Test und Verlässlichkeit Foliensatz 6: Software

Prof. G. Kemnitz

6. November 2022

[17. Vorlesung]

Contents

Vorlesung	1	2	3	4
bis Abschn.	2.?? (??)	4.?? (??)	4.?? (??)	5.?? (??)

Software

Der Begriff SW wurde 1958 von John W. Tukey als Gegenstück zu dem wesentlich älteren Begriff Hardware geprägt für

- Programme, Einstellungen,
- HW-Konfigurationen, ...

Erweiterung um Dokus des SW-Entstehungsprozesses, in denen Fehler entstehen, gesucht, vermieden und beseitigt werden:

- Spezifikation, Entwurfsskizzen,
- Quellcode, Dokumentationen, ... [Doku-Fehler ⇒ Phantomf. ⇒ SW-Fehler]

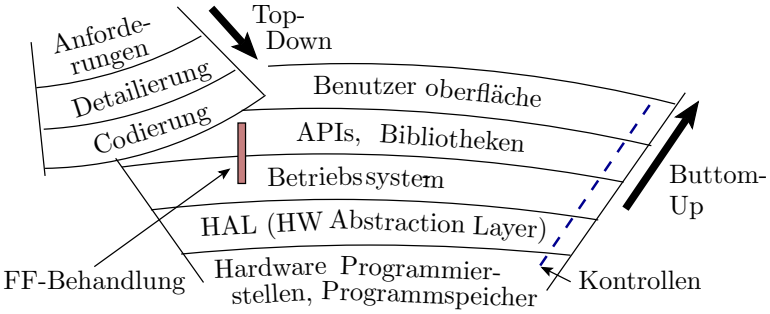
Manuelle SW-Entstehungsschritte:

- kreative Teile, nicht deterministisch,
- hohe Fehlerentstehungsrate, Kontrolle durch Review, ...

Automatisierte SW-Entstehungsschritte:

- Compilieren aus Quellcode,
- Generierung z.B. für Parser, Oberflächen, Softprozessoren,
- deterministisch, geringe Fehlerentstehungsrate, ...

Wirksamste Fehlervermeidung: Automatisierung



- Software legt funktionale Schichten über die Hardware,
- ist selbst in Schichten organisiert.
- Jede Schicht erbt die Funktionalität und die Fehler der darunter.

Der Entwurf, in dem die Fehler entstehen, ist ein Prozess zunehmender Detaillierung:

- Sammlung von Anforderungen, Use-Cases, Testbeispielen,
- Entwurfsentscheidungen für die HW, nachnutzbare SW, Programmiersprache, SW Architektur, ...
- Modularisierung und Schnittstellenfestlegung, Codierung, ...

1 Fehlervermeidung

Der Weg zu fehlerarmer, verlässlicher Software

Fehlervermeidung:

- Arbeitsvermeidung, Automatisierung, [Demonstrator, Produktiv-Code]
- Nachnutzung gereifter (fehlerarmer) Software-Bausteine,
- gereiftes Vorgehensmodell, [Good Practice, Stellschrauben]

Test und Fehlerbeseitigung:

- prüfgerechter (testfokusierter) Entwurf, [frühe Entscheidung]
- geeignete Software-Architektur, ...
- Werkzeugunterstützung
  - statischer Tests: Syntax, Wertebereiche, API-Regeln, ...,
  - dynamischer Test: Ein-Compilieren & Ausführung, ...,
  - Abschätzung Testgüte: Anweisungsüberdeckung, toter Code, ...
  - Fehlerlokalisierung: Debugger, Trace-Tools, ...
  - Built-Prozess, Versionsverwaltung, Rückbau, ...

Umgang mit FF:

- Assert, kontrollierter Programmabbruch, Fehlermeldungen mit Quell-Dateiname, Zeilennummer, Aufrufstack, ...,
- Debug-Built $\Rightarrow$ Fail Fast; Release-Built $\Rightarrow$ Fail Slow, ...

1.1 Software-Architektur

Software-Architektur

- Fehlervermeidung √
- Test und Fehlerbeseitigung √√
- Umgang mit FF √

Eine Software-Architektur gibt einen Rahmen vor für

- Aufteilung eines Systems in Teilbausteine,
- die Gestaltung der Schnittstellen zwischen den Teilsystemen

und bestimmt:

- Entwurfsaufwand, Testbarkeit, Änderbarkeit, Wartbarkeit,
- Wiederverwendbarkeit, Aufwand für nachträgliche Änderungen, ...

Grobeinteilung:

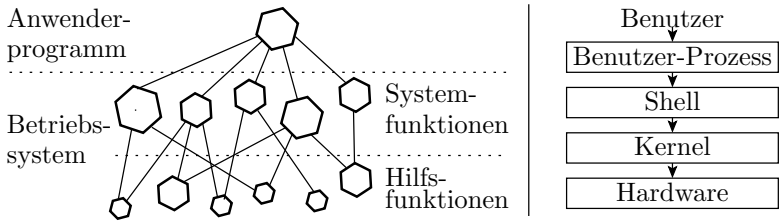
- Prozedurensammlung, Schichtenmodell,
- Client/Server-Model

Je komplexer das System, desto wichtiger ist eine (auch für Nichtautoren) verständliche Struktur.

[Änderung nach Jahren, Personalwechsel, Kommunikation, Review]

Prozedurensammlung

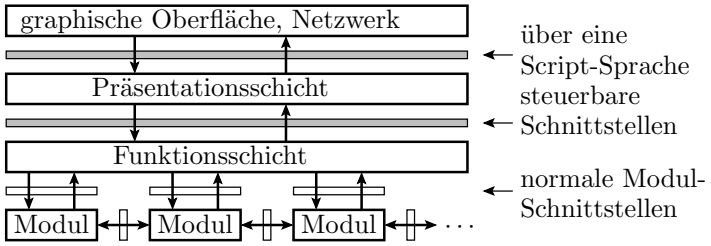
Am wenigsten restriktive SW-Architektur ist die Aufteilung der Gesamtfunktion in eine Sammlung von Prozeduren (Funktionen):



Eine Prozedurensammlung bietet Schnittstellen, aber ein Programmierer muss sich nicht an diese Schnittstellen halten.

[Bsp. Betriebssystem IO- und Speicherzugriff, aber auch direkter HW-Zugriff möglich]
[keine erkennbare Struktur, Big ball of mud, ein Anti-Pattern]

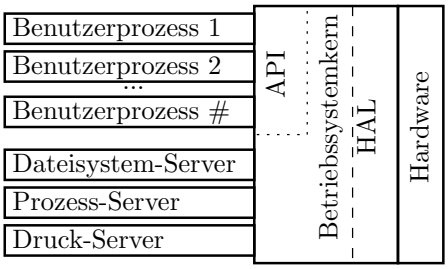
Schichtenmodell



Reglementierung der Benutzung von Prozeduren:

- Die Prozeduren sind alle einer Schicht zugeordnet.
- Einer höhere Schicht - z.B. Benutzeranwendung wie Excel oder Word - kann nur Prozeduren der Schicht darunter über eine wohl definierte Schnittstelle (API) nutzen.
- Ein Kommunikationsprogramm kann z.B. nicht direkt auf den COM-Port zugreifen. Die Applikation stellt eine Anfrage an das Betriebssystem, ob COM-Port verfügbar, ...
- Programme größer, langsamer, ... als bei Prozedurensummlung. Test, Nachnutzung, FF-Behandlung und Wartung einfacher.

