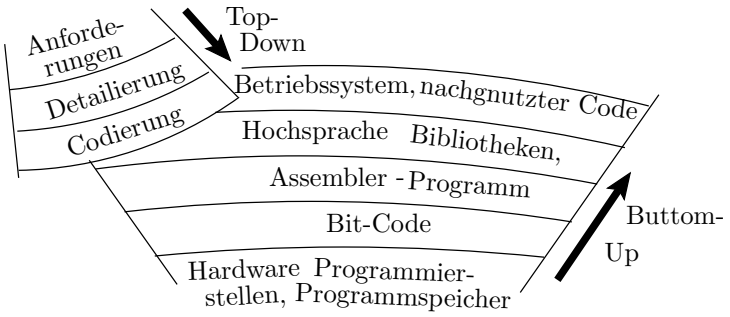
Test und Verlässlichkeit Foliensatz 6: Software

Prof. G. Kemnitz

10. April 2022

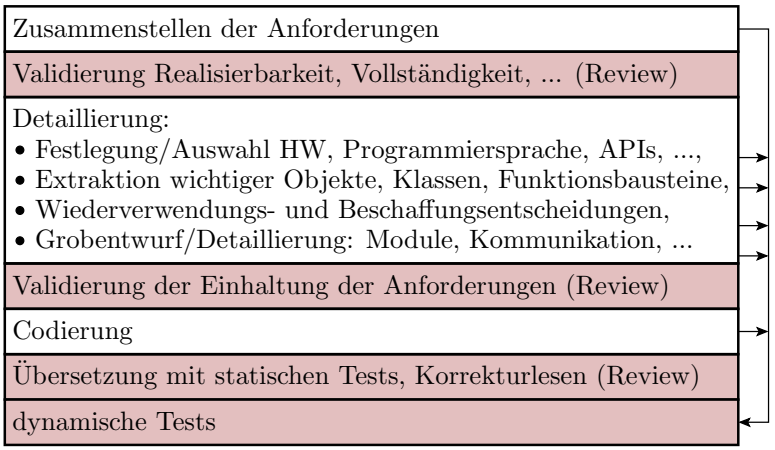
Contents

Software



- Software legt funktionale Schichten über die Hardeware,
- ist für komplexere Funktionen selbst in Schichten organisiert.
- Jede Schicht erbt die Funktionalität und die Fehler der darunter.
- Der SW-Entwurf setzt in der Regel auf eine Hochsprache, ein Betriebssysteme und vorhandene SW-Bausteine auf.
- Der HW-Entwurf gleicht dem SW-Entwurf in vielen Aspekten.

Der SW-Entwurf in Stufen



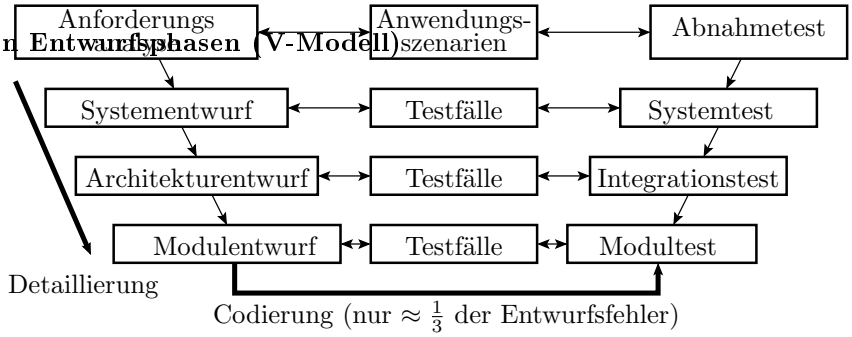
In jeder Stufe vervielfacht sich die Menge der erstellten Entwurfsdaten und die Fehleranzahl. Nach jeder Stufe folgen Kontrollen.

Fehlervermeidung

Fehlervermeidung ist die Suche eines Vorgehens, bei dem möglichst wenig Fehler entstehen (vergl. Foliensatz F1, Abschn. 5.3 Fehlervermeidung. Projekte und Vorgehensmodelle). Ansatzpunkte

- Minimierung des (manuellen) Aufwands: Automatisierung, Nachnutzung von Bausteinen aus anderen Projekten, gut ausgebildete Entwickler, ...
- Vermeidung von Nachbesserungsiterationen, vor allem bei Schnittstellen,
- Etablierung von Abläufen mit gut vorhersagbaren Ergebnissen (Arbeistaufwand, Systemgröße, Fehlerenstehungsrate)
- Lernen gute funktionierenden Abläufen, Arbeitsteilungen, ... aber auch auf Fehlern.
- Guter Kompromiss zwischen Projektdisziplin und kreativem Freiraum.
- ...

Tests aus allen Entwurfsphasen (V-Modell)



In Ergänzung zu einem einfachen Stufenmodell definiert das V-Modell, dass in jeder Stufe Testszenarien oder Testfälle zu definieren und im aufsteigenden V-Ast abzuarbeiten sind. Positive Effekte:

- deckt Testprobleme zeitig im Entwurfsfluss auf,
- zum Entwurf diversitäre Tests und Sollwerte, ...

Das V-Modell-XT ergänzt formale Beschreibungsmittel, um

- die Abläufe aus Entwurfsschritten,
- durchzuführenden Kontrollen und
- der dabei zu dokumentierenden Ergebnisse zu dokumentieren.

Das Rreifen von technologischen Abläufen erfolgt durch eine Iteration kleiner, auf Resourcen und Aufgaben angepasste Optimierungen und Erfolgskontrollen. Genormte Dokumentationen sind dafür hilfreich.

Als Richtwert entstehen bei der Detaillierung 2/3 und bei der Codierung 1/3 der Entwurfsfehler.

Beispiel 1. Typische Werte: ca. 30 bis 100 Fehler auf 1000 NLOC. Davon werden 95% gefunden und beseitigt. Von den verbleibenden 1,5 bis 5 Fehler je 1000 NLOC sind ca. 15% schwerwiegend und davon 10% aus der Detaillierungsphasen und 5% aus der Codierungsphase.

0.1 Testbare Anforderungen

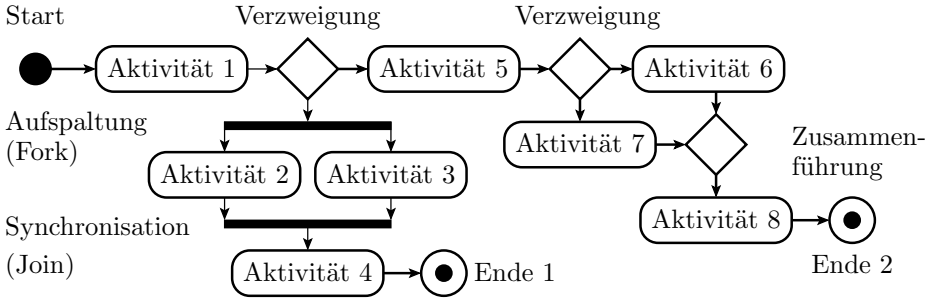
Testbare Anforderungen

Für die Beschreibung testbarer Anwendungsszenarien unterscheidet die Modellierungssprache UML zur Spezifikation, Konstruktion, Dokumentation und Visualisierung von Software-Teilen:

- Aktivitätsdiagramme,
- Sequenzdiagramme,
- Zustandsdiagramme,
- Protokollautomaten, ...

Das sind Beschreibungen, die nicht nur Aspekte der Zielfunktion beschreiben, sondern aus denen sich auch Testbeispiele ableiten lassen.

