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. Spezifikationsanalyse
   - 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. Entwurfsanalyse
   (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

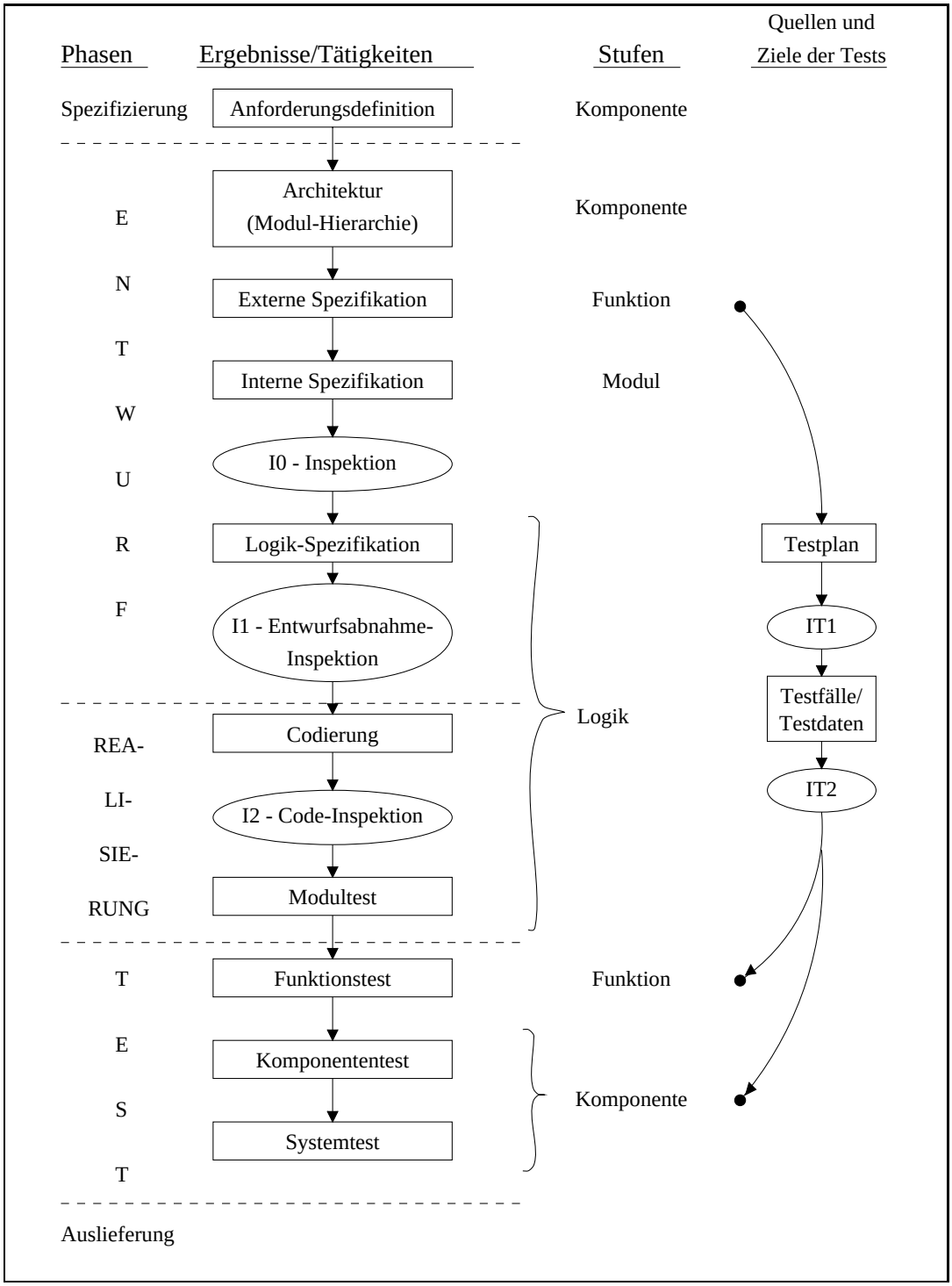


Abb. 12.1: Konstruktions- und Inspektionsschritte im Lebenszyklus der Software (nach [Fag 76], S. 105)

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

5. Verfolgung/Überprüfung
   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. Auswertung/Nachbereitung

   (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. Inspektion als Einschub beim dynamischen Testen
   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.

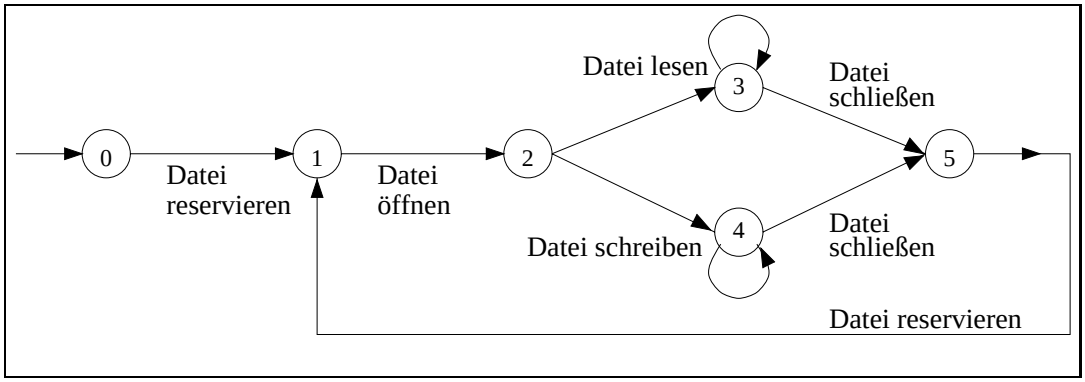


Abb. 12.2: Zustandsdiagramm der erlaubten Operationen auf Dateien

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]).

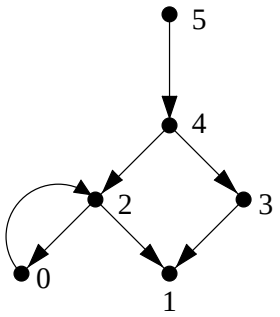
Im folgenden werden die **LIVE-/AVAIL-Algorithmen** vorgestellt, die auf Algorithmen aufbauen, die für die Programmoptimierung entwickelt wurden. Diese Algorithmen operieren mit den Begriffen *generate*, *kill* und *null*. Für die Anwendung bei der Datenflußanalyse muß dann eine Zuordnung zu den Begriffen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



tok = {A, B}
null(0) = {A, B}
kill(1) = {A, B}
generate(2) = {A}, null(2) = {B}
generate(3) = {B}, null(3) = {A}
kill(4) = {A, B}
null(5) = {A, B}

(Wegen generate(l) ⊔ kill(l) ⊔ null(l) = tok sind die nichtangegebenen Mengen leer.)

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

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

Pfadausdrücke $P(A;\rightarrow 3) = P(A;5,4) = nk$ *und* $P_r(A;\rightarrow 3) = P_r(A;5,4) = k$ *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. $A \in live(l) :\Leftrightarrow P_r(A;l\rightarrow) = g \cdot p + p'$
   wobei p *und* p' *Pfadausdrücke über* g *und* k *sind.*

2. $A \in avail(l) :\Leftrightarrow P_r(A;\rightarrow l) = p \cdot g$
   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 αur^* 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 $\alpha ur^* r$ bis zum Knoten l inklusive vorliegt, d. h. ein Pfadausdruck $\alpha ur\beta$ 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.

