Teil IV

Weitere Aspekte des Testens

Dieser Teil des Buches beschäftigt sich mit verschiedenen Aspekten des Testens im weiteren Sinne.
Kapitel 12 behandelt die statische Analyse von Programmen. Die informelle Analyse ist bei informellen Dokumenten — vor allem in frühen Phasen der Software-Entwicklung — die einzige Möglichkeit, da sie nicht durch Tests ausgeführt werden können. Für formale Dokumente, z. B. Programme, ist dagegen eine formale statische Analyse möglich und sinnvoll. Damit können einerseits Fehler entdeckt und sofort lokalisiert werden, andererseits können Kontrollflußgraphen, Datenflußgraphen etc. statisch ermittelt werden, die beim Testen benötigt werden (s. vorhergehenden Teil III). Zum Testen im weiteren Sinn gehören auch noch die Ausführung von Programmen mit symbolischen Werten und die formale Programmverifikation, die eine optimale Methode zur Zertifizierung korrekter Programme wäre, wenn sie keine theoretischen und praktischen Einschränkungen hätte.

Die Testmethoden von Teil II und III stoßen an Grenzen, wenn sie auf große Systeme angewandt werden, die aus vielen Modulen, Komponenten bzw. Subsystemen bestehen: Eine Grenze ist durch die massiv steigende Komplexität der Testverfahren gegeben, wenn sie auf komplette Systeme angewandt werden; werden dagegen die Module einzeln getestet, um die Komplexität zu reduzieren, bewirken die Aufrufe (die Benutzung) von Operationen anderer Module konzeptionelle Probleme beim Testen eines Moduls. Daher werden die besonderen Strategien und Verfahren des Modul- und Systemtests und des Integrationstests von (Sub-)Systemen und die notwendige Modellbildung für solche Systeme in Kapitel 13 behandelt.

Nebenläufige Systeme bereiten wegen ihres nichtdeterministischen Verhaltens besondere Probleme beim Testen, insbesondere wenn es verteilte Systeme oder Echtzeit-Systeme sind. Diese Systeme müssen wiederum besonders modelliert werden und die statische und dynamische Analyse dieser Systeme muß auf die besonderen Probleme mit neuen oder modifizierten Lösungsansätzen eingehen (s. Kapitel 14).

Da Testen kein Selbstzweck ist, sondern letztlich zur Fehlerreduzierung beitragen soll, sind entsprechende Verfahren zur Lokalisierung der Ursachen des Fehlverhaltens und zur Beseitigung der Fehler anzuwenden. Damit beschäftigt sich Kapitel 15.

In diesem Buch wird eine Fülle von Testmethoden vorgestellt. Aus Aufwandsgründen können aber nicht alle Methoden angewandt werden. Daher werden in Kapitel 16 Komplexitätsmaße vorgestellt, die für die Auswahl der Methoden herangezogen werden können. Außerdem werden Vorschläge für die sinnvolle Kombination verschiedener Methoden gemacht. Abschließend werden spezielle Managementfragen angesprochen, insbesondere Kriterien für die Entscheidung über die Beendigung des Testens.

Kapitel 17 gibt nach einer Zusammenfassung einen Ausblick auf die Testprobleme bei objektorientierten, funktionalen und logischen Programmen und weitere offene Probleme des Testens.

12 Statische Analyse und symbolische Ausführung

Die statische Analyse (auch statischer Test genannt) und die symbolische Ausführung eines Programms sind Vorgänge, die ein Programm nicht mit konkreten Eingabewerten, sondern „konzeptuell“ ausführen. Folgende Dokumente kommen für die statische Analyse in Betracht, da nicht erst das fertige Programm, sondern auch andere Produkte und Dokumente, die bei der Programmentwicklung in früheren Phasen anfallen, analysiert werden sollten:

1. Anforderungsspezifikation und Systemspezifikation
2. Entwurfsspezifikation (Grobentwurf und Feinentwurf)
3. Programm in Quellsprache (Quellcode)
4. Programm in Zielsprache (Objektcode)

Insbesondere die Dokumente zu 1 und 2 kommen für eine statische Analyse in Betracht, da sie i. allg. nicht ausführbar sind. Dabei sind folgende Aspekte zu überprüfen:

1. <u>Spezifikationsanalyse</u>
   - Notwendigkeit (für Systemziele)
   - Vollständigkeit (bezüglich Eingaben, Ausgaben, zu behandelnden Fällen, Umgebung, Leistung, Zuverlässigkeit, Benutzung)
   - Konsistenz (Anforderungen untereinander, Einheitlichkeit numerischer Angaben [z. B. nur cm, m oder km])
   - Durchführbarkeit (bei gegebener Hardware/Technologie)
   - Eindeutigkeit/Testbarkeit (Begriffe und Sätze eindeutig; Transformationen eindeutig; beim Testen eindeutige Entscheidbarkeit, ob eine Anforderung erfüllt ist; Wartbarkeit)
2. <u>Entwurfsanalyse</u>
   (a) Notwendigkeit (bzgl. Anforderungspezifikation)
   (b) Vollständigkeit (bzgl. Anforderungspezifikation)

(c) Konsistenz (Modulschnittstellen; Formate von Eingaben, Datenbanken und Dateien)

(d) Korrektheit (Algorithmen, mathematische Gleichungen, Kontroll-Logik des Feinentwurfs)

Es stellt sich nun die Frage, welche Methoden als statische Analysemethoden für die genannten Dokumente bzw. Objekte in Frage kommen. Im Prinzip sind folgende Methoden bzw. Vorgehensweisen dafür geeignet:

- menschliche Begutachtung (für 1 und 2)
- mathematische Analyse (für 2d)
- Numerierung der Anforderungen (für 2a, 2b)
- Analyse der Transformation der Anforderungen (für 2a, 2b, vgl. [Zur 90], Stichwort „Anforderungsflußanalyse“)
- Schnittstellenüberprüfung (für 2c)

Die Methoden können in informelle, nur manuell ausführbare Methoden und in formale, automatisch ausführbare Methoden eingeteilt werden.

12.1 Informelle Analyse

„Begangene Fehler können nicht besser entschuldigt werden als mit dem Geständnis, daß man sie als solche wirklich erkenne.“
— **Calderon**

Die informelle („manuelle“) Analyse ist bei nicht formalen Dokumenten (z. B. der Anforderungsspezifikation) notwendig. Bei formalen Dokumenten (z. B. Programmtexte) kann sie sinnvoll sein. Dieses Vorgehen ist also in jeder Phase des Software-Lebenszyklus möglich. Damit sind insgesamt die in Abb. 12.1 dargestellten Konstruktions- und Inspektionsschritte angebracht.

Um Fehler früh zu finden (und damit Kosten für schwieriges spätes Korrigieren zu sparen[1]) sind folgende Inspektionen vorzusehen:

- Inspektion der internen Spezifikation (I0)
- Inspektion der Logik-Spezifikation als Entwurfsabnahme-Inspektion (I1)
- Inspektion des Codes (I2)

[1] frühe Korrekturen sind 10 bis 100mal billiger; s. Kapitel 2.5

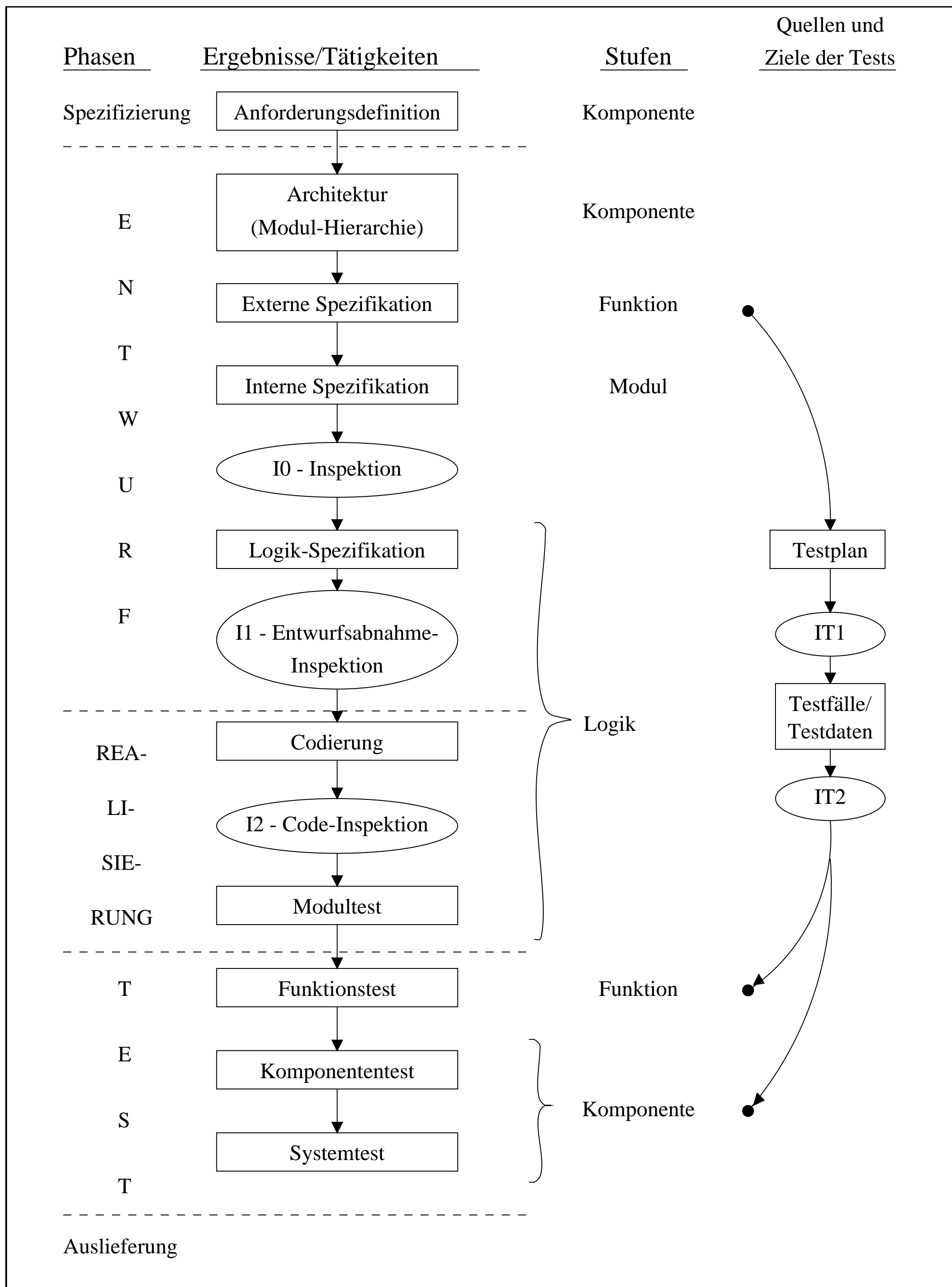

Abb. 12.1: Konstruktions- und Inspektionsschritte im Lebenszyklus der Software (nach [Fag 76], S. 105)

Außerdem sollten parallel zur Entwicklung die Testpläne, Testfälle und Testdaten entwickelt und überprüft werden:

- Inspektion des Testplans (IT1)
- Inspektion der Testfälle und Testdaten für den Funktions- und Komponententest (IT2)

Ziel von I0, I1 und I2 ist es, die Qualität der Dokumente der frühen Phasen der Entwicklung zu messen und zu beeinflussen (durch Verminderung des Fehlergehalts). Ziel von IT1 ist es, Lücken bei der Funktionsabdeckung und andere Diskrepanzen im Testplan zu entdecken und zu beseitigen. Ziel von IT2 ist es, Fehler in den Testfällen und Testdaten zu finden (z. B. zuwenig Testfälle, falsche erwartete Ausgaben). Gemeinsames Ziel von IT1 und IT2 ist es somit, die „Integrität" des Testens (und damit die Qualität der Software) zu erhöhen.

Zusätzlich sollte auch die Qualität der Dokumentation überprüft werden, da der Benutzer ein Software-Produkt nur sinnvoll verwenden kann, wenn es korrekt arbeitet und die notwendigen Eingaben und die erwarteten Ausgaben im Benutzerhandbuch korrekt und verständlich beschrieben sind. Daher sind Publikations-Inspektionen PI0, PI1 und PI2 für Benutzer-, Wartungs- und Installationshandbuch vorzusehen.

12.1.1 Vorgehen bei der Inspektion

Inspektionsteilnehmerinnen
Die an einer Inspektion teilnehmenden Personen haben vor oder während der Sitzung folgende Rollen:

1. Moderation (Moderatorin[2])
   Dies ist die „Schlüssel"-Rolle und sollte daher von einer Person mit besonderen Qualifikationen (Kompetenz in der Programmierung, Fingerspitzengefühl für persönliche Probleme) wahrgenommen werden. Um sich nicht von anderen Projektzielen beeinflussen zu lassen und persönlich unabhängig von den anderen teilnehmenden Personen zu sein, sollte die Moderation möglichst nicht von einem Projektmitglied ausgeübt werden. Zu den Aufgaben der Moderation gehört:
   - Zusammenarbeit anregen (Synergieeffekt erzielen),
   - Organisation (Dokumente prüfen, verteilen; Sitzungsräume reservieren),
   - Inspektionsbericht erstellen,
   - Änderungen verfolgen.

[2] Wie im Vorwort angekündigt, wird in diesem Kapitel bei Personen- und Berufsbezeichnungen nur die *weibliche* oder eine neutrale Form verwendet.

2. Entwurf (Entwerferin)
3. Codierung (Codiererin)
4. Test (Testerin)

I. allg. sollten höchstens vier Personen an den Inspektionen teilnehmen[3]. Eine Handbuchautorin sollte bei I0- und I1-Inspektionen mitwirken, ein Mitglied der Wartungsabteilung an I1- und I2-Inspektionen. Wenn ein Modul viele Schnittstellen hat, sollte je eine Entwicklerin der benutzten bzw. benutzenden Module teilnehmen. Wenn die Codiererin das Programmstück auch entworfen hat, soll sie in der Inspektion die Rolle der Entwerferin spielen und eine Codiererin eines ähnlichen Projektes die Rolle der Codiererin übernehmen. Wenn eine Person das Programmstück entworfen, codiert und getestet hat, soll zusätzlich eine weitere Codiererin (mit Erfahrung im Testen) die Rolle der Testerin übernehmen.

Zeitdauer der Inspektions-Sitzungen

Die Sitzungen sollten nicht länger als zwei Stunden dauern. Zwei Sitzungen à zwei Stunden sind pro Tag akzeptabel.