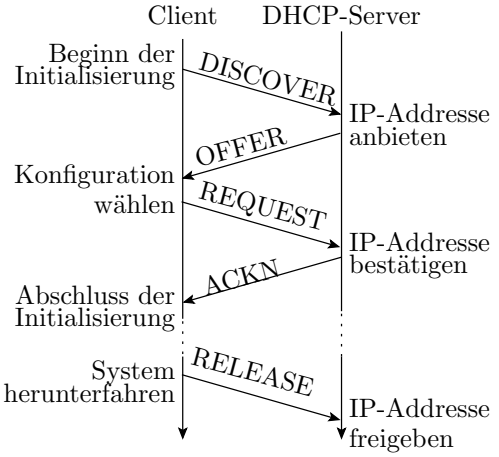
Aktivitätsdiagramm



Ein Aktivitätsdiagramm beschreibt Ablaufmöglichkeiten, die aus Aktivitäten (Schritten), Transaktion, Verzweigung, Synchronisation, Signale senden und empfangen. Aus dem Beispiel ableitbare Testfälle:

- Start, A1, A2‖A4, Ende 1
- Start, A1, A5, A7, A8, Ende 2
- Start, A1, A5, A6, A8, Ende 2

Sequenzdiagramm

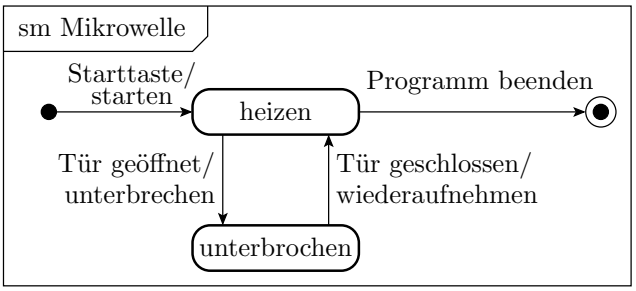


Nachricht	Beschreibung
DISCOVER	Broadcast eines Clients, der einen Server sucht
OFFER	Antwort eines Servers mitKonfigurationsvorschlag
REQUEST	Braodcast des Clients an den bevorzugten Server Ablehnung aller anderen Server
ACKN	Server liefertIP-Adresse
NAK	Der Server lehnt die IP-Adresse ab
DECLINE	Der Server hat ein Problem mit der angebotenen IP-Adresse und lehnt ab
RELEASE	Client gibt IP-Adresse frei

Sequenzdiagramme sind Interaktionsdiagramme und zeigen den zeitlichen Ablauf einer Reihe von Nachrichten (Methodenaufrufen) zwischen Objekten, Threads, Rechnern, ... in einer zeitlich begrenzten Situation. Dabei kann auch das Erzeugen und Entfernen von Objekten enthalten sein.
Ableitbare Tests:

- Korrekte Abläufe mit korrekten Daten.
- Korrekte Abläufe mit unzulässigen Daten.
- Korrekte Reihenfolge mit Zeitüberschreitungen.
- Unzulässige Reihenfolge der Nachrichten.
- Ursache-Wirkungsgraph für Server und Client für die Testauswahl (siehe später Abschn. #).

Zustandsdiagramm



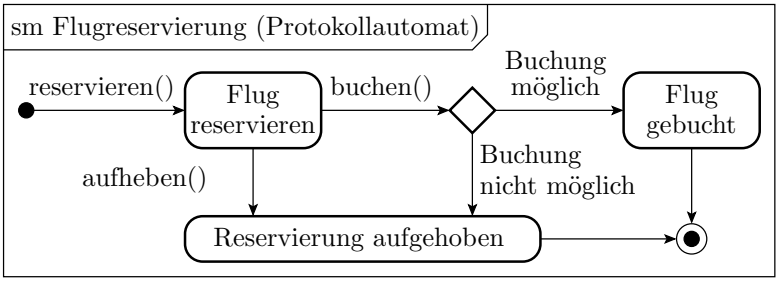
Zustandsdiagramm (Verhaltenszustandsautomat, engl. behavioral state machine) beschreibt Funktionsabläufe durch:

- Zustände,
- Kanten mit Bedingungen für Zustandsübergänge,
- Zuständen und/oder Kanten zugeordnete Aktivitäten.

Ableitbare Tests:

- Abläufe, die alle Knoten abdecken.
- Abläufe, die alle Kanten abdecken.
- Abläufe bis zu allen Knoten und Test der Reaktion auf nicht spezifizierte Übergangsbedingungen.

Protokollautomat

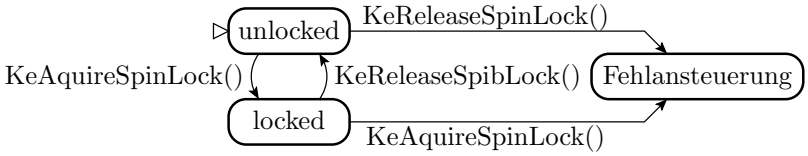


Beschreibung zulässiger Aktionsreihenfolgen. Mögliche Aktionen im Beispiel sind die Methodenaufrufe »reservieren()«, »aufheben()« und »buchen()«. Aus dem Protokollautomat im Beispiel geht hervor, dass ein Flug nur nach erfolgreicher Reservierung gebucht und dass ein einmal gebuchter Flug nicht gestrichen werden kann.

Kontrollautomaten können als Basis für Korrektheitsbeweise oder als Spezifikation für die Programmierung von Überwachungsautomaten genutzt werden.

Ableitbare Tests: zulässige Reihenfolgen, Fehlerbehandlung bei unzulässigen Reihenfolgen, ...

Statischer Test für API-Benutzungsregeln



Beispiel Benutzung der Windows-API aus [1], Kontrollautomat für die für Regel »spinlock« :

spinlock Spinlocks müssen alternierend reserviert und freigegeben werden.

spinlocksafe Vermeidung von Deadlocks mit Spinlocks.

criticalregions Problemvermeidung im Zusammenhang mit der Nutzung kritischer Regionen.

Eine zu testende Treiberfunktion

Eine Treiberfunktion ruft »KeAquire..« und »KeRelease...« u.U. mehrfach auf, in Fallunterscheidungen, Schleifen, ...
Für jeden Kontrollpfad muss der Spinlock alternierend bedient werden.

Fehlerausschluss erfordert Kontrolle für alle Pfade.

Reale Treiberfunktionen haben hunderte von Code-Zeilen. Kontrolle selbst so einfacher Regeln nicht trivial.

```
void example(){
 do {
  KeAcquireSpinLock();
  nPacketsOld = nPackets;
  req = devExt->WLHV;
  if(req && req->status){
   devExt->WLHV = req->Next;
   KeReleaseSpinLock();
   irp = req->irp;
   if(req->status > 0){
    irp->IoS.Status = SUCCESS;
    irp->IoS.Info = req->Status;
   } else {
    irp->IoS.Status = FAIL;
    irp->IoS.Info = req->Status;
   }
   SmartDevFreeBlock(req);
   IoCompleteRequest(irp);
   nPackets++;
  }
 } while(nPackets!=nPacketsOld);
 KeReleaseSpinLock();
}
```

1 Good Practice

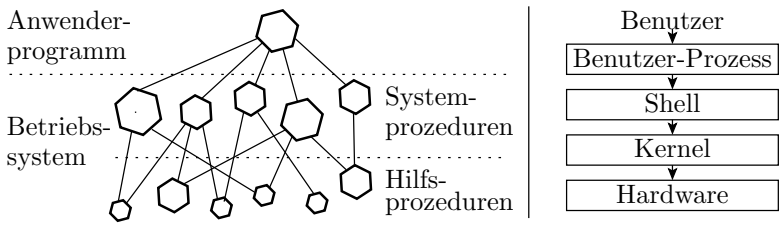
1.1 Software-Architektur

Software-Architektur

Je komplexer, desto wichtiger ein klare Struktur als Voraussetzung

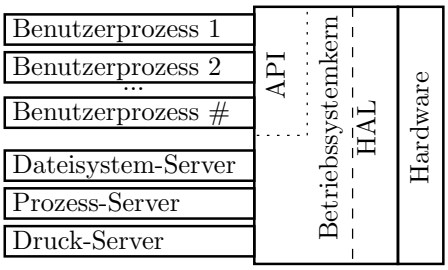
- um den Überblick zu behalten (Fehlervermeidung),
- für die Entwicklung in Teams,
- um nachträgliche Änderungen vornehmen zu können (Wartbarkeit),
- zur Fehlerisolation und für robuste Reaktionen auf FF (Fehlerbehandlung),
- die Durchführbarkeit von Tests (prüfgerechter Entwurf).