Client/Server-Modell



Server sind autonom arbeitende Dienstprogramme, Clients sind Applikationen, die Dienste der Server nutzen.

Beispiel Client/Server-Betriebssystem: Datei-, Prozess- und IO-Verwaltungen als Server. Kleiner Betriebssystemkern (Mikrokernel) für die Kommunikation zwischen den Servern, Clients und HAL.

CS-Systeme sind flexibel und leicht auf andere Plattformen portierbar.

[Fehlerisolation, Nachnutzung]

Schicht als Test- und FF-Behandlungsschnittstelle

Schichten- und Server-APIs sind wohl definierte Schnittstellen, in die sich gut verlässlichkeitssichernde Funktionen unterbringen lassen:

- Überwachung: Trace-Aufzeichung, Stream-Monitor, ...
- FF-Behandlung: kontrollierter Abbruch mit Fehlermeldung, Aufbewahrung der Daten zur Nachstellung der FF für die Fehlerbeseitigung, ...
- Test: Debugger, Script-Sprache zum isolierten Test der auf der Schicht aufsetzenden Module, ...

Schichten mit Script-Sprachen:

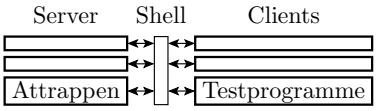
- Betriebssystem-Shell, z.B. unter Linux

```
ls -l *.pdf # Ausführen ls('-l', '*.pdf')
```

- Windows Low-Level Benutzerinteraktion:

```
send_xevents keydn Control_L
send_xevents keyup Control_L
```

[Testprogramme, Attrappen, verteilte HW]



Schichten mit Script-Sprachen (Fortsetzung):

- Bei unserem FPGA-Entwurfssystem ISE kann man alle Entwurfswerkzeuge statt über die Graphik-Oberfläche auch über eine ISE-eigene Konsole mit einer eigenen Skript-Sprache steuern: Compilieren, Routen, ...

Weitere Vorteile von Schichten- und Client/Server-Strukturen:

- Fehlerisolation,
- Portier- und Austauschbarkeit, ...

  Je komplexer die Systeme, desto mehr Planungsaufwand gehört in die SW-Architektur.

[Testprogramme, Attrappen, verteilte HW]

1.2 Entwurfsablauf

Entwurfsablauf

- Fehlervermeidung √√
- Test und Fehlerbeseitigung √
- Umgang mit FF √

Vereinheitlichtes Vorgehens durch Vorgehensmodelle (vergl. Foliensatz F1), damit :

- ähnlich oder vergleichbare Abläufe oft wiederholt werden, um
- dabei aus erkannten Fehlern zu lernen.

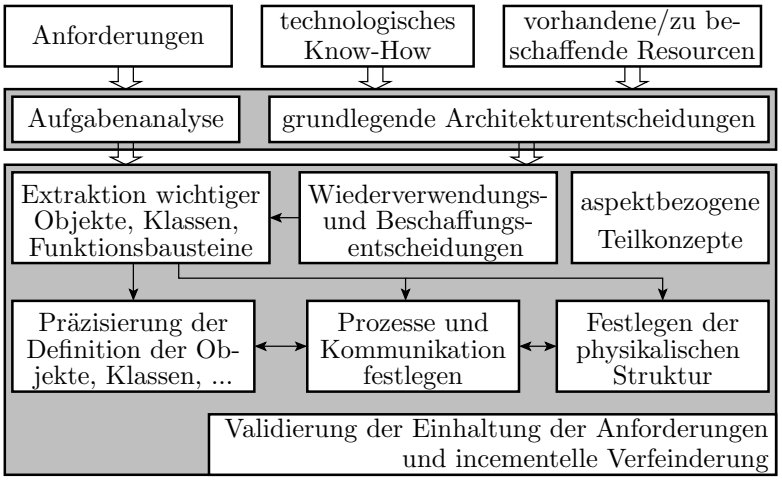
Good Practice (bewährte Techniken):

- Stufenmodell mit Zwischenkontrollen,
- Begrenzung/Minderung der Rückgriffhäufigkeit, ...

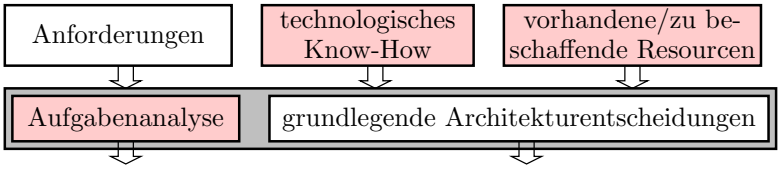
Stellschrauben:

- Reihenfolge Entwurfsentscheidungen, Aufgabenteilung,
- Verantwortlichkeiten, Zwischenkontrollen,
- Arbeits- und Fehlerkultur, kreative Freiräume,
- verwendete Sprachen, Bibliotheken, ...

Beispielablauf: Anforderung bis Architektur



Aufgabe und Ressourcen analysieren



Aufgabenanalyse:

- Wie lässt sich die Aufgabe lösen?
- Was braucht man dafür für Hardware, Entwicklungszeit?
- ...

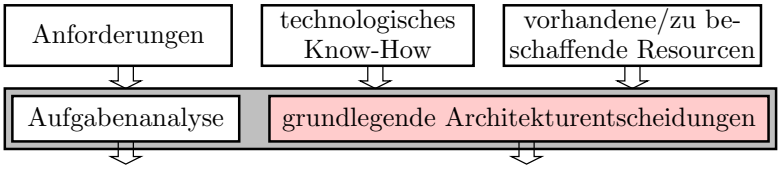
Technologisches Know-How:

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

vorhanden/zu beschaffen:

- Rechner, Software, Personal, ...

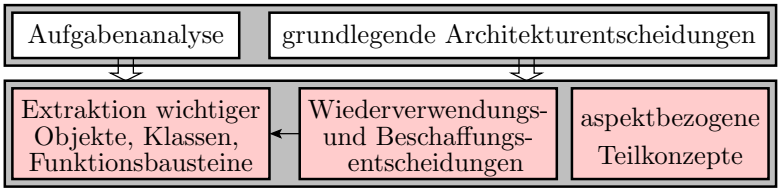
Grundsätzliche Architekturentscheidungen



Nach den Analysen folgen grundlegende Entscheidungen:

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

Detaillierung der Entscheidungen



Wiederverwendung und Beschaffung:

- Komponenten aus früheren Entwürfen,
- Vergabe von Unteraufträgen, ...