Inspektionsschritte

1. Überblick (Gruppensitzung)
2. Vorbereitung (individuell)
3. Eigentliche Inspektion (Gruppensitzung)
4. Überarbeitung/Korrektur (Autorin)
5. Verfolgung/Überprüfung (Moderatorin)
6. Auswertung/Nachbereitung (Moderatorin)
7. Inspektion als Einschub beim dynamischen Testen

Schritte und Ziele der Inspektion im einzelnen[4]

1. Überblick (Gruppensitzung)
   - Die Entwerferin gibt einen mündlichen Überblick über das spezielle Programmteil und die Einordnung in die allgemeine Problemstellung. (Dies entfällt bei einer I2-Inspektion mit denselben Personen wie bei I1.)

[3] Nach einer neuen Studie von Porter et al. finden Inspektionsgruppen mit zwei Personen ebensoviele Fehler wie Gruppen mit vier Personen — sind also kostengünstiger und ebenso effektiv (s. [Po& 95]).

[4] Die folgenden Bemerkungen beziehen sich auf I1- und I2-Inspektionen, andere Inspektionen (IT1, IT2, sowie die Dokumentations-Inspektionen PI0 bis PI2) haben im wesentlichen gleiche Eigenschaften, unterscheiden sich aber im Inspektionsmaterial und der Zahl der Teilnehmerinnen.

- Außerdem wird schriftliches Material verteilt:
  - Entwurfsdokumentation (bei I1),
  - zusätzlich Programmtext und Hinweise auf geänderte Teile und Fehler, die seit I1 gefunden wurden (bei I2).

2. Vorbereitung (individuell)
   In Einzelarbeit ist folgendes zu tun:

   - Durchlesen der Entwurfsdokumentation, um ihre Logik und Intention zu verstehen. (Dabei müssen noch keine Fehler gefunden werden.)
   - Studieren der Liste der Fehler (mit Häufigkeitsangaben), um die fehlerträchtigsten Konstrukte kennenzulernen.
   - Studieren der Checklisten (mit Anhaltspunkten, wie die Fehler gefunden werden können).

   Beispiel 12.1.1 (Auszug aus einer Checkliste für die Entwurfsinspektion I1)

   (a) *Sind alle Konstanten definiert?*
   (b) *Sind bei Eingabe-Parametern alle speziellen Werte explizit getestet/abgefragt worden?*
   (c) *Werden alle Werte nach ihrer Berechnung gespeichert?*
   (d) *Werden alle (Inkrement-)Zähler richtig initialisiert (0 oder 1)?*

   Beispiel 12.1.2 (Auszug aus einer Checkliste für die Code-Inspektion I2)

   (a) *Wird die richtige Bedingung abgefragt (x=on statt x=off)?*
   (b) *Werden korrekte Variablen abgefragt (x=on statt y=on)?*
   (c) *Sind leere then-/else-Zweige richtig verwendet worden?*
   (d) *Sind die Sprungziele (bei goto) korrekt?*
   (e) *Ist der am häufigsten ausgeführte Zweig der then-Zweig?*

3. Eigentliche Inspektion (Gruppensitzung)
   Eine von der Moderatorin ausgewählte „Leserin“ (i. allg. die Codiererin) beschreibt, wie sie den Entwurf implementieren wird bzw. implementiert hat. Sie soll dabei den Entwurf mit ihren Worten umschreiben. Jedes Stück Logikentwurf bzw. Logik und jeder Zweig wird mindestens einmal angesprochen.
   Als Material müssen während der Sitzung die Dokumentation der Entwurfsspezifikation (externe, interne und Logik-Spezifikation) sowie bei I2 die Programmtexte vorliegen.

Ziel der Sitzung ist es, Fehler zu finden:

- Dies geschieht i. allg. während des Vortrags der Codiererin.
- Fragen und Probleme werden nur so weit verfolgt, bis ein Fehler erkannt wird. (Die Ursachen des Fehlers und seine Behebung sollen nicht geklärt werden[5].)
- Fehler werden von der Moderatorin notiert und klassifiziert (z. B. als schwerwiegend oder leicht).
- Offensichtliche Lösungen und Korrekturmöglichkeiten werden allerdings notiert.

Als Ergebnis ist innerhalb eines Tages von der Moderatorin ein schriftlicher Bericht der Inspektionssitzung zu erstellen, in dem alle Fehler vermerkt und bewertet sind. Teil 1 dieses Berichts enthält die Fehlerliste, Teil 2 eine Übersicht.

Beispiel 12.1.3 (Fehlerliste: Beschreibung eines Fehlers)
LO/W/MAJ: Zeile 172: NAME-CHECK wird einmal zu wenig ausgeführt.
Fehlersorte: LO $\hat{=}$ *Logik*
Fehlerart: W $\hat{=}$ *wrong (falsch) (sonst: Missing/Extra)*
Fehlertyp: MAJ $\hat{=}$ *major (schwerwiegender (Funktions-)Fehler)*
(sonst: MIN $\hat{=}$ *minor* $\hat{=}$ *leichter (Form-)Fehler)*

Beispiel 12.1.4 (Übersicht)
Modul: Checker (Legende: M, W, E wie bei Beispiel 12.1.3)

Fehlersorte	Funktionsfehler (schwerwiegend)			Formfehler (leicht)			Offene Fragen
	M	W	E	M	W	E	
LO: Logik	1	9			1		
TV: Test und Verzweigung							
⋮							
PS: Programmiersprache		2					1
⋮							
EF: Entwurfsfehler					1		1
CK: Code-Kommentare	2			1			
SO: Sonstiges			1	1			1
Summe der Fehler	3	11	1	2	2	0	3
Status: Wiederholungs-Inspektion erforderlich							

Tab. 12.1 Übersicht über das Modul Checker (nach [Fag 76], S. 118)

Nach Schnurer sind beim Entwurf 65% der Fehler vom Typ „M“ (missing/fehlend), 33% vom Typ „W“ (wrong/falsch) und 2% vom Typ „E“ (extra/zuviel,

[5] Aus Effizienzgründen soll die Gruppe nur *ein* Ziel (Fehler finden) verfolgen.

überflüssig). Beim Codieren verschiebt sich der Schwerpunkt der Fehler: nur 33% sind vom Typ „M“, 66% vom Typ „W“ (s. [Sch 88b], S. 319). Bei offenen Fragen sollten entsprechende Spezialistinnen (außerhalb der Sitzung) hinzugezogen werden.

4. <u>Überarbeitung/Korrektur</u>
   Alle im Inspektionsbericht notierten Fehler bzw. Probleme werden von der Entwerferin (bei I1) bzw. von der Codiererin (bei I2) korrigiert bzw. gelöst.

5. <u>Verfolgung/Überprüfung</u>
   Die Moderatorin muß überprüfen, ob alle Fehler und Probleme von der Entwerferin bzw. Codiererin korrigiert bzw. gelöst wurden.

   - Wenn mehr als 5% des Produkts überarbeitet wurden, sollte die Gruppe eine neue, vollständige Inspektion durchführen.
   - Wenn weniger als 5% überarbeitet wurden, kann die Moderatorin selbst die Qualität der Überarbeitung überprüfen oder die Gruppe neu zu einer Inspektion des kompletten Produkts (oder nur der Überarbeitung) versammeln.

6. <u>Auswertung/Nachbereitung</u>

   (a) Module/Programmteile sollten gemäß der Fehlerhäufigkeit bei der Inspektion angeordnet werden. (Falls die Fehleraufdeckungseffektivität aller Inspektionen ähnlich ist, kann damit die relative Restfehleranzahl abgeschätzt werden.) Daraus können folgende Schlüsse gezogen werden:

      i. die Module mit den meisten Fehlern noch einmal inspizieren oder
      ii. die Module mit den meisten Fehlern „härter“ testen.

   (b) Die Fehlersorten(-häufigkeiten) je Modul sollten mit einer mittleren Fehlersorten-Verteilung verglichen werden.
   Bei großen Abweichungen (z. B. 31% statt normalen 18% Schnittstellenfehler) in den fehlerhaftesten Modulen können die Fehlerursachen frühzeitig aufgespürt werden und in allen Modulen ausgemerzt werden, bzw. die Programmiererinnen können gezielt nachgeschult werden.

7. <u>Inspektion als Einschub beim dynamischen Testen</u>
   Dies ist bei sehr fehlerhaftem Code sinnvoll. Code-Auswahlkriterien für diese Re-Inspektion sind:

   (a) Module mit höchster Zahl von Testfehlern (pro 1000 Code-Zeilen)
   (b) Module mit geringer Testüberdeckung (z. B. TWM_1)[6] aber folgenden Zusatzeigenschaften:

[6] siehe Definition 7.2.5 auf S. 198

i. viele Inspektionsfehler (beim ersten Mal) pro 1000 Code-Zeilen bzw.
ii. kritische Module laut Programmiererinnen-Einschätzung.

Diese Re-Inspektionen sollten wie I2-Inspektionen ablaufen, allerdings ist ein Überblick (Schritt 1) erneut notwendig, falls der erste zu lange zurückliegt.

12.1.2 Vorgehen beim Walkthrough

Ein manuelles Durchgehen (**Walkthrough**) unterscheidet sich von einer Inspektion in Schritt 3 dadurch, daß ein Gruppenmitglied einfache Testdaten vorgibt und die Gruppe anleitet, das Programmstück mit diesen Testdaten per Hand auszuführen (zu simulieren). Dabei werden Zwischenergebnisse schriftlich festgehalten. Ziel des Walkthroughs ist allerdings nicht die komplette Simulation des Programmteils, sondern das Anregen von Nachfragen an die Entwicklerin bzgl. ihrer (Entwurfs- oder Codierungs-)Entscheidungen und eine Diskussion darüber mit dem Ziel, Fehler aufzudecken.

12.1.3 Voraussetzungen, Vor- und Nachteile der informellen Analyse

Voraussetzung für erfolgreiches Überprüfen ist eine nachdrückliche Disziplin und eine klar definierte Folge von Überprüfungsoperationen. Außerdem sind die Teilnehmerinnen in effektiver Fehlersuche zu schulen. Insbesondere sind die häufig vorkommenden, mit hohen Folgekosten verbundenen Fehler zu suchen. Die „Lehrerin“ sollte die Anhaltspunkte vermitteln, welche das Vorkommen einer Fehlersorte (s. Beispiel 12.1.4) verraten. Günstig ist eine vorbereitende Inspektion eines Programmstücks, welches repräsentativ für das zu inspizierende Programm ist. Die dabei gefundenen Fehler sollten analysiert und klassifiziert werden nach Herkunft, Ursache und Anhaltspunkten. Diese Information ist Grundlage der **Testspezifikation**[7], die in den folgenden Inspektionen zu verwenden (und zu verbessern) ist. (Näheres zum Training siehe in [Rus 91], S. 29, und [AAE 82].)

Nach Fagan und Hausen sollten die Inspektionsgeschwindigkeiten aus Tabelle 12.2 beim Inspizieren von Programmen eingehalten werden (siehe [Fag 76], [Hau 83], [Fag 86]).

Die Inspektionsgeschwindigkeiten aus Tabelle 12.2 sollten nicht als „Akkord-Vorgaben“ verwendet werden, wohl aber bei der Planung der Testphase, um genügend Zeit für Inspektionen zu haben[8].

[7] Angaben darüber, worauf zu achten ist; siehe Checklisten bei Schritt 2 oben

[8] Normalerweise gibt es immer Zeitdruck am Ende eines Projektes, so daß die Gefahr besteht, daß die Inspektionen wegfallen bzw. nur oberflächlich durchgeführt werden.

Inspektionsschritt	Inspektionsgeschwindigkeit (in Code-Zeilen pro Stunde [bei I2] bzw. zu produzierenden Code-Zeilen pro Stunde [bei I1])	
	Entwurf: I1	Code: I2
1. Überblick	500	nicht nötig
2. Vorbereitung	100	125
3. Inspektion	100–200[9]	90–150[10]
4. Überarbeitung	50	62

Tab. 12.2 Inspektionsgeschwindigkeiten

Nach Hausen geht man von 70 bis 100 Fehlern in 1000 (zu produzierenden) Code-Zeilen aus (s. [Hau 83]). Also könnten bei der I1-Inspektion (bei 130 Zeilen pro Stunde) 9 bis 13 Fehler pro Stunde in der Inspektionssitzung gefunden werden, oder jedenfalls ein bestimmter Teil davon[11]. Solche Werte werden demnach vorgegeben, um ein positives Ziel zu haben.

Einer der größten Vorteile der Inspektionen ist die relativ schnelle Rückkopplung (feedback) der Ergebnisse an Entwerferinnen und Programmiererinnen, die dabei lernen,

- welche Fehler(-sorten) sie am häufigsten machen,
- wie viele Fehler sie machen,
- wie diese Fehler gefunden werden können.

Meistens verbessert eine Entwicklerin schon im selben Projekt ihre diesbezüglichen Fähigkeiten. Die Rückkopplung sollte aber ausschließlich zum Vorteil der Entwicklerin verwendet werden. Unter keinen Umständen sollte das Management über einzelne Entwicklerinnen Leistungsmessungsdaten aus Inspektionen erhalten[12].

Konsequente **Reviews**, d. h. Inspektionen und Walkthroughs, haben den Vorteil, daß an mehreren Stellen Fehler gefunden werden:

1. von der Programmiererin vor der Übergabe des Materials für die Review-Sitzung (durch sorgfältiges Überprüfen des Materials und durch den Lerneffekt aus früheren Review-Sitzungen) und bei der eigenen Vorbereitung auf die Review-Sitzung,
2. während der Review-Sitzung (durch Aufdecken nichterfüllter Annahmen und durch synergetische Effekte),

[9] Nach [Fag 76] liegt der Wert bei 130 Codezeilen, nach [Sch 88b] bei 100 und nach [Hau 83] zwischen 100 und 200 Codezeilen — bei I0 zwischen 220 und 300.

[10] Nach [Sch 88b] liegt der Wert bei 90 Codezeilen, nach [Hau 83] und [Fag 86] nur zwischen 90 und 125, nach [Fag 76] bei 150 Codezeilen.

[11] Nach Hausen bei I1 18% (vgl. Fußnote 18 auf S. 310).

[12] Man soll die Gans nicht schlachten, die goldene Eier legt.

3. Entwurfsfehler beim Code-Review (durch neue, genauere Informationen, die sich als widersprüchlich erweisen).

Die ökonomischen und qualitativen Vorteile von Inspektionen zeigten sich bei einem Experiment (siehe [Fag 76]). Es ergaben sich folgende Netto-Einsparungen[13] pro 1000 Quell-Anweisungen (ohne Kommentar):

- Inspektion der Logik (I1): +94 Programmierstunden
- Inspektion des Codes (I2): +51 Programmierstunden
- Inspektion des Modultests (I3): –20 Programmierstunden

Durch I1 und I2 wird also bei Programmierung und Test fast ein Personenmonat (von 2,4 bis 4 Personenmonaten[14]) eingespart. Dagegen erhöht I3 die Kosten insgesamt und sollte daher weggelassen werden (daher fehlt I3 in Abbildung 12.1).