Algorithmus UR:

```
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).

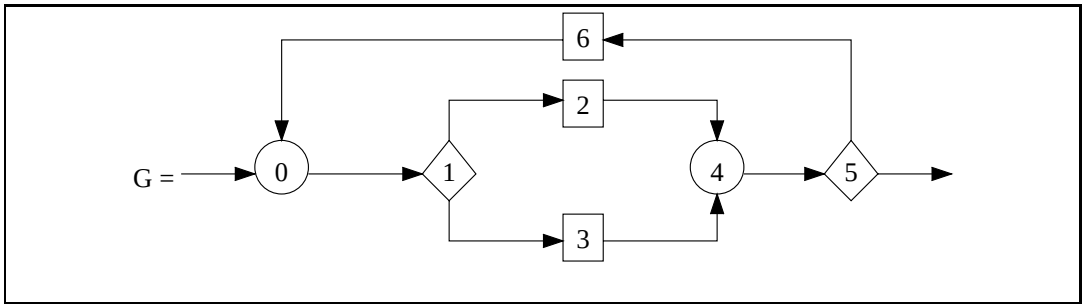


Abb. 12.4: Kontrollflußschema G

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 *def* bzw. *ref* bzw. *undef* ist
dout, rout, uout:	es gibt einen Weg durch s, so daß die letzte („out“) Aktion für v ein *def*, *ref* bzw. *undef* 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.

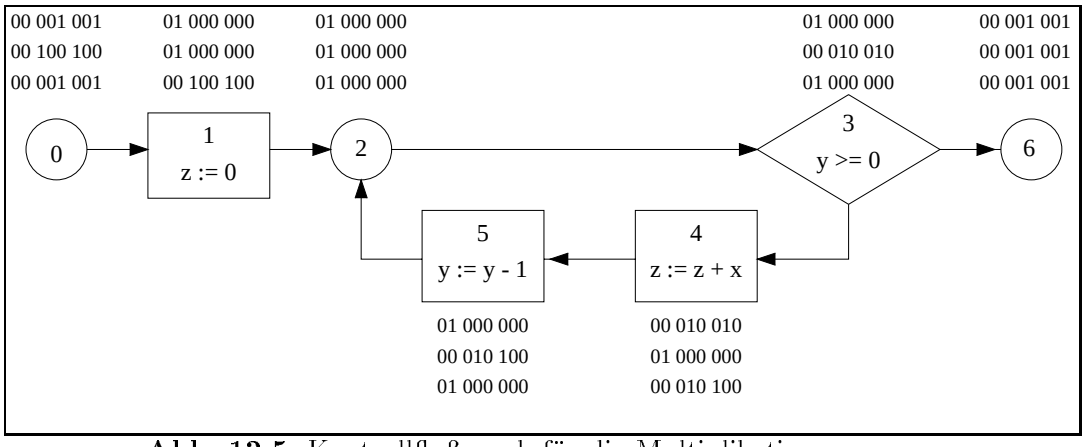


Abb. 12.5: Kontrollflußgraph für die Multiplikation $z = x * y$

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 „$\cdot$“ 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:

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

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

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

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 ∗ ... 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.

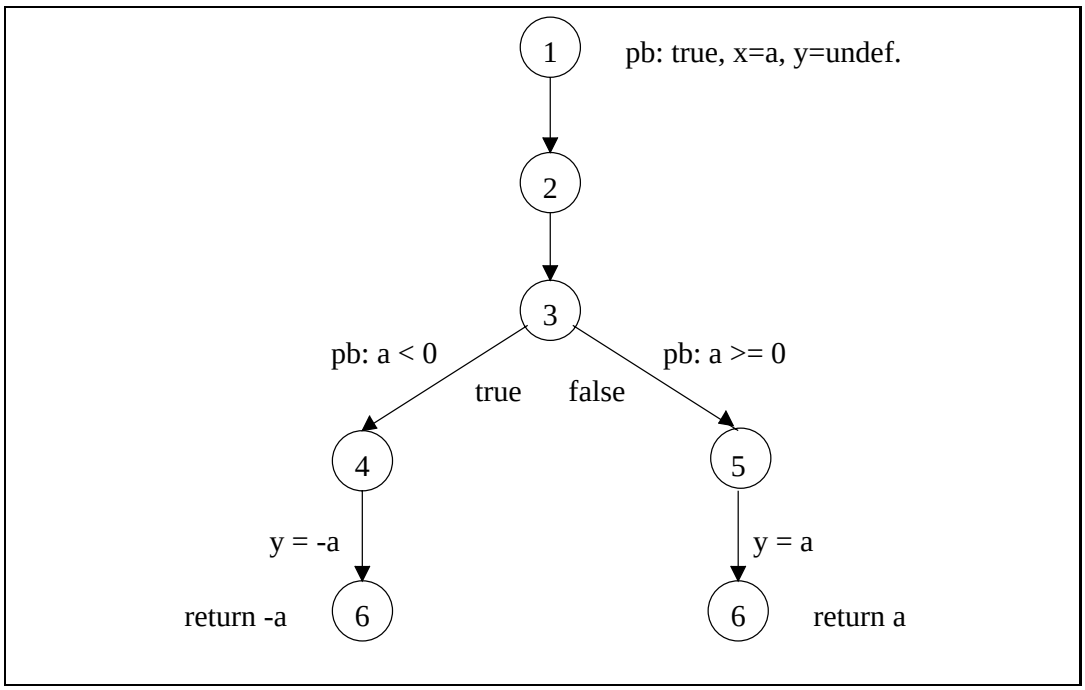


Abb. 12.6: Symbolischer Ausführungsbaum von ABSOLUTE

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

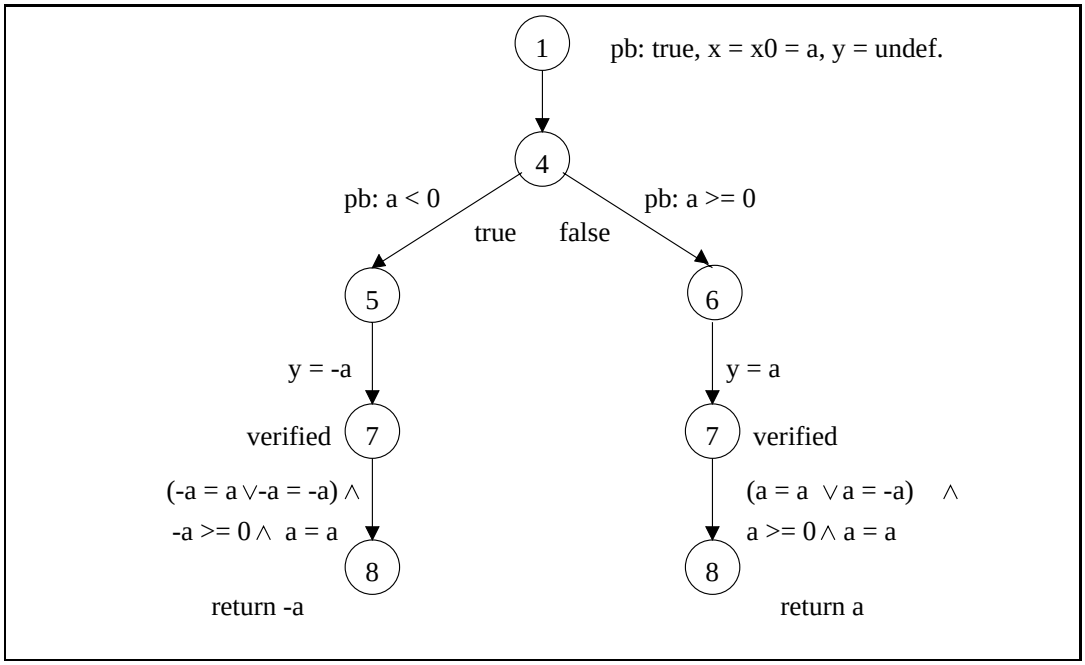


Abb. 12.7: Symbolischer Ausführungsbaum von ABSOLUTE mit PROVE

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   cut_2 ...       ASSUME (x > 0 ∧ y > 0);
3                   var a, b: integer;
4                   a: = x;
5                   b: = y;
6                   while (a ≠ b) do
7   cut_7 ...           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

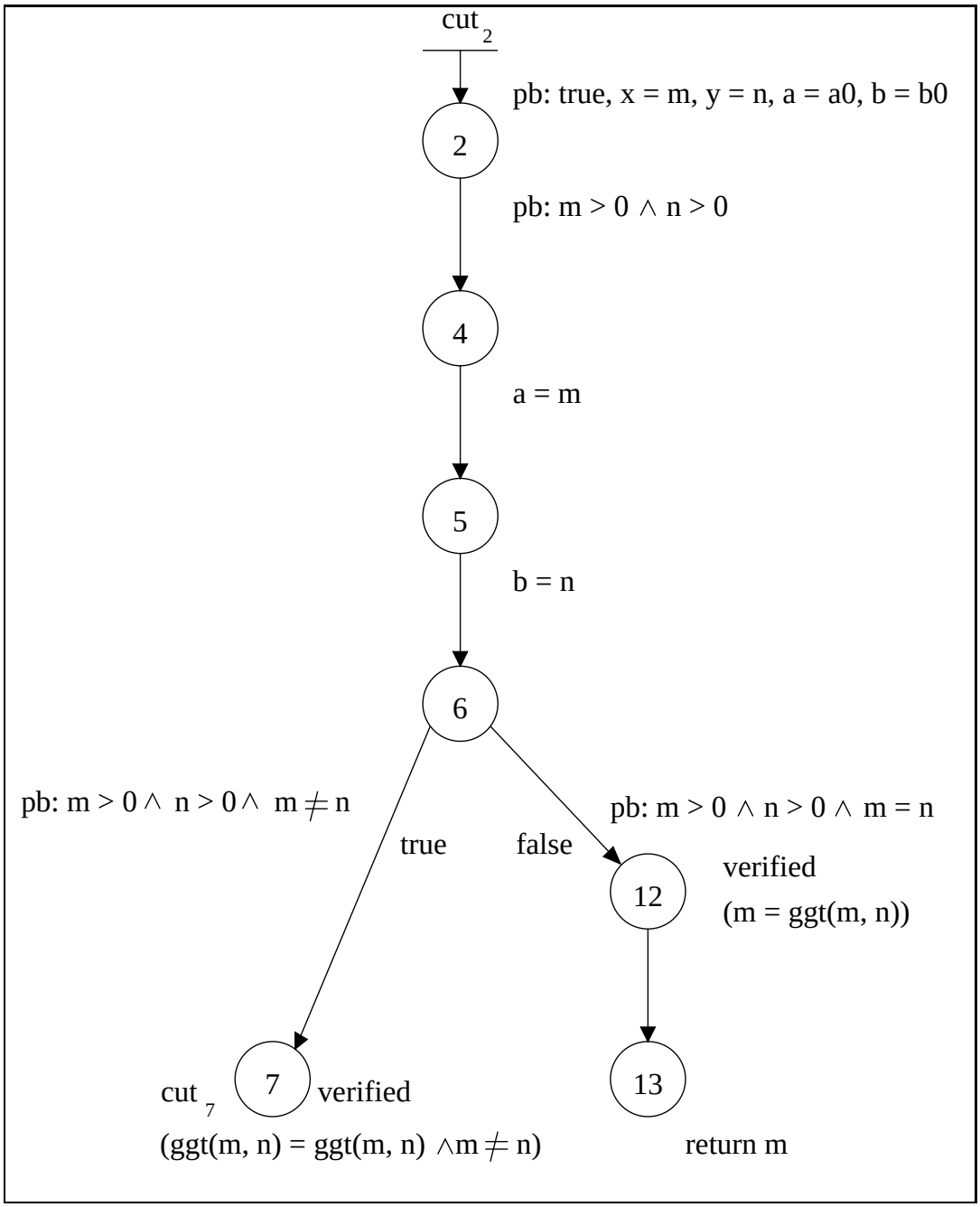


Abb. 12.8: Symbolischer Ausführungsbaum der GGT-Funktion (ausgehend von cut_2 bis zum Schnitt cut_7 und zum *return*)

Die Programmverifikation hat folgende Vorteile und Nachteile bzw. Probleme:

Vorteile:
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).

Nachteile/Probleme:

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. Szenariobasierte 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]).

13 Testen „im Großen“

13.1 Probleme beim Testen großer Systeme

Mit den in Teil II und III dieses Buches vorgeschlagenen Methoden wurde nur die Funktion (das Eingabe-/Ausgabeverhalten) der Programme getestet. Zur Überprüfung der Qualität eines Systems gehört aber mehr (s. Kapitel 3.2). Daher hat das Testen eines Systems viele Aspekte, die zur Klassifikation der Testverfahren dienen können. Es handelt sich dabei im einzelnen um:

1. Art der Prüfgegenstände und ihre Stellung im Softwarelebenszyklus:
   Bei sehr großen Systemen reicht eine Untergliederung in Module nicht aus; sie sind aus Gründen des Projektmanagements in Komponenten zusammenzufassen, die wiederum zu Teilsystemen zusammengefaßt werden. Somit kommen als Prüfgegenstände Module, Komponenten, Teilsysteme, Systeme oder geänderte Systeme in Frage. Die entsprechenden Tests heißen so wie die getesteten Teile, also z. B. **Komponententest** bei Komponenten (vgl. Abbildung 12.1 auf S. 302). Nur bei Systemen, die in der sogenannten Wartungsphase geändert werden, heißt der Test **Regressionstest**, da oft durch Änderungen neue Fehler in das System eingebaut werden und diese Regression[1] des Systemverhaltens bzw. die Tatsache, daß die bisherigen Fehler nicht mehr auftreten, getestet werden muß.

2. Merkmale der Software und das Ziel des Testens[2]:

   (a) Funktionalität
   Ist die Funktionalität korrekt und vollständig? Dies ist i. allg. dynamisch zu testen.

   (b) Schnittstellen
   Werden an den Schnittstellen der Module, Komponenten etc. die richtigen Informationen übergeben? Dies kann vor allem statisch überprüft werden (genaueres s. Abschnitt 13.5.1).

   (c) Leistung
   Wie sieht das Zeitverhalten und der Speicherplatzbedarf in typischen Fällen, aber auch in Extremfällen (bei starker Belastung des Systems) aus? Letzteres wird auch **Streßtest** genannt.

[1] wörtlich: „Rückbildung“; hier im Sinne von „Verschlechterung“ gebraucht
[2] vgl. Def. 3.2.1.3 bis Def. 3.2.1.4 auf S. 50 ff.

(d) Mengengerüst
Ist das System in der Lage, die spezifizierten oder vermutlich anfallenden Mengen von Informationen bzw. Daten zu verarbeiten? Die Systeme sollten auch bei Belastungen, die etwas größer sind als die spezifizierte Obergrenze, nicht sofort zusammenbrechen, sondern sinnvoll reagieren.

BEISPIEL 13.1.1
Ein Datenbanksystem, das bis zu 10.000 Kunden verwalten soll, ist auch mit 10.001 Kunden zu testen. Ein Betriebssystem, das gleichzeitig bis zu 100 Aufträge (Jobs) behandeln soll, ist auch mit 101 Aufträgen zu testen.

(e) Verfügbarkeit
Das System sollte nicht zu häufig zusammenbrechen („abstürzen"). Je nach Einsatzzweck sind die Verfügbarkeitsintervalle zu spezifizieren und ihre Einhaltung ist in einem Dauerbetrieb des Systems zu testen. Dazu gehört auch, die Mechanismen für den Wiederanlauf und die Regeneration evtl. zerstörter Daten zu testen.

(f) Sicherheit
Der technische Datenschutz des Systems ist zu testen. Der lesende Zugriff und insbesondere der (über-)schreibende Zugriff auf Daten darf nur den autorisierten Personen bzw. Prozessen erlaubt sein. Dies gilt sowohl für Hardware (z. B. Speicherschutz) als auch für Software (z. B. Datenbanken).

(g) Konfiguration
Software, die für mehrere Kunden entwickelt wird, ist in mehreren Varianten zu erstellen, die jeweils auf bestimmten Hardware- und Systemsoftware-Plattformen lauffähig sind. Es ist zu testen, ob die entsprechenden Varianten wirklich auf der spezifizierten Hardware (z. B. Sun XY), unter dem angegebenen Betriebssystem (z. B. Linux) und der vereinbarten Grafikoberfläche (z. B. Motif) lauffähig ist.

(h) Kompatibilität
Neue Versionen sollten mit alten Versionen anderer Komponenten weiterhin zusammen korrekt ausführbar sein.

(i) Benutzungsfreundlichkeit
Dazu zählen folgende Aspekte: Aufgabenangemessenheit, Selbsterklärungsfähigkeit, Fehlerrobustheit, Steuerbarkeit, Erwartungskonformität (genaueres siehe DIN 66234, Teil 8 bzw. ISO 9241, Teil 10, und [Op& 88]).

3. Testpersonen:

(a) Entwickler (bei eigenem oder fremdem Projekt)
Sie führen Modul-, Integrations-, System- und Regressionstests aus.

(b) Potentielle Benutzer
Sie führen den sogenannten Beta-Test von Standard-Software durch, bevor diese offiziell als Produkt vermarktet wird.

(c) Benutzer
Sie führen Abnahme- und Betriebstests aus.

4. Umgebung des Prüfgegenstands:

(a) Rechner (Entwicklungsrechner oder Zielrechner)
Nach einer Übertragung des Systems vom Entwicklungsrechner zum Zielrechner ist anschließend ein Installationstest erforderlich.

(b) Anwendungsumgebung (simuliert oder real)
Kritische Software, von deren Funktionieren Menschenleben oder hohe Kosten abhängen (z. B. Flugsoftware, Prozeßleitsysteme), sollten natürlich erst in simulierter Umgebung getestet werden.

5. Testreferenz:
Spezifikation und Benutzerhandbuch werden beim Systemtest als Testreferenz verwendet, Benutzerhandbuch und Benutzerwünsche werden beim Beta-Test und Abnahmetest referenziert.

Bei kleinen Programmen ist es sinnvoll, das ganze Programm mit den bisher beschriebenen Methoden „auf einmal“ — als eine Einheit — zu testen.

Bei großen Programmsystemen ist dieser **monolithische Test** aber schwierig bzw. ungünstig:

- Es müssen sehr viele Testdaten erzeugt werden, um z. B. alle Zweige im Programm oder etwa alle Wege (ohne Schleifen) auszuführen.
- Fehlende Testdaten (z. B. für eine 100%-Segmentüberdeckung) zu ermitteln ist schwierig, da die Eingabedaten bis zum Erreichen des entsprechenden Segments eine lange Kette von Transformationen durchlaufen (vgl. Abschnitt 11.2.1).
- Wenn bei einem Test(datum) ein Fehlverhalten auftritt, ist das Lokalisieren des verursachenden Fehlers sehr schwierig, da im Prinzip jede ausgeführte Anweisung als Fehlerquelle in Betracht kommt und die Ausgabe-Wirkung einer Anweisung bis zur (System-)Ausgabe noch mehrfach transformiert wird (genaueres siehe Kapitel 15).
- Fehler aus verschiedenen Modulen können sich gegenseitig maskieren und damit unerkannt bleiben.
- Parallelarbeit von mehreren Testern ist schwierig abzustimmen.

Daher sollte das gesamte Programm „in Teilen“ getestet werden. Die dazu gehörige Betrachtungsebene und entsprechende Modelle von großen Systemen und ihren Teilen werden in Kapitel 13.2 vorgestellt. Die verschiedenen Möglichkeiten, Teile

auszuwählen, und die Vor- und Nachteile werden in Kapitel 13.3 präsentiert. Außerdem ist zu klären, nach welchen Kriterien und mit welchen Methoden angemessene Testdaten für das funktionale Testen der Programmteile erzeugt werden sollen (siehe Kapitel 13.4 und 13.5). Abschließend werden einige Überlegungen zum System- und Abnahmetest sowie zu (Test-)Verfahren in der Installations- und Wartungsphase angestellt.

13.2 Modelle für große Systeme

Beim Testen und Überprüfen kleiner Programme kann man sich eine *mikroskopische* Betrachtungsweise leisten (s. Kapitel 4 bis 12). Als Bestandteile kommen Anweisungen und Daten in den Blick, die über den Kontrollfluß bzw. Datenfluß in Relation stehen. Würde man beim (Teil-)Systemtest großer Systeme diese mikroskopische Betrachtungsweise beibehalten, hätte man mit einer Unmenge von Anweisungen, Daten und somit Test- und Überprüfungsfällen zu tun und würde außerdem den Überblick verlieren.
Aus ökonomischen und Management-Gründen ist also eine *makroskopische* Betrachtungsweise großer Systeme nötig. Dabei werden Teile des Systems als Einheit mit ihren Relationen und Schnittstellen betrachtet. Je nach Größe des Systems und notwendigem Abstraktionsgrad bieten sich Operationen (Prozeduren oder Funktionen), Module oder noch größere Teile des Systems als Einheiten an.
Wählt man Operationen als Einheit, so stehen sie durch den *Aufruf* anderer Operationen in Relation und haben eine Schnittstelle, die aus Parametern und globalen Variablen besteht. Die Aufrufrelation läßt sich dabei z. B. durch den **Operationsaufrufgraphen** darstellen, bei dem die Operationen die Knoten des Graphen sind und eine Kante von P nach Q existiert g. d. w. Operation P die Operation Q aufruft[3]. Man beachte, daß dieser Graph Zyklen enthält, wenn eine Operation rekursiv ist bzw. es eine indirekte Rekursion über eine Kette von Operationsaufrufen gibt.
Wählt man Module als Einheit, so stehen sie wiederum durch den Aufruf von Operationen aus anderen Modulen in Relation, außerdem können auch Typdefinitionen anderer Module benutzt werden. Beide Beziehungen werden meist als **Benutzt-Beziehung** und die aufgerufenen Operationen als Teil der **Export-** bzw. **Import-Schnittstelle** der Module bezeichnet. Bei dieser Betrachtungsweise spielen also Beziehungen innerhalb eines Moduls keine Rolle. Die Benutzt-Beziehung zwischen Modulen kann — analog zur Definition des Operationsaufrufgraphen — wieder als Graph, genannt **Modulgraph**, dargestellt werden[4]. (Bei objektorientierten Syste-

[3] vgl. Kapitel 12.2.1, Nebenprodukte der Analysen.
Bei einer genaueren Betrachtung ist auch noch von Interesse, wie oft (an wieviel verschiedenen Programmstellen) Operation P die Operation Q aufruft.

[4] Die Kanten im Modulgraph sollten bei einer genaueren Betrachtung mit den Namen der verschiedenen aufgerufenen Operationen markiert werden.

men gehört zur Benutzt-Beziehung natürlich auch noch die Vererbungsbeziehung zwischen verschiedenen Objekten und Klassen.)

Beispiel 13.2.1 (Modulbeziehungen als Modulgraph)
Die Module IM1, IM2, IM3 und EX haben die in Abb. 13.1 angegebene Struktur (s. [Spi 95], S. 282).
Dann hat der Modulgraph mit *Kantenmarkierung das in Abb. 13.2 angegebene Aussehen.*

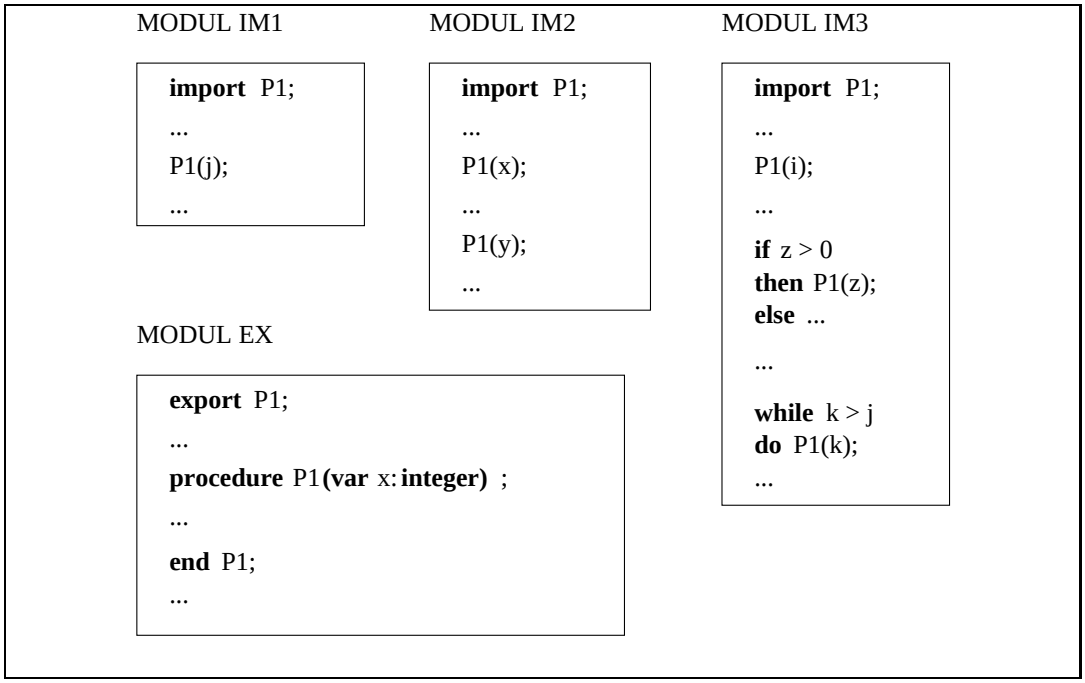


Abb. 13.1: Programmcode von vier Modulen (s. [Spi 95], Fig. 2)

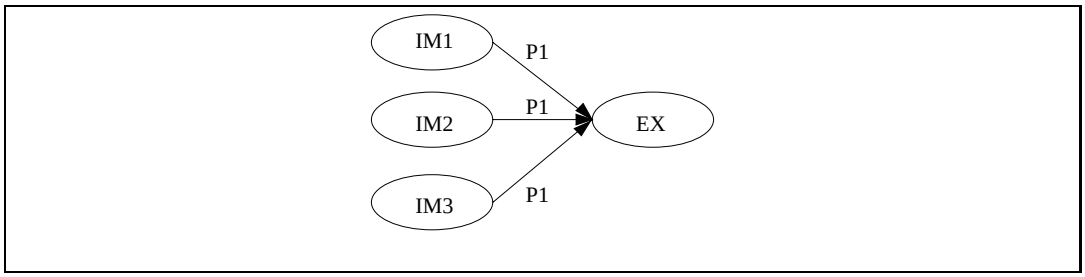


Abb. 13.2: Modulgraph zur Struktur von Abb. 13.1

Im folgenden wird stets davon ausgegangen, daß die Benutzt-Relation auf den Modulen eine Hierarchie bildet, d. h., daß keine Zyklen bestehen und daß es ein Modul an der Spitze der Hierarchie gibt, das von keinem anderen Modul benutzt wird. (Im

Falle von Zyklen müßten Operationen, die sich zyklisch aufrufen - z. B. A ruft B auf, B ruft C auf, C ruft A auf — in einem Modul zusammengefaßt werden bzw. mit Platzhaltern getestet werden. Im Beispiel müßte B durch einen Platzhalter ersetzt werden, wenn A getestet wird, aber auch A müßte durch einen Platzhalter ersetzt werden, wenn C getestet wird.)

BEISPIEL 13.2.2
Es gibt zwei Arten von Benutzt-Hierarchien:

1. *Die Benutzt-Hierarchie von Abbildung 13.3 a) ist ein Baum, d. h., jedes Modul wird nur von höchstens einem anderen Modul benutzt.*

2. *Die Benutzt-Hierarchie von Abbildung 13.3 b) ist kein Baum.*
   (In diesem Beispiel ist sie auch nicht in strengen Schichten [so daß Module aus Schicht $i+1$ nur von Modulen aus Schicht i benutzt werden] anzuordnen.)

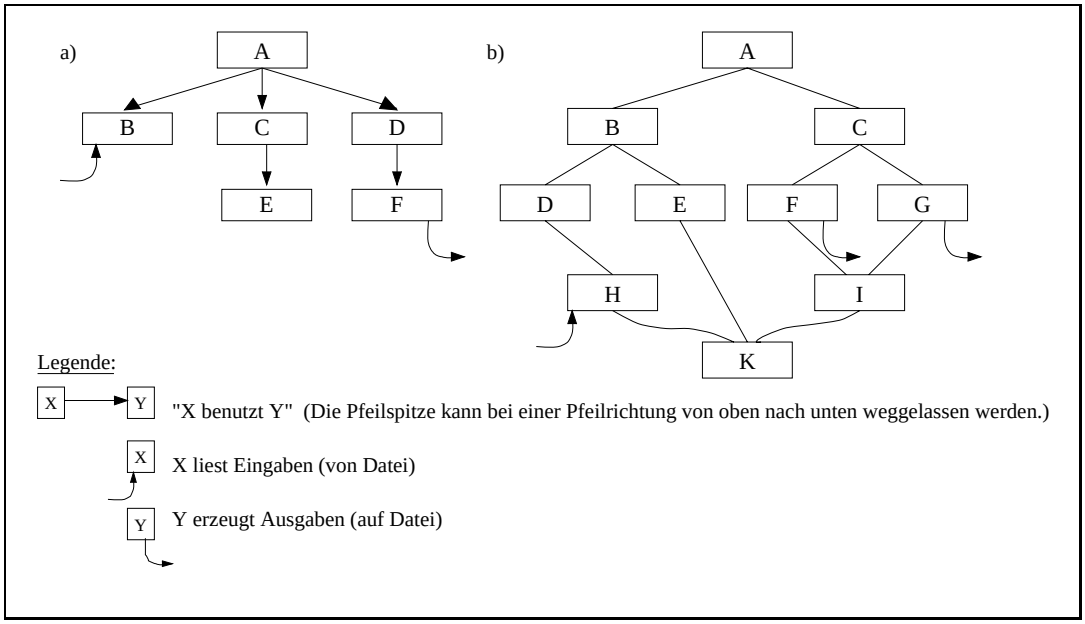


Abb. 13.3: Verschiedene Modulhierarchien

13.3 Strategien für den Modul- und Integrationstest

Im folgenden soll nun der Programmtest „in Teilen“ beschrieben werden, der oft unter den Schlagwörtern „Modultest“ und „Integrationstest“ beschrieben wird.

13.3.1 Modultest

Ein **Modultest** besteht im Testen eines einzelnen Moduls. Aufgabe des Modultests ist der Vergleich der realisierten Modulfunktionen und Datenbenutzungen mit der Modulspezifikation. (Letztere sollte als Ergebnis des Entwurfs vorliegen.) Das Vorgehen beim Modultest unterscheidet sich im Prinzip nicht vom Vorgehen beim Programmtest, d. h., es sind Testdaten nach passenden Kriterien (implementationsorientiert, spezifikationsorientiert oder fehlerorientiert) zu erstellen (siehe Teile II und III).

Es ergeben sich allerdings zwei neue Probleme bzw. Fragestellungen:

1. Wie wird das Modul mit Eingabedaten versorgt und wie wird es aktiviert?
   (Dies betrifft alle Module außer dem Modul an der Spitze[5].)
2. Was geschieht mit dem Aufruf bzw. der Benutzung „tiefer liegender“ Module der Benutzt-Hierarchie?
   (Dies betrifft alle Module außer den Modulen am unteren Ende der Hierarchie.)

Diese beiden Probleme werden durch das Konzept der Treiber und Platzhalter gelöst.

Ein **Treiber**(-Modul), welches das entsprechende Modul aufruft — und zwar mit den entsprechenden Argumenten (globale Variable, Parameter und evtl. Werte für E/A-Operationen) — löst das Problem 1.

Beispiel 13.3.1
Modul B sei:

procedure *UPDATE(x:* **integer**, *y:* **integer***);*
begin
⋮
end

mit Parametern x und y und Benutzung der globalen Variablen p vom Record-Typ Person *(bestehend aus den Komponenten* Nachname: **string**[30], Vorname: **string**[20] *und* Alter: **integer***).*
In keiner Programmiersprache ist die direkte Ein- und Ausgabe von Verbünden (Records) möglich. Die Komponenten müssen also einzeln eingelesen und zusammengesetzt werden. Ein Treiber für B hat dann beispielsweise folgendes Aussehen:

[5] Das Versorgen mit Eingabedaten, die von unteren Modulen geliefert werden, ist auch beim Modul an der Spitze der Hierarchie problematisch.