Prozedurensammlung und Schichten



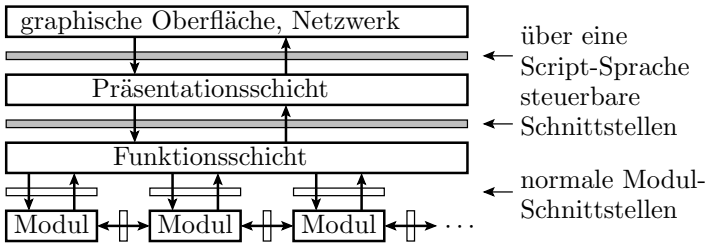
- Eine Prozedurensammlung bietet Schnittstellen, aber ein Programmierer muss sich nicht an diese Schnittstellen halten.
- Schichtenmodell: Ein Zugriff von einer anderen Schicht - beispielsweise einer Benutzeranwendung wie Excel oder Word - auf eine andere Schicht ist nur über eine definierte Schnittstelle (API) möglich. Ebenso kann beispielsweise ein Kommunikationsprogramm nicht direkt auf den COM-Port zugreifen. Die Applikation stellt eine Anfrage an das Betriebssystem, ob der COM-Port verfügbar ist.

Client/Server-Modell



Client/Server-Betriebssysteme bestehen aus kleinen Einzelteilen die autonom arbeiten können den sogenannten Servern. Der Betriebssystemkern ist klein (Mikrokernel) und sorgt für die Kommunikation zwischen den Servern und den Clients welche in Form von Applikationen die Dienste der Server beanspruchen. Client/Server-sind flexibel und leicht auf andere Plattformen portierbar.

Schichten als Testschnittstelle



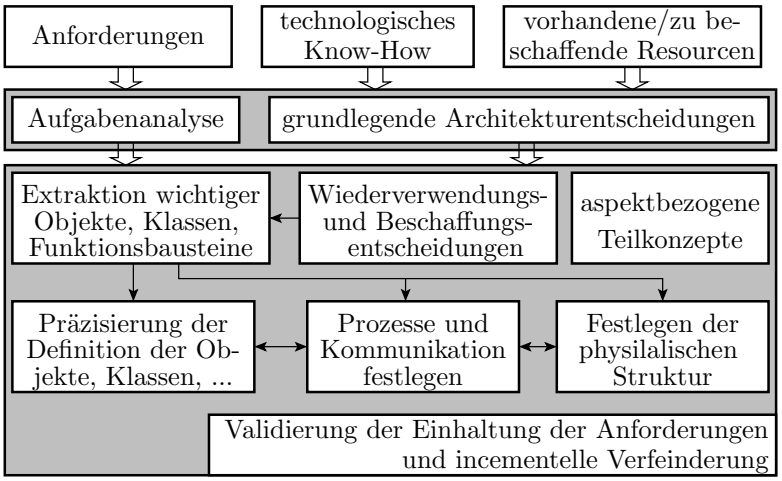
Schichten sind wohl definierte Schnittstellen, die eine Steuerung über Script-Sprache erlauben:

- Betriebssystem-Shell,
- Schnittstellen zwischen Netzwerken und Graphik-Oberflächen und der Repräsentationsschicht.
- Schnittstellen von der Repräsentationsschicht zu den Anwenderprogrammen.

Mit diesen Scriptsprachen dienen auch für den Test.

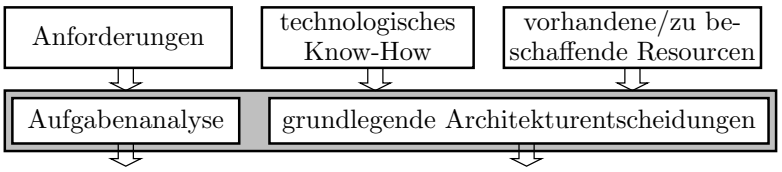
1.2 Entwurfsfluss

System- und Architekturentwurf



Zielführende Reihenfolge der Entwurfseinscheidungen.

Vorgaben und Architekturentscheidungen



technologisches Know-How:

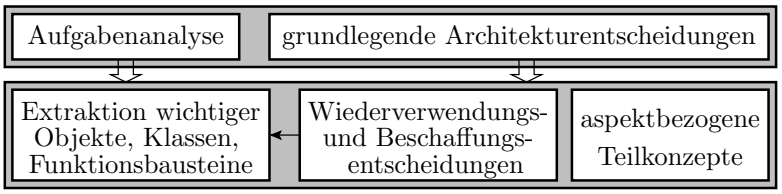
- Erfahrungen mit ähnlichen Projekten,
- nachnutzbare Software-Bausteine und Tests,
- alte Projektpläne, ...

vorhanden/zu beschaffen: Rechner, Software, Personal.

grundlegende Architekturentscheidungen:

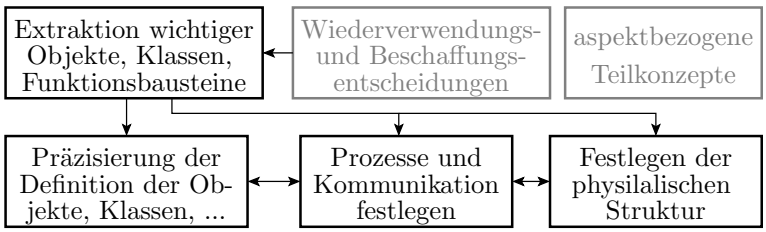
- Prozedurensammlung, Client-Server-Architektur, ...
- File-System oder Datenbank, ...
- Wiederverwendung, Vergabe von Unteraufträgen.
- Benutzerschnittstellen, Fehlerbehandlung, Fehlertoleranz, ...

Entwurf des Systemkonzepts



- Entscheidungen über Wiederverwendung, Beschaffung und Vergabe von Unteraufträgen implizieren Vorgaben für den Rest.
- Zu den aspektbezogene Teilkonzepte, über die zu Beginn zu entscheiden ist, gehören Datenhaltung, Benutzerschnittstellen, Fehlerbehandlung, Fehlertoleranz, Sicherheit, ...
- Nach Zusammenstellung aller Vorgaben werden die wichtigen Objekte, Klassen, Funktionsbausteine, ... extrahiert.

Schrittweise Verfeinerung



Intiale Festlegungen

- für Objekte, Klassen, Module, Prozesse, Schnittstellen,
- Kommunikation, Hardware-Konfiguration,

Schrittweise incrementelle Verfeinerung unter Kontrolle der Einhaltung aller Anforderungen. Ergebnis:

- Zu codierende Bausteine.
- Schnittstellen und ableitbare Beispiele für die Modul- und Integrationstests.

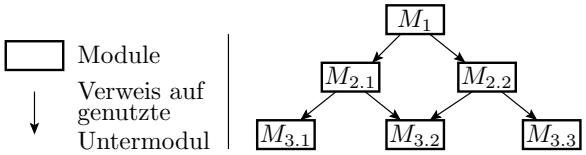
1.3 Codierung und Test

Hilfreiche Funktionen einer Entwurfsumgebung

- Statische Kontrollen bei der Übersetzung.
- Eincompilieren von Kontrollen und Fehlerbehandlung für unzulässige Aktivitäten (Division durch null, WB-Überläufe, ...).
- Fehlerisolation und Ausschluss nicht autorisierter Zugriffe auf fremde Daten.
- Unterstützung Durchführung und Archivierung von Tests.
- Versionsverwaltung für Regressionstest und den Rückbau in Fehlerbeseitigungsiterationen.
- Debugger mit Haltepunkten, Schrittbetrieb und Lese-/Schreibzugriff auf die Daten.
- Trace- und Event-Aufzeichung. Auffinden von totem Code, ...
- Unterstützung bei der Bestimmung von Code- und Fehlerüberdeckungen,
- Refractionig (Änderung von Bezeichnern).
- Unterstützung bei der Erstellung von Dokumentationen, auch für Reviews, Änderungen, ...

