Test und Verlässlichkeit Foliensatz 6: Software

Prof. G. Kemnitz

Institut für Informatik, TU Clausthal (TV_F6)

6. Februar 2024

Inhalt TV_F6: Software

Vorlesung	18	19	20
bis Abschn.	3	35	76

Software

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

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

Erweiterbar um manuelle Beschreibungen, in denen auch Fehler entstehen, gesucht, vermieden und beseitigt werden:

- Anforderungen, Realisierungsideen,
- Absprachen, ...

Manuelle SW-Entstehungsschritte (siehe Abschn. 1.5.2):

- kreative Teile, nicht deterministisch, hohe Fehlerentstehungsrate,
- Kontrolle durch Review (siehe Abschn. 4.2.1), ...

Automatisierte SW-Entstehungsschritte:

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

Wirksamste Fehlervermeidung: Automatisierung

Programmiersprache

Programmiersprache und Verlässlichkeit

Die Programmiersprache ist die Schnittstelle zwischen den manuellen und den automatisierten Entstehungsschritten.

Die Wahl der Programmiersprache bestimmt sehr wesentlich mit

- den Programmieraufwand,
- mögliche und wahrscheinliche Fehler,
- Fehlerentstehungsrate,
- Fehlerbehandlung, ...

Unterteilung der Fehlerarten nach dem typ. Umgang mit ihnen:

ST	Ausschluss durch statische Tests zur Compile-Zeit.
HT	Schadensvermeidung durch Check + Abbruch.
FT	Fehlertoleranz durch Check + MF-Korrektur.
DT	Ausschluss durch dynamische Tests.

Rust*

Programmiersprache für nachfolgende Beispiele, recht neu, noch nicht allzu verbreitet, aber:

- außergewöhnlich viel Fehlerausschluss durch statische Tests,
- automatisch einprogrammierte Kontrollen in Testübersetzungen,
- außergewöhnlich gute Unterstützung für die MF-Behandlung,
- innovative Beschreibung und Verwaltung von Tests.

Durch das Sprachkonzept rutschen einige verbeitete Fehlerarten aus den Kategorien:

- HT (Schadensvermeidung durch Check + Terminierung)
- FT (Fehlertoleranz durch Check + MF-Korrektur)
- DT (Ausschluss durch dynamische Tests)

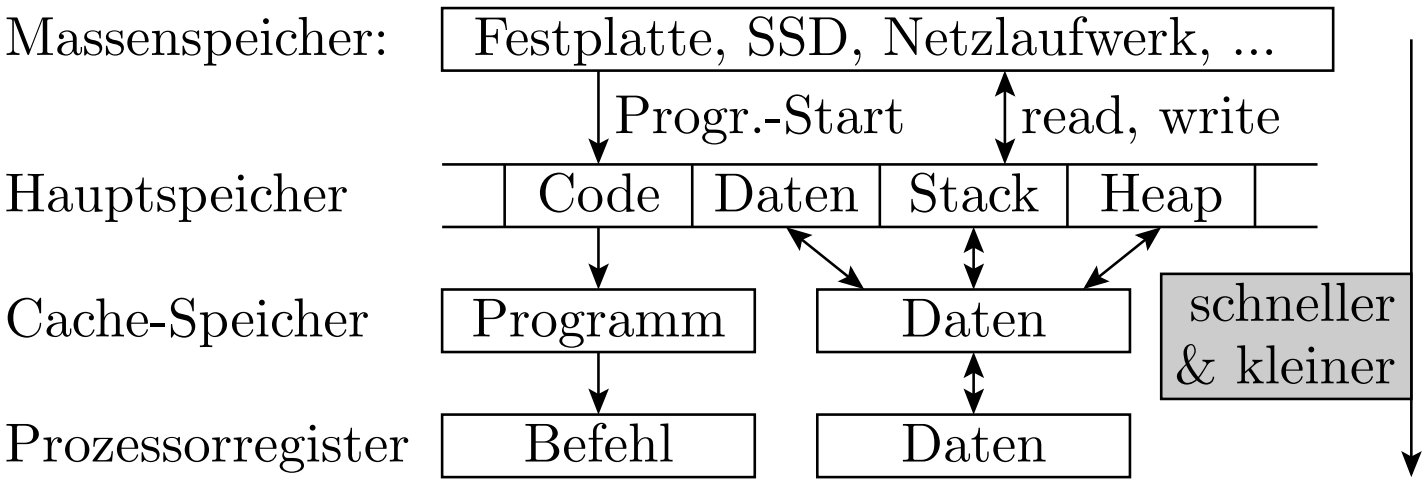
in die Kategorie »statisch nachweisbar«. Dadurch in übersetzten Programmen ausschließbar und somit unproblematisch.

Verbesserte Sprachunterstützung für MF-Behandlung, Testprogrammierung und Testdurchführung motiviert zu mehr Kontrollen und Tests.

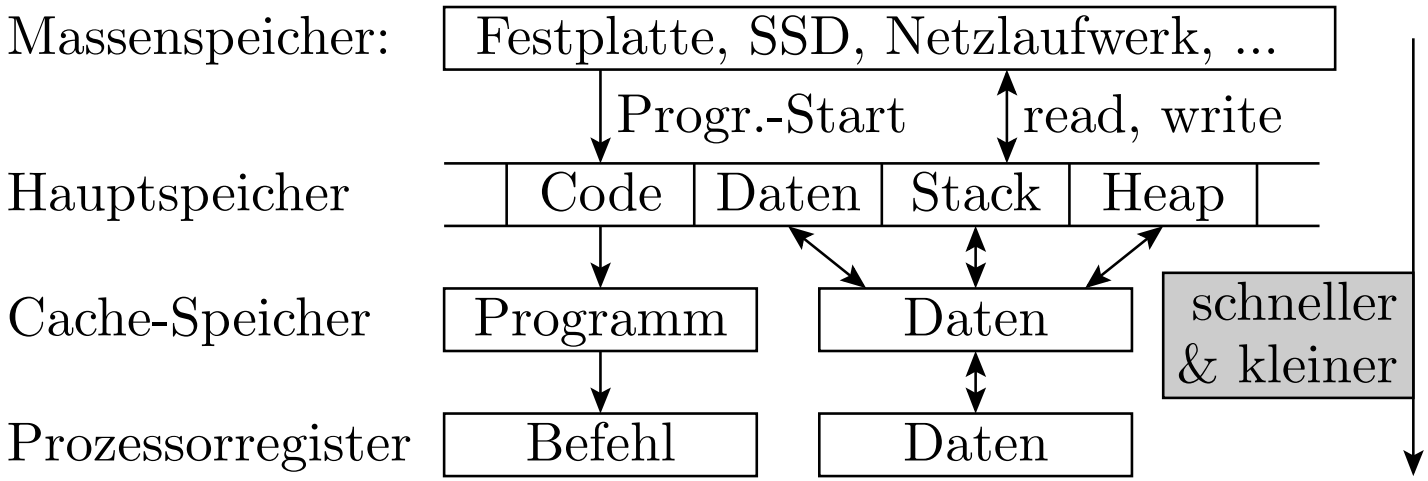
* https://rust-lang-de.github.io/rustbook-de.