```
type
    Person  = record
                  Nachname : string[30];
                  Vorname : string[20];
                  Alter : integer;
              end;
var x1, y1 : integer;
    in-file, out-file: file;
    p: Person;
begin
    open(in-file, READ);
    read(in-file, p.Nachname, p.Vorname, p.Alter);
    read(in-file, x1, y1);
    close(in-file);
    UPDATE(x1, y1);
    open(out-file, WRITE);
    write(out-file, p.Nachname, p.Vorname, p.Alter);
    write(out-file, x1, y1);
    close(out-file);
end
```

Der Treiber von B kann also einen Aufruf von UPDATE nach dem Lesen der Werte aus einer Datei ausführen. Natürlich könnte man auch eine Schleife einbauen, so daß mehrere Aufrufe mit verschiedenen Werten für UPDATE möglich sind — bis zu einem „Abbruch-Zeichen".
Der Treiber kann i. allg. von einem Werkzeug erzeugt werden, da nur die Aufrufschnittstelle bekannt sein muß. Sie ergibt sich aus den formalen Parametern und den übergebenen globalen Variablen. Durch genügend intelligente statische Analyse (s. Kap. 12.2) können die globalen Variablen bestimmt werden, die formalen Parameter liegen syntaktisch vor.

Die Lösung von Problem 2 (Aufruf „tiefer liegender" Module) sind sogenannte **Platzhalter**, welche die zu benutzenden Module „simulieren" (und zum Modul hinzugefügt werden müssen).

Die Simulation kann verschieden komplex sein:

1. Der Platzhalter meldet nur, daß er aufgerufen wurde. (Das kann z. B. beim Testen der Dialogsteuerung reichen.)

2. Der Platzhalter liefert ein konstantes Ergebnis, das im erlaubten Wertebereich liegt. (Das kann manchmal reichen, verfälscht aber oft das Ergebnis.)

3. Der Platzhalter liefert korrekte Ausgaben für eine kleine Teilmenge der erlaubten Eingaben, die beim Testen benötigt wird.

Für jedes Testdatum des aufrufenden Moduls ist also zu spezifizieren:

- mit welchen Eingaben der Platzhalter aufgerufen wird,
- welche Ausgaben der Platzhalter dafür abliefern muß.

Der Platzhalter kann automatisch erzeugt werden, wenn die Eingabe-/Ausgabe-Paare pro Platzhalter und Testdatum des aufrufenden Moduls vorliegen (vgl. Treiberkonstruktion für Problem 1 oben).

Bei der Entscheidung über die Komplexität eines Platzhalters besteht folgendes Dilemma: Wählt man einen einfachen Platzhalter (Fälle 1 und 2), kann das Modul falsche Ergebisse liefern, obwohl es korrekt arbeitet — aber der Platzhalter falsche Werte liefert; entscheidet man sich für Fall 3, benötigt man meist für die Erfüllung einer gewünschten Überdeckung des Moduls (z. B. Zweigüberdeckung) viele Testdaten und daher auch viele Ein-/Ausgabe-Paare für den Platzhalter. Dann ist — insbesondere bei ungenauer Spezifikation des zu simulierenden Moduls — die Gefahr groß, daß falsche Ausgaben vom Platzhalter zurückgegeben werden und damit den Test des aufrufenden Moduls verfälschen. Daher verzichtet man beim Modultest meistens auf die Erfüllung einer Überdeckungsforderung, die umfangreiche Platzhalter erfordert, und muß mit einem groben Test zufrieden sein, der nur zeigt, daß „das Modul läuft“ und die Schnittstellen syntaktisch stimmen. Ein genauer Modultest muß demnach bei der Integration der Module nachgeholt werden.

13.3.2 Zweck des Integrationstests

Der Integrationstest eines Teilsystems von Programmeinheiten (im folgenden stets „Module“ genannt) kann verschiedenen Zwecken dienen:

1. Test der Schnittstellen zwischen den Modulen des Teilsystems,
2. vollständiger Test des Moduls, das in das bisherige Teilsystem integriert wird, gemäß eines angestrebten Überdeckungsmaßes.

Der zweite Zweck muß oft erfüllt werden, da das isolierte Testen einzelner Module (Modultest) selten vollständig möglich ist (s. Ende von Abschnitt 13.3.1). Ansonsten sollte sich der Integrationstest natürlich auf Zweck 1 konzentrieren.

Wie gut die Zwecke 1 und 2 erfüllt werden können, hängt von der Art und Weise, insbesondere der Reihenfolge ab, mit der die einzelnen Module in das Gesamtsystem integriert werden[6]. Dies wird in den folgenden Abschnitten behandelt.

[6] Hier wird eine zweistufige Hierarchie angenommen: Module als Teile des Gesamtsystems. Bei mehreren Stufen (z. B. Module, Komponenten, Teilsysteme, System) ist für *jede* Stufe die Art und Weise und Reihenfolge der Integration festzulegen.

13.3.3 Nichtinkrementeller Integrationstest

Im folgenden wird eine einfache Strategie des Integrationstests vorgestellt, die viel Parallelarbeit ermöglicht und die Schwierigkeiten des monolithischen Tests (siehe S. 344) vermeidet.

Strategie 1: nichtmonolithischer Test („Urknall"-Methode)

1. Teste alle Module einzeln.
2. Teste danach das gesamte Programm.

In diesem Falle treten bei Schritt 1 die Probleme des Modultests auf, d. h. es sind eine Reihe von Treibern und Platzhaltern zu erzeugen:

1. Für alle Module (außer dem Modul an der Spitze) müssen Treiber erstellt werden (also $m-1$ bei m Modulen).
2. Für alle Module (außer den „Boden"-Modulen) müssen Platzhalter erstellt werden, also $m-1$ bei Modulen in einer baumförmigen Hierarchie, da jedes Modul (außer der Spitze) einmal aufgerufen wird. Bei einer nicht baumförmigen Hierarchie sind es sogar mehr als $m-1$, nämlich $\sum_{i=1}^{m} a_i$ Platzhalter, wenn das i-te Modul a_i-mal (von verschiedenen Modulen) aufgerufen wird (wobei $a_j = 0$ für die Spitze j gilt). Diese Anzahl ist gerade die Anzahl k der Kanten in dem Graphen, der die Benutzthierarchie darstellt. (Für Bäume gilt $k = m-1$, siehe Platzhalteranzahl oben bei Fall 1.)

   Beispiel 13.3.2

   (a) *In Abb. 13.3 a) liegt eine Baumhierarchie mit $m = 6$ Modulen vor:*
   - *5 Treiber werden benötigt (für B, C, D, E, F).*
   - *5 Platzhalter werden benötigt (für A: B, C, D; für B: $\emptyset$; für C: E; für D: F; für E: $\emptyset$; für F: $\emptyset$); es gilt $a_A = 0$, $a_B = a_C = a_D = a_E = a_F = 1$.*

   (b) *In Abb. 13.3 b) liegt eine Hierarchie mit $m = 10$ Modulen vor, die keine Baumhierarchie ist. Es gilt $a_A = 0; a_B = a_C = a_D = a_E = a_F = a_G = a_H = 1; a_I = 2; a_K = 3$; also werden $\sum_i a_i = 12$ Platzhalter (mehr als $m - 1 = 9$) benötigt:*
   für A: B, C; für B: D, E; für C: F, G; für D: H; für E: K; für F: I; für G: I; für H: K; für I: K, für K: $\emptyset$.

   Für Module, die für verschiedene Aufrufe durch Platzhalter simuliert werden müssen, sind verschiedene Platzhalter zu erstellen, die jeweils nur die zu dem entsprechenden Aufruf gehörigen Eingabedaten durch passende Ausgabedaten (gemäß Spezifikation des Moduls) zu beantworten haben.

BEISPIEL 13.3.3
E, H und I aus Abbildung 13.3 b) brauchen verschiedene Platzhalter für K, da sie i. allg. K mit verschiedenen Testdaten aufrufen.

Man könnte natürlich einen „Super-Platzhalter“ erstellen, der alle Testdaten für alle verschiedenen Aufrufe enthält, aber bei Parallelarbeit müßten verschiedene Testpersonen diesen Platzhalter gemeinsam erstellen, was viel Kommunikation und Abstimmung erfordert. Vorteilhaft wäre nur, daß bei *verschiedenen* Aufrufen mit *demselben* Eingabedatum abweichende (fehlerhafte) Solldaten frühzeitig erkannt und bereinigt würden.

Vorteile des nichtmonolithischen Testens:

1. Zu Beginn der Testphase (beim separaten Modultest) gibt es Gelegenheit zu vielen parallelen Aktivitäten.
2. Da keine teilweise Zusammenfassung von Modulen getestet wird, wird weniger Testaufwand benötigt als beim inkrementellen Testen (s. folgenden Abschnitt 13.3.4).

Nachteile des nichtmonolithischen Testens:

1. Fehler an den Schnittstellen werden spät entdeckt.
2. Die Lokalisierung von Fehlern, die erst beim Test des gesamten Programms auftreten, ist schwierig.
3. Der Aufwand für Treiber und Platzhalter ist höher als beim inkrementellen Testen (siehe folgenden Abschnitt).
4. Wegen der Platzhalterprobleme ist ein vollständiger Modultest genauso schwierig wie beim separaten Modultest (s. Abschnitt 13.3.1).

Nachteil 1 könnte durch einen **separaten Schnittstellentest** vermieden werden, bei dem zu jeder einzelnen Schnittstelle nur die beiden beteiligten Module als Teilsystem getestet werden. Ähnlich wie beim nichtmonolithischen Test muß man aber dafür viele Treiber und Platzhalter erzeugen. (Bei Abbildung 13.3 a sind fünf Schnittstellen durch die Modulpaare AB, AC, AD, CE und DF zu testen, mit jeweils zwei Platzhaltern für die ersten drei Paare und je einem Treiber für die Paare CE und DF, insgesamt also mit sechs Platzhaltern und zwei Treibern.) Separate Schnittstellentests sind also nur sinnvoll, wenn Schnittstellenfehler vermutet werden, aber mit den folgenden Strategien nicht gefunden werden können.

13.3.4 Inkrementeller Integrationstest

Wegen des Überwiegens der Nachteile des nichtinkrementellen Testens sind Teststrategien aus der folgenden Strategieklasse zu wählen.

Strategieklasse 2: inkrementelles Testen

1. Teste zu Beginn (irgend)ein Modul X. Dann besteht das „Teilsystem“ genau aus Modul X.

2. Füge ein Modul M hinzu, für das gilt:

   (a) M benutzt keine anderen Module, *oder*

   (b) wenn M andere Module benutzt, gehört wenigstens eines dieser benutzten Module zum bisherigen „Teilsystem“, *oder*

   (c) wenn M von anderen Modulen benutzt wird, gehört wenigstens eines dieser (benutzenden) Module zum bisherigen „Teilsystem“.

   Das neue „Teilsystem“ besteht dann aus dem alten „Teilsystem“ und Modul M. Dieses „Teilsystem“ ist nun als Einheit zu testen.

3. Wiederhole Schritt 2 mit einem weiteren (ungetesteten) Modul usw. bis das „Teilsystem“ das ganze Programm ist.

Beispiel 13.3.4
Bei der Modulhierarchie aus Abbildung 13.3 a) kann das inkrementelle Testen z. B. mit Modul C beginnen. Anschließend können die Module B oder F (wegen Eigenschaft 2a), Modul A (wegen Eigenschaft 2b) oder Modul E (wegen Eigenschaft 2a und 2c) integriert werden.
Wird B als neues Modul gewählt, muß das unzusammenhängende Teilsystem aus B und C getestet werden (mit Treiber für B und C und Platzhalter für E). Zu diesem Teilsystem {B, C} können nun Modul F (wegen Eigenschaft 2a), Modul A (wegen Eigenschaft 2b) oder Modul E (wegen Eigenschaft 2a und 2c) integriert werden.
Nach Wahl von E kann das Teilsystem {B, C, E} ohne Platzhalter getestet werden. Anschließend kann nur Modul A (wegen Eigenschaft 2b) oder Modul F (wegen Eigenschaft 2a) integriert werden.
Wird als neues Teilsystem {A, B, C, E} gewählt, das nur mit einem Platzhalter für D getestet werden kann, sind anschließend Modul D oder F integrierbar. Nach Wahl von D kann das Teilsystem {A, B, C, D, E} mit einem Platzhalter für F getestet werden, der in einem letzten Integrationsschritt durch Modul F zu ersetzen ist.

Bei obigem Beispiel findet beim Übergang von Teilsystem {B, C} zum Teilsystem {B, C, E} ein *Abstieg* in der Hierarchie statt; beim Übergang nach {A, B, C, E} findet ein *Aufstieg* in der Hierarchie statt; anschließend wieder ein *Abstieg* über D nach F. Es liegt also ein Wechsel zwischen Abstieg und Aufstieg vor, sozusagen eine

Jo-Jo-Strategie. Eine solche Strategie mag in manchen Fällen angebracht sein, z. B. wenn das zuerst gewählte Modul besonders gründlich getestet werden soll, da von seinem korrektem Verhalten viel abhängt. Im allgemeinen ist aber eine reine Strategie — also nur absteigend oder nur aufsteigend — vorzuziehen, da sie besser zu steuern ist.
Es sind also folgende Spezialfälle des inkrementellen Testens zu betrachten:

Strategie 2.1: absteigender (top down) Test

1. Beginne mit dem Modul an der Spitze der Hierarchie als „Teilsystem“.
2. Füge jeweils ein Modul, das nur von Modulen des bisherigen Teilsystems direkt benutzt wird, zum Teilsystem hinzu (bis das „Teilsystem“ das ganze System ist).

Strategie 2.2: aufsteigender (bottom up) Test[7]

1. Beginne mit einem Modul am „Boden“ der Hierarchie als „Teilsystem“.
2. Füge jeweils ein Modul M zum „Teilsystem“ hinzu, für das alle Module, die M direkt benutzt, zum bisherigen „Teilsystem“ gehören (bis das „Teilsystem“ das ganze System ist).

Beispiel 13.3.5 (zu Strategie 2.1 und Strategie 2.2)
Bei Abb. 13.3 a) ist die Reihenfolge A, B, C, D, E, F und bei Abb. 13.3 b) die Reihenfolge A, B, C, D, E, F, G, H, I, K ein absteigender Test.
Bei Abb. 13.3 a) ist die Reihenfolge E, F, B, C, D, A und bei Abb. 13.3 b) die Reihenfolge K, H, I, G, D, F, E, B, C, A ein aufsteigender Test.

Bei beiden Strategien ist die *Auswahl* bei Punkt 2 relativ unbestimmt. Beim absteigenden (top down) Test soll dies am Beispiel von Abb. 13.3 b) diskutiert werden: Beim Testen von A mit den Platzhaltern für B und C müssen die Testdaten jeweils simuliert werden, und zwar muß der B-Platzhalter für jeden Aufruf das Einlesen von Dateien (durch H, weitergegeben von D) simulieren und die „interne“ Eingabe für Modul A abliefern. Diese „interne“ Eingabe ist oft schwer zu spezifizieren. Entsprechend problematisch ist die „interne“ Ausgabe von A an den Platzhalter C, der die von F und G zu erzeugende Ausgabe des Gesamtsystems simulieren muß.

Daher sollte bei Schritt 2 gelten:
Wähle beim absteigenden[8] Test neue Module so, daß die Ein- und Ausgabe des Gesamtsystems möglichst frühzeitig zum Teilsystem gehört, wobei die Eingabe vor

[7] Aufsteigendes Testen bedeutet nicht aufsteigendes (bottom up) Entwerfen. Vielmehr verträgt es sich mit absteigendem (top down) Entwurf. Aufsteigendes Testen verhindert allerdings das Überlappen von Entwurf, Codierung und Testen (vgl. ambivalente Eigenschaft 1 beim absteigenden Testen, s. unten).

[8] Beim aufsteigenden Test ist dies nicht so wichtig, da in der Hierarchie „parallel“ liegende Module (z. B. E neben dem Teilsystem {C, F, G, I, K} bei Abbildung 13.3b) davon nicht profitieren.

der Ausgabe dazugebunden werden sollte. Entsprechendes gilt beim absteigenden und auch beim aufsteigenden Test für „kritische“ Module, d. h. Module, von deren Leistung das Gesamtsystem entscheidend abhängt.

Beispiel 13.3.6 (Abb. 13.3 b mit einem „kritischen“ Modul F)

- *absteigende Reihenfolge (z. B.): A, B, D, H, C, F, G, E, I, K*
- *aufsteigende Reihenfolge (z. B.): K, H, I, G, F, E, D, C, B, A.*

Die Strategien 2.1 und 2.2 sind sequentiell formuliert, sie können aber parallelisiert werden:

1. Beim aufsteigenden Test kann oft parallel gearbeitet werden.

   Beispiel 13.3.7 *(zu Abb. 13.3* a)
   {C, E} und {D, F} können parallel getestet werden.

2. Beim absteigenden Test kann bewußt von Strategie 2.1 abgewichen und parallel gearbeitet werden.

   Beispiel 13.3.8 *(zu Abb. 13.3 b)*
   Hier kann, statt nacheinander A mit B und dann C zu verbinden, parallel A mit B und A mit C integriert werden (und danach A mit B und C).

Im folgenden werden die Vor- und Nachteile des inkrementellen Testens und speziell des absteigenden und aufsteigenden Testens diskutiert.

Vorteile des inkrementellen Testens (allgemein):

1. Aufwand:
   Bei einer absteigenden Strategie braucht man keine Treiber, aber nur so viele Platzhalter wie beim nichtmonolithischen Testen (beim Erweitern des „Teilsystems“ können nichttangierte Platzhalter beibehalten werden). Bei einer aufsteigenden Strategie braucht man keine Platzhalter, aber nur soviele Treiber wie beim nichtmonolithischen Testen.

   Beispiel 13.3.9

   (a) *absteigender (top down) Test für Abb. 13.3* a*):*

   Platzhalter: B, C und D für den Test von A,
   E (wenn C dazu kommt),
   F (wenn D dazu kommt)
   (also $6 - 1 = 5$ Platzhalter)

(b) *aufsteigender (bottom up) Test für Abb. 13.3 b);*
z. B. Reihenfolge K, H, I, G, F, D, E, B, C, A:

Treiber: für K, für H (mit K), für I (mit K), für G (mit I und K),
für F (mit I und K), für D (mit H und K), für E (mit K),
für B (mit D, E, H und K), für C (mit F, G, I und K),
also $10 - 1 = 9$ *Treiber.*

2. Schnittstellenfehler werden bei jedem „Dazubinden“ eines neuen Moduls (also früh) entdeckt.

3. Die Fehlerlokalisierung ist einfacher.
(Vermutlich liegen die Fehler im neu dazugenommenen Modul oder sie haben mit der Schnittstelle zwischen dem neuen Modul und dem bisherigen Teilsystem zu tun.)

4. Die zuerst ausgewählten Module werden mit zusätzlichen Testdaten — also mit einem gründlicheren Modultest — getestet.

Nachteile des inkrementellen Testens (allgemein):

1. Der Testaufwand ist höher als beim nichtinkrementellen Testen.

2. Bei den Schritten 2 und 3 der Strategieklasse 2 (s. oben), dem sogenannten **(inkrementellen) Integrieren**, kann nur sehr wenig parallel gearbeitet werden.

Fazit für das inkrementelle Testen:
Nachteil 1 ist leicht zu verschmerzen (z. B. im Vergleich zu Vorteil 3 und 4). Nachteil 2 kann in Kauf genommen werden oder durch modifizierte Strategien abgemildert werden (genaueres siehe Übungen 13.2 bis 13.4). Daher ist i. allg. eine inkrementelle Strategie zu verfolgen.

Spezielle Vor- und Nachteile des absteigenden Tests und des aufsteigenden Tests sollen im folgenden betrachtet werden:

Vorteile des absteigenden Tests:

1. Fehler bzw. Probleme in den oberen Modulen der Hierarchie, insbesondere Entwurfsprobleme, werden frühzeitig und häufig aufgedeckt.

2. Die Darstellung von Testdaten ist nach der Integration der E/A-Funktionen einfacher, d. h. Testdaten haben dann die Form der Systemein- und -ausgaben. Dies ist *frühzeitig* möglich, wenn die E/A-Funktionen „oben“ in der Modulhierarchie angeordnet sind.

3. Bei verzahntem absteigenden Codieren und Testen steht für Vorführungen jederzeit ein Programmskelett zur Verfügung; das beseitigt die Ungewißheit, ob das System „läuft“[9].

Nachteile des absteigenden Tests:

1. (Komplizierte) Platzhalter müssen erstellt werden.
2. Die Darstellung von Testdaten ist vor der E/A-Integration schwierig, also in vielen Fällen, wenn die E/A-Funktionen „unten“ in der Modulhierarchie angeordnet sind — was bei einem guten Entwurf oft der Fall ist.
3. Die Erzeugung von geeigneten Testdaten für den Modultest unterer Module kann unmöglich oder schwierig sein.

   Beispiel 13.3.10

   (a) *Unmöglichkeit der Testdatenerzeugung:*
   Ein unteres Modul berechnet die Lösungen der Gleichung $ax^2 + bx + c = 0$ *(in* x*). Dieses Modul möge auch die komplexen Lösungen berechnen (wenn es keine reellwertigen gibt). Dieses Modul wird aber so benutzt, daß stets nur reellwertige Lösungen vorkommen.*

   (b) *Schwierigkeit der Testdatenerzeugung:*
   Bei Abb. 13.3 b) sei E neu hinzugebunden zum Teilsystem $\{A, B, D\}$*. Man muß nun ein Testdatum erzeugen, das einen gewünschten Test von E (zum Beispiel für einen speziellen Zweig) ausführt. Da die Systemeingabe bei H erfolgt, muß das Testdatum Eingabewerte enthalten, die in Modul D einen „Aufruf“ des Platzhalters für H mit geeigneten Eingabe- und Ausgabe-Werten bewirken. Die Ausgabe des Platzhalters H wird anschließend — mehrfach transformiert durch D und B (und evtl. auch A) — an E übergeben. Daher ist es schwierig, das geeignete Testdatum zu bestimmen.*

4. Die Beobachtung der Testausgabe ist (vor dem Einbinden der Ausgabefunktionen) schwieriger.

   Beispiel 13.3.11
   Zur Beurteilung, ob in Beispiel 13.3.10 b) das Testdatum für E richtige Ergebnisse liefert, kann nur die „interne“ Ausgabe von A an C (bei Hinzunahme von C, F und G allerdings die konkrete Systemausgabe) beobachtet werden. Aus dieser Ausgabe muß (über die Kette A, B, E) rückwärts geschlossen werden auf die „interne“ Ausgabe von E.[10]

[9] Als Nebeneffekt wirkt sich dies auch positiv auf die Motivation des Entwicklungspersonals aus.

[10] Die Ausgabe von E ist leicht zu beobachten, wenn die Werte durch zusätzlich in E eingebaute Druckanweisungen ausgegeben werden. Dann besteht aber die Gefahr der Verfälschung des Verhaltens von E, da nicht E, sondern ein modifiziertes E getestet wird.

5. Der vollständige Modultest „oberer“ Module, z. B. bezüglich Zweigüberdeckung, wird oft verschoben, da es wegen der Nachteile 1 und 2 schwierig ist, geeignete Testdaten zu erzeugen.

Ambivalente Eigenschaften des absteigenden Tests:

1. Entwurf und Programmierung (samt Test) können „verzahnt“ (fast parallel) ausgeführt werden.
   Dies ist nachteilig, wenn beim Codieren unterer Module festgestellt wird, daß der Entwurf oberer Module geändert werden muß, bzw. dies wünschenswert wäre. Der erbrachte Codier- und Testaufwand unterstützt das Beharren auf dem schlechten Entwurf und ein Ändern der unteren Module zur Behebung des Entwurfsfehlers „oben“. Damit handelt man sich für spätere Zeiten Probleme beim Verstehen und Ändern des Systems ein.
   Eine kontrollierte Verzahnung von Entwurf und Programmierung (samt Test) in der Form des **Prototyping** ist allerdings sinnvoll, um den späteren Benutzern frühzeitig ein getestetes Teilsystem vorführen zu können. Damit können Abweichungen zwischen Spezifikation und Entwurf einerseits und den eigentlichen Anforderungen andererseits frühzeitig festgestellt werden. Außerdem können grobe Entwurfsfehler frühzeitig aufgedeckt werden.

2. Module werden nur unter den Bedingungen getestet, unter denen sie auch im Gesamtsystem benutzt werden.

   Beispiel 13.3.12
   Das Gleichungslösungs-Modul aus Beispiel 13.3.10 (a) braucht keine komplexen Lösungen zu erzeugen bzw. muß dafür nicht getestet werden, da es dies in der vorliegenden Systemumgebung nicht leisten muß.
   Wird das Modul aber später in anderen Programmen wiederverwendet (sinnvolle Einsparung von Aufwand), gilt es evtl.[11] *fälschlicherweise als „getestet“ und „im Einsatz bewährt“, obwohl es Fehler in Zweigen enthalten kann, die nie getestet wurden.*

Vorteile des aufsteigenden Tests:

1. Größere Entwurfs- und Codier-Mängel bei den „unteren“ Modulen werden frühzeitig aufgedeckt.

2. Testeingaben zur Ausführung von bestimmten Anweisungen oder (Teil-)Wegen des zuletzt dazugebundenen (neuen) Moduls sind leichter zu erzeugen, da der Treiber dieses Modul direkt aufruft.

3. Das Überprüfen der Testergebnisse ist leichter (vgl. 2.).

[11] bei einem fehlerhaften Informationsfluß im Wiederverwendungs-Management

Nachteile des aufsteigenden Tests:

1. Treibermodule müssen erzeugt werden.
2. Bei einer Verzahnung von Codierung und Test existiert ein Programmskelett erst, wenn das letzte Modul (das Spitzenmodul) dazugebunden wurde.
3. Fehler bzw. Probleme in den oberen Modulen der Hierarchie, insbesondere Entwurfsprobleme, werden erst sehr spät aufgedeckt (vgl. Vorteil 1 beim absteigenden Testen).

Fazit der Beurteilung der Strategien:
Das inkrementelle Testen ist bei großen Systemen die beste Strategie, nichtmonolithisches Testen ist dagegen die schlechteste Strategie (abgesehen vom monolithischen [„Alles-auf-einmal"] Testen[12]). Da sowohl das (inkrementelle) absteigende Testen als auch das (inkrementelle) aufsteigende Testen gravierende Nachteile hat, sollten beide Strategien kombiniert werden, um die jeweiligen Vorteile zu erhalten und die Nachteile abzumildern. Als schwache Form der Kombination bietet sich das Sandwich-Testen an (siehe Übung 13.3). Sollen aber sowohl Entwurfsfehler, Schnittstellenfehler als auch Modulfehler mit großer Wahrscheinlichkeit gefunden werden, sind beide inkrementellen Strategien vollständig anzuwenden.

Will man aus Kostengründen nur *eine* inkrementelle Strategie anwenden, hängt die Auswahlentscheidung davon ab, wo die problematischen Module vermutet werden: oben oder unten in der Hierarchie (siehe Vorteil 1 des absteigenden Testens und Vorteil 1 des aufsteigenden Testens). Daher müssen im konkreten Einzelfall die Vor- und Nachteile entsprechend gewichtet werden. Alle generellen Aussagen (z. B. die, daß aufsteigendes Testen immer besser ist als absteigendes Testen) sind somit vermutlich falsch. Zur Frage „Wann ist welche Strategie am besten?" sind daher Untersuchungen und Auswertungen von kontrollierten Experimenten bzw. Projekten notwendig, was allerdings eine aufwendige Sache ist[13].

[12] siehe S. 344

[13] Ein Beispiel ist eine Untersuchung von Rowland/Zuyuan, die für künstliche Modulhierarchien folgendes ergeben hat: Für Hierarchien mit fünf Ebenen findet ein aufsteigender Test, bei dem nach jedem Teilsystemtest die gefundenen Fehler korrigiert werden, etwa doppelt so viele Fehler wie ein monolithischer Test mit 10 Testläufen mit je 200 zufällig erzeugten Testdaten, bei dem nach jedem Testlauf die Fehler entfernt werden. Für Modulhierarchien mit nur vier Ebenen sind die Unterschiede zwischen den beiden Teststrategien dagegen nicht signifikant (s. [RoZ 89]).

13.4 Aufwand, Fehlerarten und Voraussetzungen des Integrationstests

In diesem und dem folgenden Unterkapitel werden die Methoden (Kriterien und Verfahren) vorgestellt, mit denen der Integrationstest durchgeführt werden sollte. Beim (Einzel-)Modultest sind dies die Methoden von Teil II und III und Kapitel 12 dieses Buches (vgl. Abschnitt 13.3.1). Für den Integrationstest könnte man dieselben Methoden auf das ganze System anwenden, was sich aber aus Aufwandsgründen verbietet (s. Abschnitt 13.4.1). Wendet man die Methoden nur separat auf die einzelnen Module an, werden spezielle Integrations- und Schnittstellenfehler nicht erkannt, die erst bei der Kombination der Module sichtbar werden (s. Abschnitt 13.4.2). Daher sind — unter gewissen Voraussetzungen (s. Abschnitt 13.4.3) — neue Methoden für die Entdeckung von Integrations- und Schnittstellenfehlern anzuwenden sowie Methoden, die spezielle Berechnungsfälle eines Moduls testen, die wegen der Platzhalterproblematik beim Modultest nicht ausgeführt wurden (vgl. Abschnitt 13.3.2, Zweck 2 auf Seite 350).

13.4.1 Aufwand

Durch die Kombination von Modulen ergibt sich beim Integrationstest ein Aufwandsproblem, das mit folgendem Beispiel erläutern werden soll.

Beispiel 13.4.1 (Aufwand beim Pfadtesten)
Angenommen Modul A benutzt die Operationen B und C, wobei A die Operationen nacheinander aufruft und die Kontrollflußgraphen von B und C vier bzw. sechs Wege enthalten (siehe Abb. 13.4). Dann hat der Kontrollflußgraph dieses sequentiellen Aufrufs — ohne Verfeinerung der Aufrufe „call B“ und „call C“ — nur einen Weg (die Sequenz „call B; call C“), mit Verfeinerung der Aufrufe enthält er dagegen $4 * 6 = 24$ *Wege. Beim Pfadtesten von A, B und C (getrennt für sich) benötigt man nur* $1 + 4 + 6 = 11$ *Tests. Beim Integrationstest von A, B und C würde beim Kriterium „alle Wege testen“ ein großer Aufwand (4 * 6 = 24 Tests) entstehen, wenn man annimmt, daß alle Wegekombinationen von B und C ausführbar sind.*

Allgemein erhält man bei der sequentiellen Verknüpfung von m Operationen folgenden Aufwand, wenn der Kontrollflußgraph jeder Operation jeweils n Wege enthält:

1. n^m Tests beim Test aller Wegekombinationen (Pfadüberdeckung) für den verfeinerten Kontrollflußgraphen,
2. $m * n + 1$ Tests bei der Pfadüberdeckung der einzelnen Operationen (ohne Verfeinerung).

Es ist klar, daß n^m Tests bei großem m und schon bei mittelgroßem n praktisch nicht durchführbar sind.

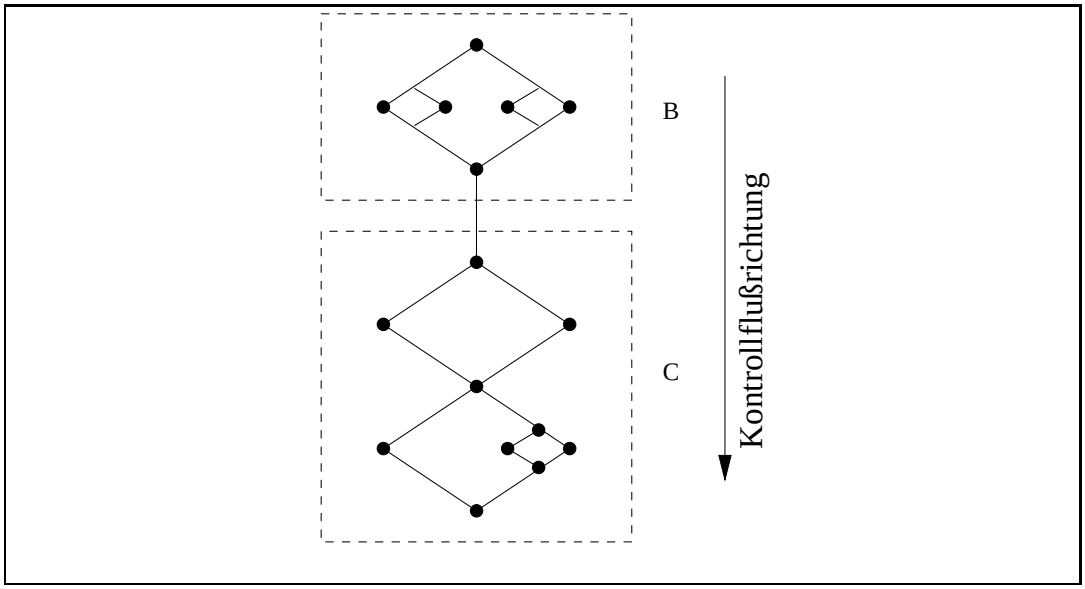


Abb. 13.4: Kontrollflußgraph der sequentiellen Verknüpfung von B und C

13.4.2 Fehlerarten beim Integrationstest

Durch den Integrationstest sollen Abweichungen der realisierten Modulstruktur und Funktionalität eines Teilsystems von Entwurf und Spezifikation dieses Teilsystems aufgedeckt werden. Dies kann sich auf den Kontroll- und Datenfluß des Teilsystems beziehen, insbesondere auf Reihenfolgebedingungen, Inhalt und Form der Schnittstellen und (falsche) Annahmen von Modulen über die Funktionalität anderer Module.
Als **Schnittstellenfehler** sind zu beachten:

- syntaktische Fehler (falsche Parameteranzahl, falsche Parametertypen),
- semantische Fehler (vertauschte Parameter gleichen bzw. verträglichen Typs),
- Aufruf mit undefinierten Parameterwerten (bei Eingabeparametern).

Eine falsche Annahme eines Moduls über die Funktionalität eines Sortiermoduls kann z. B. darin bestehen, daß aufsteigende Sortierung erwartet wird, aber absteigende Sortierung realisiert ist.

Falsche Werte von Schnittstellenvariablen stellen ebenfalls ein Problem dar, wenn entsprechende Fälle beim Modultest noch nicht getestet wurden. Beispielsweise kann durch einen Berechnungsfehler[14] in einem Modul ein Bereichsfehler in der aufgerufenen Operation (genannt: **Integrationszeit-Bereichsfehler**) entstehen.

[14] siehe Definition 7.2.1 auf S. 195

BEISPIEL 13.4.2 (INTEGRATIONSZEIT-BEREICHSFEHLER)

Modul M:

```
...
read (i);
c := i;
P (c, a);
writeln (a);
...
```

Prozedur P(mc, ma):

```
...
if mc < 4
then ma := 1
else ma := 2;
...
```

Dabei sind mc und ma die „formalen“ Parameternamen in P, die den „aktuellen“ Parameternamen c und a beim Aufruf durch M entsprechen. Der Wert von c = mc werde nur in P aber nicht später in M benutzt.

Ein angenommener Berechnungsfehler in M ($c := i$ ist falsch, $c := i + 1$ ist richtig) wird zu einem Bereichsfehler im integrierten Programm, wobei das korrekte Arbeiten von P vorausgesetzt sei: Für Werte von i mit $3 \leq i < 4$ ergeben sich Werte von c und mc mit $mc < 4$ (im Fehlerfall) und $mc \geq 3 + 1 = 4$ (wie es korrekt sein müßte), d. h. fälschlicherweise wird der then-Zweig und nicht der else-Zweig in P ausgeführt und der falsche ma-Wert wird als falscher a-Wert (1 statt 2) in M ausgegeben.
Durch Betrachten von M allein ist nicht festzustellen, daß der kleine Bereich $3 \leq i < 4$ ein „fehleraufdeckender“ Teilbereich[15] für den Fehler im Eingabeparameter c der Prozedur P ist.

Im Beispiel wird also ein Eingabeparameter einer aufgerufenen Operation vor dem Aufruf falsch berechnet, aber nur in einigen Fällen gibt die aufgerufene Operation einen falschen Ausgabewert zurück, in anderen Fällen bleibt der Fehler unentdeckt, da er sich nicht auswirkt.

13.4.3 Voraussetzungen eines erfolgreichen Integrationstests

Damit ein Integrationstest erfolgreich sein kann, müssen folgende Anforderungen an den Entwurf und die Strukturierung des gesamten Software-Systems erfüllt sein[16],

[15] genaueres dazu siehe [WeO 80] und Kapitel 16.3

[16] Die Anforderungen 1 und 2 sind widersprüchlich: Um Forderung 2 zu erfüllen, braucht man evtl. viele kleine Module, zwischen denen viele Kopplungen bestehen. Um dagegen Forderung 1 zu erfüllen, braucht man evtl. relativ große, umfassende Module, die keinen hohen inneren Zusammenhang aufweisen.

die nur durch konstruktive Qualitätsmanagementmaßnahmen (s. Abschnitt 3.2.2) zu erreichen sind:

1. Module mit wenigen Kopplungen an andere Module
   (Sonst sind zu viele Schnittstellen zu überprüfen.)

2. Module mit hohem inneren Zusammenhang
   (Sonst gibt es pro Schnittstelle zu viele Parameter.)

3. Keine (bzw. sparsame) Verwendung von globalen Variablen
   (Sonst sind die Schnittstellen zu komplex oder nicht übersichtlich.)

4. Kein Zugriff auf andere Module über mehrere Schichten bzw. durch indirekte Rekursion
   (Sonst gibt es komplexe Aufrufreihenfolgen für Operationen.)

Außerdem muß natürlich klar sein, gegen was zu prüfen bzw. zu testen ist, d. h. eine klare Spezifikation der Module und ihrer Beziehungen muß in Form von Funktions- und Schnittstellenbeschreibungen sowie Aufrufbedingungen für Folgen von Operationsaufrufen vorliegen.

13.5 Testmethoden für den Integrationstest

Wie beim Testen von Modulen und kleinen Programmen können statische und dynamische Methoden angewandt werden. Statische Verfahren haben auch beim Integrationstest den Vorteil, daß die Ermittlung und Ausführung von Testfällen und Testdaten entfällt (vgl. Kapitel 12). Andererseits können einige Fehler besser oder ausschließlich mit dynamischen Verfahren ermittelt werden. Diese Verfahren setzen meist voraus, daß gewisse Informationen oder Modelle (z. B. die Datenflußbeziehungen oder der Modulgraph) vorab statisch ermittelt werden.

In den folgenden Abschnitten 13.5.1 und 13.5.2 werden die verschiedenen statischen und dynamischen Integrationstestmethoden erläutert. Sie sind meistens Varianten der Verfahren aus Kapitel 4 bis 12, die für den Integrationstest modifiziert wurden. Die Methoden werden im folgenden für den intermodularen Integrationstest formuliert, d. h. Module werden als kleinste Einheiten betrachtet. Die Methoden können aber auch für den Test des Zusammenspiels von Operationen *innerhalb* eines Moduls verwendet werden, wenn die entsprechenden Modelle (z. B. Operationsaufrufgraph anstelle des Modulgraphs) verwendet werden.

13.5.1 Statische Integrationstestmethoden

Bei der statischen Überprüfung kommen verschiedene Verfahren in Frage:

- Syntaxprüfung der Schnittstellen (durch Übersetzer bzw. Binder)
  Damit können grobe Fehler wie falsche Parameteranzahl oder -typen bei streng getypten Programmiersprachen erkannt werden.

- Vergleich der (realisierten) Modulkopplungen mit dem Entwurf (d. h. den geplanten Modulkopplungen)
  Die Abweichungen können gewollt oder unabsichtlich — also vermutlich fehlerhaft — sein; das ist zu überprüfen, und zwar mit den Techniken von Kap. 12.1 und 12.2.

- Anzeige verdeckter Abhängigkeiten
  Verwaltet und exportiert z. B. ein Modul A eine (globale) Variable, die von zwei anderen Modulen B und C verwendet wird, so besteht zwischen B und C eine Abhängigkeit, die aus dem Programmtext nicht direkt zu entnehmen ist.

- Intermodulare Datenflußanalyse
  Bei dieser Vorgehensweise werden Anomalien im Datenfluß untersucht (siehe Abschnitt 12.2.2). Beim Integrationstest wird speziell der Datenfluß bei der Übergabe und Verarbeitung von **Schnittstellenvariablen** betrachtet (das sind die Parameter und globalen Variablen).

  Beispiel 13.5.1 (dd-Anomalie bei der Integration)
  Es seien die Module M1 und M2 wie folgt definiert:

```
M1: export P;
    ⋮
    procedure P(var out: int);
        begin
        ⋮
            out := result;
        end P;
M2: import P;
    ⋮
    begin
        ⋮
        P(outcome);
        outcome := x;
        ⋮
```

Beim Zusammenspiel von M1 und M2 entsteht eine dd-Anomalie, da die letzte Aktion in Prozedur P von M1 eine Wertzuweisung an den Ausgabeparameter ist (define) und die erste Aktion in M2 nach dem Aufruf der Prozedur P mit Ausgabeparameter out bzw. outcome wiederum eine Wertzuweisung (define) an outcome ist. Also liegt eine dd-Anomalie bei outcome vor.
Die Anomalie kann in diesem Fall durch getrennte Analyse von P und M2 allein gefunden werden: Die Analyse von P ergibt, daß out als Ausgabeparameter von P spezifiziert ist, der auch stets durch P einen Wert zugewiesen bekommt; die Analyse von M2 ergibt dann (unter der Annahme, daß P stets out bzw. outcome definiert) eine doppelte Definition von outcome.

Im allgemeinen Fall (wenn die Operationsparameter nicht genau als Eingabe- oder Ausgabeparameter deklariert sind) muß eine kombinierte Analyse der beiden Module bzw. Operationen erfolgen. Für einen Operationsaufruf bedeutet dies:

1. Im aufrufenden Modul müssen alle möglichen *letzten* Aktionen bzw. Zustände der Schnittstellenvariablen *vor* dem Aufruf ermittelt werden (*define*, *reference* oder *undefine*).

2. In der aufgerufenen Operation müssen alle möglichen *ersten* Aktionen der Schnittstellenvariablen ermittelt werden (*define*, *reference* oder *undefine*).

3. Kann als letzte Aktion vor dem Aufruf ein *define* vorkommen, liegt eine (potentielle[17]) *dd*-Anomalie vor. (Entsprechendes gilt für die *du*- und *ur*-Anomalie.)

Für die Rückkehr von einem Operationsaufruf sind analog zu obigem Vorgehen die möglichen *letzten* Aktionen in der Operation und die möglichen *ersten* Aktionen im aufrufenden Modul *nach* dem Operationsaufruf zu ermitteln. Durch Kombination ergeben sich wieder die (potentiellen) Datenflußanomalien.
Problematisch sind bei dieser Analyse globale Variablen, die nicht in der deklarierten Schnittstelle als Parameter vorkommen, da sie bei jedem Operationsaufruf eine Rolle spielen können. Diese Variablen sind also vorab für jede Operation zu bestimmen: es sind diejenigen Variablen, die nicht als Parameter oder lokale Variable deklariert sind, aber dennoch in der Operation benutzt werden.
Eine besondere Behandlung erfordern Werte, die bei einem Operationsaufruf unverändert durchgereicht werden. In diesem Fall ist der Kontrollfluß bis zum nächsten Operationsaufruf, bei dem der Wert verwendet wird, weiter zu verfolgen (evtl. über mehrere Aufrufebenen hinweg).

[17] Da nicht jeder Weg, für den eine Anomalie festgestellt wird, ausführbar ist, sind dies nur potentielle Anomalien, deren Ausführbarkeit durch symbolische Berechnung (vgl. Kapitel 12.3) oder durch Testen festgestellt werden muß.

13.5.2 Dynamische Integrationstestmethoden

Beim dynamischen Testen kann man den ablaufbezogenen (d. h. kontrollfluß- oder datenflußbasierten), den wertbezogenen und den funktionsbezogenen Integrationstest unterscheiden (vgl. Teil II und III des Buches).

13.5.2.1 Ablaufbezogener Integrationstest

Der ablaufbezogene Integrationstest wird danach differenziert, ob er auf dem Kontrollfluß oder dem Datenfluß aufsetzt.
Bei einer kontrollflußbasierten Vorgehensweise, die sich am Modulgraph (s. Kapitel 13.2) — bzw. Verfeinerungen oder Ergänzungen davon — orientiert, können folgende Beziehungen zwischen Modulen getestet werden:

Definition 13.5.2.1 (Kontrollflusskriterien)

1. **alle Module:** *jedes Modul muß mindestens einmal bei einem Test aufgerufen werden;*
2. **alle Relationen:** *jedes Modul muß mindestens einmal von jedem Modul (von dem es überhaupt aufgerufen wird) bei einem Test aufgerufen werden (dies entspricht der Ausführung jeder Kante im Modulgraph);*
3. **alle Relationen mehrfach:** *zu jeder Operation P, die von einem anderen Modul M aufgerufen wird, muß es einen Test geben, bei dem M Operation P mindestens einmal aufruft (dies entspricht der Ausführung jeder Kante im Modulgraph mit jedem als Markierung angegebenen Aufruf, vgl. Fußnote 4 in Kapitel 13.2);*
4. **alle Importe mehrfach:** *zu jeder Operation P, die von einem anderen Modul M aufgerufen wird, muß es Tests geben, bei denen jede Aufrufstelle von P in M mindestens einmal ausgeführt wird (dies erfordert eine Ergänzung des Modulgraphen aus Kapitel 13.2 um die Lage und Anzahl der Aufrufstellen von externen Operationen in den Modulen);*
5. **alle Aufrufreihenfolgen:** *jede mögliche Reihenfolge von Operationsaufrufen über Modulgrenzen hinweg ist bei einem Test auszuführen (d. h. alle Wege im Modulgraph sind auszuführen).*

   Beispiel 13.5.2.1 (Kontrollflusskriterien)
   Betrachtet man die Modulstruktur aus Abbildung 13.1 und 13.2 nur aus Sicht von Modul EX, so gilt folgendes: Kriterium „alle Module“ ist erfüllt, wenn die Prozedur P1 von IM1, IM2 oder IM3 aufgerufen wird; Kriterium „alle Relationen“ erfordert dagegen, daß IM1, IM2 und IM3 Prozedur P1 aufrufen; gleiches gilt für „alle Relationen mehrfach“, da EX nur eine Prozedur exportiert; um „alle Importe mehrfach“ zu erfüllen, muß in IM2 sowohl der Aufruf P1(x) als auch P1(y)

erfolgen und in IM3 müssen alle drei Aufrufe P1(i), P1(z) und P1(k) ausgeführt werden, d. h. bei der Verzweigung muß $z > 0$ gelten und die while-Schleife muß mindestens einmal (mit $k > j$) durchlaufen werden.

Die in Definition 13.5.2.1 genannten Kriterien sind nach ansteigenden Anforderungen angeordnet. Die ersten beiden Kriterien sind ziemlich schwach, das fünfte Kriterium verlangt evtl. unendlich viele Tests, falls Zyklen im Modulgraphen existieren. In diesem Fall sind die Wege zu begrenzen — mit analogen Techniken wie beim kontrollflußbezogenen Programmtesten (s. Abschnitt 7.2.4, Schleifenüberdeckung). Alternativ dazu kann eine spezifikationsbezogene Auswahl von Aufrufreihenfolgen erfolgen, um Fehler aufzudecken, die durch eine falsche Reihenfolge von Operationsaufrufen entstehen. Folgendes ist dafür zu ermitteln und zu testen:

1. Alle (graphentheoretisch) möglichen Reihenfolgen von Operationsaufrufen sind zu ermitteln (anhand der Kontrollflußgraphen, Operationsaufrufgraphen und Modulgraphen).

2. Nachdem alle Reihenfolgen ermittelt wurden, sind zumindest alle laut Spezifikation *verbotenen*, aber ausführbaren Reihenfolgen festzustellen. Falls die zulässigen Reihenfolgen durch einen Pfadausdruck P spezifiziert sind, können dazu (analog zum Vorgehen in Kapitel 5.1) die Folgen betrachtet werden, die im zugehörigen Automaten $A_v(P)$ in den Fehlerzustand führen. Ob sie in der Implementation ausführbar sind, kann durch Ausprobieren (Testen) oder durch symbolisches Ausrechnen festgestellt werden (siehe Kapitel 12.3).

Obwohl die beiden letzten Kriterien von Definition 13.5.2.1 umfangreiche Anforderungen an Tests stellen, sind entsprechende Tests nicht ausreichend, da jeder Operationsaufruf nur mit einer einzigen Wertekombination für die Schnittstellenvariablen erfolgen muß. Spezielle Berechnungsfehler, die sich nur bei einem bestimmten Datenfluß fortpflanzen, werden also nicht erkannt.

Daher ist eine datenflußbasierte Vorgehensweise angebracht, der folgende Konzepte zugrunde liegen:

1. Statisch festgestellte potentielle Datenflußanomalien sollen ausgeführt werden (siehe Fußnote 17 in Abschnitt 13.5.1).

2. Der Datenfluß bei Operationsaufrufen an der Schnittstelle von Modulen ist zu testen. Dabei wird eine analoge Vorgehensweise wie beim Ermitteln von Datenflußanomalien (s. Abschnitt 13.5.1) angewandt:

   (a) Im *aufrufenden* Modul müssen die letzten (bzw. ersten) Aktionen auf den Schnittstellenvariablen vor (bzw. nach) dem Aufruf ermittelt werden (*define* bzw. *reference*).

(b) In der *aufgerufenen* Operation müssen ebenfalls die ersten (bzw. letzten) Aktionen auf den Schnittstellenvariablen ermittelt werden (*reference* bzw. *define*).

Dann sind alle (bzw. einige) Wege von den ermittelten letzten *define*-Aktionen im aufrufenden Modul zu den ermittelten ersten *reference*-Aktionen in der aufgerufenen Operation beim Test auszuführen. Die Wegeauswahl kann sich dabei an den Datenflußkriterien aus Kapitel 8 orientieren (s. Übung 13.7). Entsprechendes gilt für Wege von den letzten *define*-Aktionen der aufgerufenen Operation zu den ersten *reference*-Aktionen des aufrufenden Moduls nach dem Aufruf.

13.5.2.2 Wertbezogener Integrationstest

Beim wertbezogenen Vorgehen werden *bestimmte* Werte für Parameter verwendet, mit denen mit größerer Wahrscheinlichkeit Fehler aufgedeckt werden können. Um Sonderfälle zur Ausführung zu bringen, die beim Modultest wegen der Platzhalterproblematik bisher nicht getestet wurden, kommen die folgenden herausragenden Werte in Frage:

- Grenzwerte (vgl. Abschnitt 4.2.2 und die Bereichsüberdeckung in Kapitel 9.2)
- Extremwerte (0, 1, -MAXINT, +MAXINT, etc.)
- Fehlerfälle (z.B. negative Werte)

Beim Integrationstest sind außerdem Fehler aufzudecken, die dadurch entstehen, daß eine Operation P mit falschen Werten der Schnittstellenvariablen aufgerufen wird (s. Beispiel 13.4.2 auf S. 362). Wenn die Operation P nur in gewissen Fällen darauf mit fehlerhaften Ergebnissen reagiert, werden diese Fälle beim Modultest des aufrufenden Moduls A meistens nicht getestet. (Beim Test von Operation P wird der Fehler natürlich auch nicht bemerkt, da ja P auf korrekte Eingaben richtig reagiert.) Damit stellt sich beim Integrationstest die Aufgabe, aus der (Kontroll- und Datenfluß-)Struktur der aufgerufenen Operation Werte für die Eingabevariablen abzuleiten, so daß sich ein Fehler bei einer Eingabevariablen *mit Sicherheit* als Fehler bei einer Ausgabevariablen bemerkbar macht.

13.5.2.3 Funktionsbezogener Integrationstest

Der funktionsbezogene Integrationstest ist ein spezifikationsorientierter Test, der das korrekte Zusammenspiel der von den einzelnen Modulen realisierten Teilfunktionen bzw. Abweichungen von der spezifizierten Funktionalität feststellen soll. Abweichungen können von folgender Art sein:

- mangelnde Funktionalität, d. h. ein Modul liefert erwartete Teilfunktionen nicht;
- zuviel Funktionalität, d. h. eine Teilfunktion wird vom Modul zusätzlich ausgeführt, obwohl dies nicht spezifiziert bzw. erwartet ist;
- die Funktionalität des Moduls ist falsch (vgl. das Beispiel des Sortiermoduls in Abschnitt 13.4.2, Seite 361).

Für jede exportierte Operation P eines Moduls sollte überprüft werden, ob die Funktionalität mit den Erwartungen aller Module, die P importieren, übereinstimmt bzw. davon abweicht. Diese Überprüfung sollte durch eine informelle Analyse (Inspektion, s. Kapitel 12.1) und durch Tests erfolgen, wobei die Testdaten aus den ablauf- und wertbezogenen Tests verwendet werden können.

13.5.2.4 Testdatenerzeugung für den Integrationstest

Die bisher vorgestellten Testmethoden für den Integrationstest haben nur Kriterien formuliert, *was* zu testen ist, ohne anzugeben, *wie* (mit welchen Testdaten) die Tests auszuführen sind.
Um Aufwand zu sparen, ist folgendes Vorgehen zu empfehlen, wenn eine Schnittstelle zwischen einem Modul M und einer aufgerufenen Operation P getestet werden soll:

1. Aus den bisher erstellten Testdaten für den Modultest von M sind diejenigen Testdaten auszuwählen, die den Aufruf von P (bzw. dessen Stellvertreter beim Modultest) zur Ausführung bringen.
2. Zu den ausgewählten Testdaten sind die Parameterwerte zu bestimmen, mit denen P bzw. sein Stellvertreter aufgerufen wird.
3. Wird mit den festgestellten Parameterwerten ein geforderter Pfad in P ausgeführt (z. B. von einer Variablendefinition zu einer -referenz bei der datenflußbasierten Vorgehensweise), ist ein Testdatum gefunden, das einen Aspekt eines Testkriteriums erfüllt, indem es einen entsprechenden Weg durch M und P ausführt.
4. Werden mit Schritt 3 nicht alle Aspekte eines Testkriteriums erfüllt, sind weitere Testdaten durch symbolische Ausführung der entsprechenden Wege durch Modul M und Operation P zu bestimmen (vgl. Kapitel 11.2 und 12.3).

Beim wertbezogenen Integrationstest müssen nicht unbedingt bestimmte Wege ausgeführt werden, dafür aber Anweisungen mit bestimmten Werten (z. B. Grenzwerten). Dies kann aber bei der symbolischen Berechnung beachtet werden (vgl. Kapitel 11.2).
Der allgemeine Ansatz der **Fehlerfortpflanzung** von falschen Eingabevariablewerten bis zu den Ausgabevariablen verlangt allerdings einen besonderen Lösungsansatz,

etwa die folgende Idee: Bei m Eingabevariablen[18] einer aufgerufenen Operation gibt es nur m Arten, wie Eingabefehler auftreten können. Jeder Eingabefehler ist eine (Linear-)Kombination dieser m Eingabe-Fehlerarten (genannt **Fehlerrichtungen**).

Beispiel 13.5.2.2
$m = 2$, Eingabevariablen E1, E2, fehlerhafte Eingabevariablen: $E1' = E1 + e1$, $E2' = E2 + e2$. Dabei ist $(E1\ E2)^T$ der Vektor[19] der korrekten Eingabe und $(e1\ e2)^T = e1 * (1\,0)^T + e2 * (0\,1)^T$ der „Fehler“, wobei $(E1'\ E2')^T = (E1\ E2)^T + (e1\ e2)^T$ die fehlerhafte Eingabe ist. Die Fehlerrichtungen $(1\,0)^T$ und $(0\,1)^T$ sind dabei unabhängig. (Vorausgesetzt ist die Unabhängigkeit von E1 und E2.)

Das Vorgehen zum Finden von Integrationsfehlern besteht nun darin, möglichst nur einen (höchstens m) Weg(e) durch den Kontrollflußgraphen der aufgerufenen Operation zu bestimmen, auf dem jeder Eingabefehler[20] zu einem in der Ausgabe erkennbaren Berechnungsfehler führt. Ein solcher Weg heißt (**fehler-)sensitiv**.

Beispiel 13.5.2.3 (Bereichsfehler im Programm von Beispiel 13.4.2)
Eingabe: c = *mc*
Ausgabe: a = *ma*

Die Grenze der beiden Wege (then-Zweig/else-Zweig) ist zu testen. Dafür sind zwei Testdaten notwendig:

t1: $c = mc = 4$ *(auf der Grenze),*
t2: $c = mc = 4 - \epsilon$ *(unterhalb der Grenze mit kleinstmöglichem $\epsilon > 0$).*

Für die fehlerhafte Eingabe $c' = c + c_1$ erhält man dann (falls $|c_1| \geq \epsilon$, d. h. der Fehler ist größer als die „Testgenauigkeit“ ϵ):

1. *$c_1 > 0$: Bei t2 wird statt des then-Zweigs (ma = 1) im Fehlerfall der else-Zweig (ma = 2) ausgeführt.*
2. *$c_1 < 0$: Bei t1 wird statt des else-Zweigs (ma = 2) im Fehlerfall der then-Zweig (ma = 1) ausgeführt. (Siehe Beispiel 13.4.2 auf S. 362: richtig ist $c = i + 1$, falsch ist $c = i = (i + 1) - 1$, d. h. der Fehler ist $c_1 = -1$).*

[18] Eingabeparameter und globale Variablen
[19] Ein Vektor $\binom{E1}{E2}$ wird hier als transponiertes Tupel $(E1\ E2)^T$ geschrieben.
[20] jede Kombination der m Eingabefehlerarten, d. h. jede Abweichung von der korrekten Eingabe in irgendeiner (oder mehreren) Fehlerrichtung(en)

Der vorgestellte Ansatz wirft leider viele ungelöste Probleme auf (genaueres siehe [HaZ 81]):

- Berechnung und Existenz der fehlersensitiven Wege,
- Testdatenerzeugung für das aufrufende Modul (oben wurden nur die Parameterwerte *beim Aufruf* bestimmt) (vgl. Kapitel 11.2),
- Das Auftreten eines Fehlers *nach* einem Operationsaufruf ist eventuell abhängig vom gewählten Weg im Kontrollflußgraphen der Operation.

Daher ist obige Methode bisher nur von theoretischem Interesse, d. h. es müssen alternative Verfahrensschritte erforscht werden.

13.6 System- und Abnahmetest sowie Verfahren in der Installations- und Wartungsphase

Der **Systemtest** ist der abschließende Test des Gesamtsystems aus Entwicklersicht. Bei der Übergabe des Systems an die Auftraggeber bzw. Käufer sollte von diesen ein **Abnahmetest** durchgeführt werden, bei Standard-Software ein entsprechender **Beta-Test** durch potentielle Benutzer. Die zu beachtenden Kriterien bei beiden Tests, die Unterschiede und Gemeinsamkeiten sind in Tabelle 13.1 aufgelistet.

Kriterium	**Systemtest**	**Abnahmetest**
Ziel	Zeigen, daß das System *nicht* einsatzbereit ist.	
Personen	Qualitätssicherungsabteilung des Entwicklers	Auftraggeber und/oder Benutzer
Prüfobjekt	System nach Modultest (mit 20–50 Fehlern pro 1000 Codezeilen)	System nach Systemtest: stabil bzw. reifend[21]
Testreferenz bzw. Testdatenquelle	Pflichtenheft, Benutzerhandbuch, Fehlbedienungshandlungen/-eingaben	
	Spezifikation	aktuelle Benutzerwünsche
Ziele und Aspekte	(a), (c) bis (i) von (2) aus Kapitel 13.1	
	—	Verträglichkeit mit der Organisation
Umgebung (Rechner)	Entwicklungsrechner	i. allg. Zielrechner
Anwendungsumgebung	i. allg. simuliert	i. allg. real
(juristische) Folgen eines akzeptablen Testergebnisses	Freigabe des Systems (Auslieferung an den Kunden)	Beginn der Garantiezeit (und Zahlungsverpflichtung)

Tab. 13.1 Unterschiede und Gemeinsamkeiten von Systemtest und Abnahmetest

In der Installations- und Wartungsphase sind weitere Operationen zu befolgen, um einen reibungslosen und möglichst sicheren Einsatz des Systems zu gewährleisten. Dazu gehören folgende Verfahren:

- Kontrollprozeduren für den laufenden Betrieb einführen (z. B. einen Systemmanager, der eine Ressourcen- und Benutzungskontrolle realisiert)
- Leistungsfähigkeit des Systems dokumentieren:
  - Bericht über die Leistungsfähigkeit
  - Statusbericht(e) über die Benutzung
  - Liste der nicht ausgeführten Änderungsanforderungen
- Softwareverbesserungen und -erweiterungen mit Hilfe eines Qualitätskontrollplans steuern: Eine Änderung wird vorgenommen, wenn eine Anforderung vom Entwicklungsmanagement bejaht wird. Die zu ändernde Software durchläuft dann folgende Phasen: bearbeiten, testen (durch Regressionstest[22]), formell freigeben, Testbibliothek ändern und die Dokumentationsbibliothek ändern.

13.7 Übungen

Übung 13.1:

Betrachten Sie die Modulhierarchien a) und b) aus Abbildung 13.5.

(a) Ermitteln Sie die Anzahl aller möglichen Reihenfolgen beim aufsteigenden und absteigenden Integrieren der Module.

(b) Welche dieser Reihenfolgen ist möglich, wenn die Eingabemodule und dann die Ausgabemodule möglichst früh integriert werden sollen?

(c) Geben Sie für Fall (a) Möglichkeiten zur Parallelarbeit an.

Übung 13.2:

Welche Vor- und Nachteile gegenüber dem absteigenden (top down) Testen hat das **modifizierte absteigende Testen**, bei dem erst alle Module einzeln getestet werden und dann das (normale) absteigende Testen angewandt wird?

Übung 13.3:

Um die Vorteile des absteigenden Testens und des aufsteigenden Testens zu kombinieren, bietet sich das **Sandwich-Testen** (mit folgenden Schritten) an:

[21] stabil: 0 bis 0,5 Fehler pro 1000 Codezeilen, reifend: 0,5 bis 3 Fehler pro 1000 Codezeilen

[22] s. Kapitel 13.1, Aspekt 1. (Genaueres dazu siehe in [RoH 94].)

- Zerlege die Benutzt-Hierarchie durch einen waagerechten Schnitt in zwei Teilmengen.
- Wende den absteigenden Test auf die Module oberhalb des Schnitts an und parallel dazu den aufsteigenden Test auf die Module unterhalb des Schnitts.
- Ersetze zum Schluß die Platzhalter für den oberen Teil komplett durch die Module des unteren Teils (oder nacheinander, falls dabei keine neuen Platzhalter erforderlich werden) oder ersetze nacheinander die Treiber für den unteren Teil durch Module (und Treiber) des oberen Teils.

(a) Für die Modulhierarchie b) aus Abbildung 13.5 sei der obere Teil A, B, C, D, E und der untere Teil somit F, G, H. Welche Reihenfolgen sind beim entsprechenden Sandwich-Testen möglich?

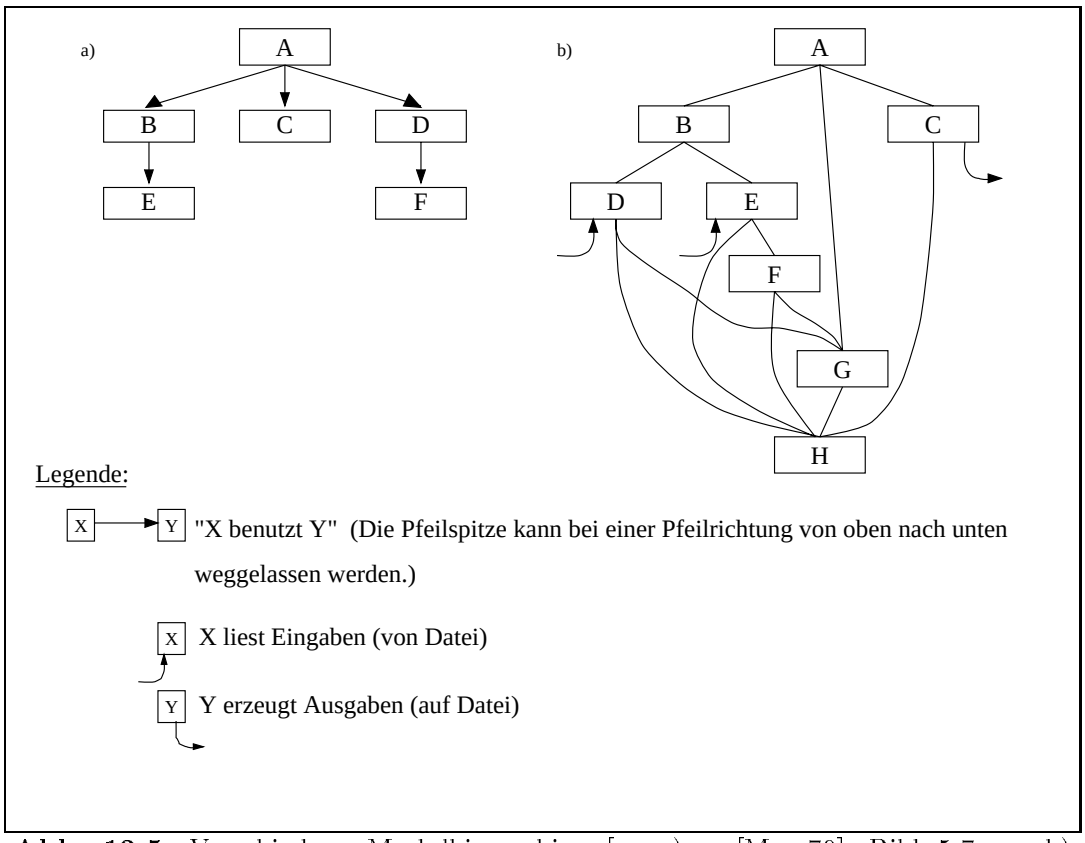


Abb. 13.5: Verschiedene Modulhierarchien [zu a) s. [Mye 79], Bild 5.7; zu b) vgl. [Mye 76], Fig. 10.5]

(b) Welche Reihenfolgen sind bei Aufgabe (a) möglich, wenn nur A, B, C den oberen Teil bilden?

(c) Welche Vor- und Nachteile hat das Sandwich-Testen gegenüber dem (reinen) aufsteigenden bzw. dem (reinen) absteigenden Testen? Welche Vor- und Nachteile hat eine Modifikation des Sandwich-Testens, bei der im oberen Teil das *modifizierte* absteigende Testen (s. Übung 13.2) angewandt wird?

Übung 13.4:

Das **vertikale Scheibentesten** besteht aus folgenden drei Schritten:

- Zerlege die Benutzt-Hierarchie in Teilgraphen, die jeweils die Spitze der ursprünglichen Hierarchie als Wurzel haben und insgesamt die Benutzt-Hierarchie überdecken.
- Teste parallel alle Teilgraphen (jeden Teilgraphen nach beliebiger inkrementeller Strategie).
- Teste die Integration der Teilgraphen (schrittweise oder alle auf einmal integriert).

1. Definieren Sie geeignete Teilgraphen für die Modulhierarchien von Abbildung 13.5 und wenden Sie darauf das vertikale Scheibentesten an.
2. Welche Vor- und Nachteile hat das vertikale Scheibentesten gegenüber den anderen inkrementellen Integrationsstrategien?

Übung 13.5:

Ermitteln Sie die potentiellen Datenflußanomalien für die Integration von Modul M und Prozedur P (siehe Abbildung 13.6) mit dem Verfahren aus Abschnitt 13.5.1.