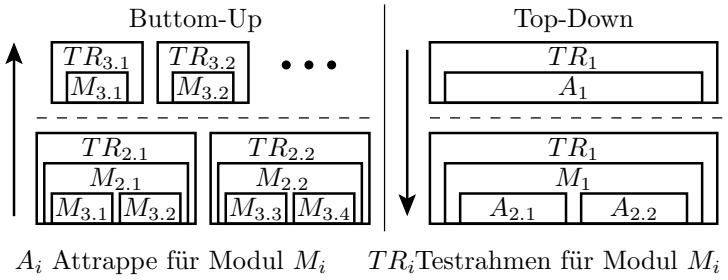
Bottum-up oder Top-down

Jeder Code-Baustein muss ausprobiert werden:



Strategien für die Entwurfs- und Testreihenfolge:

- Bottom-Up: Beginn mit dem Entwurf und Test der untersten Module. Test der übergeordneten Module mit den bereits getesteten Untermodulen.
- Top-Down: Beginn mit dem Entwurf übergeordneter Module und Test mit Attrappen für die Untermodule. Schrittweise Ersatz der Attrappen durch getestete Untermodule.



A_i Attrappe für Modul M_i TR_iTestrahmen für Modul M_i

Praktisches Vorgehen:

- erst beispielbasierte Tests mit Ergebnisausgabe, um das Testobjekt zu untersuchen,
- dann Erweiterungen auf zielgerichtete Kontrolle zuzusichernder Eigenschaften,
- Ergänzung Fehlerbehandlung im Testobjekt und Tests dafür,
- dann Fussifizierung um ungewollte Eigenarten aufzudecken.

Je mehr Attrappen der Test erfordert, um so schlechter ist der Code.

Regeln für die Codierung

- Einfach, ohne überflüssige Schnörkel. Gut testbar. Gut änderbar.
- Verzicht auf Code für eventuelle künftige Erweiterungen, weil das voraussichtlich toter Code wird.
- Ausnahme Schnittstellen, weil nachträgliche Schnittstellenänderungen viel Nacharbeit mit hohem Fehlerentstehungsrate bedeuten.
- Wenn man das dritte mal dasselbe Stück Code schreibt, ist es Zeit für die Auslagerung in eine Hilffunktion, weil dann etwa klar ist, wie diese aussehen muss.
- Tests immer nach dem Prinzip »Fail Fast« programmieren, d.h. mit strengen Kontrollen und Abbruch bei FF.
- Sorgfäliger Entwurf externer Schnittstellen auch mit Rücksicht auf künftige Verwendung.
- Größenbegrenzungen: Funktionen ≤ 30 NLOC, Modul ≤ 500 NLOC, je schlechter testbar (z.B. nicht im Schrittbetrieb) um so kleiner und übersichtlicher.
- Fokus zuerst auf Korrektheit, dann erst auf Schnelligkeit.
- Codierung nur der benötigten Funktion statt Universallösungen mit einer Komplexität, die nicht erforderlich ist.
- Wenn ein Test versagt, zugrundeliegende Fehler sofort suchen beseitigen.
- Zum Test der Tests sollte jeder Test einmal mit einem wohlüberlegten Bug im Testobjekt zum versagen gebracht werden.
- ...

»Anti-Patter«

Das sollte man vermeiden:

- Big ball of mud: Ein System ohne erkennbare Struktur.
- Eingabe-Hack: Mögliche ungültige Eingaben nicht behandelt.
- Schnittstelle überladen: So überdimensioniert, dass die Implementierung extrem schwierig wird.
- Programmierarbeit, die mit besseren Werkzeugen vermeidbar wäre.
- Nutzung von Programmiermustern und Methoden, ohne sie zu verstehen.
- Benutzung von Konstanten ohne Erleuterung. ...

MISRA-Standard

Insgesamt über 100 Regeln, zum Teil verpflichtend, zum Teil Empfehlungen für C-Programme für Automotive-Anwendungen:

- Bezeichnerlänge max. 31 Zeichen (längere Bezeichner werden von manchen Compilern nach 31 Zeichen abgeschnitten, Risiko, dass Compiler unterschiedliche Variablen zu einer zusammenfasst.
- Unterschiedliche Bezeichner für unterschiedliche Objekte:

```
int16_t i; {
 int16_t i; // Hier zwei Variablen i definiert.
            // Nach MISRA-Standard unzulässig.
 i = 3;     // Denn, welche ist hier gemeint?
}
```

- Jeder Variablen ist vor ihrer Nutzung ein Wert zuzuweisen, ...

Vermeidung unsicherer Konstrukte

Die bekannteste Funktion, die Sicherheitslücken in C-Programmen verursacht, ist die Bibliotheksfunktion

```
char * strcpy(char *dest, char *src);
```

beim Kopieren von Eingabezeichenketten in einen Puffer auf dem Stack. Wegen der fehlenden Längenkontrolle lässt sich damit auf der Stack hinter dem Puffer überschreiben und die Rücksprungadresse der Funktion verändern.

Problemvermeidung durch statische Code-Analyse:

- Suche alle Aufrufe von strcpy (die Eingabedaten in Puffer kopieren).
- Ersatz durch

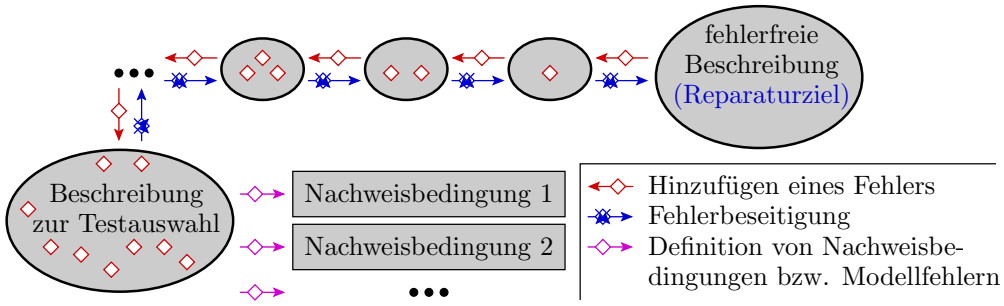
  ```
  char *strncpy(char *dest, char *src, int n);
  ```

  n – Puffergröße.

2 Testauswahl

2.1 Mutationen

Mutationen statt Modellfehler



- Testauswahl für eine fehlerhafte Entwurfsbeschreibung.
- Statt der Modellfehler lassen sich nur Mutationen einer *potentiell fehlerhaften Beschreibung* konstruieren.
- Für vergessene Aspekt lassen sich keine ähnlich nachweisbaren Mutationen ableiten.
- Gleichfalls nicht erzeugbar sind simulierbare Mutationen für Nicht-Code-Beschreibungen (Anforderungslisten, ...).

Mutationen für Programme

Mutationen auf der Hochsprachenebene sind geringfügige Verfälschungen im Programmtext:

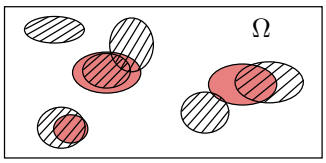
- Verfälschung arithmetischer Ausdrücke (x=a+b ⇒ x=a*b)
- Verfälschung boolescher Ausdrücke (if(a>b){} ⇒ if(a<b){})
- Verfälschung der Wertezuweisung (value=5 ⇒ value=50)
- Verfälschung der Adresszuweisung (ref=obj1 ⇒ ref=obj2)
- Entfernen von Schlüsselworten (static int x=5 ⇒ int x=5)

Bestimmung der Modellfehler- (Mutations-) überdeckung:

- Wiederhole für jede Fehlerannahme:
  - Erzeuge mutiertes Programm und übersetze.
  - Teste, bis zur ersten erkennbaren Ausgabeabweichung zwischen Mutation und Original oder bis Testsatz abgearbeitet.