Als Qualitätsvorteil der Inspektionen I1 und I2 ergab sich, daß die erstellten Programme 38% weniger Fehler[15] enthielten als vergleichbare Programme, die nur mit einem **schwachen Walkthrough**[16] überprüft worden waren.
Die relative Effektivität der Inspektionen I1 und I2 war sehr hoch (bei einem Anwendungs-Programm mit ca. 4400 Zeilen ohne Kommentar): von den 46 gefundenen Fehlern wurden

- 38, d. h. 83%, durch I1 und I2-Inspektionen gefunden[17],
- 8, d. h. 17%, durch Modultest und Vorbereitung zum Akzeptanztest gefunden,
- 0 durch Akzeptanztest gefunden[18].

Untersuchungen von Russell ergaben, daß eine bei der Inspektion aufgewendete Personenstunde 33 Stunden Arbeit bei späteren Fehlerkorrekturen erspart. Im Vergleich zum Modultest (durch Entwicklerinnen) ist die Fehlerfindungs-Effizienz ($\hat{=}$ gefundene Fehler pro Personenstunde) etwa um den Faktor 4 größer, im Vergleich zum Systemtest um den Faktor 2. Bei komplexen Testumgebungen kann Inspektion sogar

[13] Einsparungen bei der Codierzeit abzüglich erhöhtem Aufwand für Inspektions- und Überarbeitungszeiten

[14] da pro Jahr ca. 3.000 bis 5.000 Zeilen von einer Person geschrieben werden können

[15] Bis zu einem Zeitpunkt „7 (Test-)Monate nach Abschluß des Modul-Tests“ gemessen. Bei IBM ergab sich seit 1976 sogar eine Fehlerreduktion um ca. 66% ($\frac{2}{3}$); siehe [Fag 86].

[16] ohne genaue Rollen für Teilnehmerinnen, insbesondere ohne Moderation; ohne Checklisten, Fehlerlisten; ohne Überprüfung der Korrekturen und Fehlerauswertung

[17] bei einem Programm mit ca. 6.000 Zeilen sogar 93% aller überhaupt gefundenen Fehler, generell 60–90% aller Fehler (nach [Fag 86])

[18] Nach Hausen werden 12%, 18% und 25 bis 30% aller Fehler bei der Inspektion von Grob- und Feinentwurf und Programmen (I0-, I1-, I2-Inspektionen) gefunden und die restlichen 40 bis 45% erst durch Testen (siehe [Hau 83]).

bis zum Faktor 20 effizienter als das Testen sein (s. [Rus 91]). Für die Erkennung bestimmter Fehler, die nicht durch Ausführen des Programms festgestellt werden können (z. B. Abweichung von Programmier-Standards), ist die Inspektion natürlich unvergleichbar effektiver.

Aufgrund von Fallstudien in der Fa. GTE hat Daly die in Tabelle 12.3 dargestellten Werte für die aufgewendete Zeit, die gefundenen Fehler und die aufgewendeten Kosten pro Modul ermittelt (s. [How 79]). Damit sind die Vorteile des manuellen

Methode	Zeit (%)	gefundene Fehler (%)	Kosten pro Modul ($)
Entwurfs-Review	17	45	185 bzw. 410 [19]
Code-Lesen	8	45	160
Testen im Simulationslabor	75	10	1150

Tab. 12.3 Vergleich verschiedener Überprüfungsverfahren

Überprüfens gegenüber anderen Methoden deutlich aufgezeigt.

Eine Studie von Shooman (zitiert in [How 79]) bestätigt, daß Code-Lesen sehr viel kostengünstiger als computergestütztes Fehlerfinden ist:

- manuelle Fehlerfindung: 11 $ pro Fehler
- computergestützte Fehlerfindung: 252 $ pro Fehler

(bei einem Stundenlohn von 18 $ und Kosten von 1000 $ pro CPU-Stunde).

Ein Problem ergibt sich oft bei den Inspektionen oder Walkthroughs:

- Programmiererinnen fühlen sich meist in einer Programmiersprache sicherer als in einer Spezifikations- oder Entwurfssprache. Daher finden sie frühzeitiges Codieren meist sinnvoller als nochmaliges Überprüfen des Entwurfs bzw. Testen sinnvoller als Inspizieren des Codes.
  (Gegenargument: Frühzeitige Fehlersuche ist wichtig, s. Kapitel 2.5.)
- Managerinnen sind von den Techniken nicht immer überzeugt: Es kostet Zeit und der Erfolg (in Form von erhöhter Qualität der Software und niedrigeren Test- und Wartungskosten) ist nicht sofort sichtbar.
  (Gegenargument: siehe langfristige Kosten- und Qualitätsvorteile)

Mit letzterem Problem hatten Entwicklerinnen und Entwickler bei Sperry Univac zu tun (siehe [Har 82]). Dort wurden deshalb folgende Abwandlungen der Review-Methode vorgenommen, was die in Klammern angegebenen Nachteile hatte.

[19] pro Unterprogramm/Modul bzw. Modul/Segment

1. Entwurfs- und Code-Review wurden gleichzeitig durchgeführt.
   (Der Entwurf wurde nicht mehr genau geprüft: „Da er schon codiert ist, muß er o.k. sein". Es wurden keine Entwurfsalternativen mehr überlegt; das hätte zuviel Umstellungsaufwand erfordert, da schon codiert war. Es wurde nur der Code für sich geprüft, nicht die Übereinstimmung des Codes mit dem Entwurf.)

2. Ausnahmen (kein Review) wurden für einige Programmteile erlaubt.
   (Die Wartung von ungeprüften Programmteilen war später viel schwieriger. Die Autor[inn]en von geprüften Programmteilen fühlten sich ausgesondert: „Warum werden gerade meine Programme geprüft?". Dadurch ergab sich eine Cliquenbildung: Autor[inn]en mit geprüften Teilen $\leftrightarrow$ Autor[inn]en mit ungeprüften Teilen.)

3. Es wurde keine Fehler- und Aktionsliste (Liste der nötigen Korrekturen und Änderungen für Programmteile) geführt.

4. Die (Entwurfs- und Programm-)Dokumentation wurde nur in einem Exemplar per Umlauf an alle „Reviewer/innen" verschickt, die ihre Bemerkungen auf dem einen Exemplar bzw. auf einem zusätzlichen Blatt notierten.

5. Es gab keine (Review-)Sitzungen, sondern jede beteiligte Person prüfte allein, individuell das Programmteil.
   (Es wurden keine Fragen an Autor[inn]en gestellt, höchstens auf dem Umlauf-Dokument; aber auch das blieb später — mangels Rückantwort — aus. Unbequeme Probleme wurden nicht angesprochen. Kleinere extra eingestreute Fehler wurden daher nicht gefunden.)

Fazit: Die Abwandlung (Verkürzung) der Methode ist offenbar nicht vorteilhaft in Bezug auf die Qualität. Es ergab sich (nach [Har 82]), daß in den ungeprüften Programmteilen (10% des Systems) 75% der später entdeckten Fehler enthalten waren, dagegen in den geprüften Programmteilen (90% des Systems) nur 25% der Fehler[20].

Bei der vorgestellten manuellen Inspektion in Gruppen können neben den Täuschungen (s. Kapitel 2.4) eine Reihe weiterer Probleme auftreten:

- Konzentration auf Funktionsfehler und Vernachlässigen anderer Qualitätsmerkmale wie z. B. Wartbarkeit, Portabilität, Wiederverwendbarkeit.

- Kein systematisches Suchen nach allen Fehlerarten, evtl. Verzetteln beim Diskutieren von Konventionen für Kommentare oder von anderen Trivialitäten.

- Dominanz in Gruppen, d. h. eine starke, gut vorbereitete Teilnehmerin verhindert die nützlichen Beiträge anderer Teilnehmerinnen, und schlecht vorbereitete Personen können sich unbemerkt zurückhalten.

[20] Einige komplizierte Schnittstellen-Routinen wurden vorsichtshalber (nach Änderungen) immer wieder geprüft, so daß es schon hieß: „déja re-vu" (statt „déja vu" = schon mal gesehen).

Alle drei Probleme können durch **Einzelinspektor-Phasen** gelöst werden. Dabei soll eine Person das vorliegende Dokument anhand einer Checkliste auf einen (oder wenige) Aspekt(e) hin überprüfen. Die folgenden Aspekte bieten sich — in dieser Reihenfolge — als Untersuchungsziele für entsprechende Phasen an.

1. Syntaktische Aspekte der Dokumentation (Grammatik, Rechtschreibung, Formatierung)
2. Layout des Quellcodes
3. Code-Lesbarkeit (Variablennamen, Abkürzungen, Namensstandards)
4. Programmierstil (keine unnötigen *goto*'s, keine Zuweisungen in Booleschen Ausdrücken, etc.)
5. Korrektheit von Programmkonstrukten (z. B. Inkrementierung von Schleifenvariablen)

Als Phase 6 kann sich daran die übliche Gruppensitzung anschließen, in der die funktionale Korrektheit das Untersuchungsziel ist.

12.2 Formale Analyse

12.2.1 Fehleranalyse

Folgende Analysearten kommen bei einer formalen Vorgehensweise in Betracht:

1. Typ-, Variablen-, Prozedur- und Datei-Analyse (Fehleranalyse im engeren Sinne)

   (a) Analyse der Typen (d. h. Wertebereich und erlaubte Operationen).
   Dies erledigt i. allg. ein guter Compiler (jedoch nicht bei Zuweisungen in der „Steinzeit"-Sprache C).
   Um fehlerhafte Verwendungen noch genauer feststellen zu können, empfehlen sich folgende zusätzliche Typangaben:

   - *Index* für Arrayindizes,
   - *Zähler* für Zählvariablen in Schleifen,
   - *Einheiten*, z. B. Längeneinheit „cm".
     (Damit läßt sich z. B. folgender Fehler bei einer Zuweisung a := w/z aufdecken, wenn gilt:
     - Einheit(a) = cm/sec
     - Einheit(w) = cm
     - Einheit(z) = g = Gramm [statt sec].)

(b) Analyse von Lebens- und Gültigkeitsbereichen von Variablen
(Dies erledigt i. allg. der Compiler. Beispielsweise kann bei Zeigervariablen ein referenziertes Objekt eine kürzere Lebenszeit als der Zeiger haben, wenn man Speicherplatz explizit [mit *dispose* oder *free*] freigeben darf.)

(c) Analyse von Prozeduren
Dabei gibt es die folgenden Fälle:

i. Prozeduren als Parameter
(Zu prüfen ist, ob die formalen und aktuellen Parameter der Prozedur selbst wieder typgleich sind.)

ii. Funktionsprozeduren
(Zu prüfen ist, ob sie einen Seiteneffekt haben.)

iii. Schnittstellen (bei getrennter Übersetzung)
(Dies erledigen Compiler bzw. Binder, allerdings oft nur zum Teil.)

(d) Analyse von Zugriffen auf Dateien
Operationen auf Dateien dürfen nur in einer bestimmten Reihenfolge ausgeführt werden, z. B. „lesen" erst nach „öffnen".
Das Zustandsdiagramm von Abb. 12.2 beschreibt z. B. eine mögliche vorzugebende Reihenfolgespezifikation; dies ist anhand der Kontrollflußwege in den zugreifenden Programmen statisch zu überprüfen.

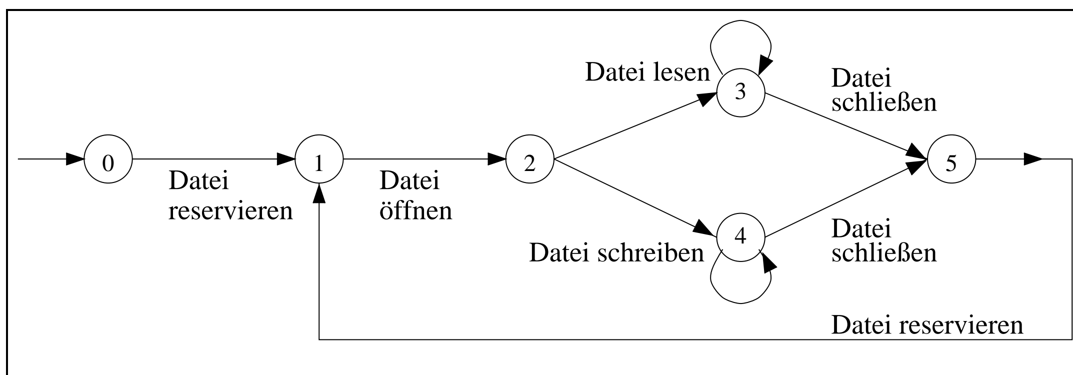

Abb. 12.2: Zustandsdiagramm der erlaubten Operationen auf Dateien

2. Analyse der Programmstruktur
Dabei werden folgende Eigenschaften bzw. Fragen untersucht:

(a) Ist das Programm strukturiert (d. h. ohne *goto*'s)?
(Falls nein, erfolgt ggf. eine automatische Restrukturierung.)

(b) Hat das Programm unerreichbare Anweisungen (bei Verwendung von *goto*'s)?

(c) Hat das Programm nichtausführbare Wege?

(d) Welche Schleifen terminieren in jedem Fall?[21]

[21] Wegen der Unentscheidbarkeit des „Halteproblems" ist dies natürlich nur z.T. (in einfachen Fällen) entscheidbar.

3. Analyse, ob Programmierrichtlinien eingehalten werden
   (a) Formale Richtlinien für Kommentare
   (b) Layout-Richtlinien
   (c) Einhaltung von Komplexitätsgrenzen[22]
   (z. B. Codelänge pro Modul, Schachtelung von Schleifen)
   (d) Namenskonventionen und verbotene Konstrukte
   (z. B. Verbot von Lese-/Schreib-Operationen ohne „sichere“ Speicherverwaltung)

Als Nebenprodukt obiger Analysen kann folgende Dokumentation erzeugt werden:

- Kontrollfluß des Programms als Graph (Kontrollflußgraph[23])
- Komplexitätsmaße[24] (z. B. Angaben über die Tiefe der Schleifenschachtelung)
- Graph der Prozedur- bzw. Funktionsaufrufe (**Operationsaufrufgraph**[25])
  (Dieser Graph enthält alle Operation(snam)en als Knoten und eine Kante von P nach Q g. d. w. Operation P die Operation Q aufruft.)
- Hinweise auf (un-)problematische Schleifenterminierungen:
  i. *for*-Schleifen mit festen Grenzen (unproblematisch),
  ii. datenabhängige Schleifengrenzen (dafür muß die Terminierung dann „per Hand“ bewiesen werden).

