fassen musste, war man gezwungen, immer wieder einmal das sogenannte GOTO-Statement zu benutzen. Die Entwicklung der zu Bild 17 gehörenden Textform ist in Bild 18 gezeigt.

```
PROGRAM BEGIN

    A;
    B;
    IF a1 THEN GOTO S1;
    C;
    D;
    IF a2 THEN GOTO S2;
    E;

PROGRAM END
```

```
PROGRAM BEGIN

    A;
    B;
    IF a1 THEN GOTO S1;
    C;
S5: D;
    IF a2 THEN GOTO S2;
    E;
    GOTO S3;

S1: F;
    IF a3 THEN GOTO S4;
    G;
    GOTO S5;

S3: PROGRAM END
```

```
PROGRAM BEGIN

    A;
S6: B;
    IF a1 THEN GOTO S1;
    C;
S5: D;
    IF a2 THEN GOTO S2;
    E;
    GOTO S3;

S1: F;
    IF a3 THEN GOTO S4;
    G;
    GOTO S5;

S2: H;
    J;
    GOTO S6;

S3: PROGRAM END
```

```
PROGRAM BEGIN

    A;
S6: B;
    IF a1 THEN GOTO S1;
    C;
S5: D;
    IF a2 THEN GOTO S2;
    E;
    GOTO S3;

S1: F;
    IF a3 THEN GOTO S4;
    G;
    GOTO S5;

S2: H;
S7: J;
    GOTO S6;

S4: K;
    GOTO S7;

S3: PROGRAM END
```

Bild 18 Textform zu Bild 17

Im folgenden werde ich das Kriterium der Wohlstrukturiertheit einführen, und wenn wir dann dieses Kriterium auf das Programm in Bild 17 anwenden, werden wir feststellen, dass dieses Programm nicht wohlstrukturiert ist.

Edsger Dijkstra, einer der Informatik-Pioniere aus der Frühzeit des Programmierens, schrieb einen Aufsatz mit dem Titel: „The GOTO-Statement Considered Harmful“, mit dem er die Unterscheidung zwischen wohlstrukturierten und chaotisch strukturierten Programmen einführte. Leider führte der Titel von Dijkstras Aufsatz dazu, dass häufig die Wohlstrukturiertheit schlicht durch die Abwesenheit von GOTO-Statements definiert wird. Dadurch wird natürlich das Verständnis des Wesens nicht gefördert, denn Wohlstrukturiertheit kann ja nicht einfach durch das Vermeiden eines programmier-sprachlichen Wortes erreicht werden.

Wir betrachten nun das Kriterium, an Hand dessen wir feststellen können, ob ein imperatives Programm wohlstrukturiert ist oder nicht. Ich habe schon gesagt, dass dieses Kriterium nicht in der Abwesenheit des GOTO-Statements liegt. Ein imperatives Programm wird als wohlstrukturiert bezeichnet, wenn es schrittweise verfeinert werden kann, wobei bei jedem Verfeinerungsschritt nur eine von drei zulässigen Verfeinerungsformen vorkommen dürfen. Ein Verfeinerungsschritt besteht jeweils darin, dass eine Transition durch ein Teilnetz ersetzt wird. Das ersetzende Teilnetz muss entweder eine Sequenz oder eine Fallunterscheidung oder eine Wiederholung sein. In Bild 19 ist eine solche schrittweise Verfeinerung anhand eines Beispiels gezeigt. Die Verfeinerung kann in Form eines Strukturbaums dargestellt werden, worin jeder Knoten eine Transition erfasst. Die Blätter im Strukturbaum sind diejenigen Transitionen des Programms, die nicht weiter verfeinert werden können. Im Baumgraphen unterscheiden wir die Knoten durch unterschiedliche Knotensymbole: runde Knoten stehen für verfeinerbare Transitionen, rechteckige Knoten stehen für Blätter.

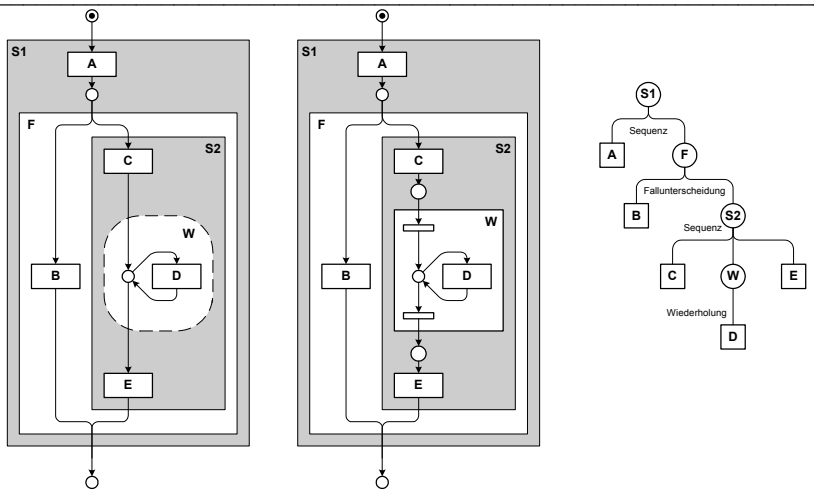
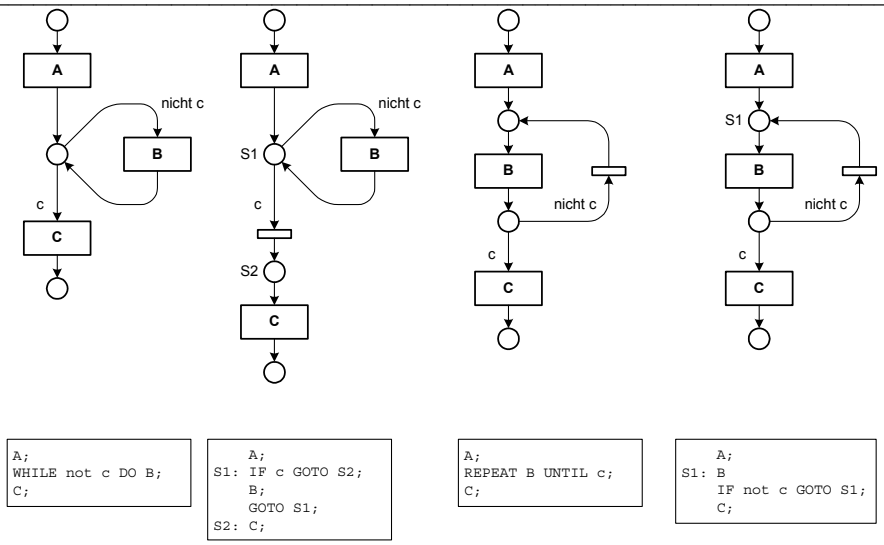