Kostet viel Rechenzeit, ist aber prinzipiell durchführbar.

Gezielte Testauswahl



Ω Menge der Eingabewerte / Teilfolgen die einen Fehler nachweisen können

Nachweismenge eines Modellfehlers

Nachweismenge eines tatsächlichen Fehlers

Gezielte Testauswahl sucht für jede Mutation mindestens eine Eingabe aus deren Nachweismenge. Die Wahrscheinlichkeit für den Fehlernachweis hängt von den Überschneidungen der Nachweismengen ab.

Fehlende Anforderungen, Ausnahmebehandlungen, ... teilen sich keine Nachweisbedingungen mit mutierten Anweisungen und bleiben so bei der Testauwahl unberücksichtigt.

Für potentielle Fehler vom Typ »fehlt in der fehlerhaften Beschreibung« ist keine gezielte Testauswahl möglich.

Zufällige Testauswahl (Fuzzing)

- Alle potentiellen Fehler und alle Mutationen haben Nachweismengen, die sich mehr oder weniger überschneiden.
- Trotz des Fehlens ähnlich nachweisbarer Mutationen ist eine vom Fehlermodell abhängige Testzeitskalierung zu erwarten (vergl. Foliensatz 2, Abschn. 2.4 Isolierter Test):

$$\begin{aligned} h(\zeta) &\sim h_{\mathrm{Mut}}(c\cdot\zeta) \\ \mathbb{E}[FC(n)] &\approx \mathbb{E}[FC_{\mathrm{Mut}}(c\cdot n)] \end{aligned}$$

Zufallstest sind auch für den Fehlertyp »fehlt in der fehlerhaften Beschreibung« geeignet. Software und Hardware-Entwürfen sollten immer einem ausreichend langen Zufallstest unterzogen werden.

Fehlende Sollfunktion

Das Sollergebnis für einen Test ergibt sich aus der Zielfunktion, nicht aus der Umsetzung.

Erfordert eine diversitäre Sollwertberechnung. Maskierung durch übereinstimmende Fehlfunktionen schwer ausschließbar.

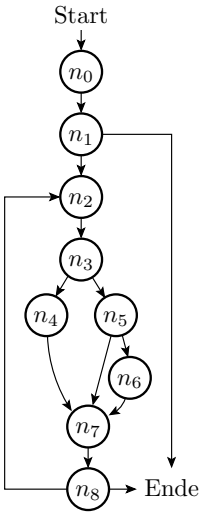
Konzeptionelle Ansätze:

- Erstellen der Testfälle und Sollwerte vor dem Entwurf, unabhängig vom Entwurf, durch getrennte Personen, ...
- Zusatzkontrollen der Testergebnisse: Format, Plausibilität der Werte, ...
- Entwicklung disversitärer Lösungsbeschreibungen zur Testauswahl und/oder Sollwertbestimmung.
- Review (aufgezeichneter) Istwerte.
- ...

2.2 Kontrollfluss

Ein Beispielprogramm und sein Kontrollflussgraph

```
int Ct_A, Ct_B, Ct_N;
int ZZ(int Ct_max){
    char c;
n0: Ct_A=0; Ct_B=0; Ct_N=0;
n1: while (Ct_N<Ct_max){
n2:  c=getchar();
n3:  if (is_TypA(c))
n4:   Ct_A++;
n5:  else if (is_TypB(c))
n6:   Ct_B++;
n7:  Ct_N++;
n8: } //Test Abbruchbedingung
}
```



Auswahlkriterien für Tests

1. Anweisungsüberdeckung: Jede Anweisung muss mindestens einmal ausgeführt werden. Beispiel: Start, n_0, n_1, n_2, n_3, n_4, n_7, n_8, n_2, n_3, n_5, n_6, n_7, n_8, Ende

2. Kantenüberdeckung: Jede Kante muss mindestens einmal durchlaufen werden. Beispiel: Start, n_0, n_1, n_2, n_3, n_4, n_7, n_8, n_2, n_3, n_5, n_6, n_7, n_8, n_2, n_3, n_5, n_7, n_8, Ende

3. Entscheidungsüberdeckung: Jede Entscheidung muss mindestens einmal von jeder Bedingung abhängen.

Beobachtbarkeit verfälscher Anweisungsergebnisse nicht gefordert, d.h. Fehlerannahmen besser als zu erwartende Fehler nachweisbar.

Bestimmung der Überdeckung bei manueller/zufälliger Testauwahl:

- Die Anweisungs- und Kantenüberdeckung lässt sich durch Einfügen von Zählern in das Programm vor der Compilierung bestimmen.
- Automatisierbar.

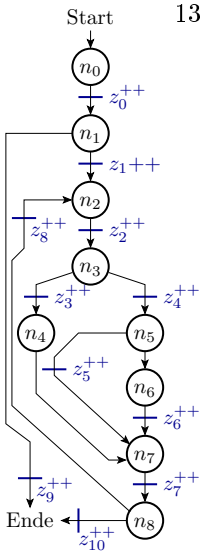
Kontrolle der Testergebnisse:

- einprogrammierte Überwachungsfunktionen,
- Trace-Aufzeichnung und Review aufgezeichneter Daten,
- Regressionstest.

Erweiterung der Kontrolle auf Anweisungsergebnisse möglich:

- Schrittbetrieb und manuelle Kontrolle der Zwischenergebnisse,
- Trace-Aufzeichnung und Inspektion aller Zwischenergebnisse.
- Einbeziehung der Beobachtbarkeit (siehe nächster Abschnitt).

Bestimmung der Kantenüberdeckung



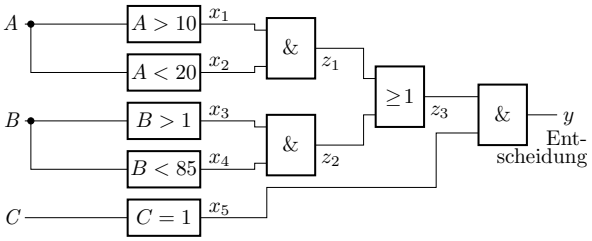

```
int z[11]={0,0,0,0, ...};
...
int ZZ(int Ct_max){char c;
n0: Ct_A=0;Ct_B=0;Ct_N=0;z(0)++;
n1: while (Ct_N<Ct_max){ z(1)++;
n2:  c=getchar(); z(2)++;
n3:  if (is_TypA(c)){
n4:   z(3)++; Ct_A++;}
      else {z(4)++;
n5:   if (is_TypB(c)){
n6:    Ct_B++; z(5)++;}
      else z(6)++;
     }
n7:  Ct_N++; z(7)++;
n8: ...[1]}
}
```

Bedingungsüberdeckung

Ein logischer Ausdruck, z.B.