Datenobjekte

Speicherverwaltung



- Moderne Rechner speichern die kompletten Daten als Dateien in Massenspeichern. Für aktive Programme werden Code-Kopien im Hauptspeicher gehalten und Platz für Daten reserviert.
- Von genutzten Code- und Datenbereichen werden Kopien im schneller zugreifbaren Caches gehalten.
- Der Prozessor holt je Takt ein Befehlswort, führt eine Berechnung mit Prozessorregistern aus oder kopiert ein Datenwort zwischen Prozessorregister und Daten-Cache.

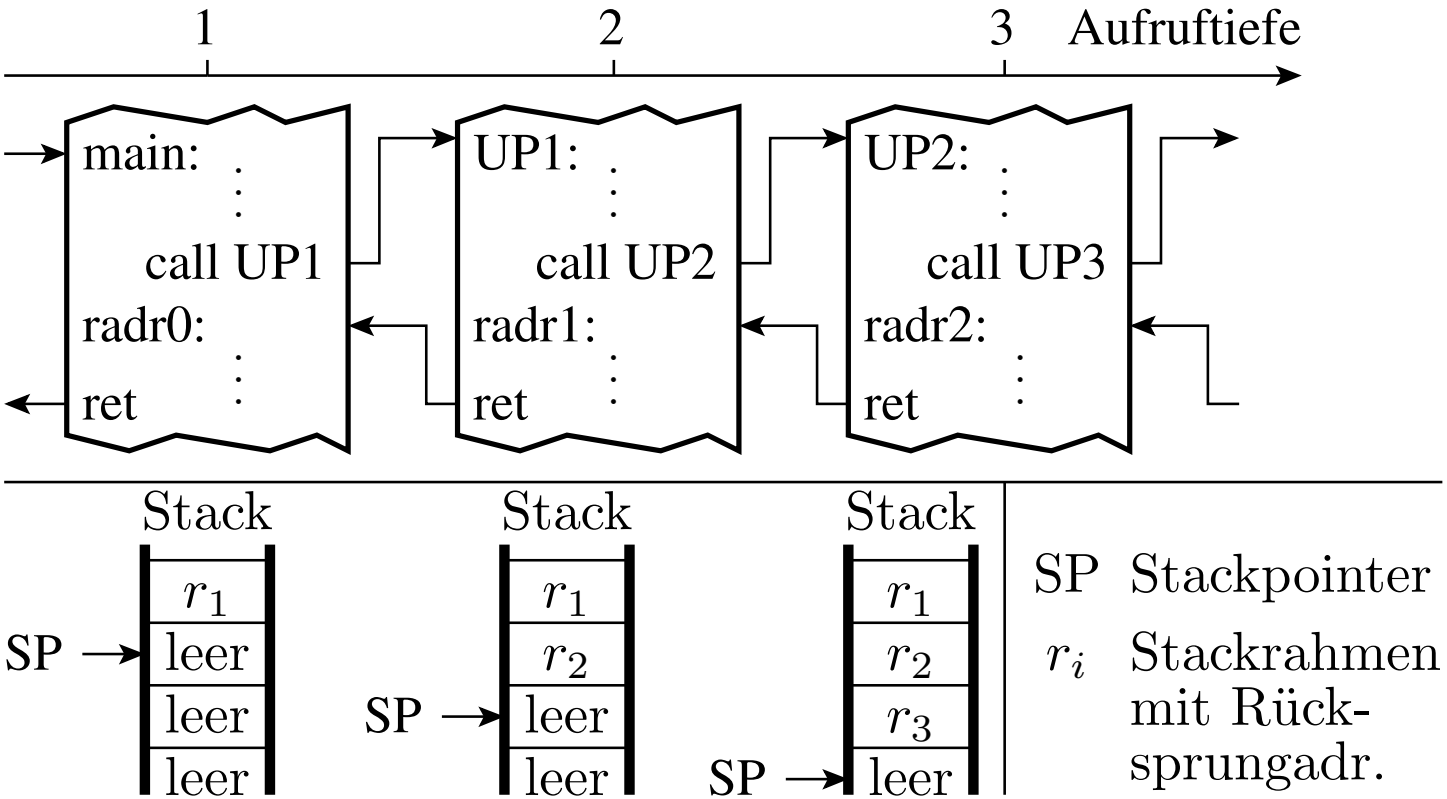


Der Compiler reserviert je Programm vier Speichersegmente:

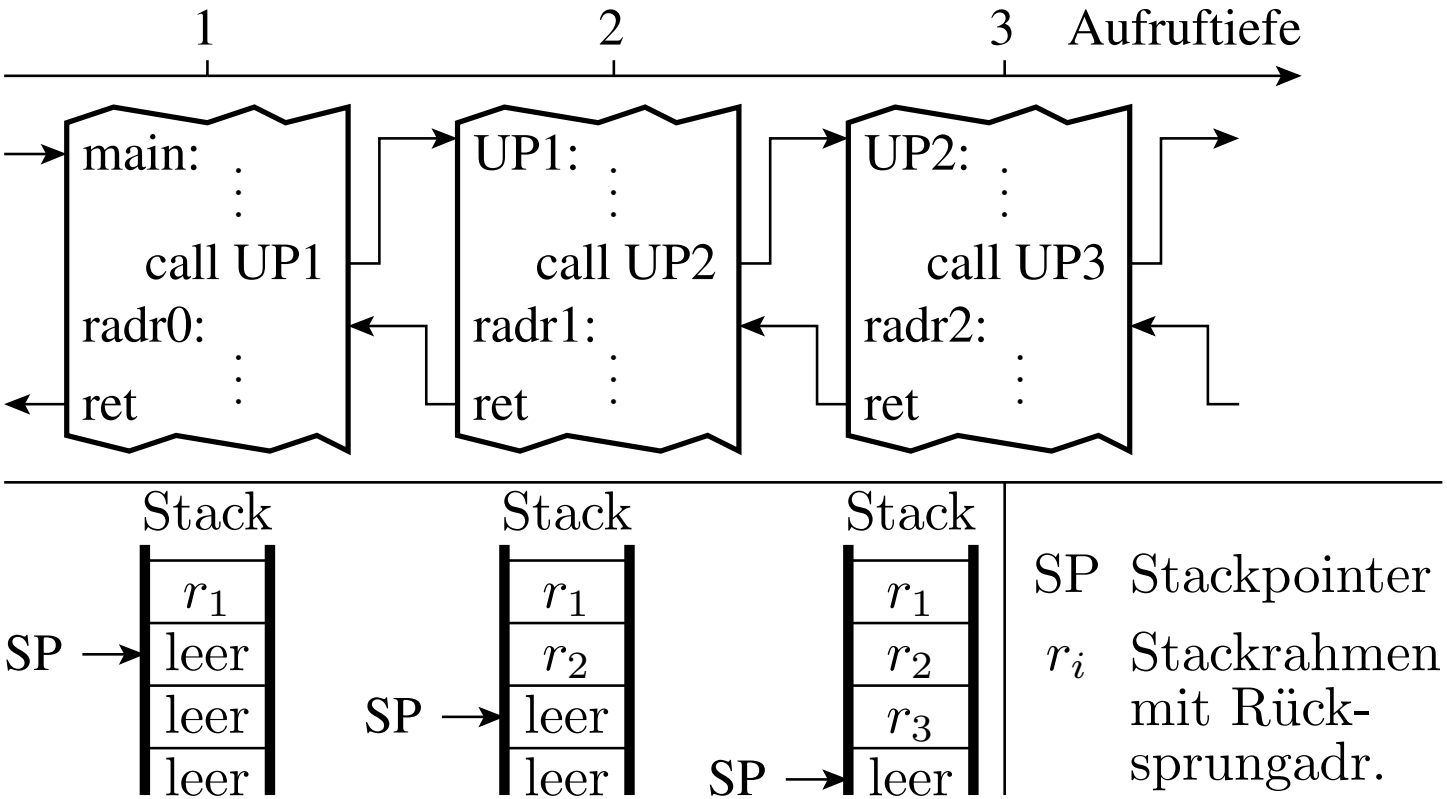
- Codesegment: Befehle und Konstanten, nur lesbar.
- Datensegment: globale Variablen mit konstanten Adressen.
- Stack: Stapelspeicher für Rücksprungadressen und lokale Variablen mit zur compile-Zeit bekannter Größe.
- Heap: zur Laufzeit vergebbarer Speicher für Datenobjekte variabler Größe.