4. Ausdrucksanalyse
   Hierbei sind folgende Fragen zu analysieren:
   (a) Werden Grenzen von Arrays überschritten?
   (b) Werden Real-Zahlen auf (Un)-Gleichheit abgefragt, z. B.
   „**if** r = s **then** ... “?
   (Dies ist zu korrigieren. Eine richtige [d. h. zuverlässige] Abfrage lautet:
   „**if** absolutbetrag(r-s) $< \epsilon$ **then** ... “.)
   (c) Wie sieht der Datenfluß im Programm aus?
   Dabei wird pro Variable untersucht, wann sie
   - einen neuen Wert erhält („*def*“),
   - referenziert bzw. benutzt wird („*ref*“),
   - ihren Wert verliert („*undef*“).

[22] genaueres dazu siehe Kap. 16.1
[23] Dieser Kontrollflußgraph wird für die Kriterien von Kap. 7 benötigt.
[24] Genaueres zu Definition und Verwendung der Maße siehe in Kap. 16.1 und 16.2.
[25] Prozeduren und Funktionen sind **Operationen**.

Zusammen mit den Informationen aus dem Kontrollflußgraphen erhält man damit den Datenflußgraphen, der für die Kriterien aus Kap. 8 benötigt wird. Eine spezielle Analyse ermittelt stattdessen die Datenflußanomalien.

12.2.2 Datenflußanalyse

Ziel der Datenflußanalyse ist die Aufdeckung von **Datenflußanomalien** bei der Benutzung einer Variablen:

- eine Referenz („*ref*") vor der ersten Definition („*def*"),
- zwei aufeinanderfolgende Definitionen ohne zwischenzeitliche Referenz,
- eine Definition gefolgt von einer Freigabe („*undef*") ohne zwischenzeitliche Referenz.

Vorgehensweise bei der Datenflußanalyse:

1. Die Wege („Pfade") durch den Kontrollflußgraphen werden durchlaufen.
2. Für jede Variable wird dabei notiert, ob sie definiert oder referenziert oder undefiniert wird, wobei die Abkürzungen d, r und u verwendet werden. (Achtung: Bei Zuweisungen wird die rechte vor der linken Seite betrachtet.) Damit erhält man pro Variable und Pfad einen Pfadausdruck über dem Alphabet $\{d, r, u\}$, der nur Sequenzen beschreibt (vgl. Definition 5.1.1 auf S. 114).

   Beispiel 12.2.1

   (a) *Anweisung: A := A + B; Pfadausdruck: rd (für A), r (für B)*

   (b) *Anweisungsfolge:*
   A := B + C; B := A + D; A := A + 1; B := A + 2;
   Pfadausdruck: drrdr für A, rdd für B

3. Eine **Anomalie** liegt bei folgenden Pfadausdrücken vor:

   (a) $\alpha ur\beta$ (**ur-Anomalie**)

   (b) $\alpha dd\beta$ (**dd-Anomalie**)

   (c) $\alpha du\beta$ (**du-Anomalie**)

   wobei α, β beliebige[26] Folgen der Symbole r, d, u sind.

[26] Auch die leere Folge ist erlaubt; formal gilt also, daß α und β aus $\{r, d, u\}^*$ sind.

Folgende Programmkonstrukte sind bei der Datenflußanalyse schwierig zu behandeln:

1. Aufrufe von Operationen

   (a) Wird der Kontrollflußgraph für die Operation an der Aufrufstelle eingesetzt, so führt das zu einer „Aufblähung“ der Kontrollstruktur, insbesondere bei geschachtelten Aufrufen. Bei rekursiven Operationen ergibt sich meistens ein unendlich großer Gesamtgraph.

   (b) Wenn der Operationsaufrufgraph (siehe Abschnitt 12.2.1) bekannt ist, kann das Problem vereinfacht werden, allerdings werden nicht mehr alle Datenflußanomalien erkannt (genaueres siehe [FoO 76]).

2. Dynamisch wachsende Strukturen
   Für Datenstrukturen, die zur Laufzeit wachsen können (wie Listen, Bäume, Graphen), läßt sich der Zugriff auf einzelne Elemente nicht statisch analysieren. Daher werden solche Strukturen als „monolithische“ Einheit — wie eine Variable — behandelt. Damit lassen sich aber nicht alle Datenflußanomalien korrekt modellieren.

3. Arrays
   Bei dynamischen Arrays gibt es die gleichen Probleme wie im Fall 2. Aber selbst bei Arrays mit fester Größe, also etwa Array A mit 20 Elementen, gibt es Analyseprobleme. Der Ansatz, die 20 Elemente wie getrennte Variablen zu behandeln, funktioniert nur bei Zugriffen mit festem Index, etwa bei $B := A[1] + 1$. Bei einem Zugriff $B := A[K] + 1$ mit einer Variablen K, deren Wert zur Laufzeit eingelesen wird, ist unklar, welches Element gemeint ist. Daher ist ein Array in jedem Fall als monolithische Einheit (wie bei 2) zu behandeln — mit denselben Einschränkungen bei der Erkennung von Datenflußanomalien.

4. Schleifen
   Beim Vorkommen von Schleifen kann es sehr viele — evtl. unendlich viele — vollständige Wege geben, die durchlaufen werden. Es ist zu klären, wieviel Schleifendurchläufe für die Datenflußanalyse ausreichend sind. Dieses Problem wird beim LIVE-/AVAIL-Algorithmus und bei der algebraischen Methode von Forman richtig gelöst (s. unten), nicht jedoch beim Algorithmus von Howden (s. [How 78c]).

Im folgenden werden die **LIVE-/AVAIL-Algorithmen** vorgestellt, die auf Algorithmen aufbauen, die für die Programmoptimierung entwickelt wurden. Diese Algorithmen operieren mit den Begriffen *generate*, *kill* und *null*. Für die Anwendung bei der Datenflußanalyse muß dann eine Zuordnung zu den Begriffen *def*, *ref* und *undef* vorgenommen werden.

Voraussetzung:
Zu jedem Knoten l des Kontrollflußgraphen sind folgende Mengen definiert, die disjunkt sind: $generate(l)$, $kill(l)$, $null(l)$. Dabei gilt $generate(l) \cup kill(l) \cup null(l) = tok$, wobei tok eine endliche Menge von „Token" ist. (In der Anwendung für die Datenflußanalyse sind dies die Variablen.)

Notation beim Graph-Durchlauf
Für ein Token A wird beim Durchlauf durch einen Knoten l notiert:

Symbol g, falls $A \in generate(l)$,
Symbol k, falls $A \in kill(l)$,
Symbol n, falls $A \in null(l)$.

P(A; w) beschreibt den entsprechenden Pfadausdruck beim Durchlauf des Weges w für Token A. Bei einem **reduzierten Pfadausdruck** läßt man die Symbole n weg. Betrachtet man *alle* Wege des Kontrollflußgraphen, die von einem Knoten k ausgehen bzw. zu k hinführen, lassen sich die möglichen alternativen und iterierten „Aktionen" g, k und n durch einen sequentiellen Pfadausdruck (im Sinne von Definition 5.1.1 bzw. als regulärer Ausdruck mit $\cdot$, $+$, $*$, s. S. 322) beschreiben, der ebenfalls reduziert werden kann durch Weglassen des Symbols n.

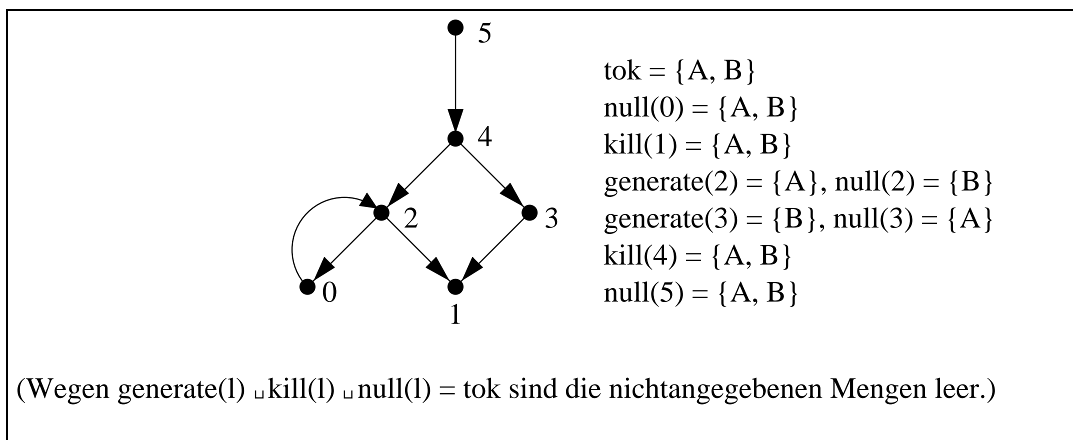

Abb. 12.3: Kontrollflußgraph mit Mengen *generate*, *kill*, *null*

Beispiel 12.2.2 (Pfadausdrücke zum Kontrollflussgraph aus Abb. 12.3 und Notationen $P_r(A; l \rightarrow)$ und $P_r(A; \rightarrow l)$)
Pfadausdruck: $P(A; 5, 4, 2, 0, 2, 1) = nkgngk$,
reduzierter Pfadausdruck: $P_r(A; 5, 4, 2, 0, 2, 1) = kggk$,
Pfadausdruck $P(A; 5 \rightarrow) = k(g(ng)^* + n)k$ *beschreibt alle Aktionen für* A*, ausgehend von Knoten 5,*
Pfadausdruck $P_r(A; 5 \rightarrow) = k(gg^* + \epsilon)k = kgg^*k + kk$ *ist der entsprechende reduzierte Pfadausdruck,*

Pfadausdrücke $P(A; \to 3) = P(A; 5, 4) = nk$ und $P_r(A; \to 3) = P_r(A; 5, 4) = k$ beschreiben alle Aktionen (außer null bei P_r) für A auf Wegen, die zu Knoten 3 hinführen.

Lebendigkeit und Verfügbarkeit von Token bzw. Variablen

Es gilt $A \in live(l) \Leftrightarrow$ einer der Wege, der von Knoten l ausgeht, enthält als erste „Aktion" (abgesehen von *null*) das Symbol g (für *generate*). (Die „Lebendigkeit" von A in l bedeutet, daß A noch „in der Zukunft" [auf wenigstens einem Weg] generiert werden kann.)

Entsprechend gilt $A \in avail(l) \Leftrightarrow$ alle Wege, die zu l hinführen, enthalten als letztes Symbol (abgesehen von *null*) das Symbol g. (Die „generierte" Variable ist also in l stets verfügbar, egal wie der Kontrollfluß zu l hinführt.)

Mit Pfadausdrücken lassen sich die Eigenschaften *live* und *avail* folgendermaßen beschreiben (mit Hilfe der Notationen aus Beispiel 12.2.2):

DEFINITION 12.2.1 (LIVE, AVAIL)

1. $A \in live(l) :\Leftrightarrow P_r(A; l \to) = g \cdot p + p'$
   wobei p und p' Pfadausdrücke über g und k sind.

2. $A \in avail(l) :\Leftrightarrow P_r(A; \to l) = p \cdot g$
   wobei p ein Pfadausdruck über g und k ist.

Berechnung von $live(l)$ und $avail(l)$ für alle Knoten l

Zur Berechnung benutzt man die folgenden iterativen Algorithmen. In jedem Schritt werden dabei für alle Knoten j neue Werte für $live(j)$ bzw. $avail(j)$ berechnet. Das Verfahren wird so lange fortgesetzt, bis sich die neuen Werte nicht mehr von den alten Werten unterscheiden (für alle Knoten), d. h. bis $change = false$ gilt. Damit ist der „kleinste Fixpunkt" als korrekte Lösung berechnet[27]. Beim LIVE-Algorithmus beginnt man mit leeren Mengen ($live(j) := \emptyset$ für alle $j = 0$ bis $|N| - 1$, wobei N die Menge der Knoten ist), beim AVAIL-Algorithmus beginnt man mit der Menge, die alle Token umfaßt ($avail(j) := tok$ für alle $j = 1$ bis $|N| - 1$), nur für den Startknoten 0 beginnt man mit der leeren Menge ($avail(0) = \emptyset$), da zum Startknoten keine Wege hinführen (vgl. Def. 12.2.1, Teil 2, und die Erläuterung davor).

Man kann sich nun überlegen, daß die mit (*) bezeichnete Zuweisung im LIVE- bzw. AVAIL-Algorithmus die korrekte iterative Umsetzung der Definition 12.2.1 ist[28].

[27] Dies ist nur die Idee, der Korrektheitsbeweis findet sich bei Hecht und Kildall (s. [HeU 75], [Kil 73]).

[28] Dabei bezeichnen $S(j)$ und $P(j)$ die Menge der Nachfolgerknoten (successor) bzw. Vorgängerknoten (predecessor) eines Knotens j.

Die Algorithmen lauten somit komplett:

Algorithmus LIVE:

```
for j := 0 to |N| do live(j) := ∅;
change := true;
while change do
    begin
        change := false;
        for j := 0 to |N| do
            begin
                previous := live(j);
                (*) live(j) := ⋃_{l∈S(j)}[(live(l) \ kill(l)) ∪ generate(l)];
                if live(j) ≠ previous then
                    change := true;
            end;
    end;
```

Algorithmus AVAIL:

```
avail(0) := ∅;
for j := 1 to |N| do avail(j) := tok;
change := true;
while change do
    begin
        change := false;
        for j := 1 to |N| do
            begin
                previous := avail(j);
                (*) avail(j) := ⋂_{l∈P(j)}[(avail(l) \ kill(l)) ∪ generate(l)];
                if avail(j) ≠ previous then
                    change := true;
            end;
    end;
```

Beispiel 12.2.3 (LIVE-Algorithmus)
Für den Kontrollflußgraphen aus Abbildung 12.3 ergeben sich die in Tabelle 12.4 dargestellten live-Mengen vor der k-ten Ausführung von Schritt (*) im LIVE-Algorithmus (beim ersten Durchlauf der while-Schleife).

In Tabelle 12.4 gilt beispielsweise für $k = 6, j = 4$:
$live(4) = \{A, B\}$, *da* $3 \in S(4), B \in generate(3) = \{B\}$ *und* $2 \in S(4)$,
$A \in live(2) \setminus kill(2) = \{A\} \setminus \emptyset = \{A\}$

Knoten j	Schritt k						
	1	2	3	4	5	6	7
0	∅	A	A	A	A	A	A
1	∅	∅	∅	∅	∅	∅	∅
2	∅	∅	∅	A	A	A	A
3	∅	∅	∅	∅	∅	∅	∅
4	∅	∅	∅	∅	∅	A, B	A, B
5	∅	∅	∅	∅	∅	∅	∅

Tab. 12.4 Berechnungsschritte beim LIVE-Algorithmus

Bei $k = 6$ sind die endgültigen live-Mengen ermittelt, aber dies wird erst bei erneutem Durchlaufen der while-Schleife mit sechs Ausführungen von Schritt (*) des Algorithmus festgestellt (durch change = false).