```
Modul M                  procedure P (var x: integer);
...                      var y: integer;
var i: integer;          begin
begin                    read(y);
if k = 10                if y > 10
then i := k;             then y := x;
P(i);                    else x := 10;
...                      ...
```

Abb. 13.6: Integration von Modul M und Prozedur P (s. [Spi 92b], S. 95)

Übung 13.6:

Stellen Sie fest, ob der in Abbildung 13.7 angegebene Kontrollflußgraph des Programmsystems Reihenfolgen von Operationsaufrufen zuläßt, die laut Spezifikation nicht erlaubt sind. Die Spezifikation ist durch den Pfadausdruck $P = (Q|R|S)$ gegeben, wobei:

$$Q = (P1; ((P11; P21; P211); (P11; P21; P212)^{+})^{+}; P3; P31; [P32; P211])$$

$$R = (P2; P21; P211; P3; P31; P32; P211)$$

$$S = (P2; [P21; P212]; P3; P31; [P32; P211])$$

Übung 13.7:
Wenden Sie die datenflußbezogenen Integrationstestkriterien, die den Kriterien *alle Definitionen*, *alle B-/einige E-Referenzen*, *alle E-/einige B-Referenzen* und *alle Referenzen* entsprechen (vgl. Kapitel 8.2 und Abschnitt 13.5.2.1), auf die Schnittstelle von Modul IM und Modul EX aus Abbildung 13.8 an. Geben Sie die entsprechenden Testfälle pro Kriterium an.

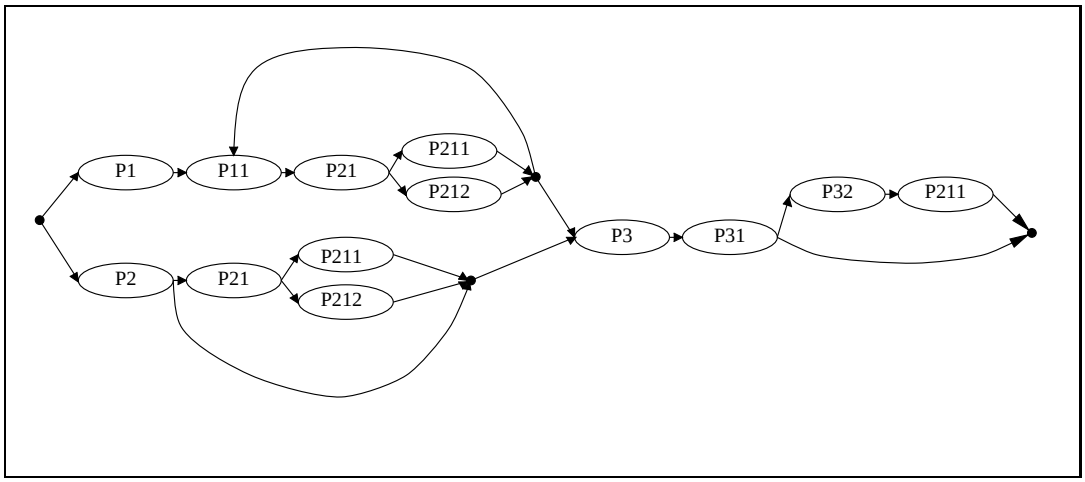


Abb. 13.7: System-Kontrollflußgraph (nach [Spi 90], Abb. 4)