Code kommt in einen nicht flüchtigen Speicher oder wird vom Betriebssystem geladen. Daten werden vom Programm initialisiert, bearbeitet und ein- und ausgegeben. Bei der Ein- und Ausgabe ist auch das Programm für die MF-Behandlung zuständig.

Stack

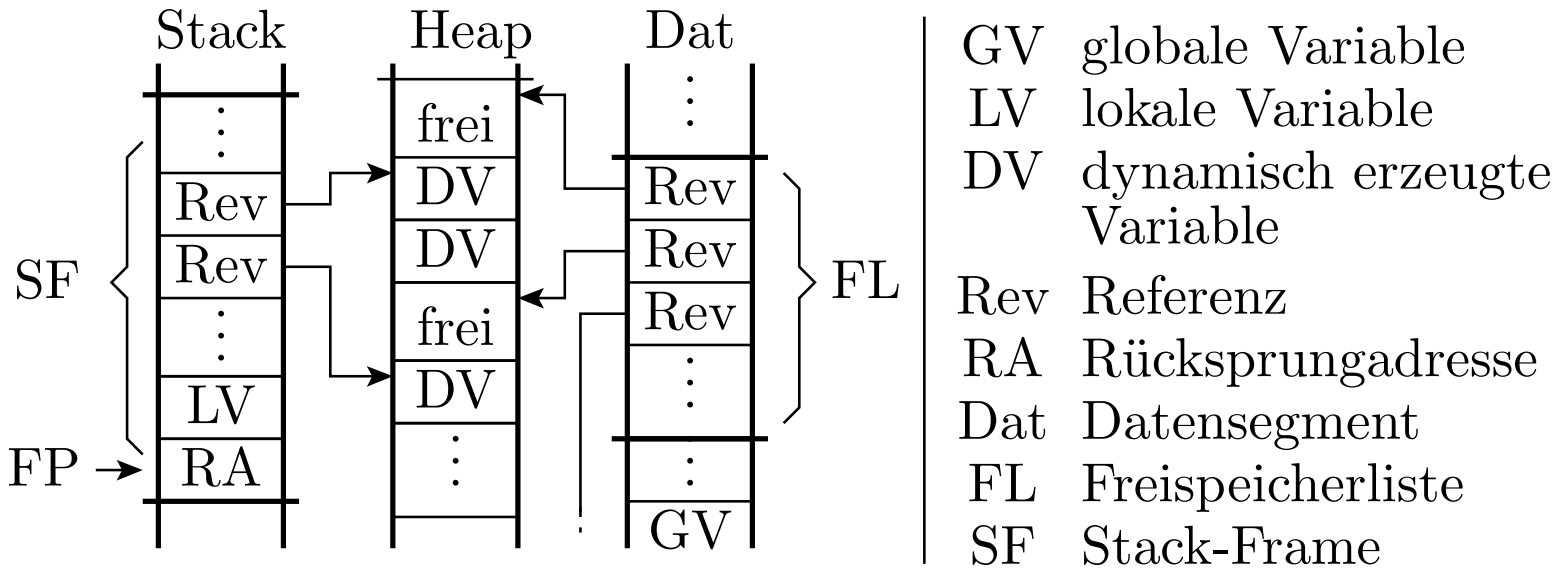


Abgesehen vom Startup-Code für die erste HW-Initialisierungen ist ein Programm eine Sammlung von Funktionen, die aufgerufen werden und nach Beendigung zum Aufrufpunkt zurückspringen.



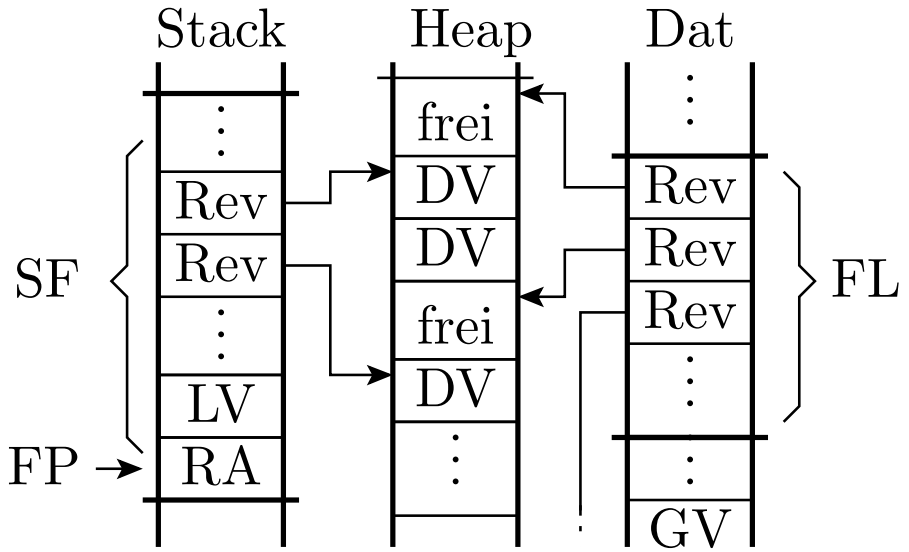
- Stapelspeicher für Rücksprungadresse und für die Fortsetzung nach Rücksprung benötigte Registerzustände.
- Erlaubt Aufruf von Funktionen in Funktionen, auch rekursiv.
- Der Stack wird auch zur Ablage lokaler Variablen genutzt.
- Adressierung relativ zum Frame-Pointer (FP).
- Nur die Funktionen auf dem Stack belegen Speicher.

Heap



- Von der SW verwalteter Datenbereich,
- Zugriff über Zeiger im Datensegment, auf Stack und Heap.
- Zeiger sind bis Speicherreservierung und ab Speicherfreigabe ungültig. Adressverlust ohne Freigabe => Speicher-Leck.

Stack und Heap schaffen Flexibilität, mindern den manuellen Programmieraufwand (Fehlervermeidung), sind aber selbst Fehlerquellen.



GV	globale Variable
LV	lokale Variable
DV	dynamisch erzeugte Variable
Rev	Referenz
RA	Rücksprungadresse
Dat	Datensegment
FL	Freispeicherliste
SF	Stack-Frame

Fehlerquellen Heap:

- Speicherlecks: keine Freigabe nicht mehr benötigter Daten (DT*),
- Referenzen auf ungültige Variablen (DT*)
- Heap-Fragmentierung: Zersplitterung der freien Heap-Bereiche. Zunahme der Rechenzeit für die Suche ausreichend großer freier Plätze bis »keine Erfolg« bei genug freiem Speicher (FT*).