Im folgenden soll nun mit Hilfe des LIVE- und AVAIL-Algorithmus das eigentliche Problem (Erkennung der Datenflußanomalien) gelöst werden.

Die nichtinitialisierte Referenz (*ur*-Anomalie) ist ein echter Fehler, daher wird hier die Lösung (mit dem AVAIL-Algorithmus) angegeben.

Satz 12.2.1
Sei generate(l) = def(l) und kill(l) = undef(l) für alle Knoten l des Kontrollflußgraphen gesetzt[29].
Dann gilt für jeden Knoten l und jede Variable A:

> $A \notin$ avail(l) *und* $A \in$ *ref(l) g. d. w. ein Weg im Kontrollflußgraphen mit einem Pfadausdruck* $\alpha ur\beta$ *für A existiert, wobei gilt:* $\alpha, \beta \in \{r, d, u\}^*$ *und die Referenz r von A erfolgt im Knoten l. (Es liegt also eine ur-Anomalie vor.)*

Beweis: $A \notin avail(l) \Leftrightarrow$ es gibt einen Weg, der zu l hinführt und als letzte Aktion für A (außer *null*) *kill* enthält, d. h. für die „Aktionen" k, g und n (*kill, generate, null*) hat dieser Weg den Pfadausdruck $\alpha k n^*$ mit $\alpha \in \{r, d, u\}^*$; *kill* entspricht „*undef*", *generate* entspricht „*def*", also gilt: *null* entspricht „*ref*" oder „keine Benutzung". Somit liegt für A (in r, d, u) ein Pfadausdruck $\alpha u r^*$ vor, d. h. er endet nach u eventuell mit einer Folge von Symbolen r. Dieser Pfadausdruck gilt bis vor den Knoten l. — $A \notin avail(l)$ und $A \in$ *ref(l)* gilt also genau dann, wenn ein Pfadausdruck $\alpha u r^* r$ bis zum Knoten l inklusive vorliegt, d. h. ein Pfadausdruck $\alpha u r \beta$ mit $\alpha, \beta \in \{r, d, u\}^*$, wobei die Referenz r von A im Knoten l erfolgt.

q. e. d.

[29] Bei geschachtelten Prozeduren oder Blöcken gilt an deren Anfang für den entsprechenden Knoten j: *undef(j) = kill(j) = L*, wobei L die Menge der lokalen Variablen ist. Bei Prozeduren mit Parametern ist für den Anfangsknoten 0 die Menge *def(0)* bzw. *generate(0)* die Menge der Parameter und globalen Variablen mit definierten Werten.

Die *ur*-Anomalien können also mit folgendem Algorithmus gefunden werden.

<u>Algorithmus UR:</u>

```
begin
    for n := 0 to |N| do
        begin
            kill(n) := undef(n);
            generate(n) := def(n);
        end;
    { Ende Initialisierung }
    call AVAIL;
    for n := 0 to |N| do
        if ref(n) \ avail(n) ≠ ∅
        then print(„uninitialized references to variables“
                    , ref(n) \ avail(n)
                    , „at node“, n, „are possible“);
end;
```

Die *dd*- und *du*-Anomalien können mit Hilfe des LIVE-Algorithmus berechnet werden (s. Übung 12.6).

Das aufwendige Durchlaufen des Kontrollflußgraphen (mit einigen Iterationen beim LIVE- und AVAIL-Algorithmus) ist nicht notwendig, wenn ein strukturiertes Programm vorliegt, dessen Konstrukte nur einen Eingang und einen Ausgang haben. In diesem Fall können die Datenflußanomalien schrittweise „von innen nach außen“ berechnet werden.

Die folgende **algebraische Methode** (nach Forman) zur Bestimmung von Datenflußanomalien kann auch auf allgemeine Kontrollflußdiagramme (mit *goto*'s) angewandt werden, wenn der Kontrollfluß als regulärer Ausdruck beschrieben wird[30].

$\cdot$ (Konkatenation) steht für Sequenz im Kontrollfluß,
$+$ (Alternative) steht für eine Verzweigung im Kontrollfluß und
$*$ (Kleene-Stern) steht für Iteration im Kontrollfluß.

Beispiel 12.2.4

Zum Kontrollflußschema G aus Abbildung 12.4 gehöriger regulärer Ausdruck R(G):

$$R(G) = 0 \cdot 1 \cdot (2+3) \cdot 4 \cdot 5 \cdot [6 \cdot 0 \cdot 1 \cdot (2+3) \cdot 4 \cdot 5]^*$$

Folgende Datenflußaussagen werden für jede Variable v und jedes Konstrukt s (in entsprechenden Booleschen Werten bzw. Bits) algebraisch notiert:

[30] Dies entspricht der Bestimmung der erkannten regulären Wortmenge zu einem erkennenden, endlichen Automaten, ist also formal berechenbar (s. [Bra 84], Satz 5.3.7 (von Kleene); [Weg 93], Satz 5.3.3 bzw. [Weg 96], Kap. 5.5).

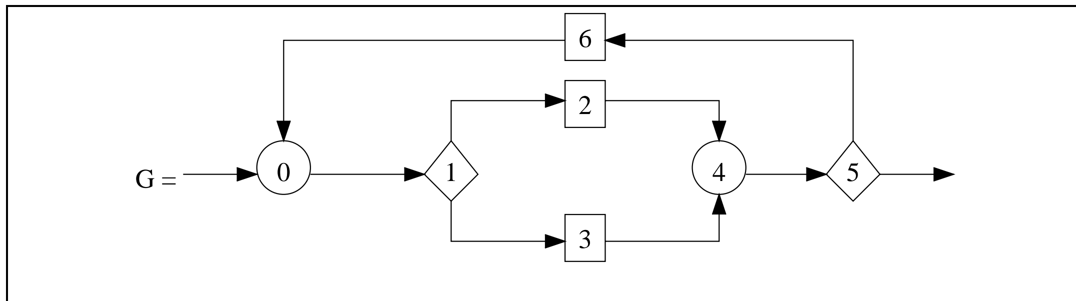

Abb. 12.4: Kontrollflußschema G

an:	es liegt eine Datenflußanomalie im Konstrukt s vor
ee:	es gibt einen Weg durch s ohne Aktion auf Variable v
din, rin, uin:	es gibt einen Weg durch s, so daß die erste Aktion („in") für v ein *d̲ef* bzw. *r̲ef* bzw. *u̲ndef* ist
dout, rout, uout:	es gibt einen Weg durch s, so daß die letzte („out") Aktion für v ein *d̲ef*, *r̲ef* bzw. *u̲ndef* ist

Eine 8-stellige Binärzahl kann diese Angaben pro Variable und Konstrukt repräsentieren. Für die Zuweisung $v := v{+}1$ gilt z. B. für v: 00 010 100, wobei die 1 an vierter Stelle angibt, daß $rin{=}true$ ist (wegen $v+1$ auf der rechten Seite der Zuweisung) und die 1 an drittletzter Stelle gibt an, daß $dout = true$ ist (wegen v auf der linken Seite). Für ein Konstrukt s wird ein Vektor aus solchen achtstelligen Binärzahlen angelegt, wobei jedes Vektorelement eine Variable repräsentiert.

Beim Zusammensetzen von Konstrukten B und C zu einem Konstrukt A kann man die neuen Werte der Bits der Binärzahlen nach folgenden Regeln berechnen. Dabei bezeichnet $s.x$ den Booleschen Wert x für die betrachtete Variable und das Konstrukt s; $\vee, \wedge$ bezeichnen die logischen Operatoren *oder* und *und*.

Konkatenation/Sequenz:
Sei $A = B \cdot C$, dann gilt:

- $A.an = B.an \vee C.an \vee (B.dout \wedge C.din) \vee (B.uout \wedge C.rin) \vee (B.dout \wedge C.uin)$

  $B.dout \wedge C.din$ entspricht der dd-Anomalie,
  $B.uout \wedge C.rin$ entspricht der ur-Anomalie,
  $B.dout \wedge C.uin$ entspricht der du-Anomalie.
  Mit $B.an$ und $C.an$ werden die bekannten Anomalien weitergereicht.

- $A.ee = B.ee \wedge C.ee$

- $A.din = B.din \vee (B.ee \wedge C.din)$

- $A.rin = B.rin \vee (B.ee \wedge C.rin)$

- $A.uin = B.uin \vee (B.ee \wedge C.uin)$
- $A.dout = (B.dout \wedge C.ee) \vee C.dout$
- $A.rout = (B.rout \wedge C.ee) \vee C.rout$
- $A.uout = (B.uout \wedge C.ee) \vee C.uout$

Falls in B die Variable nicht angesprochen wird (d. h. nur $B.ee$ ist *true*), gilt also $A.x = C.x$ für alle $x \in \{an, ee, din, rin, uin, dout, rout, uin\}$. Entsprechend gilt $A.x = B.x$, wenn in C die Variable nicht angesprochen wird (s. Übung 12.8).

Alternative/Verzweigung:
Sei $A = B + C$, dann gilt:
$A.x = B.x \vee C.x$ für alle $x \in \{an, ee, din, rin, uin, dout, rout, uout\}$.

Iteration:
Sei $A = B^*$. Dann gilt nach Definition:

$$B^* = \epsilon + B + B \cdot B + B^3 + B^4 + \cdots = \epsilon + \sum_{i=1}^{\infty} B^i$$

wobei ϵ den Fall „keine Ausführung von B“ repräsentiert.

Satz 12.2.2
Für die Ermittlung der Datenflußanomalien, die in der Schleife B^* oder vor und nach der Schleife B^* entstehen, reicht die Betrachtung von $E+B \cdot B$, wobei $E = 01\,000\,000$, d. h. nur $ee = true$.
Alle Datenflußanomalien werden also durch 0 oder 2 Iterationen erzeugt.

Beweis: siehe [For 84] und [Ste 81], S. 166 f.

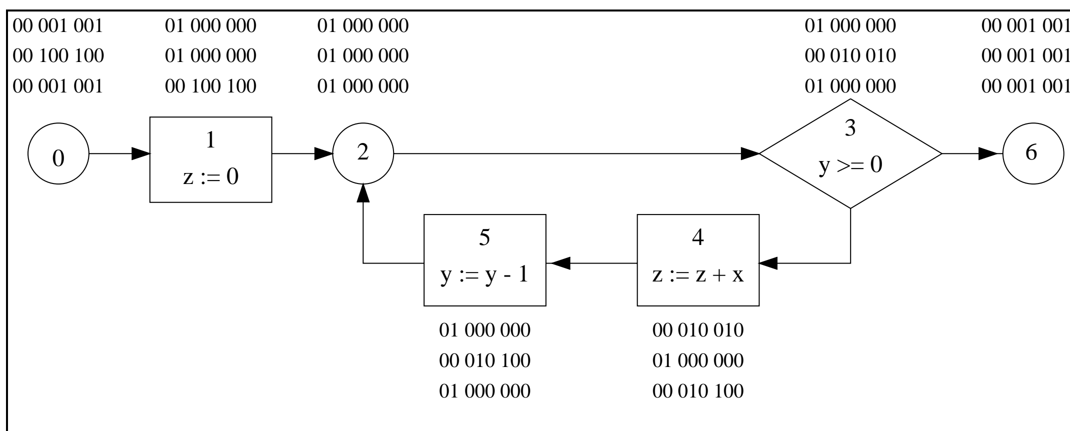

Abb. 12.5: Kontrollflußgraph für die Multiplikation $z = x * y$

Beispiel 12.2.5 (mit Datenflussbinärzahlen pro Knoten)
An den einzelnen Knoten in Abbildung 12.5 stehen untereinander die achtstelligen Binärzahlen für die Variablen x, y und z. Zum Beispiel bedeutet $x \hat{=} 00\ 001\ 001$, $y \hat{=} 00\ 100\ 100$ und $z \hat{=} 00\ 001\ 001$ bei Knoten 0, daß x und z undefiniert sind (uin = uout = 1, sonst alles 0) und daß y definiert ist (din = dout = 1, sonst alles 0), da hier angenommen wird, daß x (und z) fälschlicherweise als lokale Variablen und nicht als Parameter implementiert wurden.

Damit erhält man für $D(4 \cdot 5)$, die Matrix der drei waagerecht angeordneten Bitfolgen, welche die x-, y- und z-Binärzahlen für die Datenflußaussagen der Sequenz von 4 und 5 repräsentieren:

$$D(4 \cdot 5) \hat{=} \begin{pmatrix} 00\ 010\ 010 \\ 00\ 010\ 100 \\ 00\ 010\ 100 \end{pmatrix}$$

Die Komponenten des Ergebnisses $D(4 \cdot 5)$ sind nach obigen Regeln für die Konkatenation zu berechnen. Insbesondere gilt für die Konkatenation „·“ mit allen achtstelligen Binärzahlen b und $E = 01\ 000\ 000$:
$b \cdot E = E \cdot b = b$ (vgl. Übung 12.8).

Entsprechend erhält man — durch Einsetzen von $D(4 \cdot 5)$ — folgendes für die Iteration:

$$D(2 \cdot 3 \cdot (4 \cdot 5 \cdot 2 \cdot 3)^*) = D(2 \cdot 3 \cdot [E + (4 \cdot 5 \cdot 2 \cdot 3)^2]) \hat{=} \begin{pmatrix} 01\ 010\ 010 \\ 00\ 010\ 010 \\ 01\ 010\ 100 \end{pmatrix}$$

$$\text{da } D(2 \cdot 3) \ \hat{=} \ \begin{pmatrix} 01\ 000\ 000 \\ 00\ 010\ 010 \\ 01\ 000\ 000 \end{pmatrix} \ \textit{und}$$

$$D(4 \cdot 5 \cdot 2 \cdot 3) \ \hat{=} \ \begin{pmatrix} 00\ 010\ 010 \\ 00\ 010\ 010 \\ 00\ 010\ 100 \end{pmatrix} \ \hat{=} \ D[(4 \cdot 5 \cdot 2 \cdot 3)^2]$$

Also erhält man für den Ausdruck $B = 1 \cdot 2 \cdot 3 \cdot (4 \cdot 5 \cdot 2 \cdot 3)^*$:

$$D(B) \hat{=} \begin{pmatrix} 01\ 000\ 000 \\ 01\ 000\ 000 \\ 00\ 100\ 100 \end{pmatrix} \cdot \begin{pmatrix} 01\ 010\ 010 \\ 00\ 010\ 010 \\ 01\ 010\ 100 \end{pmatrix} = \begin{pmatrix} 01\ 010\ 010 \\ 00\ 010\ 010 \\ 00\ 100\ 100 \end{pmatrix}$$

Schließlich erhält man für das gesamte Programm, d. h. für den Ausdruck $P = 0 \cdot B \cdot 6$:

$$D(P) = \begin{pmatrix} 00\ 001\ 001 \\ 00\ 100\ 100 \\ 00\ 001\ 001 \end{pmatrix} \cdot \begin{pmatrix} 01\ 010\ 010 \\ 00\ 010\ 010 \\ 00\ 100\ 100 \end{pmatrix} \cdot \begin{pmatrix} 00\ 001\ 001 \\ 00\ 001\ 001 \\ 00\ 001\ 001 \end{pmatrix}$$

$$= \begin{pmatrix} 10\ 001\ 011 \\ 00\ 100\ 010 \\ 00\ 001\ 100 \end{pmatrix} \cdot \begin{pmatrix} 00\ 001\ 001 \\ 00\ 001\ 001 \\ 00\ 001\ 001 \end{pmatrix} = \begin{pmatrix} 10\ 001\ 001 \\ 00\ 100\ 001 \\ 10\ 001\ 001 \end{pmatrix}$$

Bei der Verknüpfung von 0 *und* B *wird die (ur-)Anomalie für x registriert, bei der Verknüpfung von* $0 \cdot B$ *und* 6 *die (du-)Anomalie für z. Damit wird erkannt, daß x kein Eingabeparameter und z kein Ausgabeparameter ist.*

12.2.3 Bewertung der formalen statischen Analyse

Die formale statische Analyse hat folgende Vorteile:

+ Sie ist eine vergleichsweise wenig aufwendige Methode.
+ Sie erfordert keine Änderungen der Arbeitsgewohnheiten der Programmiererinnen, da die Analyse automatisch erstellt wird.
+ Sie erfordert nur minimalen zusätzlichen Aufwand (keine Änderung des Quellcodes; kein Erstellen von Testtreibern, Platzhaltern, Ausgabe-Code).
+ In „einem Lauf“ können mehrere Fehler entdeckt werden.
+ Unausführbare Systeme können behandelt werden (bei fehlenden Unterprozeduren, bei nebenläufigen Programmen mit Verklemmungen [genaueres siehe Kapitel 14.3]). Daher kann und sollte die statische Analyse zu Beginn der Testphase — vor dem dynamischen Testen — eingesetzt werden.