Aspektbezogene Teilkonzepte, über die zu Beginn zu entscheiden ist:

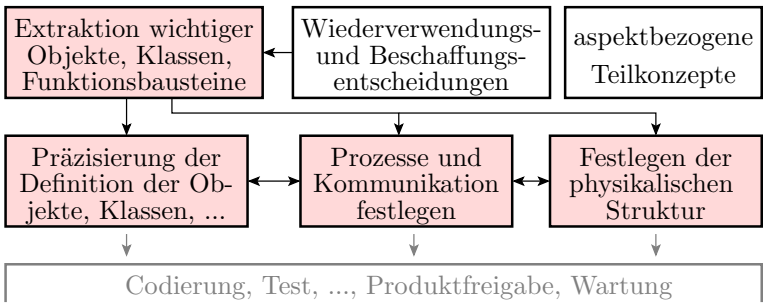
- Datenhaltung, Benutzerschnittstellen,
- Fehlerbehandlung, Fehlertoleranz, Sicherheit, ...

impliziere Vorgaben die bei der Extrahierung

- wichtigen Objekte, Klassen,
- Funktionsbausteine, ...

zu berücksichtigen sind.

Schrittweise Verfeinerung



Incrementelle Verfeinerung der intialen Festlegungen für

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

unter Kontrolle der Anforderungen verfeinert. Ergebnis:

- Schnittstellen + Zielfunktionen für Programmieraufgaben und
- Beispiele für die Modul- und Integrationstests.

Neue Trends

- DevOps (Developement and Operation): Erweiterung der Vorgehensmodelle um Einsatzfreigabe und Reifeprozess, insbesondere auch die Werkzeugunterstützung dafür (Built-Prozess, Versionsverwaltung, ...)
- TDD (Test Driven Developement): Beginn der Entwicklung neuer Funktionen mit Testbeispielen, die auch als eine Art Spezifikation gesehen werden [DD16].
- BDD (Behavioral Drive Developement): Bescheibung der Gesamtfunktion durch unabhängig oder nacheinander entwickelbare Zielfunktionen. Herausforderungen für Software-Architektur, Fehlerisolation, ... [Wanja Bec03, Sma14, EAD14]

Trend: Fehlerentstehungsursachen⇓, Automatisierung⇑, Nachnutzung⇑, Testbarkeit⇑, Fehlerisolation⇑, ...
Es gibt nicht das optimale Vorgehensmodell, sondern gutes Vorgehen entsteht durch lange Lernprozesse.

1.3 Testbare Anforderungen

Testbare Anforderungen

- Fehlervermeidung $\surd$
- Test und Fehlerbeseitigung $\surd\surd$
- Umgang mit FF ($\surd$)

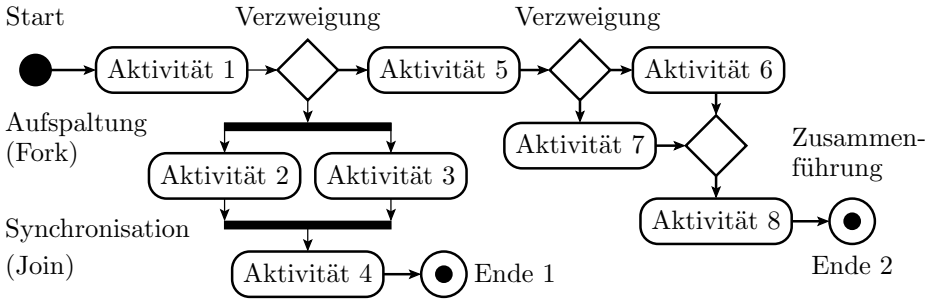
Beschreibung Zielfunktion als Sammung testbarere Anforderungen
für das Gesamtsystem, Module, Schichten, Server, Clients, ...
V-Modell: Zuordnung von Tests zu jeder Entwurfsstufe:

- Anforderungsanlyse: Abnahmetests
- Zwischenschritte: Tests für Teilsysteme und Interaktionen
- Codierung: Modultests.

Modellierungssprache UML zur Spezifikation, Konstruktion, Dokumentation und Visualisierung von Software-Teilen:

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

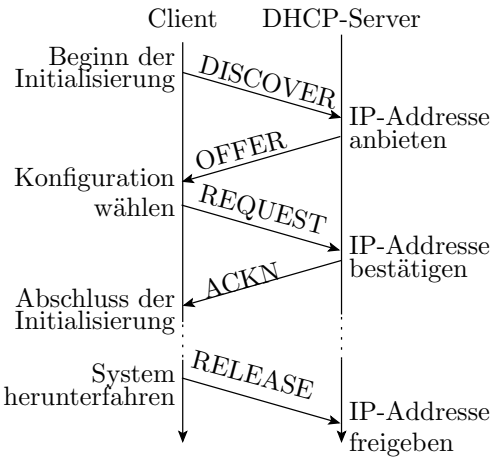
Aktivitätsdiagramm



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

- Start, A1, A2‖A3, A4, Ende 1
- Start, A1, A5, A7, A8, Ende 2
- Start, A1, A5, A6, A8, Ende 2

Sequenzdiagramm



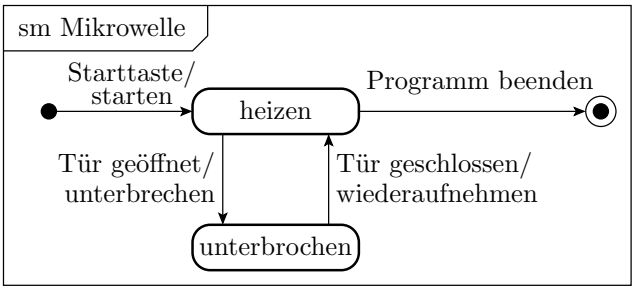
Nachricht	Beschreibung
DISCOVER	Broadcast eines Clients, der einen Server sucht
OFFER	Antwort eines Servers mitKonfigurationsvorschlag
REQUEST	Braodcast des Clients an den bevorzugten Server Ablehnung aller anderen Server
ACKN	Server liefertIP-Adresse
NAK	Der Server lehnt die IP-Adresse ab
DECLINE	Der 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. 2.5 »UW-Analyse«).

Zustandsdiagramm



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

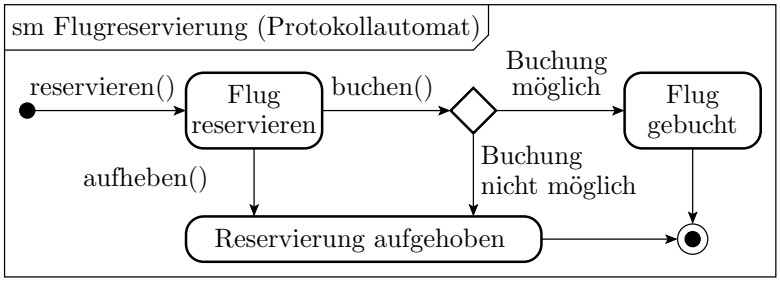
- Zustände,
- Kanten mit Bedingungen für Zustandsübergänge,
- Zuständen und/oder Kanten zugeordnete Aktivitäten.

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.

(siehe später Abschn. 2.6 »Automaten«).