ST	Ausschluss durch statische Tests zur Compile-Zeit.
HT	Schadensvermeidung durch Check and Abbruch.
FT	Fehlertoleranz durch Check + MF-Korrektur.
DT	Ausschluss durch dynamische Tests.
*	üblicher Umgang.

Datenobjekte in Rust

Konstanten: benutzte Zahlenwert, Zeichen, Auszählungswerte:

```
3136, ..., 2.45, ..., "Text"
```

Variablen:

- global (static), feste Adresse im Code-Segment,
- lokal, Stack, feste Adresse im Stack-Frame der Funktion
- dynamisch, Adresszuordnung zur Laufzeit auf dem Heap.

```
static x: u16 = 3136; // globale, unveränderbar
let mut yyy: u32 = 0; // lokal, veränderbar
let yyy = yyy + 1254; // Wertänderung
let v = vec![1, 2, 3];// Vektor auf dem Heap
```

Besonderheiten von Rust:

- **static** erzeugt globale Datenobjekte (feste Adresse),
- **let** erzeugt Datenobjekte zur Laufzeit (Stack oder Heap),
- Veränderbarkeit (**mut**ability) ist explizit festzulegen.

Der Kontrakt »unveränderbar« ermöglicht zusätzliche statische Tests.

Keine hängenden Zeiger und Speicherlecks

Erzeugung von Heap-Objekten standardmäßig als initialisierte Konstante mit Adresszeiger auf dem Stack als lokale Variable:

```
let a:<typ> = <komplexes Datenobjekt>;
```

Veränderbarkeit muss extra angegeben werden:

```
let mut b:<typ> = <komplexes Datenobjekt>;
```

Nur einen Besitzer:

```
let b1 = b; // b existiert danach nicht mehr
```

Bei Übergabe eines Heap-Objekts an eine Funktion geht der Besitz an die Funktion über und muss bei Rückkehr zurückgegeben werden. Rückgabewert ist ein Ausdruck ohne abschließendes Semikon:

```
    xxx   // Rückgabewert, auch Tupel
}         // schließende Klammer
```

Beim Verlassen einer Funktion (oder Blocks) werden alle »im Besitz befindlichen« Objekte auf dem Heap gelöscht.

(Echte Kopien zur getrennten Bearbeitung und Weitergabe möglich.)

Referenzen

Referenzen sind Zeigerkonstanten auf Heap-Objekte, die statt der Objekte an Funktionen übergeben werden können:

```
fn get_len(s: &String)->usize {//Referenz auf s
  s.len()                      //Rückgabewert
}
```

- Referenzen besitzen das Objekt nicht, sondern borgen es nur.
- Wenn eine Referernz den Gültigkeitsbereich verlässt, wird nur die Referenz, aber nicht das Objekt selbst gelöscht, also keine Rückgabe erforderlich.
- Innerhalb eines Gültigtigkeitsbereichs sind mehrere nur lesbare Referenzen auf eine Objekt erlaubt.
- Bei einer **mut**able Referenz keine weiteren Referenzen erlaubt.
- Ausschluss gelöschte Objekte mit noch gültigen Referenzen, ...

Insgesammt lassen sich so Speicherlecks, hängender Zeiger, Read-after-Write-Wettlaufe, ... durch statische Tests ausschließen. Programmierung gewöhnungsbedürftig.

Datentypen und Operationen

Elementare Typen fester Größe:

- ganzzahlig: 8, 16, ... bit, vorzeichenfrei oder Zweierkomplement,
- Gleitkomma: 32, 64, ... bit aus Vorzeichenbit, Mantisse und Charakteristik, Sonderwerte: NaN, $+\infty$ und $-\infty$.
- Aufzählungen z.B. die Wahrheitswerte »true« and »false«,
- Zeichentypen für die Darstellung eines Textzeichens.

An elementare Typen sind Operationen und Kontrollmöglichkeiten auf Zulässigkeit gebunden:

- Kombination von Operanden- und Ergebnistyp (ST)
- Zulässigkeit konstanter Werte (ST)
- Zulässigkeit berechneter Wert (HT*)

NAN	Ungültig (not a number), z.B. Ergebnis der Division durch null.
ST	Ausschluss durch statische Tests zur Compile-Zeit.
HT	Schadensvermeidung durch Check and Abbruch.
*	Rust compiliert in Testversionen WB-Überlaufskontrollen für Operationen, wenn nicht ausdrücklich unterbunden (fail-fast). In den Release-Code jedoch nicht (fail-slow).

Zusammengesetzte Typen

Verbund von Objekten unterschiedlichem Typs:

```
// nur lesbares Tuple
let tup: (i32, f64, u8) = (500, 6.4, 1);
```

Feld mit Elementen desselben Typs:

```
// Feld mit veränderbaren Werten
let mut a: [i32; 5] = [1, 2, 3, 4, 5];
```

Operationen: Erzeugung, Übergabe an Funktionen und Rückgabe, Elementeauswahl, ... Kontrollmöglichkeiten auf Zulässigkeit:

- unzulässige Elementeauswahl (ST),
- unzulässige Index-Konstante (ST),
- unzulässiger berechneter Index (HT),
- unzulässige Zuweisungen (ST*, HT)...

ST Ausschluss durch statische Tests zur Compile-Zeit.

HT Schadensvermeidung durch Check and Abbruch.

* In Rust sind auch zusammengesetzte Objekte standardmäßig unveränderlich". Veränderbaren (mutable) Objekten dürfen nur Werte, aber nicht Typen oder Größe neu zugewiesen werden.

Kollektionen

Sammlung von Daten variabler Größe und Anzahl, die während der Programmausführung wachsen oder schrumpfen können mit unterschiedlichen Fähigkeiten. Die mitgelieferten Bibliotheken bieten:

- Vektor: gemischte Elementetypen, Erzeugen, Elemente anhängen und löschen, indizierter Zugrif Iteration über alle Elemente, ...
- Zeichenkette: Erzeugen, verketten, parsen, Zeichen suchen, ...
- Hash-Tabelle (hash map): Datenzugriff über Schlüssel.

Kollektionen nutzen den Heap und löschen automatisch ihre Heap-Daten, wenn der Gültigkeitsbereich verlassen wird.

Komplexe Beschreibungsmittel wie die Verwaltung von Listen, Zeichenkettenverarbeitung incl. Parse-Funktionen, Hash-Tabellen z.B. für Wörterbücher, ... mindern den Programmieraufwand für das individuelle Programm und damit die Fehlerannzahl.

Die größere zu erwartende Fehleranzahl im komplexeren Compiler verlangt längere Reifeprozesse, ... Rust ist noch sehr jung.