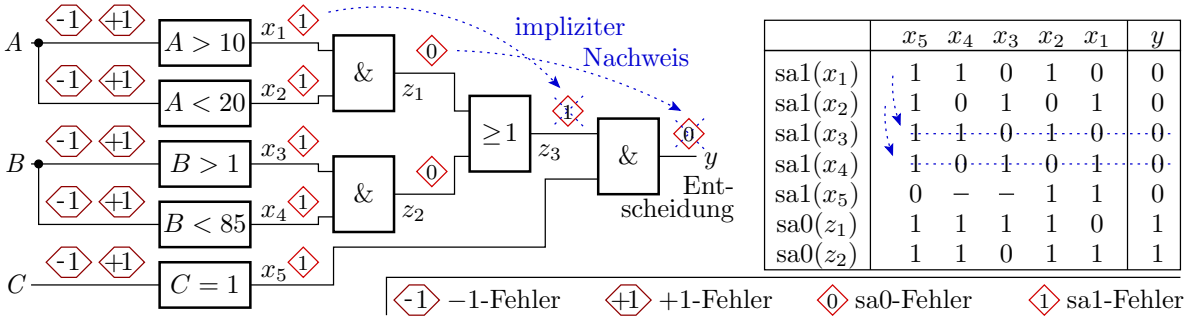
```
n1: if (((A>10) && (A<20)) || ((B>1) && (B<85))
        && (C==1)) {
n2:  ... }
    else {
n3: ...  }
```

ist nachbildbar durch einen Schaltplan aus Gattern und Vergleichern:



Berechnungsfluss mit eingezeichneten Fehlern

Die Bestimmung der Bedingungsüberdeckung lässt sich auf die Modellierung von Haftfehlern und Off-By-One-Fehlern (±1-Fehler) zurückführen.



	x_5	x_4	x_3	x_2	x_1	y
sa1(x_1)	1	1	0	1	0	0
sa1(x_2)	1	0	1	0	1	0
sa1(x_3)	1	1	0	1	0	0
sa1(x_4)	1	0	1	0	1	0
sa1(x_5)	0	–	–	1	1	0
sa0(z_1)	1	1	1	1	0	1
sa0(z_2)	1	1	0	1	1	1

Weiter wie bei Haftfehlern

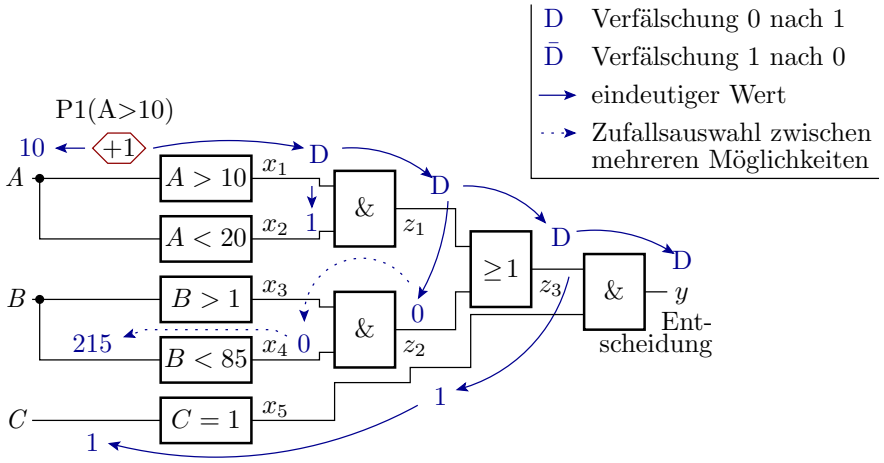
- Zusammenfassen identischer Fehler. Streichen redundanter und implizit nachweisbarer Fehler (vergl. Foliensatz F2, Abschn. 1.3 und dieser Foliensatz, Abschn. 2.2).

- Die ∓1-Fehler implizieren den Nachweis aller Haftfehler nach den Vergleichsoperatoren, ...

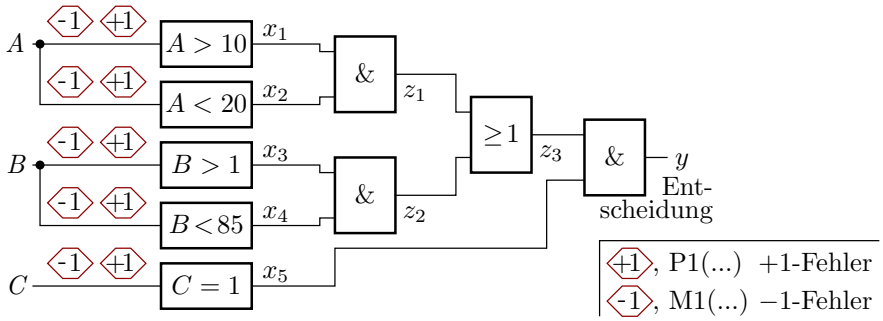
[1] Zur Unterbringung aller Zähler Schleife in Maschinenbefehle auflösen.

- Eventuelle redundante Fehler deuten auf Möglichkeiten zur Programmvereinfachung.
- Fehlersimulation und Testberechnung für die so zusammengestellte Modellfehler- (Mutations-) Menge innerhalb der logischen Ausdrücke könnte mit den für digitale Schaltungen etablierten Verfahren erfolgen.
- Das ist aber noch nicht Stand der Technik.

Testsuche

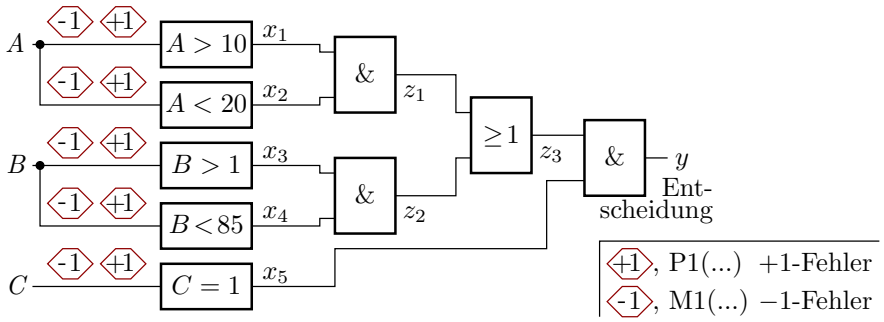


Für den »+1«-Fehler ist am Fehlerort ein Wert einzusetzen, bei dem sich Erhöhnung um eins das Vergleichsergebnis ändert. Vom Vergleichsergebnis wird ein D-Pfad zum Entscheidungsausgang durch zurücktreiben von Steuerpfaden zu Eingängen sensibilisiert.



	A	B	C	x_1	x_2	x_3	x_4	x_5	z_1	z_2	z_3	y
P1(A>10)	10	215	1	D	1	-	0	1	D	0	D	D
M1(A>10)												
P1(A<20)												
M1(A<20)												
P1(B>1)												
M1(B>1)												
P1(B<85)												
...												

Lösung



	A	B	C	x_1	x_2	x_3	x_4	x_5	z_1	z_2	z_3	y
P1(A>10)	10	215	1	D	1	-	0	1	D	0	D	D
M1(A>10)	11	0	1	$\bar{D}$	1	0	-	1	$\bar{D}$	0	$\bar{D}$	$\bar{D}$
P1(A<20)	19	86	1	1	$\bar{D}$	-	0	1	$\bar{D}$	0	$\bar{D}$	$\bar{D}$
M1(A<20)	20	1	1	1	D	0	-	1	D	0	D	D
P1(B>1)	11	1	1	-	0	D	1	1	0	D	D	D
M1(B>1)	23	2	1	-	0	$\bar{D}$	1	1	0	$\bar{D}$	$\bar{D}$	$\bar{D}$
P1(B<85)	19	84	1	0	-	1	$\bar{D}$	1	0	$\bar{D}$	$\bar{D}$	$\bar{D}$
...												

Testanforderungen für heutige Software

Nach Standard DO-178 B gilt als ausreichend[2]:

- 100% Anweisungsüberdeckung für nicht sicherheitskritische Systeme,
- 100% Zweigüberdeckung für Software, die bedeutende Ausfälle verursachen kann,
- 100% Bedingungsüberdeckung für flugkritische Software.

Nach Stand der Technik noch nicht gefordert:

- Beobachtbarkeit der Anweisungsergebnisse an Ausgängen.
- Ergebniskontrolle im Schrittbetrieb.
- Mehrfachausführung zur Erhöhung der Wahrscheinlichkeit der Fehleranregung und Beobachtbarkeit.

2.3 Def-Use-Ketten

Def-Use-Ketten

Def-Use-Tupel: Datenstruktur, die aufeinanderfolgende Paare von Schreib- und Lesezugriffen einer Variable beschreibt. Programmbeispiel »größter gemeinsamer Teiler[3] «:

```
     int ggt(int a, int b){
n0:    int c = a;
n1:    int d = b;
n2:    if(c == 0)
n3:      return d;
n4:    while(d != 0){
n5:      if(c > d)
n6:        c = c - d;
n7:      else
n8:        d = d - c;
       }
n9:  return c;
     }
```

Var	Def	Use
d	n1	n3
d	n1	n4
d	n1	n5
d	n1	n6
d	n1	n8
d	n8	n4
d	n8	n5
d	n8	n6
d	n8	n8
c	n0	n2
...	...	...