Protokollautomat

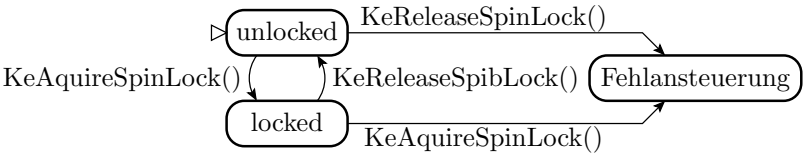


Beschreibung zulässiger Aktionsreihenfolgen. Mögliche Aktionen im Beispiel sind die Methodenaufrufe »reservieren()«, »aufheben()« und »buchen()«. Aus dem Protokollautomat im Beispiel geht hervor, dass ein Flug nur nach erfolgreicher Reservierung gebucht und dass ein einmal gebuchter Flug nicht gestrichen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

[Das Beispiel ist einer der statischen Tests für Gerätetreiben unter Windows, damit die Treiber von Microsoft als unbedenklich für die Zuverlässigkeit des Betriebssystemsh eingestuft werden.]

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.

[Fehler in uralten Treibern gefunden]

```
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();
}
```

Zusammenfassung

Die betrachteten UML-Modelle für Anwendungsszenarien sind anschauliche Beschreibungselemente für das gewünschte Verhalten und gleichzeichig eine gute Basis, um daraus

- Tests und Überwachungsfunktionen zu gewinnen bzw.
- die Vollständigkeit der Test zu überprüfen.

Zuverlässigkeit ist, wie in Vorabschnitten gezeigt, proportional zur Testanzahl durch die Anzahl der nicht beseitigten Fehler mal Anteil der nicht erkennbaren FF. Höhere Zuverlässigkeit ⇒ mehr Testen und mehr überwachen.

Die ganzen Ansätze, Zielfunktionen so zu beschreiben, dass sich daraus auch Tests, Überwachungsfunktionen, ... ableiten lassen (UML, aber auch Test / Behavioral Drive Developement, V-Modell) haben dafür Potential.

1.4 Codierung und Test

Codierung

Fehlervermeidung $\surd$

- bei der Codierung entstehen ca. 30% und
- ca. 10 bis 100 Fehler je 1000 NLOC.
- Automatisierung, ...

Test und Fehlerbeseitigung $\surd\surd$

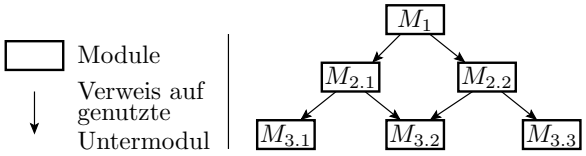
- Modularisierung und Testrahmen,
- Hilfreiche Funktionen einer Entwurfsumgebung,
- Automatisierung, ..

Sowohl Fehlervermeidung als auch Fehlerbeseitigung

- Regeln für die Codierung (Good Practice),
- Anti-Pattern, ...

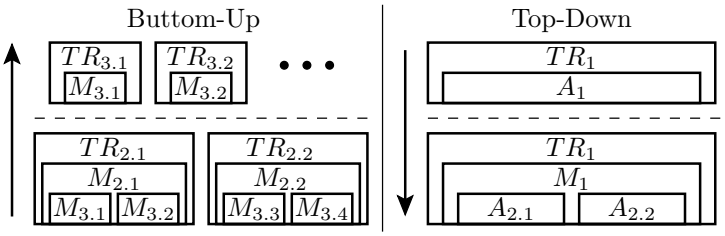
Modularisierung und Testrahmen

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.

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,
- Refactoring (Änderung von Bezeichnern).
- Unterstützung bei der Erstellung von Dokumentationen, auch für Reviews, Änderungen, ...

Regeln für die Codierung (Good Practice)

- 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-Pattern«

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

[C fast Assembler, Preis »böse« Fehler: Pointer, Speicherlecks, Sicherheitslücken,...]

MISRA: Insgesamt über 100 Regeln für C-Programme für Automotive zur Vermeidung »böser« Fehler, zum Teil verpflichtend, zum Teil Empfehlungen:

- 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.
 i = 4;
}
i = 3;       // Welchen Wert hat welche Variable i?
```

- 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.

- ...

[18. Vorlesung]

2 Testauswahl

Testauswahl für Software

Das Gütemaß eines Tests ist sein Fehlerüberdeckung

$$FC = \frac{\#NF}{\#F}$$

$\#NF$ – Anzahl der nachweisbaren Fehler; $\#F$ – Anzahl der vorhandenen Fehler.

Bei SW (und HW Entwürfe) ist die fehlerorientierte Testauswahl und Bewertung (noch) unüblich. Statt dessen:

- empirische Überdeckungskrit. (z.B. Anweisungsausführung) und
- empirische Auswahlregeln wie, jede Automatenkante ausprobieren.

Prinzipielle Probleme der fehlerorientierten Testauswahl für SW:

- Keine fehlerfreie Beschreibungen,
  - um Fehler zu modellieren und
  - Sollwerte für die Testergebnisse zu bestimmen.

Symbolische und Fuzzy-Tests, Operationsprofil

Ein symbolischer Tests definiert

- Eingabebedingungen,
- nachzuweisendes (Fehl-) Verhalten und
- Kontrollkriterien für das Ergebnis

ohne die Festlegung konkreter Testeingaben und Sollwerte.

Fuzzifizierung:

- Suche konkreter Eingaben zu den Eingabebedingungen bzw. den nachzuweisenden (Fehl-) Verhaltens,
- Ergebnis kann auch ein langer Zufallstest mit typischen Eingaben für ein Operationsprofile sein.

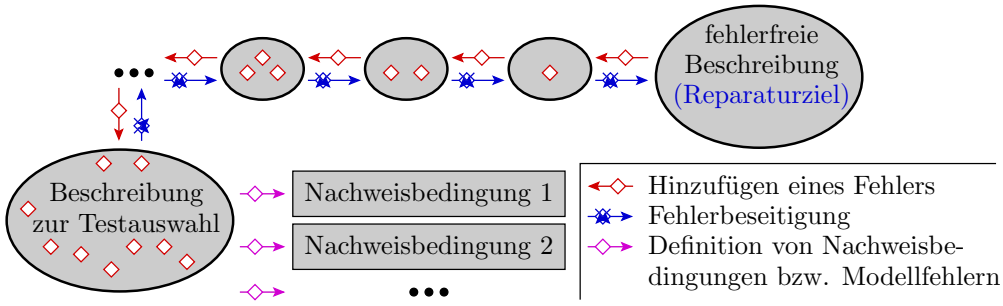
Operationsprofil:

- Art der Systemnutzung, genauer
- relative Nutzungshäufigkeit der unterschiedlichen Arten von SL.

Ähnlich wie bei Wichtung für HW-Selbsttest hängen die FF-Raten der einzelnen Fehler im System erheblich vom Operationsprofil ab.

2.1 Mutationen

Mutationen statt Modellfehler



Die Beschreibungen von SW für die Testauswahl und Bewertung enthalten die zu findenden Fehler:

- Statt der Modellfehler lassen sich nur Mutationen einer potentiell fehlerhaften Beschreibung konstruieren.
- Für vergessene Aspekt lassen sich keine ähnlich nachweisbaren Mutationen ableiten.

[Nachweisbeziehungen Mutationen ⇔ Fehlern im Vergleich zu ICs]
[Mutationen für Nicht-Code-Beschreibungen, z.B. Anforderungen, ...]
[Review Check-Schwerpunkte ⇔ fehlerorierte. Testauswahl]

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)

  [viele Möglichkeiten, redundante Fehler]

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.

Kontrolle der Testausgaben

Aus einer fehlerhaften Beschreibung lassen sich keine Sollwerte für einen Soll/Ist-Vergleich ableiten. Alternativen:

- Diversitäre Bestimmung der Sollwerte aus der Zielfunktion (in der Regel manuell). [viel Arbeit für lange Zufallstests]
- Sollwertbestimmung mit Golden Device oder Regressionstest,
- Inspektion der Testausgaben, [manuell]

- Kontrollen, die auch unter Betriebsbedingungen funktionieren:
  - Wertebereiche, Prüfkennzeichen,
  - Zeitüberwachung, Loop-Test, ...

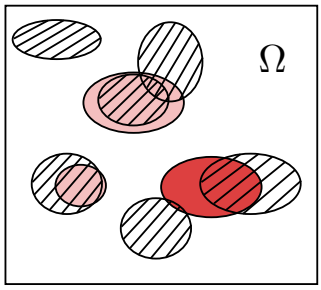
Kontrollfragen

1. Was lässt sich über Aufwand, FFC und die Phantom-FF-Rate der für dynamische SW-Tests nutzbaren Kontrollen aussagen?
2. Was lässt sich aus Sicht der Kontrollmöglichkeiten als Maßnahmen für den prüfgerechten Entwurf ableiten?

2.2 Nachweisbeziehungen

Nachweisbeziehungen

[Die und folgende Folien fast gleich Foliensatz F5, Abschn. Nachweisbez.]



Ω Menge aller Testeingabewerte

Eingabewerte, die den Fehler nachweisen

Eingabewerte, die den Fehler eventuell (unter Zusatzbedingungen) nachweisen

Nachweismenge ähnlich nachweisbarer Mutationen

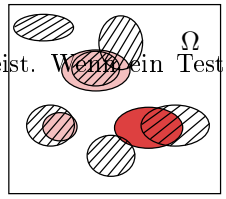
Mutationsmodelle sollten eine große Anzahl von Mutationen erzeugen, die so ähnlich wie die zu erwartenden Fehler in Programmen:

- fehlende oder falsche Anforderungen,
- fehlende oder fehlerhafte Umsetzung und
- fehlende, falsche und überflüssige Anweisungen, ... ,

nachweisbar sind.

Ähnlichkeit bedeutet, große anteilige Schnittmengen der Nachweismengen und entsteht durch übereinstimmende Anregungs- Beobachtungs- und lokale Nachweisbedingung.

Fehlerorientierte Testsuche



Für jeden Fehler i gibt es $\#MF_i \geq 0$ ähnlich nachweisbare Mutationen j, für die jeder gefundene Test Fehler i mit p_{ij} nachweist. Wenn ein Test gefunden wird, werden $m-1$ weitere Tests gesucht und auch gefunden.

A_j Tests für Mutation j gefunden:

$$\mathbb{P}(A_j) = FC_{\mathrm{M}}$$

B_{ij} m Tests für Mutation j weisen Fehler i nach:

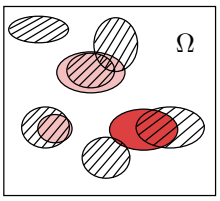
$$\mathbb{P}(B_{ij}) = 1-(1-p_{ij})^m$$

N_i Nachweis von Fehler i:

$$N_i = \bigcup_{j=1}^{\#MF}(A_j \cap B_{ij}) = \overline{\bigcap_{j=1}^{\#MF}\overline{(A_j \cap B_{ij})}}$$

$$p_i = \mathbb{P}(N_i) = 1-\prod_{j=1}^{\#MF}(1-(FC_{\mathrm{M}}\cdot(1-(1-p_{ij})^m)))$$

Zu erwartende Fehlerüberdeckung



$$p_i = 1 - \prod_{j=1}^{\#MF} (1 - (FC_{\mathrm{M}} \cdot (1 - (1 - p_{ij})^m)))$$

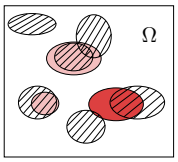
$$\mathbb{E}[FC] = \frac{1}{\#F} \cdot \sum_{i=1}^{\#F} p_i$$

Für fehlende Aspekte (Anforderung, Verzweigung, Berechnungen, Ausnahmebehandlungen, ...) lassen sich durch Mutation kaum Modellfehler ableiten, die sich Anregungs- und Beobachtungsbedingungen teilen.

Angenommen 50% der tatsächlichen Fehler sind von Typ »es fehlt etwas in der Beschreibung«, dann gilt für für diesen Anteil nur »zufälliger Nachweis«:

$$\mathbb{E}[FC] \leq 50\%$$

Zufällige Testauswahl



Nachweiswahrscheinlichkeit für Fehler und Mutationen gleich FF-Rate, im Bild repräsentiert durch die Größe der Nachweismengen.

Fehler- und Mutationsüberdeckung, pareto-verteilter Nachweislänge:

$$\begin{aligned} \mathbb{E}[FC(n)] &= 1 - C_{\mathrm{MU}} \cdot (1 - \mathbb{E}[FC_{\mathrm{MU}}(n)]) \\ &= 1 - c_{\mathrm{MU}}^{-k} \cdot (1 - \mathbb{E}[FC_{\mathrm{MU}}(n)]) \\ \mathbb{E}[FC(c_{\mathrm{MU}} \cdot n)] &= \mathbb{E}[FC_{\mathrm{MU}}(n)] \end{aligned}$$

k – Formfaktor Pareto-Verteilung; n – Testanzahl; $c_{\mathrm{MU}} = \frac{n_{[\mathrm{eff}]}}{n_{\mathrm{T}}} = \frac{\bar{\zeta}_{\mathrm{MU}}}{\bar{\zeta}}$ – Testlängenskalierung, Verhältnis mittleren FF-Rate Mutationen $\bar{\zeta}_{\mathrm{MU}}$ und mittleren FF-Rate realer Fehler $\bar{\zeta}$; C_{MU} – Fehleranteilskalierung.

Wenn die Mutationen typische Programmierfehler sind:

$$\mathbb{E}[FC(n)] \approx \mathbb{E}[FC_{\mathrm{MU}}(n)]$$

2.3 Kontrollfluss

Kontrollflussbasierte Testauswahl

Nach Standard DO-178 B gilt heute für SW-Tests als ausreichend, um sich der Produkthaftung zu entziehen:

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

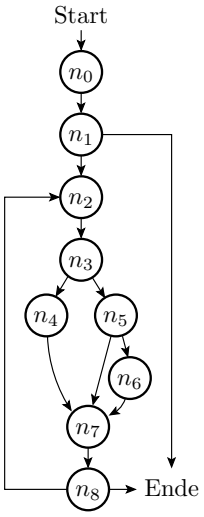
Alle drei Überdeckungsmaße

- sind am Kontrollfluss eines Programms definiert,
- lassen sich einfach bestimmen und
- vernachlässigen ähnlich wie der Toggle-Test für digitale Schaltungen die Beobachtbarkeit lokal nachweisbarer FF.

Angedachte Erweiterung: »Def-Use«-Tupel/Ketten zur Ergänzung von von Beobachtungsbedingungen.

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
}
```



Kontrollflussgraphen automatisch erzeugbar.