Aufzählungen in Rust

Aufzählung von Werten, in der jedem Wert ein Tupel von Datenobjekte zugeordnet ist:

```
enum IpAddr {              // Datentyp für IP-Adresse
    V4(u8,u8,u8,u8),       // V4-Adresse als vier u8
    V6(String),            // V6-Adresse als String
}
```

Vordefinierte Aufzählungstypen für die Fehlerbehandlung:

```
enum Option<T> {    // T generischert Typ
    None,     // kein gültiger Wert
    Some(T), // gültiger Wert vom Typ T
}
enum Result<T, E> {
    Ok(T),    // ohne MF ein Objekt vom Typ T
    Err(E),   // wenn MF, Beschreibung der MF
}
```

Verarbeitung mit »match«

Abgleich von Werten mit einer Reihe von Mustern und Ausführung des zum Muster passenden Codes, z.B. Verarbeitung möglicherweise nicht existierender Daten:

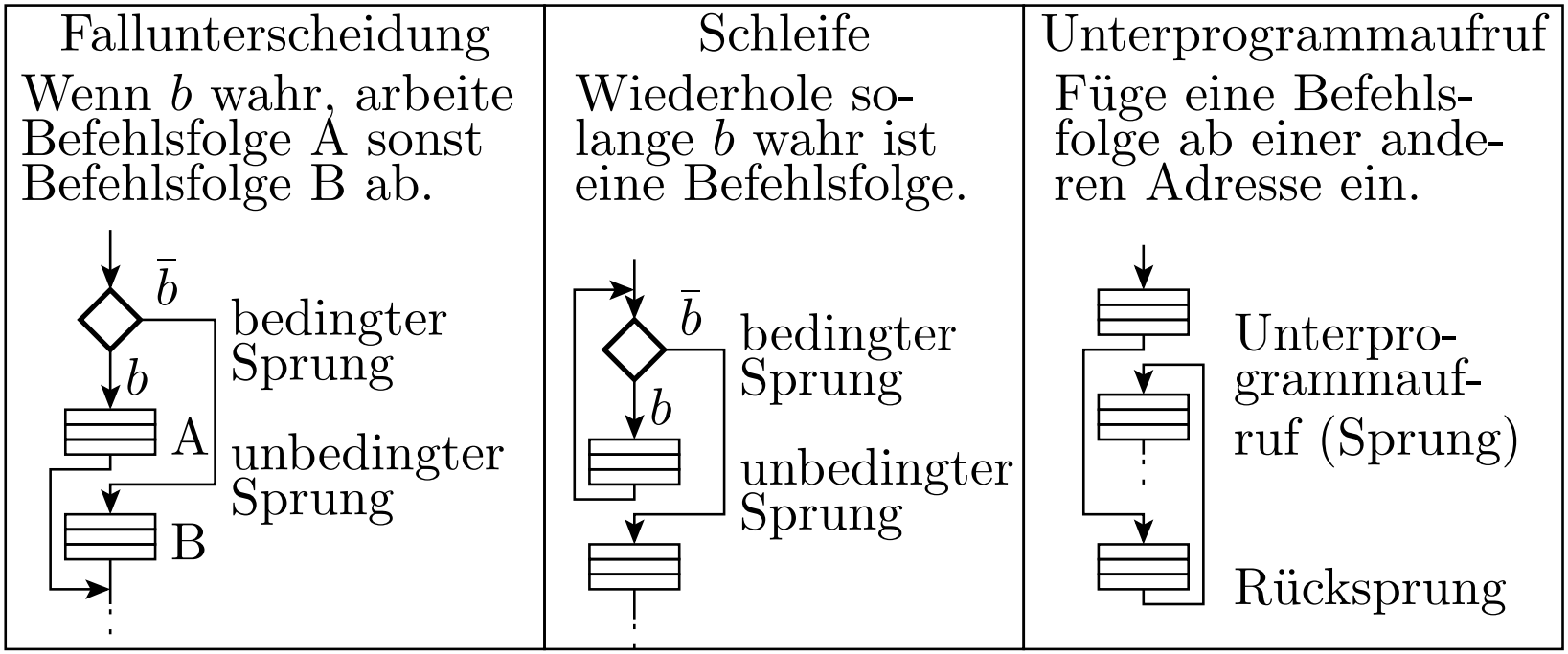
```
fn plus_one(x: Option<i32>) -> Option<i32> {
  match x {
      None => None,           // für x=None
          Ausdruck None und
      Some(i) => Some(i + 1),// für x=Some(i)
          wird i verarbeitet
  }
}
```

Ein Match-Ausdruck muss für alle Mustermöglichkeiten einen Ausdruck von selben Typ liefern. Kontrollen auf Zulässigkeit:

- unvollständige Musterabdeckung (ST),
- Fälle mit abweichendem Rückgabetyp (ST).

Kontrollfluss

Steuerung des Kontrollflusses



- Nach jedem Befehl wird danach der Folgebefehl abgearbeitet.
- Ausnahme Sprünge: unbedingt, bedingt, zu einem Unterprogramm oder zurück
- Manuelle Programmierung extrem fehlerträchtig. Hochsprachen beschränken die zulässigen Ablaufstrukturen in der Regel auf Fallunterscheidungen, Schleifen und Unterprogramme.

If-else und let-if in Rust

Bedingte Abarbeitung wie in allen höheren Programmiersprachen:

```
if number % 4 == 0 {
    println!("Zahl ist durch 4 teilbar");
} else if number % 3 == 0 {
    println!("Zahl ist durch 3 teilbar");
} else {
    println!("Zahl ist nicht durch 4 oder 3
        teilbar");
}
```

- Bedingungen müssen Ausdrücke vom Typ boolean sein (ST).

Bedingte Zuweisung (Besonderheit von Rust):

```
let number = if condition { 5 } else { 6 };
```

- Zusätzlich zur Compile-Zeit bestimmbarer Ergebnistyp (ST).
  - Abdeckung aller Möglichkeiten.
  - Gleicher Typ aller Ergebnisausdrücke.

Schleifen in Rust

Wie in allen höheren Programmiersprachen:

```
loop {...,                      // Schleife
  if <Bedingung> break          // Abbruchbedingung
};
while <Bedingung> {...}; // Abweisschleife
```

Besonderheiten mit Bezug zur Fehlervermeidung und MF-Behandlung:

- Schleifen mit Rückgabewert: Ausdruck hinter break, um z.B. in Fehlersituationen aus der Schleife zu eine Fehlerbehandlung zu wechseln und danach die Schleife mit Abbruchwert fortzusetzen.
- Schleifen-Label, um bei Verschachtelung mehrerer Schleifen festlegen zu können, welche Schleife mit break zu verlassen ist.
- For-Schleife über alle Elemente einer Kollektion (einfacher und weniger fehleranfällig als mit Index-Variable, oft genutzt):

```
let a = [10, 20, 30, 40, 50];
for element in a {
   println!("Der␣Wert␣ist:␣{element}");
}
```

MF-Behandlung

MF-Behandlung

Nach jeder Kontrolle muss eine MF-Behandlung abzweigen:

- kontrollierter Abbruch
  - Retten von Daten,
  - mit Freigabe Stack, Heap, Geräte-Reservierungen.
  - Schließen aller Dateien,
  - Bei Verbindung zur physikalischen Welt, sicherer Zustand.
  - Informationszusammenstellung zur MF (Fehlermeldung, Ausgabe Aufrufstacks, Core-Dump vom Speicher, ..)
- oder MF-Behandlung und Fortsetzung.

Der erste Schritt nach Erkennen einer MF ist die Entscheidung, ob Abbruch oder Fortsetzung.

Die Unterstützung durch Programmiersprache und Betriebssystem entscheiden über

- den Umfang einprogrammierter Kontrollen, den Aufwand dafür und
- die Anzahl der Fehler in den Funktionen zur MF-Behandlung.