Es gibt natürlich auch Nachteile bei der statischen Analyse:

– Begrenzte Interpretation dynamischer Ereignisse

Beispiel 12.2.6

Programm
```
R := 0;
for I := 1 to 100 do
begin
if I = 1 then A := 5;
R := A * (I - 1) + R;
end;
```

Äquivalentes Programm
```
R := 0;
A := 5;
for I := 1 to 100 do
R := A * (I - 1) + R;
```

Die (begrenzte) statische Analyse meldet für Variable A einen Datenflußfehler vom Typ „undefinierte Referenz", da es im linken Programm einen (nichtausführbaren) Weg an der bedingten Anweisung $A := 5$ vorbei zur Anweisung $R := A * \ldots$ gibt. (Im äquivalenten rechten Programm kommt dieser Weg nicht vor.)

- Ungewißheit der Diagnose
  Die automatische Diagnose muß vom Menschen noch bewertet werden[31]. Beispielsweise kann ein Programm aus Effizienzgründen eine *dd*-Anomalie enthalten, wie etwa der LIVE- und der AVAIL-Algorithmus bezüglich der Variablen *change* (vgl. Übung 12.7).
- Abhängigkeit vom Programmierstil
  Wenn Programmiererinnen nicht strukturiert programmieren, sondern verwickelte Kontrollsequenzen und dubiose Sprachkonstrukte verwenden, wird der Nutzen der formalen statischen Analyse durch eine Fülle von Warnungen und Fehlermeldungen vereitelt. Das Auftreten einer Fülle von Meldungen ist also selbst schon ein Hinweis auf (meist unerwünschte) Programmiertechniken. (Dieser Hinweis ist also ein Vorteil.)

12.3 Symbolische Ausführung

Die symbolische (Programm-)Ausführung wurde bereits in Abschnitt 11.2.1 als Hilfsmittel für die Testdatenermittlung beim „Überdecken" eines Programms gemäß C_0- oder C_1-Kriterium benutzt. Hier wird diese Methode als eigenständige Methode zur Überprüfung von Programmen vorgestellt. In Kapitel 12.4 wird dann die Verwendung bei der formalen (Programm-)Verifikation erläutert.
Die symbolische Ausführung eines Programms ist eine Erweiterung des Begriffs der Ausführung eines Programms mit konkreten Werten: Es werden symbolische Werte für die Eingabevariablen verwendet, mit denen dann „entsprechend" zu rechnen ist. Was bedeutet das für einzelne Konstrukte?

1. Symbolische Ausrechnung von Ausdrücken/Zuweisungen:

   Beispiel 12.3.1

   $$C := A + 2 * B$$

   Die symbolischen Werte seien mit $w(\ldots)$ bezeichnet, d. h. vor Ausführung von $C := A + 2 * B$ sei $w(A) = a$ und $w(B) = b$.
   Dann besagt die symbolische Ausführung der obigen Zuweisung:

   $$w(C) = w(A) + 2 * w(B) = a + 2 * b$$

[31] Unter dem Gesichtspunkt der Erhaltung von Arbeitsplätzen ist dies sogar ein Vorteil.

Sei $D := C - A$ eine danach auszuführende Anweisung, dann gilt dafür:

$$w(D) = w(C) - w(A) = a + 2 * b - a = 2 * b$$

Bei der Berechnung von w(D) wurde eine Umformung (Vereinfachung) vorgenommen. Diese ist mathematisch korrekt, muß aber nicht unbedingt der Computer-Arithmetik entsprechen, die Rundungsfehler erzeugen kann.

2. Symbolische Ausrechnung von Bedingungen für Verzweigungen
Verzweigungen treten bei den Kontrollstrukturen *if-then-else, while, repeat*, etc. auf. Das Vorgehen wird exemplarisch an der Anweisung **if** B **then** A1 **else** A2 demonstriert (bei *while, repeat*, etc. läuft die Auswertung analog). Folgende drei Fälle können für den Wert des symbolischen Ausdrucks für B, d. h. für $w(B)$, auftreten:

   (a) Es läßt sich zeigen, daß stets $w(B) = true$ gilt: Dann reduziert sich die Ausrechnung der obigen Anweisung auf die symbolische Ausrechnung von $A1$.

   (b) Es läßt sich zeigen, daß stets $w(B) = false$ gilt: Dann muß nur $A2$ symbolisch ausgerechnet werden.

   (c) Es läßt sich nicht zeigen, daß stets $w(B) = true$ bzw. $w(B) = false$ gilt: In diesem Fall sind beide Ausgänge der Verzweigung zu verfolgen, um eine komplette Auswertung der *if*-Anweisung zu erhalten.

   Fall c1): ($true$-Ausgang) $A1$ ist symbolisch auszurechnen, wobei dabei und im folgenden die sogenannte **Pfadbedingung** $w(B) = true$ als gültig anzunehmen ist.

   Fall c2): ($false$-Ausgang): $A2$ ist symbolisch auszurechnen, wobei dabei und im folgenden die Pfadbedingung $w(B) = false$ als gültig anzunehmen ist.

Wenn das Programm keine Schleifen hat, bricht das Verfahren für Verzweigungen nach endlich vielen Schritten ab, da es dann nur endlich viele Wege mit endlich vielen Verzweigungspunkten auf den einzelnen Wegen gibt. Die Menge aller dieser Wege kann dann zusammen mit den symbolischen Berechnungen als endlicher Baum dargestellt werden.

Beispiel 12.3.2 (Funktion zur Berechnung des Absolutbetrags)

```
1   procedure ABSOLUTE (x: integer): integer
2       var x, y: integer;
3       if x < 0
4           then y := -x;
5           else y := x;
6       return y;
```

Der Baum der symbolischen Ausführungen von ABSOLUTE ist in Abbildung 12.6 dargestellt. Dabei bezeichnet pb die Pfadbedingung.

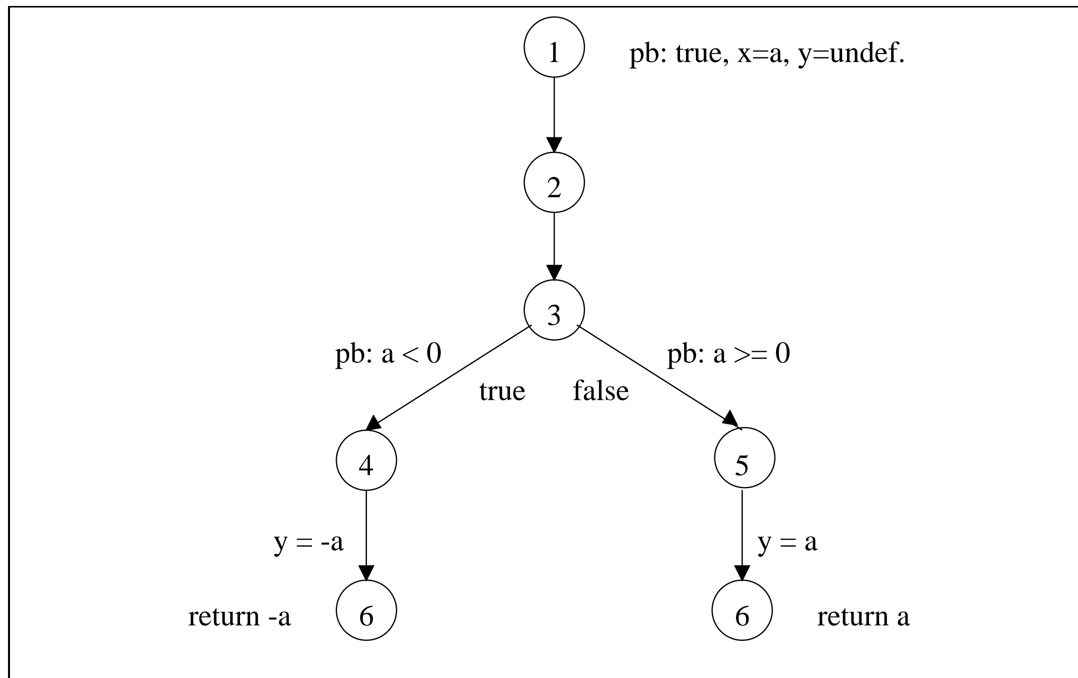

Abb. 12.6: Symbolischer Ausführungsbaum von ABSOLUTE

Wenn das Programm Schleifen enthält, die beliebig oft durchlaufen werden können (z. B. **for** i := 1 **to** n)[32], bricht das Verfahren bei der Ausrechnung der Verzweigungsbedingung nicht (nach endlich vielen Schritten) ab, da es dann Wege mit beliebig vielen Verzweigungspunkten auf dem Weg gibt. Zur Darstellung aller Wege bräuchte man also einen unendlichen Baum. Daher muß man sich auf eine endliche Anzahl von Durchläufen beschränken.

3. Symbolische Ausrechnung der Sequenz zweier Anweisungen
   Für $S = S1; S2$ sind die symbolischen Werte für die Variablen gemäß Anweisung $S1$ zu berechnen und bei der Berechnung gemäß Anweisung $S2$ zu benutzen; entsprechend ist mit den Pfadbedingungen zu verfahren: Sei $p(S_i)$ die zu S_i gehörige Pfadbedingung $(i = 1, 2)$, dann ist $p(S_1) \wedge p(S_2)$ die zu $S = S_1; S_2$ gehörige Pfadbedingung, wobei allerdings $p(S_2)$ mit den (veränderten) Werten aus der Berechnung in S_1 zu berechnen ist.

[32] Eine *for*-Schleife „**for** i := 1 **to** 10“ erzeugt dagegen wieder nur endlich viele Wege endlicher Länge.

4. Symbolische Ausrechnung von Operationsaufrufen
   Ein Aufruf einer Operation (Prozedur oder Funktion) kann bei der symbolischen Berechnung auf zwei Arten behandelt werden:
   (a) Der Aufruf wird ersetzt, und zwar durch eine symbolische Rechnung gemäß der Parameterbelegung und den Anweisungen im Rumpf der Operation. Dabei entstehen bei Verzweigungen wieder neue Pfadbedingungen und Schleifen können nur mit endlich vielen Fällen (stichprobenartig) durchgerechnet werden.
   (b) Beim Aufruf werden nur die aktuellen symbolischen Werte der Parameter eingesetzt, der Ausdruck wird ansonsten nicht weiter „aufgelöst“ bzw. ersetzt. Das Ergebnis der gesamten symbolischen Berechnung ist also ein Ausdruck, in dem Operationssymbole (Prozedur- oder Funktionsnamen) vorkommen.

Die symbolische Ausführung hat folgende Vor- und Nachteile.

Vorteile:

+ Eine formale Programmspezifikation ist nicht erforderlich. (Der berechnete Ausdruck muß nur betrachtet und überprüft werden).

+ Ein symbolischer „Test“ deckt eine Vielzahl normaler Testdaten ab.

+ Es werden insbesondere Fehler entdeckt, bei denen für eine Teilmenge der Eingaben die Ergebnisse falsch berechnet werden.

+ Das Vorgehen unterstützt das Finden von Schleifen-Invarianten für die formale Programmverifikation (genaueres dazu s. Kap. 12.4).

Nachteile:

– Das Überprüfen der Ergebnisse ist schwieriger als beim konventionellen Testen, da symbolische Ausdrücke mit der Spezifikation verglichen werden müssen. Das ist besonders schwierig, wenn unaufgelöste Operationsaufrufe im symbolischen Ausdruck — aber nicht in der Spezifikation — vorkommen.

– Die verwendete Programmiersprache muß formal definiert sein, damit ein symbolischer Interpreter arbeiten kann.

– Die Methode hat mehr Voraussetzungen als das konventionelle Testen (symbolischer Interpreter, Interpretation der Ergebnisse).

– Als alleinige Testmethode ist die symbolische Ausführung nicht ausreichend (ein funktionsorientierter Test fehlt).

– Durch Umformungen während der symbolischen Ausführung (z. B. $X+1.99+0.01$ zu $X+2$) werden Maschineneigenschaften (z. B. Effekte der „Real-Arithmetik“) nicht geeignet berücksichtigt.

– Es muß ein **Theorembeweiser** zur Verfügung stehen, der alle notwendigen Umformungen erzeugen kann (z. B. $a + 2 * b - a = 2 * b$ in Beispiel 12.3.1) und korrekt arbeitet. Leider kann es für hinreichend mächtige Programmiersprachen keinen vollständigen automatischen Theorembeweiser geben (sonst wäre ja z. B. die Äquivalenz von Programmen entscheidbar, vgl. Fußnote 13 auf Seite 36). Daher kann es bei einem verwendeten Theorembeweiser vorkommen, daß mit ihm weder $w(B) = true$ noch $w(B) = false$ bei einer Verzweigung mit Prädikat B beweisbar ist und daher beide Verzweigungsausgänge gewählt werden, obwohl tatsächlich einer nicht ausführbar ist.

Welche Konsequenz hat der letzte Nachteil?