Kontrollflussbasierte Testvollständigkeitsmaße

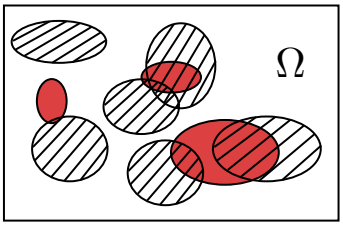
1. Anweisungsüberdeckung: Jede Anweisung muss mindestens einmal ausgeführt werden. Beispiel: Start, n_0, n_1, n_2, n_3, n_4, n_7, n_8, n_2, n_3, n_5, n_6, n_7, n_8, Ende

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.

[gefundenen Konrollflussabläufe ⇒ symbolische Tests]
[Fuzzifizierung, Mehrfachausführung $FC \uparrow$]

Testvollständigkeitsmaß und Fehlerüberdeckung



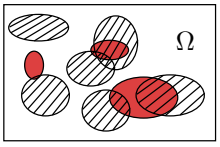
Ω alle Eingabemöglichkeiten
Eingabemenge Fehlernachweis
Eingabemenge Überdeckungskriterium (Anweisungsausführung etc.)

Überdeckungskriterien j sind die Ausführung von Anweisung j, Verzweigung j bzw. Bedingung j. Relationen zwischen Fehlern und Überdeckungskriterien:

- Fehler ohne ähnlich nachweisbaren Überdeckungskriterien ($\#MF = 0$), z.B. fehlende Anweisungen, Verzweigungen, ...
- Fehler mit $\#MF_i \geq 1$ ähnlich nachweisbaren Überdeckungskriterien (z.B. überflüssige und falsche Anweisungen), p_{ij} nicht größer als Beobachtbarkeit für falsche Anweisungsergebnisse, Verzweigungen, ...

[Diskussion $p_{i,i}$: Kontrolle Anweisungsergebnisse im Debugger, nur Gesamtausgaben]
[Soll/Ist-Vergleich, nur Ist-Werte-Review, nur Formatkontrollen, ...]

Gezielte Testauswahl



$$p_i = 1 - \prod_{j=1}^{\#MF} \left(1 - \left(FC_\mathrm{K} \cdot \left(1 - (1 - p_{ij})^m\right)\right)\right)$$

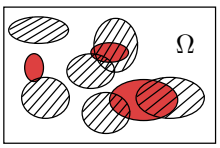
$$\mathbb{E}\left[FC\right] = \frac{1}{\#F} \cdot \sum_{i=1}^{\#F} p_i$$

FC_K – Kriterienüberdeckung, gefordert meist 100%. Die zu erwartende Fehlerüberdeckung $\mathbb{E}\left[FC\right]$ hängt außer von FC_K erheblich ab vom

- Anteil der Fehler ohne ähnlichen Überdeckungskriterien und
- für die Fehler mit ähnlichen Überdeckungskriterien von
  - der Anzahl der Tests m je Überdeckungskriterium und
  - der Wahrscheinlichkeit der Beobachtbarkeit des falschen Anweisungsergebnisses, der falschen Verzweigung etc.

$FC_\mathrm{K} = 1$ allein erlaubt bei gezielter Testauswahl wenig Rückschlüsse auf die Fehlerüberdeckung.

Zufallstest

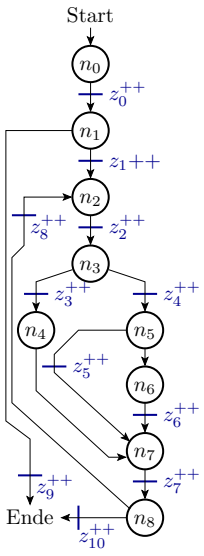


Hinzunahme zufälliger (intuitiv gewählter) Testbeispielen, bis Überdeckung ausreichend. Beziehung Kriterienüberdeckung ⇔ Fehlerüberdeckung:

$$\begin{aligned}\mathbb{E}\left[FC\left(n\right)\right] &= 1 - C_\mathrm{K} \cdot \left(1 - \mathbb{E}\left[FC_\mathrm{K}\left(n\right)\right]\right)\\ &= 1 - c_\mathrm{K}^{-k} \cdot \left(1 - \mathbb{E}\left[FC_\mathrm{K}\left(n\right)\right]\right)\\ \mathbb{E}\left[FC\left(c_\mathrm{K} \cdot n\right)\right] &= \mathbb{E}\left[FC_\mathrm{K}\left(n\right)\right]\end{aligned}$$

k – Formfaktor der Pareto-Verteilung; n – Testsatzlänge; $c_\mathrm{K} = \frac{n_\mathrm{[eff]}}{n_\mathrm{T}} = \frac{\bar{\zeta}_\mathrm{K}}{\bar{\zeta}}$ – Testlängenskalierung, Verhältnis der mittleren Erfüllungsrate Überdeckungskriterien $\bar{\zeta}_\mathrm{K}$ und mittleren FF-Rate realer Fehler $\bar{\zeta}$; C_KA – Fehleranteilskalierung.

Fehlerüberdeckung $FC \approx FC_\mathrm{K}$ erfordert eine Verlängerung der abgeschätzen Testsatzlänge n für FC_K um einen schwer abzuschätzenden Faktor $c_\mathrm{K} = \frac{\bar{\zeta}_\mathrm{K}}{\bar{\zeta}} \gg 1$.