Abbruch mit dem Makro panic

```
fn main() {
    panic!("abstürzen␣und␣verbrennen");
}
```

erzeugt eine Fehlermeldung mit

- Quelldateinahme und Zeilennummer des Panic-Aufrufs und
- dem Ausgabetext hinter **panic**.

Mit der Umgebungsvariable »RAST_BACKTRACE=1« wird der komplette Aufruf-Stack ausgegeben.

Automatisch eingefügte Kontrollen, z.B. auf »unzulässiger Index-Wert« bewirken immer einen Panic-Abbruch:

```
fn main() {
    let v = vec![1, 2, 3];
    v[99];    // Indexfehler, Abbruch mit panic
}
```

Rückgabetyp Result

Funktionen, die fehlschlagen können, z.B. das Öffnen einer Datei, erhalten den Ergebnistyp:

```
enum Result<T, E> {
    Ok(T),    // ohne MF ein Objekt vom Typ T
    Err(E),   // wenn MF, Beschreibung der MF
}
```

Beispiel:

```
let greeting_file_result = File::open("hallo.txt");
let greeting_file = match greeting_file_result {
  Ok(file) => file,
   Err(error)=>panic!("Problem␣beim␣Öffnen:{:?}",error),
};
```

Fehlerausgabe, wenn die Datei nicht existiert:

```
thread 'main' panicked at 'Problem beim Öffnen:
Os {code: 2, kind: NotFound, message: "No such
file or directory" }', src/main.rs:8:23
```

Kurzschreibweise mit »?«:

Der Operator »?« hinter einer Zuweisung eines Wertes vom Typ »result« bewirkt einen Funktionsabbruch mit Rückgabe »Err(E)«:

```
fn read_username_from_file()->Result<String,io::Error>{
  let mut username_file = File::open("hallo.txt")?; //**
  let mut username = String::new();
  username_file.read_to_string(&mut username )?;     //**
  Ok(username) // Sonst Rückgabe gelesener Datei-Text
}
// ** Abbruch bei Fehler mit Err(result)
```

Voraussetzung:

- Die Funktion muss auch Rückgabetyp »result« haben und
- der Datentyp »E« für die Fehlerbeschreibung muss übereinstimmen.

Rust bietet kompakte übersichtliche Beschreibungsmittel für eine sehr flexible Fehlfunktionsbehandlung.

Test

Rust-Beschreibungsmittel für Tests

Gemeint sind dynamische Tests. In SW sind Tests Funktionen, die

- die zu testenden Funktionen mit Bespielwerten ausführen und
- die Ergebnisse kontrollieren.

In Rust erhalten Testfunktionen die Annotation »#[test]«:

```
#[test]
fn test_add2() {
  let test_tup = ((1, 2), (3, 5));// Testeingaben
  for (a, b) in test_tup {     // für alle Testeingaben
    let sum = add(a, b);       // Ausführung des Testobjekts
    assert_eq!(sum, a + b);    // Ergebniskontrolle
  }
}
```

Das Kommando »cargo test« führt alle Tests im Projekt aus mit Protokollausgabe: der ausgführten Tests, ob bestanden, ...

Der Test der MF-Behandlung mit Programmabruch verlangt auch Tests der MF-Behandlung mit »panic« als Sollverhalten:

```
fn add_u8(x: u8, y:u8)-> u8 {x + y}

#[test]
#[should_panic]
fn test_add2_overflow() {
  let sum = add_u8(128, 128); // Ergebnisüberlauf, panic
}                             // bei Testübersetzung
```

Die Annotation [should_panic] wandelt »panic« in Test bestanden und erfolgreiche Ausführung in die Protokollausgabe »Test nicht bestanden« um.

Bei Funktionen mit Rückgabetyp »result« erfolgt der Test der Fehlerbehandlung mit ganz normalem Soll-/Ist-Vergleich.

Weitere Makros für die Testauswertung:

```
assert!(<einzuhaltende Bedingung>) // Kontrolle "wahr"
assert_neq(<Wert 1>, Wert 2>);     // Kontrolle ungleich
```

Optionen für die Testausführung

- Auswahl oder Unterdrückung eines Teils der Tests.
- Ausführung parallel oder nacheinander im selben Thread.
- Modultests: isolierter Test einzelner Funktionen.
- Integrationstest: Test des Gesamtsystems über die externen Schnittstellen.

Zusammenfassung

Zusammenfassung

Die Programmiersprache ist die Schnittstelle zwischen den manuellen und den automatisierten Entwurfsschritten. Sie bestimmt wesentlich:

- Programmieraufwand, mögliche und wahrscheinliche Fehler,
- Fehlerentstehungsrate, Fehleranzahl, Testaufwand,
- eingebaute Kontroll- und Fehlerbehandlungsfunktionen.

Am Beispiel der Programmiersprache Rust wurde gezeigt, dass

- geeigneten Beschreibungsmittel und
- ein guter Kompromis aus hartem und sanften Zwang

Fehlerarten vermeiden bzw. vom Compiler nachweisbar machen.