[2] Bei »ausreichendem« Test kann sich der Hersteller der Produkthaftung im Falle eines Unfalls durch einen nicht erkannten Fehler entziehen.

[3] Aus https://de.wikipedia.org/wiki/Def-Use-Kette vom 17.10.2015.

Berechnung und Verwendung von Def-Use-Ketten

Berechnung aller Def-Use-Tupel:

```
Wiederhole für alle Lesezugriffe aller Variablen:
    suche die Anweisungen, die den Wert geschrieben haben könnten
```

Verwendung als Testvollständigkeitskriterien:

- Für alle »Defs« mindestens ein »Use«.
- Für alle »Use« mindestens ein »Def«.
- Alle Def-Use-Tupel.
- Def-Use-Überdeckung als Testgüte (wenig populär).

Statische Code-Analyse:

- »Use« ohne »Def« ist ein Initialisierungsfehler.
- »Defs« ohne »Use« sind redundanter Code.

Fehlerlokalisierung:

- Rückverfolgung des Def-Use-Graphen zur Suche der Entstehungsursachen von Verfälschungen.

Beispiel: Rückverfolgung in »größter gemeinsamer Teiler«:

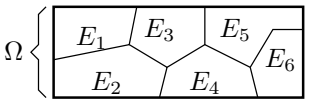
```
int ggt(int a, int b){
n0:   int c = a;
n1:   int d = b;
n2:   if(c == 0)
n3:     return d;
n4:   while(d != 0){
n5:     if(c > d)
n6:       c = c - d;
n7:     else
n8:       d = d - c;
      }
n9:   return c;
    }
```

Wenn »n9« FF, dann sind die möglichen »Defs«, an denen Unterbrechungspunkte beim nächsten Testdurchlauf zu setzen sind, »n0« und »n6«

2.4 Äquivalenzklassen

Testauswahl mit Äquivalenzklassen

- Äquivalenzklasse: Eingabemenge ähnlich zu verarbeitender Daten.
- Fehlerannahme A: Fehler in der Verarbeitung werden mit jedem Beispiel der Klasse mit hoher Wahrscheinlichkeit nachgewiesen.
- Fehlerannahme B: Spezifikations- und Implementierungsfehler sind oft falsch gesetzte Bereichsgrenzen.



Ω Eingaberaum
E_i Äquivalenzklasse

Testauswahl / Basis:

- Fertiges Programm: Testauswahl vergleichbar mit der für »Bedingungsüberdeckung« (siehe Seite 13).
- Spezifikation: Erstellen einer zum Testobjekt diversitären Fallbeschreibung. Weiter wie »Bedingungsüberdeckung«.

Spezifikationsbasierte Testauswahl

1. Zusammenstellung der Eingabedaten, Ausgabedaten, Berechnungsvorschriften und Bedingungen, die bei der Berechnung zu unterscheiden sind.

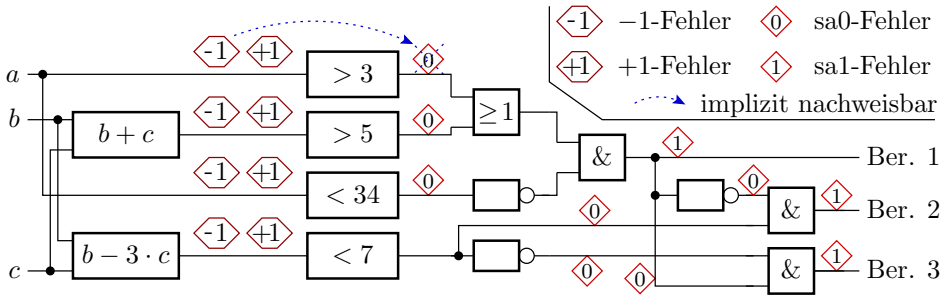
2. Bildung von Äquivalenzklassen durch Unterteilung der Eingabewertebereiche, beschreibbar durch ein Programm mit Fallunterscheidungen und Dummy-Funktionen für die Ausführung.

3. Konstruktion eines Tests mit 100% Anweisungs-, Zweig- oder Bedingungsüberdeckung für die so entstandene Programmbeschreibung.

4. (Manuelle) Sollwertbestimmung entsprechend Eingabebereich und zugeordneter Sollfunktion.

Beispiel einer aus der Spezifikation gewonnen Äquivalenzklassenbeschreibung

```
int fkt(int a, int b, int c){
 if((a>3)||(b+c>5))&& !(a<34))printf("Berechn. 1");
 else if(b-3*c<7)             printf("Berechn. 2");
 else                         printf("Berechn. 3");
}
```



Testauswahl / Überdeckungskontrolle weiter wie »Bedingungsüberdeckung« ab Seite 13.

2.5 UW-Analyse

Ursache-Wirkungs-Analyse

Bei der UW-Analyse wird wie bei dem spezifikationsbasierten Äquivalenzklassenverfahren aus der Zielfunktion eine zum Testobjekt diversitäre Beschreibung abgeleitet.

Der empirische Ansatz ist anders.

Statt nach Wertebereichen und diesen zugeordneten Verarbeitungsfunktionen wird die Zielfunktion sortiert nach:

- Auslösern für Aktionen (Ursachen) und

- ausgelösten Aktionen (Wirkungen).

Auslöser (Ursachen) sind Eingabewertebereiche (ähnlich Äquivalenzklassen), die bestimmte Sollreaktionen zur Folge haben sollen.

Wirkungen sind einzeln spezifizierte Zielfunktionen, ergänzte selbstverständliche Funktionen und Fehlerbehandlungen.

Jede Ursache und Wirkung wird durch eine binäre Variable (nicht eingetreten/eingetreten) beschrieben.

Zwischen den Ursachen und Wirkungen werden logische Verknüpfungen formuliert.

Die Testauswahl selbst ähnelt denen für »Bedingungsüberdeckung« und »Äquivalenzklassen« und lässt sich auch wieder auf die für Haftfehler zurückführen.

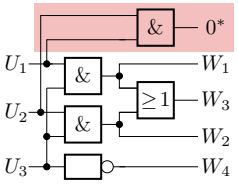
Beispiel »Zähle Zeichen«

- Wirkungen:
  W_1: Anzahl_TypA +1[4]
  W_2: Anzahl_TypB +1
  W_3: Gesamtzahl +1
  W_4: Programm beenden

- Ursachen:
  U_1: Zeichen ist vom Typ A
  U_2: Zeichen ist vom Typ B
  U_3: Zeichenanzahl < Maximalwert

- Sich ausschließende Ursachen: UND-Verknüpfung muss »0« sein.

- Eine Ursache-Wirkungs-Analyse deckt Mehrdeutigkeiten und Widersprüche in der Spezifikation auf.