1. Bei der Verwendung der symbolischen Ausführung zur Erzeugung von Testdaten (s. Kap. 11.2) kann kein Eingabedatum gefunden werden, welches das entsprechende Pfadprädikat erfüllt. (Hoffentlich merkt man das, sonst kann man lange vergeblich suchen.)

2. Bei der Verwendung für die Verifikation kann die Korrektheit des Programms nicht bewiesen werden, wenn in einem (unausführbaren) Pfad falsche Berechnungen vorgenommen werden, die der Ausgabebedingung (Zusicherung) widersprechen. Das Programm kann aber auf allen ausführbaren Pfaden (also immer) korrekt arbeiten.

Howden berichtet folgendes von Experimenten mit dem System DISSECT, welches auf 12 einfache Programme und Programmteile aus dem Buch von Kernighan/Plauger ([KeP 78]) angewandt wurde (s. [How 77]). Die Programme enthalten insgesamt 22 Fehler. Durch symbolische Ausführung werden 13 Fehler gefunden, die sich folgendermaßen klassifizieren lassen:

- 9 Berechnungsfehler[33], d. h. auf dem richtig ausgewählten Weg im Programm wird eine falsche Funktion berechnet,

- 3 fehlerhafte Initialisierungen,

- 1 Bereichsfehler[34], d. h. ein Bereich wird falsch ausgewählt, weil eine Bedingung falsch berechnet wird.

[33] siehe Definition 7.2.1 auf 195
[34] siehe Definition 7.2.1

Von den neun Berechnungsfehlern werden vier *nur* durch symbolisches Ausführen, aber nicht (bzw. mit geringer Wahrscheinlichkeit) durch Testen gefunden: wenn die Berechnung auf einem Weg fast immer das richtige Ergebnis berechnet (und nur für wenige Eingaben nicht), der symbolische Ausdruck aber offensichtlich falsch ist (zwei Fälle); wenn erkennbar falsche Formeln bei reellwertigen Approximationsproblemen verwendet werden, das Testergebnis aber „richtig aussieht", da die Abweichung erst etwa 10 Stellen hinter dem Komma auffällt.

Bei den neun nicht gefundenen Fehlern sind die Berechnungsausdrücke oder die Booleschen Ausdrücke bei den Verzweigungen falsch. Beides ist im symbolischen Ausdruck genauso gut oder schlecht wie im Programmtext als Fehler zu erkennen, da sich beide Ausdrücke in diesem Fall eins-zu-eins entsprechen. Außerdem werden fehlende Pfade schlecht erkannt, da sie nicht ausgeführt werden (können).

Abschließend eine provokative Frage:
Warum benutzt man die symbolische Ausführung als Hilfsmittel für das Testen mit konkreten Daten (siehe Konsequenz 1 auf Seite 331)? Es ist doch anscheinend sinnvoller, gleich mit den symbolischen Werten zu rechnen, da man damit ganze Klassen von konkreten Daten simuliert.

Es gibt dafür folgende Gründe:

1. Bei der symbolischen Ausführung mit Werkzeugen verläßt man sich

   (a) auf die korrekte Wiedergabe der Semantik der Programmiersprache[35],

   (b) auf die korrekte Vereinfachung der symbolischen Ausdrücke,

   (c) auf die korrekte Ermittlung der Werte von Bedingungen (an den Verzweigungsstellen) durch den Theorembeweiser.

   Bei (a) und (b) liegen eventuell Schwächen vor, z. B. bei der Modellierung von Überlauf (Overflow) und Real-Arithmetik, bei (c) liegt notwendigerweise eine Schwäche vor. Daher sind Tests mit konkreten Werten notwendig, um eventuelle Fehler aufzudecken.

2. Symbolische Ausführung behandelt nur die funktionale Korrektheit des Programms. Leistungsmessungen können nur mit konkreten Testdaten vorgenommen werden.

3. Der Vergleich der symbolischen Ausgaben mit der Spezifikation kann mühsam und fehleranfällig sein, da die Ausgaben (wegen (b)) eventuell ungenügend vereinfacht sind oder Fehler nicht auffallen (siehe den ersten Nachteil auf S. 330 und oben

[35] bzw. der Semantik des Compilers, die davon abweichen kann

die Experimente von Howden)[36]. Für konkrete Ausgaben ist die Überprüfung anhand der Spezifikation oft einfacher.

12.4 Formale Verifikation

Grundlage von Korrektheitsbeweisen von Programmen ist (wie bei der symbolischen Ausführung) eine klare, axiomatische Definition der Semantik der Konstrukte der Programmiersprache. Außerdem muß natürlich die Spezifikation für das Programm in formaler Form vorliegen. Diese Spezifikation soll in der Form von Zusicherungen (engl.: assertions) vorliegen. **Zusicherungen** sind Formeln der Prädikatenlogik erster Stufe über den Variablen[37], die im Programm verwendet werden. Das geforderte Resultat des Programms wird als Ausgabezusicherung formuliert in der Form

PROVE (<Boolean>),

wobei <Boolean> die Formel darstellt.
Falls die (Eingabe-)Variablen gewisse Bedingungen erfüllen müssen, wird noch eine Eingabezusicherung formuliert in der Form

ASSUME(<Boolean>) .

Die formale Programmverifikation besteht aus zwei Teilen:

Teil 1 (partielle Korrektheit): Es ist zu beweisen, daß nach jeder Beendigung des Programms die Ausgabezusicherung gilt, wenn zu Beginn die Eingabezusicherung gültig ist.

Teil 2 (totale Korrektheit): Es ist zusätzlich zu beweisen, daß das Programm unter allen Umständen beendet wird (d. h. keine unendliche Schleife enthält).

Das genauere Vorgehen wird durch folgende Bemerkungen erläutert:

1. Bei Programmen *ohne Schleifen* gibt es nur endlich viele (Berechnungs-)Wege durch den Programmgraphen. Daher kann folgendermaßen vorgegangen werden:
   (a) Für jeden Weg ist das Programm symbolisch auszuführen mit der Eingabezusicherung als anfänglicher Pfadbedingung.
   (b) Aus dem berechneten symbolischen Ergebnis muß dann logisch die Ausgabezusicherung folgen (dies ist zu zeigen).

[36] Druckt man arithmetische Ausdrücke durch ein Werkzeug in der (in der Mathematik üblichen) *zweidimensionalen* Form aus, fallen Klammerungsfehler doch auf: Ausdruck $MERKE(I) - 1/10 + 1$ [statt richtig $(MERKE(I)-1)/10+1$] würde in folgender Form dargestellt: $MERKE(I) - \frac{1}{10} + 1$ [statt $\frac{MERKE(I)-1}{10} + 1$].

[37] und — bei Eingabevariablen — ihren Anfangswerten

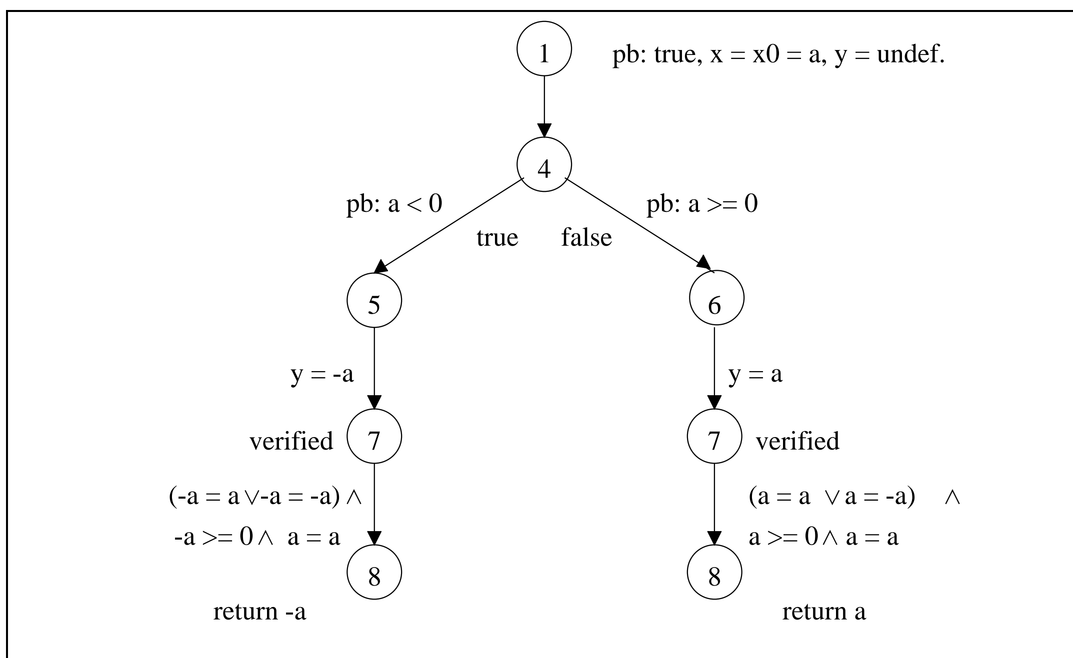

Abb. 12.7: Symbolischer Ausführungsbaum von ABSOLUTE mit PROVE

Beispiel 12.4.1 (Prozedur zur Berechnung des Absolutbetrages)
Eingabezusicherung: keine Bedingung
Ausgabezusicherung:

$(x0 \geq 0 \Rightarrow y = x0) \wedge (x0 < 0 \Rightarrow y = -x0) \wedge x = x0$
(Dabei ist $x0$ der Wert von x zu Beginn der Berechnung.)

Prozedurtext (mit Zusicherungen):

```
1    procedure ABSOLUTE (x: integer): integer;
2    ASSUME(true);
3        var x, y: integer;
4        if x < 0
5            then y := -x;
6            else y := x;
7    PROVE ((y = x0 ∨ y = -x0) ∧ y ≥ 0 ∧ x = x0);
8        return y;
```

Der symbolische Ausführungsbaum von ABSOLUTE ist in Abb. 12.7 dargestellt. Dabei bezeichnet „pb“ wieder die Pfadbedingung. Im Unterschied zu Abbildung 12.6 enthält der Ausführungsbaum auch die zu beweisenden Ausdrücke aus der PROVE-Angabe in Zeile 7. Bei Knoten 7 im linken Pfad muß aus der Pfadbedingung „$a < 0$“ die Bedingung „$-a \geq 0$“ gefolgert werden; alles andere sind Tautologien.

2. Bei Programmen *mit Schleifen* „schneidet" man die Schleifen auf (engl.: cut) und fügt an diesen Schnitten (cuts) sogenannte **induktive Zusicherungen** ein. Damit gibt es im Kontrollflußgraphen des Programms nur endlich viele Wege endlicher Länge, die von einer Zusicherung zu einer anderen[38] Zusicherung führen. Für jeden solchen Weg kann man wieder die symbolische Programmausführung wie bei 1 (s. oben) anwenden, jetzt **Schnitt-symbolische Ausführung** genannt. Wenn dabei alle Endzusicherungen aus den Anfangszusicherungen folgen, ist die partielle Korrektheit des Programms bewiesen.

BEISPIEL 12.4.2 (PROZEDUR GGT ZUR BERECHNUNG DES GRÖSSTEN GEMEINSAMEN TEILERS ZWEIER GANZER ZAHLEN)
Der Programmtext mit Schnitten (cuts) und induktiven Zusicherungen lautet:

```
1   procedure GGT (x, y: integer): integer;
2   cut2 ...      ASSUME (x > 0 ∧ y > 0);
3                 var a, b: integer;
4                 a: = x;
5                 b: = y;
6                 while (a ≠ b) do
7   cut7 ...          ASSERT(ggt(a, b) = ggt(x, y) ∧ a ≠ b);
8                       if a > b
9                       then a : = a − b;
10                      else b := b − a;
11                end;
12                PROVE(a = ggt(x, y));
13  return ... return a;
```

Bei cut_7 in Abb. 12.8 ist die folgende sogenannte **Verifikationsbedingung** *durch einen Menschen oder durch einen Theorembeweiser zu beweisen. Die Prämisse besteht dabei aus der Pfadbedingung und den symbolischen Variablenwerten auf dem Weg von 2 nach 7; die Folgerung ist die zu beweisende ASSERT-Zusicherung in Zeile 7:*

$$m > 0 \wedge n > 0 \wedge m \neq n \wedge a = m \wedge b = n \wedge x = m \wedge y = n$$

$$\Rightarrow ggt(a, b) = ggt(x, y) \wedge a \neq b$$

Dies ergibt sich wegen $a = m = x$ und $b = n = y$ und durch Einsetzen von $a = m$ und $b = n$ in $m \neq n$. **q. e. d.**

[38] nicht unbedingt verschieden von der ersten

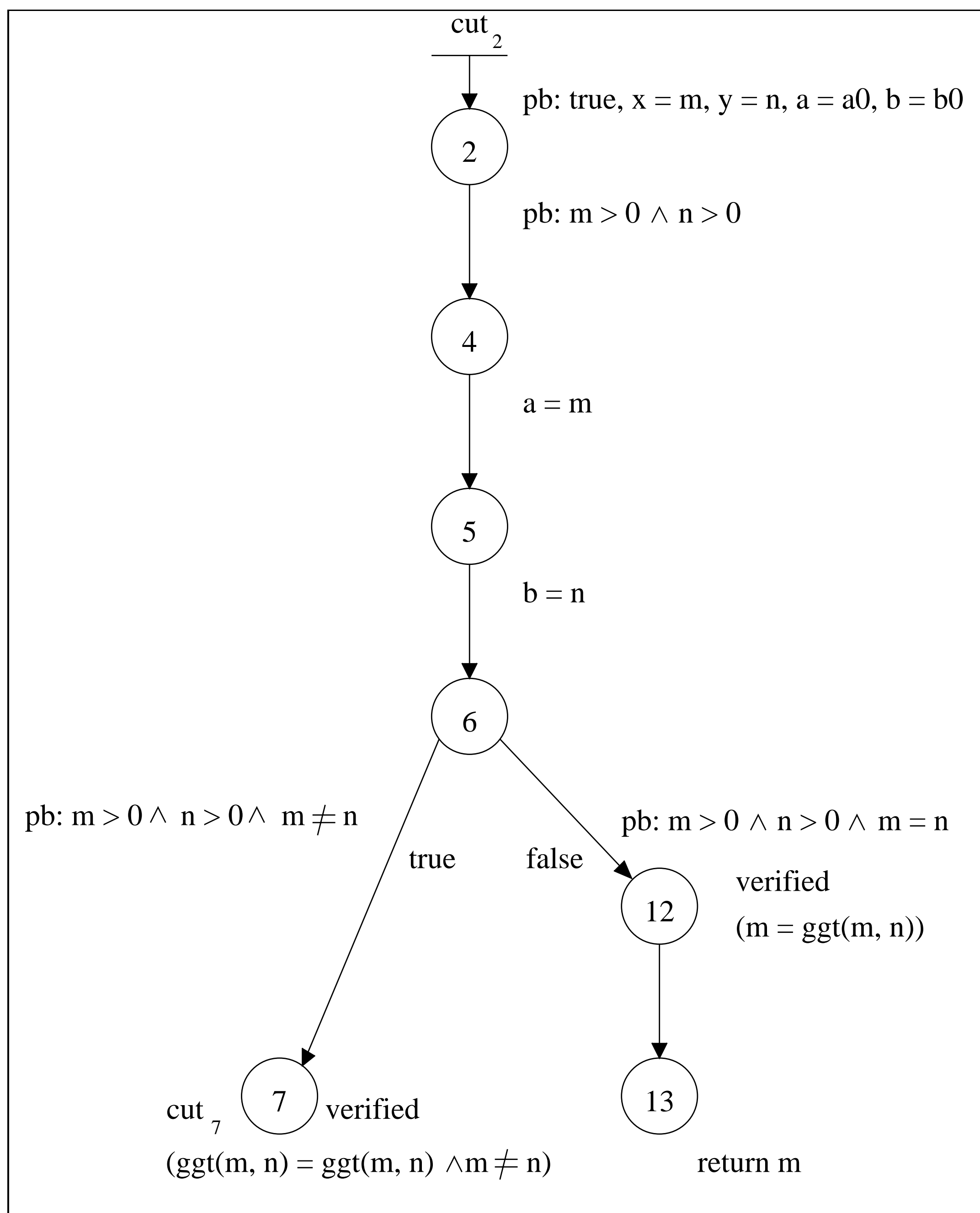

Abb. 12.8: Symbolischer Ausführungsbaum der GGT-Funktion (ausgehend von cut_2 bis zum Schnitt cut_7 und zum *return*)

Die Programmverifikation hat folgende Vorteile und Nachteile bzw. Probleme:

<u>Vorteile:</u>
Das Programm wird (bezüglich der Eingabe- und Ausgabezusicherungen) als korrekt für alle Eingabefälle bewiesen. (Dies gilt für den Quellcode, für den Objektcode gilt dies natürlich nur, falls der Compiler dieselbe Programmsemantik hat wie das Verifikationswerkzeug).