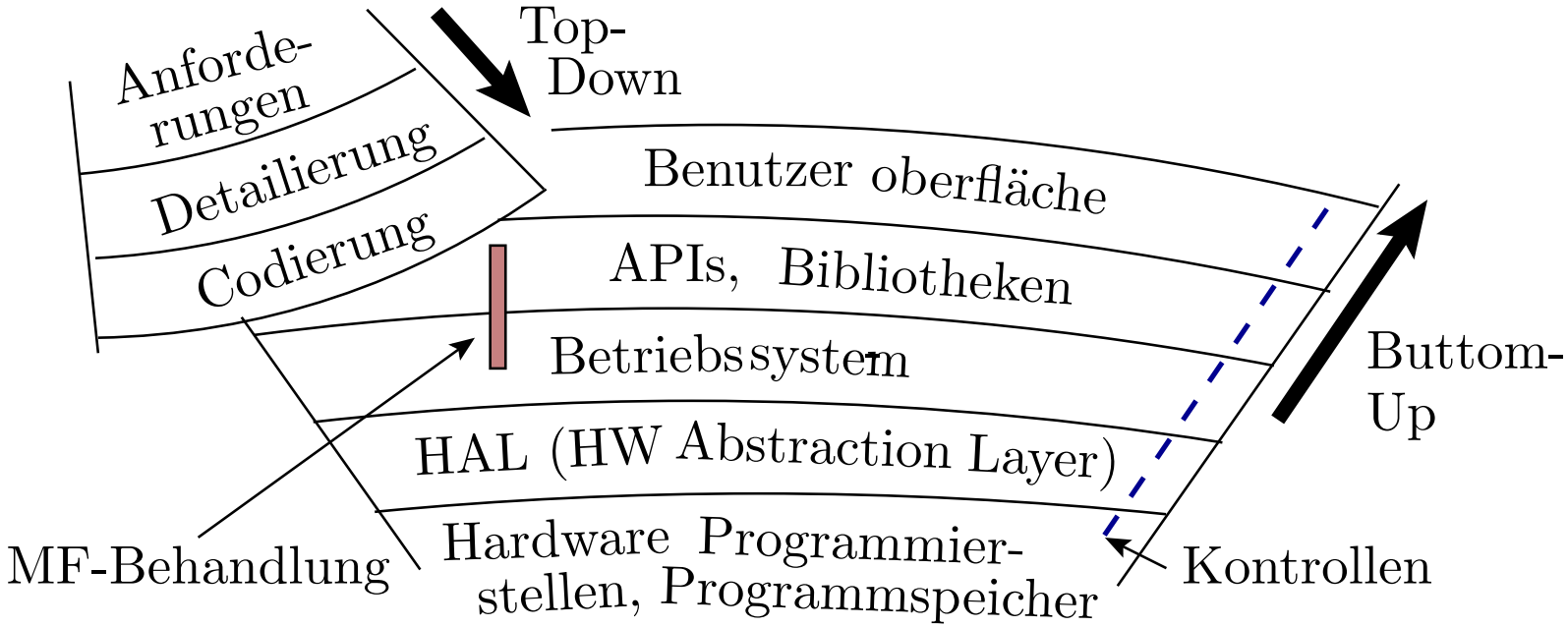
Rust liefert auch gute Anregungen für

- die Beschreibung und Modellierung der Behandlung von Fehlfunktionen und
- die Verwaltung und Durchführung von Tests.

Vieles davon lässt sich auch mit anderen Programmiersprachen beschreiben.

Vorgehen

2. Vorgehen



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

Der Entwurf beginnt mit einem Prozess zunehmender Detaillierung:

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

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: Eincompilieren & Ausführung, ...,
  - Abschätzung Testgüte: Anweisungsüberdeckung, toter Code, ...
  - Fehlerlokalisierung: Debugger, Trace-Tools, ...
  - Built-Prozess, Versionsverwaltung, Rückbau, ...

Umgang mit Fehlfunktionen:
- kontrollierter Programmabbruch oder
- Problemumgehung und Fortsetzung,
- Protokollierung von Informationen über die Ursache, ...

Software-Architektur

Software-Architektur

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, ...

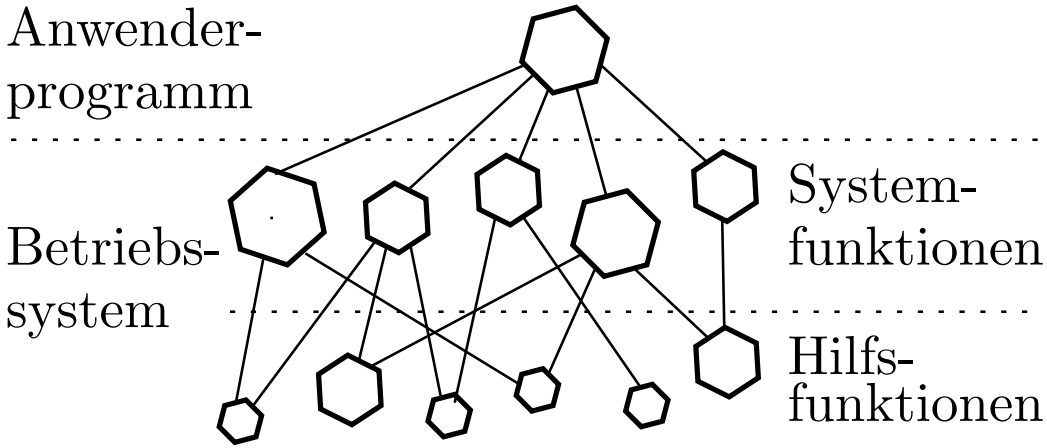
Grobeinteilung:

- Prozedurensammlung, Schichtenmodell,
- Client/Server-Modell, ...

Eine gute Architektur ist die Basis für langfristig wartbare, flexible und verständliche Systeme.

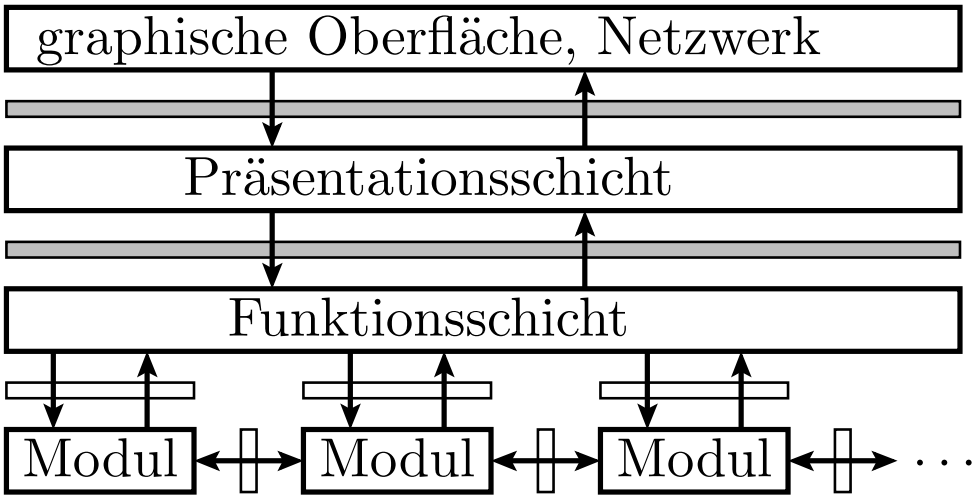
Prozedurensammlung

Die 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.
Ein Betriebssystem als Prozedurensammlung bietet Zugriffsfunktionen auf die Hardware der IO-Geräte und den Speicher, aber auch den direkten HW-Zugriff.

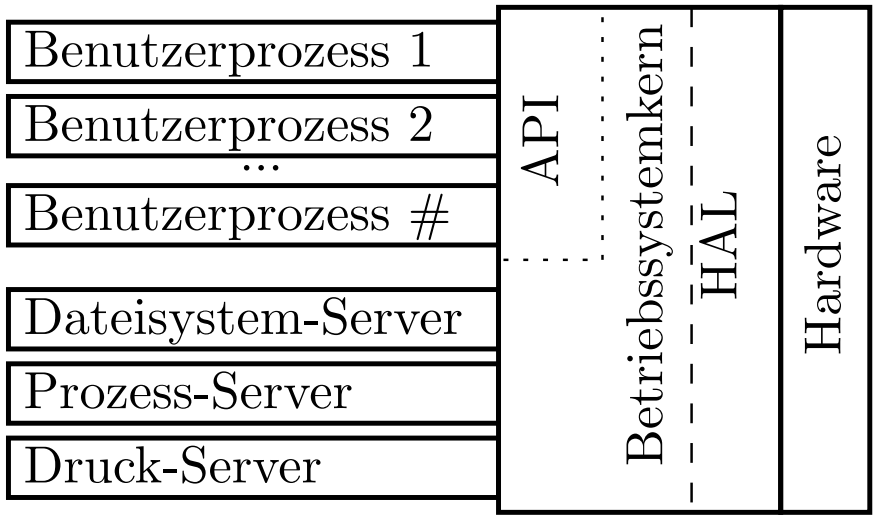
Schichtenmodell



Reglementierung der Benutzung von Prozeduren:

- Die Prozeduren sind alle einer Schicht zugeordnet.
- Eine höhere Schicht - z.B. Benutzeranwendungen wie Excel oder Word - können nur Prozeduren der Schicht darunter über eine wohl definierte Schnittstelle nutzen.
- Ein Kommunikationsprogramm kann nicht direkt auf den COM-Port zugreifen. Die Applikation stellt eine Anfrage an das Betriebssystem, ob COM-Port verfügbar, ...