Bild 19 Baumstrukturiertes imperatives Programm

Das Petrinetz und der Strukturbaum sind äquivalente Darstellungen ein und desselben strukturellen Sachverhaltes, d.h. aus dem Baum kann das Petrinetz gewonnen werden, und aus dem Petrinetz kann der Baum gewonnen werden.

Wenn Schleifen in Petrinetzen vorkommen, kann es erforderlich sein, das Petrinetz zuerst noch strukturell zu erweitern, bevor man eine klare schrittweise, ausschließlich Transitionen betreffende Verfeinerung vornehmen kann. Die Begründung hierfür erkennt man in Bild 19. In dem Petrinetz gibt es eine Sequenz aus der Transition C, der Schleife W zur Wiederholung der Transition D, und der Transition E. In der links dargestellten Petrinetzform erscheint aber die Schleife nicht als verfeinerbare Transition. Dies ist erst in der rechten Darstellung im Bild 19 erkennbar. Hier ist die Schleifentransition dadurch entstanden, dass zwei unbeschriftete Transitionen, die keinen Anweisungen entsprechen, eingefügt wurden. Der zugehörige Baum steht rechts daneben.

Der Sachverhalt, dass bei Schleifen möglicherweise Abkürzungsformen im Petrinetz oder im Programmtext vorkommen, bei denen eine Sequenz und eine Schleife zu einer graphischen Einheit verschmolzen sind, führt auch bei den programmiersprachlichen Formulierungen zu Varianten. Man betrachte hierzu das Bild 20. In Bild 20 ist auch gezeigt, dass man diese Strukturen programmiersprachlich durchaus unter Verwendung des GOTO-Statements formulieren kann. Diese Verwendung des GOTO-Statements ist sehr wohl zulässig, d.h. dadurch wird die Wohlstrukturiertheit nicht zerstört. Bei der maschinennahen Programmierung gibt es solche Anweisungen wie `WHILE` und `REPEAT` ohnehin nicht, d.h. dort muss mit dem GOTO-Statement, welches dort Sprunganweisung genannt wird, gearbeitet werden.




```
A;
WHILE not c DO B;
C;
```

```
    A;
S1: IF c GOTO S2;
    B;
    GOTO S1;
S2: C;
```

```
A;
REPEAT B UNTIL c;
C;
```

```
    A;
S1: B
    IF not c GOTO S1;
    C;
```

Bild 20 Schleifenvarianten

Was ist der Grund, weshalb imperative Programme baumstrukturiert formuliert werden sollen? Jede Transition symbolisiert eine berechenbare Funktion, die auf ein bestimmtes Argument angewandt werden soll. Eine Transition, die eine Struktur aus kleineren Transitionen ist, stellt eine Funktion dar, die aus einfacheren Funktionen zusammengesetzt ist. Da auch das ganze Programm als nur eine einzige Transition betrachtet werden kann, stellt auch das Programm eine Funktion dar. Durch die Baumstrukturierung wird es nun möglich, zu beweisen, dass ein gegebenes Programm tatsächlich die geforderte Funktionsberechnung realisiert und sonst nichts. Korrektheitsbeweise für Algorithmen beruhen immer auf dieser Baumstrukturierung.

Neben der imperativen Programmierung, die verständlicherweise über 90 % aller Programmierung ausmacht, gibt es für besondere Anwendungsbereiche noch andere Formen der Programmierung. Bevor wir nun in die Behandlung der sog. funktionalen Programmierung eintreten, betone ich noch einmal, was das Wesen der imperativen Programmierung ist. Jede Anweisung verlangt eine Zustandsänderung, und es werden nacheinander Zustandsänderungen verlangt. Dadurch wird der Anfangszustand, der zu Beginn des Programmlaufes vorliegt, schrittweise in den Endzustand überführt. Eine Anfangssituation kennzeichnen wir immer durch die lateinische Bezeichnung PRE, und eine Endesituation durch die Bezeichnung POST. Wenn man Zustandsveränderungen hintereinander hängt, bedeutet das, dass der POST-Zustand einer Zustandsänderung zum PRE-Zustand der folgenden Zustandsänderung wird. Auf diese Weise hangelt man sich von einem Zustand zum nächsten durch das ganze Programm.

****