* Eingabe kann nicht gleichzeitig Typ A und B sein

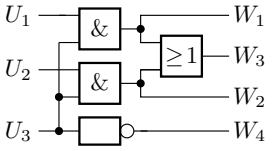
Test mit allen einstellbaren Ursachen

U_1	0	1	0	1	0	1	0	1
U_2	0	0	1	1	0	0	1	1
U_3	0	0	0	0	1	1	1	1
W_1	0	0	0		0	1	0	
W_2	0	0	0		0	0	1	
W_3	0	0	0		0	1	1	
W_4	1	1	1		0	0	0	

Beispielimplementierung als C-Funktion

```
     int Ct_A, Ct_B, Ct_N;

     int ZZ(int Ct_max){
     char c;
     Ct_A=0; Ct_B=0; Ct_N=0;
U3:  while (Ct_N<Ct_max){
       c=getchar();
U1:   if (is_TypA(c))
W1:    Ct_A++;
U2:   else if (is_TypB(c))
W2:    Ct_B++;
W3:   Ct_N++;
W4:   }
     }
```



Test mit allen einstellbaren Ursachen

U_1	0	1	0	1	0	1	0	1
U_2	0	0	1	1	0	0	1	1
U_3	0	0	0	0	1	1	1	1
W_1	0	0	0		0	1	0	
W_2	0	0	0		0	0	1	
W_3	0	0	0		0	1	1	
W_4	1	1	1		0	0	0	

Testbeispiel konkret /symbolisch

Funktions-aufruf	Eingabe	Sollzähl-werte	Ursachen U_1	U_2	U_3	Wirkungen W_1	W_2	W_3	W_4
ZZ(3)	z='0'	A=1 B=0 N=1	1	0	1	1	0	1	0
	z='A'	A=1 B=1 N=2	0	1	1	0	1	1	0
	z='x'	A=1 B=1 N=3	0	0	1	0	0	1	0
		Ende	–	–	0	0	0	0	1
ZZ(1)	z='1'	A=1 B=0 N=1	1	0	1	1	0	1	0
		Ende	–	–	0	0	0	0	1
ZZ(1)	z='B'	A=0 B=1 N=1	0	1	1	0	1	1	0
		Ende	–	–	0	0	0	0	1
ZZ(0)		Ende	–	–	0	0	0	0	1

U_1	Zeichen ist vom Typ A (Ziffer)	W_1	Ct_A++
U_2	Zeichen ist vom Typ B (Großbuchstabe)	W_2	Ct_B++
U_3	max. Zählwert nicht erreicht	W_3	Ct_N++
–	es wird kein Zeichen gelesen	W_4	Ende

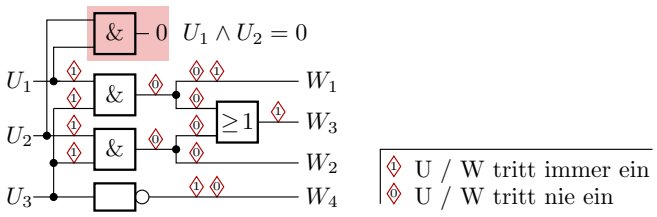
[4] Im Programmbeispiel wird Typ A Ziffer und Typ B Großbuchstabe sein.

Ungereimtheiten / Haftfehler

Erkennbare Ungereimtheiten:

- Im UW-Graph können bei »$U_3 = 0$« (max. Zählwert erreicht) Zeichen vom Type A oder B eingegeben werden, im Programm nicht. Wie lautet das gewünschte Sollverhalten?

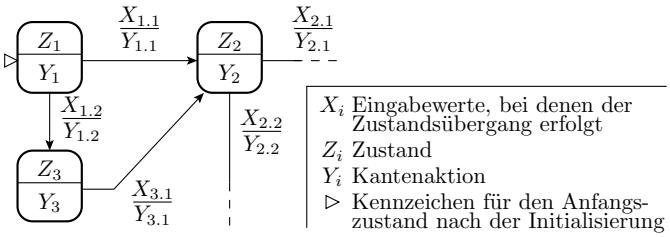
Haftfehler im UW-Graph (identisch nachweisbare Fehler zusammengefasst):



- Im Beispiel würde ein Test mit allen Kombinationen von Ursachen auch alle nachweisbaren Haftfehler erfassen.
- Für eine große Anzahl von Ursachen kann die Anzahl der Haftfehler auch wesentlich kleiner als die Anzahl der Ursachenkombinationen sein.
- Nach Berechnung der gleichzeitig zu (de-) aktivierenden Ursachen folgt die Suche geeigneter Eingaben und Kontrollen.

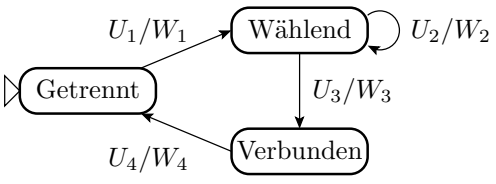
2.6 Automaten

Zielfunktion als Automat



Das Automatenmodell beschreibt die Zielfunktion eines Systems durch Mengen von Eingaben, Ausgaben, Zuständen und Zustandsübergängen. Zustandsübergänge werden durch Eingaben ausgelöst. Bei den Übergängen und in den Zuständen werden Aktionen gesteuert. Wie im UW-Modell werden bei Automaten für die Testauswahl die Ursachen (Bedingungen für Zustandsübergänge) und die Wirkungen (gesteuerte Aktionen) binarisiert.

Verbindungsaufbau und -abbau beim Telefonieren

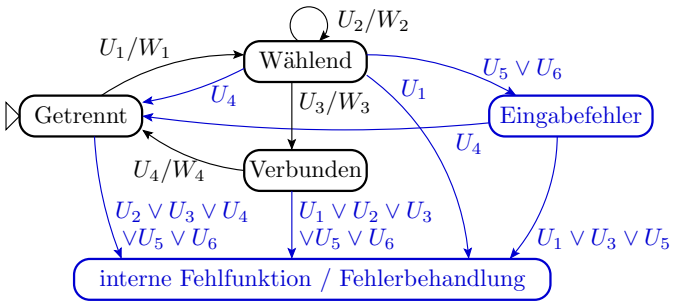


U_1	Abnehmen
U_2	Ziffer wählen
U_3	Rufnummer gültig
U_4	Auflegen
W_1	Rufnummer zurücksetzen
W_2	Ziffer zur Rufnummer hinzufügen
W_3	Verbindung aufbauen
W_4	Verbindung trennen

- Test der Sollfunktion: $U_1 \rightarrow U_2 \rightarrow \ldots \rightarrow U_2 \rightarrow U_3 \rightarrow U_4$
- Verhalten für andere Eingabefolgen?

- Abnehmen, Wählen, Auflegen ($U_1 \rightarrow U_2 \rightarrow U_4$)
- Abnehmen, Wählen, Wählen, falsche Nummer)
- ...

⇒ Ablaufgraph ist noch unvollständig



U_1	Abnehmen	W_1	Rufnummer zurücksetzen
U_2	Ziffer wählen	W_2	Ziffer zur Rufnummer hinzufügen
U_3	Rufnummer gültig	W_3	Verbindung aufbauen
U_4	Auflegen	W_4	Verbindung trennen
U_5	Rufnummer ungültig		
U_6	Timeout		

- Ergänzung um Knoten und Kanten für alle denkbaren Ursachen und Wirkungen. Präzisierung der Spezifikation.

Test aller Zustandsübergänge, Wirkungen, ...

- Abheben,Wählen, Wählen, Rufnummer gültig, Auflegen.
- Abheben, Wählen, Auflegen.
- Abheben, Wählen, Wählen, Timeout, Auflegen.
- Abheben, Wählen, Rufnummer ungültig, Auflegen.

Test der Reaktion auf interne Fehlfunktionen

- Initialisieren, Auflegen.
- Initialisieren, Rufnummer gültig, ...

Auswahlregeln sind wie bei der kontrollflussorientierten Auswahl:

- Ausprobieren aller Kanten (in Analogie zu 100% Zweigüberdeckung) oder
- jeder Übergang muss mindestens einmal von jeder Bedingung abhängen (Analogie Bedingungsüberdeckung, zurückführbar auf das Haftfehlermodell).

Aus einem Automatengraphen sind wie bei der UW-Analyse nur Rahmenvorschriften für die Konstruktion der eigentlichen Testbeispiele ableitbar, nämlich Folgen von auszulösenden Ursachen für die Kantenübergänge und erwartete Wirkungen in Form der den Kanten und Zuständen zugeordneten Aktionen.

Der zufällige Fehlernachweis für Automaten wird durch Markov-Ketten beschrieben (vergl. Foliensatz TV_F1).

Literatur

[1] T. Ball. Thorough static analysis of device drivers. In *EuroSys*, pages 73–85, 2006.