```
Modul IM                Modul EX

import P;               procedure P (var x: integer);
 ...                     ...
var i: integer;         begin
begin                   read(y);
if k < 10               if y > 10
then i := k;            then y := y − x;
else i := k/10;         else if x < 10
P(i);                   then ...
 ...                     ...
```

Abb. 13.8: Aufrufendes Modul IM und aufgerufenes Modul EX (s. [Spi 95], S. 280)

13.8 Verwendete Quellen und weiterführende Literatur

Die verschiedenen **Aspekte**, die beim (Teil-)**Systemtest** zu beachten sind, beschreiben z. B. Myers und Wallmüller (s. [Mye 76], [Mye 79], [Wal 90]). Schmitz et al. zählen die Aufgaben des Testens und die Testphasen in Kapitel 3 und 4 von [SBM 82] auf.
Die benötigten **Modelle** für den **Integrationstest** werden von folgenden Autoren vorgestellt: Modulgraphen von Linnenkugel/Müllerburg (als „p-graphs“, s. [LiM 90]), Aufrufreihenfolgebeschreibungen von Spillner (s. [Spi 90]), interprozedurale Flußgraphen (IFG) bzw. erweiterte DEF-REF-Graphen (*extended def-use graphs*) zur Datenflußbeschreibung von Harrold/Soffa und Ural/Yang (s. [HaS 89], [UrY 93]). Die exakte Bestimmung von Datenflußbeziehungen beim Verwenden von Zeigern (z. B. in C) nehmen Pande et al. vor (s. [PLR 94]).
Die Definition des **Modultests** stammt schon von Stevens et al. (s. [SMC 74]), sie wurde hier um die Benutzung der Typdefinition (Sortendefinition) eines anderen Moduls erweitert (siehe dazu, zu einer etwas anderen Definition von Modul und zur Frage der [nicht-]disjunkten Zerlegung eines Programms in Module [Kel 84], S. 3 ff.). Frühe Ansätze zur Automatisierung des Modultests mit Treibern und Platzhaltern finden sich z. B. bei Panzl und Sneed/Kirchhoff (s. [Pan 78], [SnK 79]).
Auf den Zweck des Integrationstests wird z. B. in [LiM 90] und [Spi 95] hingewiesen. Die möglichen **Strategien beim Integrationstest**, spezielle Vor- und Nachteile des **absteigenden** und des **aufsteigenden Tests** und die Vorschläge zur Strategieauswahl stammen von Myers ([Mye 76], Kap. 10; [Mye 79], Kap. 5). Das **vertikale Scheibentesten** wurde dagegen von Stephenson vorgeschlagen (s. [Ste 80]). Genaueres zu den Vorteilen des **inkrementellen Testens**, insbesondere zur Fehlerlokalisierung, findet man bei Haley und Zweben (s. [HaZ 81]). Die ambivalenten Eigenschaften des absteigenden Tests werden von Zweben angegeben, allerdings als Vorteile betrachtet (s. [Zwe 81]). Von Quirk stammt der Vorschlag, bei einem gründlichen Test beide inkrementellen Verfahren (absteigend und aufsteigend) nicht nur teilweise — wie beim **Sandwich-Testen** — sondern vollständig anzuwenden (s. [Qui 85], S. 134 f.).
Das **Aufwandproblem** für einen vollständigen Test eines (Teil-)Systems nach gängigen programmbezogenen Kriterien wird in [LiM 90], S. 712, angesprochen. Die zu beachtenden **Fehlerarten** und **Voraussetzungen** beim Integrationstest beschreibt Spillner (s. [Spi 95], S. 278 f.). Die Definition von speziellen **Integrationszeit-Bereichsfehlern** und **-Berechnungsfehlern** findet sich in [HaZ 81].
Die vorgestellten **statischen Integrationstestmethoden** stammen von Spillner und Harrold/Soffa (s. [HaS 89], [Spi 90], [Spi 92a]), die **ablaufbezogenen** Methoden finden sich in [HaS 89], [LiM 90] und [Spi 95]. Eine Einschränkung der **interprozeduralen Datenflußanalyse** auf bestimmte (vorgegebene) Wege nehmen Horwitz et al. vor (s. [HRS 95]). Die Konzepte zum **wertbezogenen** und **funktionsbezoge-**

nen Integrationstest sowie zur **Testdatenerzeugung** wurden von Spillner übernommen (s. [Spi 92a]). Der spezielle Ansatz der **Fehlerfortpflanzung** von falschen Eingabevariablenwerten stammt von Haley/Zweben (s. [HaZ 81]).
Eine gute Einführung in den **System-**, **Installations-** und **Abnahmetest** findet man in [Mye 79], Kapitel 6, sowie in [Myn 90], Kapitel 7.9. Ein systematischer Ansatz für Systemtest und Abnahmetest wird von A. Celentano[23] et al. in [CGL 81] beschrieben, weitere Hinweise dazu findet man z. B. in [PaR 91], Kapitel 8.5, und in [FLS 91b], Kapitel 3.3. Empfehlungen zum **Betriebstest** und zu Installation und **Wartung** geben Cave/Maymon in Kapitel 8.4 von [CaM 88]. McCabe und Schulmeyer stützen den Systemtest auf **Structured Analysis** (eine Art der Anforderungsanalyse, -dokumentation und -spezifikation durch Datenflußdiagramme); dabei soll gezeigt werden, daß alle „Funktionen“ des Systems ausgeführt wurden. Dies entspricht dem Ausführen aller „Segmente“ auf unterer Ebene, wobei die Entsprechung Funktion $\hat{=}$ Segment vorliegt (siehe [McS 82]). Eine Bewertung verschiedener Techniken zur Auswahl von **Regressionstests** nehmen Rothermel/Harrold vor (s. [RoH 94]).

[23] Augusto Celentano, Polytecnico de Milano, nicht Adriano Celentano (Sänger und Schauspieler).