<u>Nachteile/Probleme:</u>

1. Theoretische Probleme:
   (a) Die Verifikationsbedingungen sind i. allg. nicht entscheidbar. Falls die Bedingung falsch ist (d. h. wenn das Programm falsch ist), hält ein korrekter Theorembeweiser eventuell nicht an.
   (b) Die induktiven Zusicherungen können nicht automatisch (sondern nur mit Intuition) ermittelt werden.
   (c) Der totale Korrektheitsbeweis ist nicht immer möglich (wegen des Halteproblems).
2. Praktische Probleme:
   (a) Der Berechnungsaufwand für den Theorembeweiser ist enorm (ebenso für den Menschen). Daher sind alle vorliegenden Beweise für Programme „Spiel"-Beweise für Mini-Programme wie z. B. GGT.
   (b) Die Korrektheit eines automatischen Beweises setzt die Korrektheit des Theorembeweisers voraus. (Wenn er schon nicht alles beweisen kann, also manchmal „passen" muß, sollte er wenigstens nichts falsches beweisen.)
   (c) Wenn das Programm falsch ist, wird ein Beweisversuch bezüglich der korrekten Zusicherungen stets fehlschlagen. Bei großen Programmen wird der erste Beweisversuch vermutlich immer scheitern, da das Programm mit ziemlicher Sicherheit (mindestens einen) Fehler hat.
   (d) Folgendes wird i. allg. passieren, selbst wenn der Theorembeweiser korrekt ist und anhält:
      i. Meldung eines Syntaxfehlers im Programm oder in den Zusicherungen (warum nicht auch dort?);
      ii. ein Laufzeitfehler im Theorembeweiser, da nicht alle Fehler syntaktisch abgefangen werden können (wie bei Programmfehlern und normalen Compilern);
      iii. (als bestes) die Meldung „Programm nicht verifiziert", falls der Theorembeweiser mit einer Zeitschranke arbeitet, um unendliche Beweisversuche abzufangen.

(e) Probleme der Fehlerbeseitigung:
Die Meldung „Programm nicht verifiziert“ hilft nicht direkt, die Ursachen der Fehler zu finden. Die Aussicht, gute Fehlerlokalisierer für Beweise (proof debugger) zu entwickeln, die diese Lücke schließen würden, ist nicht sehr gut: Eine Modularisierung der Programme kann zwar die Beweise verkleinern, dafür handelt man sich aber ein Schnittstellenproblem ein (Konsistenz der Beweise für verschiedene Module).

(f) Ein als korrekt bewiesenes Programm kann dennoch falsche Resultate liefern:
   i. Die Semantik der Programmiersprache kann falsch modelliert sein.
   ii. Der Compiler, das Betriebssystem oder die Hardware können anders als angenommen arbeiten. Der Speicher kann für bestimmte Programmanwendungen zu klein sein und einen Programmabsturz verursachen[39].
   iii. Die Eingabe- und Ausgabezusicherungen in ihrer prädikatenlogischen Form können die eigentliche (meist verbale) Anforderungsspezifikation des Auftraggebers falsch wiedergeben. In der Literatur sind viele solcher Fälle wiedergegeben worden (siehe z. B. [GeY 76]).

Als Fazit läßt sich also feststellen:
Die formale Verifikation besitzt eine Reihe von Nachteilen und Problemen und macht daher das Programmtesten nicht überflüssig, obwohl letzteres nicht hinreichend ist. Da insbesondere das Testen von Schleifen ein großes Problem ist, sollte die Induktionstechnik der Verifikation dabei zumindest *informell* angewendet werden. Es sind also die üblichen informellen mathematischen Beweise zu führen, die nicht jeden prädikatenlogischen Schluß umfassen (s. [DLP 79]). Bei sicherheitsrelevanter Software sollte stets eine (formale) Verifikation kritischer Teile erfolgen, damit zumindest das algorithmische Konzept als zertifiziert gelten kann (vgl. Kapitel 2.6). Interaktive Theorembeweiser können dabei eine nützliche Rolle spielen und einen Teil der oben erwähnten Nachteile umgehen.

[39] Ein zu kleiner Stack bzw. die falsche Behandlung des Stack-Überlaufs war die Ursache für den zweitägigen Ausfall des neuen Bundesbahnstellwerks in Hamburg-Altona im März 1995 (s. *SEN*, Vol. 20, No. 3, Juli 1995, S. 8).

12.5 Übungen

Übung 12.1:

Führen Sie für ein Programm bzw. ein Modul eigener Wahl (mit ca. 200 Zeilen Code) eine Inspektion durch. Überlegen Sie sich dafür geeignete Einzelinspektor-Phasen mit entsprechenden Untersuchungszielen.

Übung 12.2:

Ermitteln Sie für ein Programm bzw. eine Prozedur mit ca. 100 Zeilen

(a) das Kontrollflußschema, wobei ganze Segmente durch nur einen Knoten darzustellen sind (s. Def. 7.1.1);

(b) die Tiefe der Schleifenschachtelungen;

(c) problematische und unproblematische Schleifenterminierungen.

Übung 12.3:

Ermitteln Sie für ein Programm bzw. ein Modul mit mindestens fünf Prozeduren den Graphen der Prozeduraufrufe.

Übung 12.4:

Ermitteln Sie für den Kontrollflußgraphen von Abbildung 12.3 die *avail*-Mengen für die Knoten 0 bis 5 mit Algorithmus AVAIL.
Beachten Sie, daß Knoten 5 der Startknoten ist, der im Algorithmus mit 0 bezeichnet wird.

Übung 12.5:

Wenden Sie Algorithmus UR zur Ermittlung der *ur*-Anomalien auf das Programm aus Beispiel 6.3.1 auf S. 158 an. Unterstellen Sie dabei, daß Array a und Variable n zu Beginn definiert sind, nicht jedoch die Variablen $r0, r1, r2, r3$.
Hinweis: Sie müssen zuerst einen Kontrollflußgraphen (mit geeigneter Darstellung der *for*-Schleifen) bilden und jedem Knoten k die Mengen *DEF(k), REF(k), UNDEF(k)* bzw. $generate(k), kill(k)$ und $null(k)$ zuordnen.

Übung 12.6:

Geben Sie (analog zum Algorithmus UR) zwei Algorithmen DD und DU an, die mit Hilfe des LIVE-Algorithmus die *dd*-Anomalien bzw. die *du*-Anomalien ermitteln.

Übung 12.7:

Ermitteln Sie die *dd*-Anomalie für die Variable *change* in den Algorithmen LIVE und AVAIL durch direkte Betrachtung der passenden Pfadausdrücke.
Wie müssen die Programme geändert werden, damit die *dd*-Anomalie beseitigt wird?
Warum sind (trotz der überflüssigen Doppeldefinition von *change*) die jetzigen Realisierungen der Algorithmen effizienter, wenn man die Definition und die Referenz einer Variablen als gleich zeitaufwendig annimmt?

Übung 12.8:
Beweisen Sie die folgenden Regeln für die Datenflußaussagen der Konkatenation $A = B \cdot C$ von Konstrukten B und C für eine Variable (s. S. 324):
für alle $x \in \{an, ee, din, rin, uin, dout, rout, uout\}$ gilt:

$A.x = C.x$ wenn für B nur die Datenflußaussage $B.ee = true$ ist, d. h. die Binärzahl 01 000 000 repräsentiert B.
$A.x = B.x$ wenn für C nur die Datenflußaussage $C.ee = true$ ist, d. h. die Binärzahl 01 000 000 repräsentiert C.

Übung 12.9:
Ermitteln Sie die *dd*-Anomalie(n) im LIVE-Algorithmus für Variable *change* mit Hilfe der algebraischen Methode, die auf regulären Ausdrücken beruht.
Hinweis: Bestimmen Sie vorweg einen Kontrollflußgraphen zum LIVE-Algorithmus (unter geeigneter Darstellung der *for*-Schleifen) und den dazu gehörigen regulären Ausdruck bzw. arbeiten Sie von innen nach außen. Beachten Sie, daß in dem Startknoten des Kontrollflußgraphen die Variable *change* undefiniert ist. Benutzen Sie die Rechenregeln aus Übung 12.8.

Übung 12.10:
Geben Sie den symbolischen Ausführungsbaum der GGT-Funktion (von Beispiel 12.4.2) an, der bei cut_7 mit der ASSERT-Zusicherung als Pfadbedingung beginnt und bei cut_7 oder *return* endet (vgl. Abbildung 12.8 für cut_2). Beweisen Sie damit, daß auf jedem Weg, der von cut_7 ausgeht, die ASSERT-Zusicherung bei cut_7 (am Wegesende) bzw. die PROVE-Bedingung bei *return* erfüllt ist.

12.6 Verwendete Quellen und weiterführende Literatur

Howden hat schon 1978 darauf hingewiesen, daß bei der **statischen Analyse** verschiedenartige Dokumente zu untersuchen sind (s. [How 78c]). Die Aspekte, die dabei zu beachten sind (z. B. Vollständigkeit, Konsistenz) werden z. B. von Myers und Parrington/Roper beschrieben (s. [Mye 79], Kap. 6; [PaR 91], Kap. 3.5).
Fagan hat die Voraussetzungen für erfolgreiches manuelles Überprüfen behandelt und den Vorteil der **Inspektion** gegenüber dem herkömmlichen „**Walkthrough**" herausgestellt (s. [Fag 76], [Fag 86]). Die Unterschiede von Inspektion und Walkthrough haben auch Adrian et al. beschrieben (s. [ABC 82]). Angaben zu (Structured) Walkthroughs, zu **Peer-Group-Reviews** und zu **Audits** findet man bei Yourdon und Frühauf et al. (s. [You 78], [FLS 91b]). Die Vorteile eines konsequenten Reviews werden von Hart beschrieben (s. [Har 82]). Die Probleme bei Gruppen-Inspektionen und ihre Lösung durch **Einzelinspektorphasen** bzw. Szenario-basierte Methoden beschreiben Knight/Myers bzw. Porter et al. (s. [KnM 91],

[Po& 95]). Sie machen — ebenso wie Johnson/Tjahjono — auch Vorschläge zur **computergestützten Inspektion** (s. [JoT 93]). Um vollständige, genaue Anforderungsspezifikationen zu erhalten, den Gruppenzusammenhalt zu stärken und Wissenslücken beim Ausfall einer Person des Entwicklungsteams zu vermeiden, schlagen Martin/Tsai die n-fache Inspektion der Anforderungsspezifikation durch n Gruppen vor (s. [MaT 90]). Vorschläge zum Review von Entwürfen machen Parnas/Weiss (s. [PaW 85]). Eine kritische Bewertung bzw. einen Überblick über die Inspektionstechnik geben Hausen bzw. Schnurer (s. [Hau 83], [Sch 88b]).
Die zusätzlichen Typangaben für eine bessere Typüberprüfung hat Howden vorgeschlagen (s. [How 78c]). Die **statische Analyse** zur Überprüfung von **Programmier-Richtlinien** verwenden z. B. Marktscheffel und Wystrychowski et al. (s. [Mar 87], [WBM 95]). Die **Datenflußanalysetechniken**, die den LIVE- und den AVAIL-Algorithmus von Hecht benutzen, stammen von Osterweil et al. (s. [OFT 81]), die **algebraische Methode**, die auf regulären Ausdrücken bzw. strukturierten Programmen basiert, wurde von Forman entwickelt (s. [For 84]).
Die Darstellung der **symbolischen Ausführung** orientiert sich an Darringer/King (s. [DaK 78]). Von Howden stammt eine Untersuchung von Fehlern, die mit dieser Methode gefunden wurden (s. [How 77]). Die Vor- und Nachteile der symbolischen Ausführung wurden von Balzert übernommen (s. [Bal 82]).
Die Darstellung der **formalen Verifikation** beruht auf Hantler/King (s. [HaK 76]). Die Idee zum Aufschneiden der Schleifen stammt von R. W. Floyd (s. [Flo 67]). Die Probleme der formalen Verifikation sind von Tanenbaum, DeMillo et al., Fetzer, Wilk und Wing et al. aufgezählt worden (s. [Tan 76], [DLP 79], [Fet 88], [Wil 90], [WiV 95]). Letztere schlagen deshalb modellbasiertes Überprüfen (model checking) als Alternative vor. Ein interaktives Verifikationssystem stellen z. B. Halpern et al. vor (s. [Ha& 87]). Eine Übersicht über den Stand der Entwicklung von Methoden und Werkzeugen gibt [BrJ 95].
Eine konsequente Anwendung der Techniken der statischen Analyse (ohne Testen) durch die Entwicklerinnen und Entwickler schlagen Selby et al. unter der Bezeichnung „**Cleanroom Software Development**“ vor (s. [SBB 87]).