Bestimmung der Kantenüberdeckung

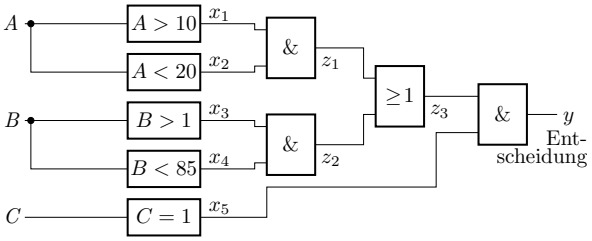
```
int z[11]={0,0,0,0, ...};
...
int ZZ(int Ct_max){char c;
n0: Ct_A=0;Ct_B=0;Ct_N=0;z(0)++;
n1: while (Ct_N<Ct_max){ z(1)++;
n2:  c=getchar(); z(2)++;
n3:  if (is_TypA(c)){
n4:   z(3)++; Ct_A++;}
      else {z(4)++;
n5:   if (is_TypB(c)){
n6:    Ct_B++; z(5)++;}
      }else z(6)++;
n7:  Ct_N++; z(7)++;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

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

Bedingungsüberdeckung
Ein logischer Ausdruck, z.B.

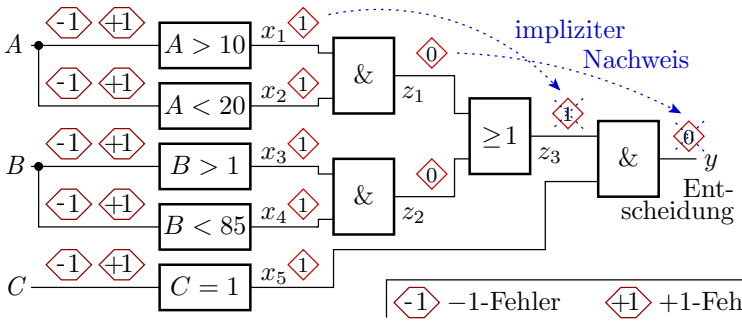
```
n1: if (((A>10) && (A<20)) || ((B>1) && (B<85))
        && (C==1)) {
n2:   ... }
    else {
n3: ...   }
```

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

[Beseitigung identisch nachweisbarer und redundanter Fehler]
[Fehlerparallele Simulation in Bitscheiben]
[Suche symbolische Test mit D-Algorithmus]

Für die SW-Testsauswahl könnten Begriffe, Algorithmen und Erfahrungen von der Testauswahl digitaler Schaltungen übernommen werden.

2.4 Def-Use-Ketten

Def-Use-Ketten

[Zusätzlichen Modellierung von Beobachtungsbedingungen]

Def-Use-Tupel: Datenstruktur, die aufeinanderfolgende Paare von Schreib- und Lesezugriffen einer Variable beschreibt. Programmbeispiel »größter gemeinsamer Teiler[2] «:

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

Berechnung von Def-Use-Tupeln und -Ketten

Berechnung aller Def-Use-Tupel:

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

Berechnung von Def-Use-Ketten:

```
Wiederhole für alle »Defs«
    Suche einen Pfad aus Def-Use-Tupeln zu einer beobachtbarer Ausgabe
```

Außer als Überdeckungskriterien sind Def-Use-Tupel/Ketten nutzbar

- für statische Code-Analysen:
  - »Use« ohne »Def« ist ein Initialisierungsfehler.
  - »Defs« ohne »Use« sind redundanter Code.
- Fehlerlokalisierung:
  - Rückverfolgung von FF zur Entstehungsursache im Def-Use-Graphen.

Verwendung zur Fehlerlokalisierung

Rückverfolgung des Def-Use-Graphen zur Entstehungsursache der FF am Beispiel »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«, nachdenen Testausgaben vor dem nächsten Testdurchlauf einzuprogrammieren sind, »n0« und »n6«

Überdeckungskriterien und Fehlerüberdeckung

Mögliche Mengen von Überdeckungskriterien:

- alle »Defs« mindestens ein [alle] »Use«.
- alle »Use« mindestens ein Def
- alle »Defs« mindestens eine [alle] Def-Use-Kette.

[Vergl. Anweisungeüberd.: p_{ij} größer, Anteil Fehler ohne ähnliche Ü-Kriterien gleich]
[redundante Fehler, Überdeckungskriterien pro NLOC $\leftrightarrow$ Zellen- und Pfadverz.]

Am zielführendsten für eine hohe Korrelation Kriterienüberdeckung $\Leftrightarrow$ FC: »für alle »Defs« eine Def-Use-Kette«:

Algorithmus zum Zählen Def-Beobachtbarkeit:

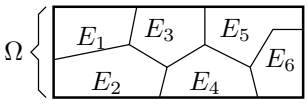
- Führe für alle Variablen eine Liste von »Defs«, von denen der aktuell gespeicherte Wert abhängt.
- Bei Ausgabe einer Variablen, Inrement der Zähler aller »Defs« aus der Def-Liste der Variablen.

[Für gezielte Testauswahl Suche von $m \geq 1$ Tests je Überdeckungskriterium.]
[Wie Anweisungsüberd. etc. automatisch in ein Test-Build einprogrammierbar]

2.5 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

Äquivalenzklassenbasierte Testauswahl ist möglich für

- ein fertiges Programm: Testauswahl und FC vergleichbar mit »Bedingungsüberdeckung«,
- aber auch für Dokumente im Entwurfsfluss davor (Spezifikation, Aufgabenanlyse, ...) bei geeigneter prüfgerechter Beschreibung.

Spezifikationsbasierte Testauswahl

Voraussetzung geeigneter Beschreibungsstruktur der Zielfunktion, z.B.:

```
wenn       <Bedingung 1> dann <Berechnung 1>;
sonst wenn <Bedingung 2> dann <Berechnung 2>;
sonst ...
```

Testauswahl (wie es eigentlich getan werden müsste):

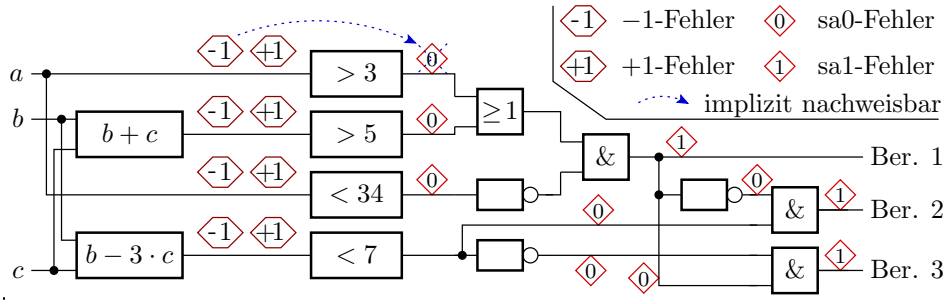
- Ableitung einer Menge symbolischer Tests,
- Fuzzifizierung, Sollwertberechung, ... erst nach Schnittstellendefinition,
- Bei Aufteilung in Module, für alle Module:
  - Zielfunktion als Wenn-Dann-Beschreibung,
  - Ableitung einer Menge symbolischer Tests, ...

Wunschvorstellungen an eine solche Wenn-Dann-Beschreibung:

- diversitär zum eigentlichen Entwurf,
- Testauswahl, Fuzzifizierung und Sollwertberechnung automatisierbar,
- Bedingungen nicht auf Äuivalenzklassen beschränkt.

Wenn-Dann-Beschreibung zur Testauswahl

```
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");
}
```



- Transformation Fallunterscheidungen ⇒ log. Berechnungsfluss.
- Fehlersimulation / Testberechung wie für Haftfehler.
- symbolische Testeingaben: Bedingungen für Eingabewerte.
- symbolische Testausgaben: ausgeführte Operationen.
- Fuzzifizierung: Expansion zu (vielen) konkreten Tests.

2.6 UW-Analyse

Ursache-Wirkungs-Analyse

Spezifikationsbasierte Testauswahl auf Basis einer verallgemeinerten Wenn-Dann-Beschreibung:

- Ursachen (wenn): Auslösern von Aktionen.
- Wirkung (dann): ausgelöste Aktionen.

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

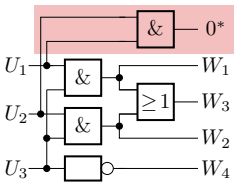
Zwischen den Ursachen und Wirkungen werden logische Verknüpfungen formuliert.

Fehlersimulation / Berechung symbolischer Tests wie für Äquivalenzklassen:

- symbolische Tests für Haft- und Off-By-One-Fehler,
- Fuzzifizierung.

Beispiel »Zähle Zeichen«

- Wirkungen:
  W_1: Anzahl_TypA +1[3]
  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.



* 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	

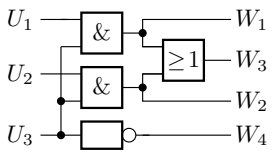
Eine Ursache-Wirkungs-Analyse deckt auch Mehrdeutigkeiten und Widersprüche in der Spezifikation auf.

[3] Im Programmbeispiel wird Typ A »Ziffer« und Typ B »Großbuchstabe« sein.

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

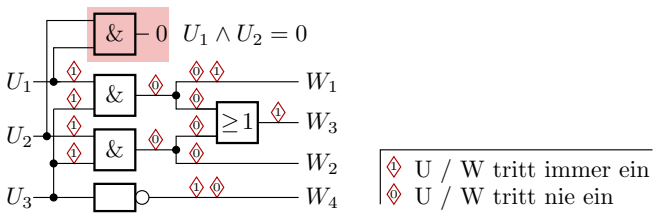
U_1	Zeichen ist vom Typ A (Ziffer)	W_1	Ct_A++
U_2	Zeichen ist vom Typ B (Großbuchstabe)	W_2	Ct_B++
U_3	max. Zählwert nicht erreicht	W_3	Ct_N++
–	es wird kein Zeichen gelesen	W_4	Ende

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.

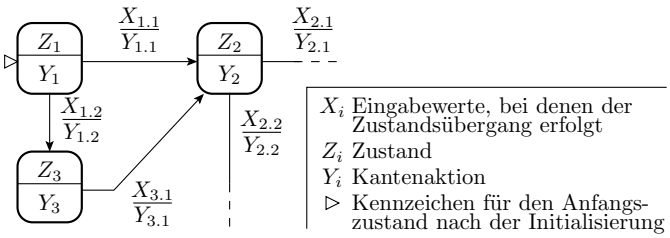
- Nach Berechnung der gleichzeitig zu (de-) aktivierenden Ursachen folgt die Suche geeigneter Eingaben und Kontrollen.

UW-Bescheibung hat Potential

- diversitär zum zu testenden Code zu sein,
- für die Automatisierbarkeit Testauswahl, Fuzzifizierung, ...

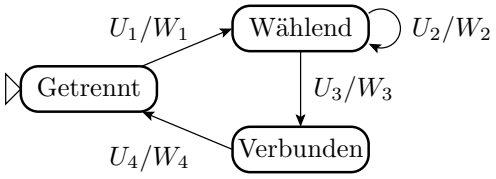
2.7 Automaten

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

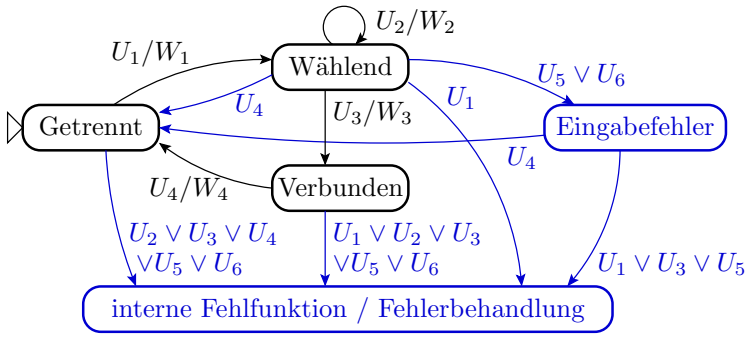


U_1 Abnehmen
U_2 Ziffer wählen
U_3 Rufnummer gültig
U_4 Auflegen
W_1 Rufnummer zurücksetzen
W_2 Ziffer zur Rufnummer hinzufügen
W_3 Verbindung aufbauen
W_4 Verbindung trennen

- 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)
  - ...

$\Rightarrow$ Ablaufgraph ist noch unvollständig

Ergänzen von Fehlerzuständen



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		

Symbolische Tests aller Zustandsübergänge

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

[Kantenübergänge für jede Ursache]

Fuzzifizierung

Aus einem Automatengraphen sind wie bei der UW-Analyse nur symbolische Tests ableitbar, z.B. als Liste zu kontrollierender Übergang-Bedingungs-Tupel:

```
Getrennt, Wählend, U1;
Wählend, Wählend, U2;
...
```

Die konreten Testeingaben und Testabläufe werden erst bei der Fuzzifizierung festgelegt.

2.8 Zusammenfassung

Testauswahl für SW

Höhe Verlässlichkeit verlangt nicht nur eine auflistbare Anzahl von Einzeltests, sondern wochenfüllende Tests mit voller Systemgeschwindigkeit. Das wiederum verlangt Automatisierung bzw. Rechnerunterstützung von der Testauswahl über die Testdurchführung bis zur Generierung der Fehler- und Testvollständigkeitsausgaben .

Die automatisierbaren Verfahren für die Testauswahl benötigen

- den fertigtigen Programm-Code oder
- eine (diversitäre) Beschreibung der Zielfunktion.

und erlauben die Ableitung symbolischer Tests, aus denen über Fuzzifizierung konkrete Testeingaben gewonnen werden.

Code-basierte Testauswahl

Für die code-basierte Testauswahl genügen heute noch 100% Anweisungs-, Zweig- oder Bedingungsüberdeckung. Aus Sicht der Wahrscheinlichkeitsbeziehungen zwischen den code-basierten Überdeckungskriterien und der Fehlerüberdeckung und auch Sicht der Umsetzbarkeit erscheinen folgende Erweiterungen als zielführend:

- Suche mehrere symbolischer Tests je Überdeckungskriterium,
- Fuzzifizierung der symbolischen Tests zu längeren Zufallstests,
- Erweiterung der Beobachtungskriterien um Def-Use-Ketten.

Spezifikationsbasierte Tests

Als (diversitäre) aus der Spezifikation ableitbare Beschreibung der Zielfunktion eignen sich

- Wenn-Dann-Beschreibungen (Äquivalenzklassen, UW-Graphen),
- Zustandsdiagramme von Automaten und
- vermutlich auch die anderen UML-Funktionsbeschreibungen (Aktivitätsdiagramm, Sequenzdiagramm, Protokollautomat).

Die Ableitung der symbolischen Tests lässt sich weitgehend auf die Testauswahl für Haftfehler für digitale Schaltungen zurückführen.

Forschungs- und Entwicklungsbedarf besteht für

- weitere / bessere Beschreibungen für Zielfunktionen, aus denen sich auch symbolische Tests ableiten lassen,
- A/R Fuzzifizierung,
- A/R Sollwertenberechung und A/R andere Kontrollen,
- A/R Erzeugung Test-Builts und Testdurchführung.

(A/R – Automatisierung oder Rechnerunterstützung).

Literatur

Literatur

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