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.

API	Programmiersschnittstelle.
HAL	Hardwareabstraktionsschicht.

Schnittstelle für Test- und MF-Behandlung

Schichten, Server-APIs sind wohl definierte Schnittstellen auch für:

- Überwachung: Trace-Aufzeichung, Stream-Monitor, ...
- MF-Behandlung: kontrollierter Abbruch mit Fehlermeldung,
- Modultests, z.T. über Script-Sprachen, d.h. ohne übersetzen.

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
```

- Bei unserem FPGA-Entwurfssystem ISE lassen sich die Entwurfswerkzeuge auch über Konsolenbefehle mit einer eigenen Skript-Sprache steuern: Compilieren, Routen, ...

Je komplexer die Systeme, desto mehr Aufwand erfordern Test und Überwachung auch bei der SW-Architektur.

Entwurfsablauf

Entwurfsablauf

Vereinheitlichtes Vorgehens durch Vorgehensmodelle (siehe Abschn. 1.5.3), damit

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

Good Practice (bewährte Techniken):

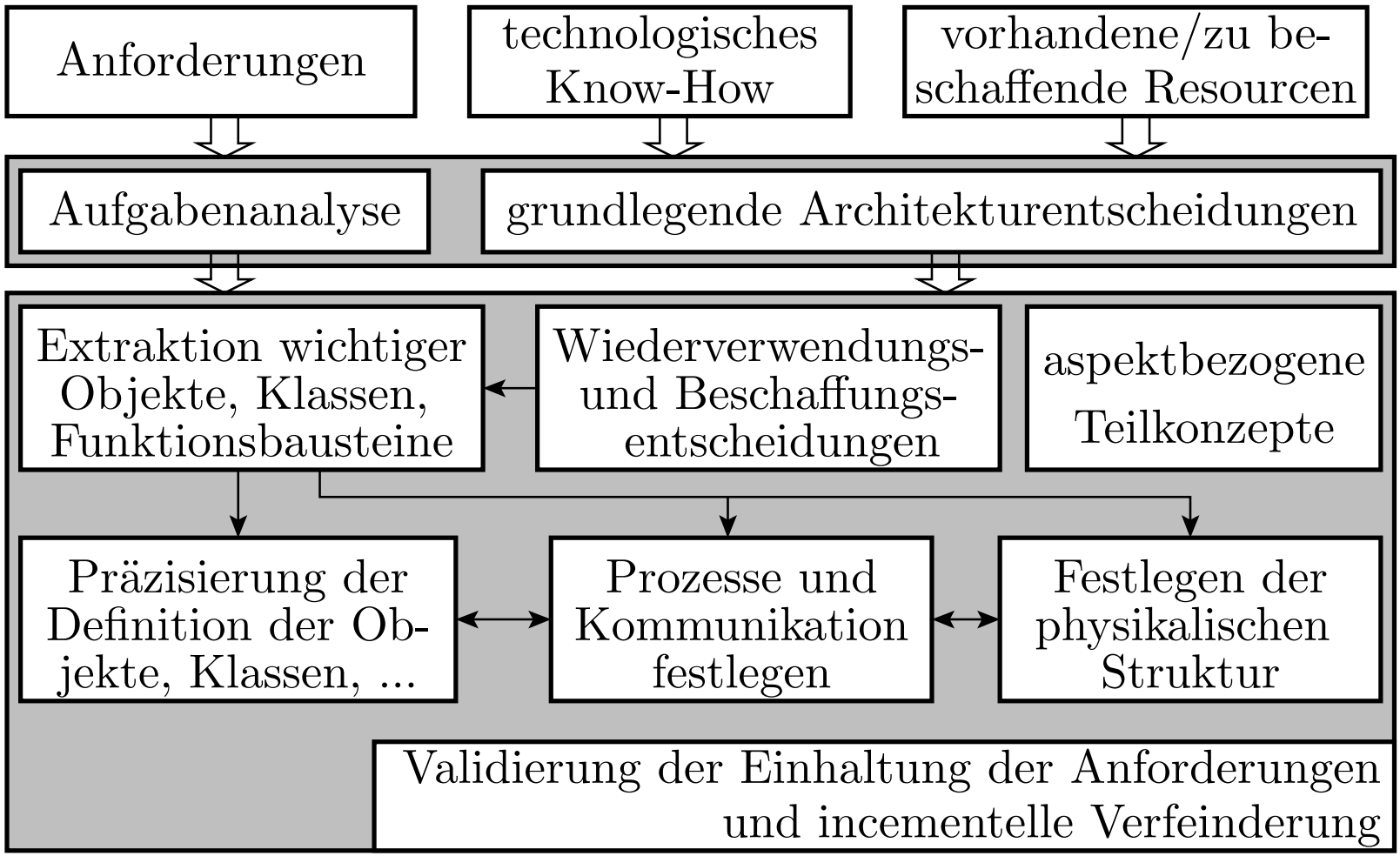
- Stufenmodell mit Zwischenkontrollen,
- Begrenzung/Minderung der Rückgriffhäufigkeit, ...

Stellschrauben:

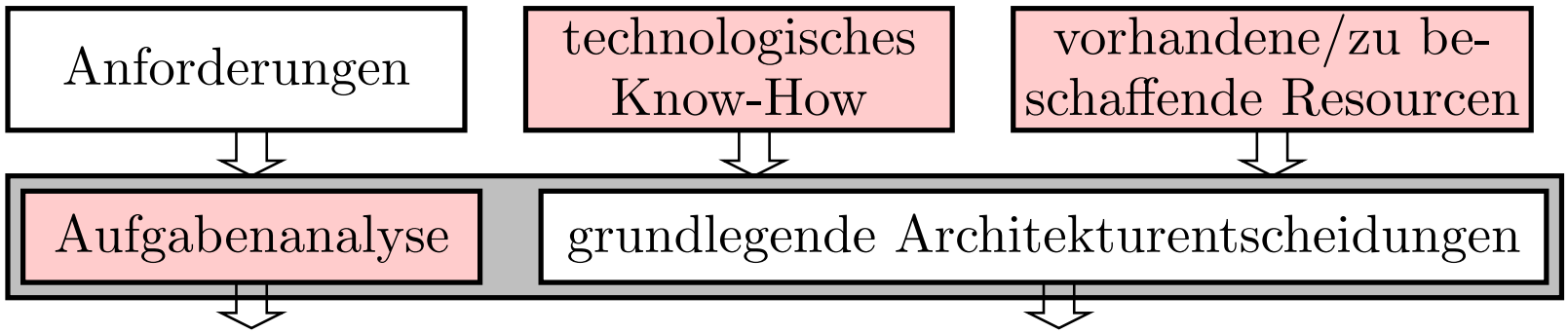
- Reihenfolge Entwurfsentscheidungen, Aufgabenteilung,
- Verantwortlichkeiten, Zwischenkontrollen,
- Arbeits- und Fehlerkultur, kreative Freiräume,
- verwendete Sprachen, Bibliotheken, ...

Wichtiges technologisches Knowhow eines SW-Unternehmens.

Beispielablauf: Anforderung bis Architektur



Aufgabe und Ressourcen analysieren



Aufgabenanalyse:

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

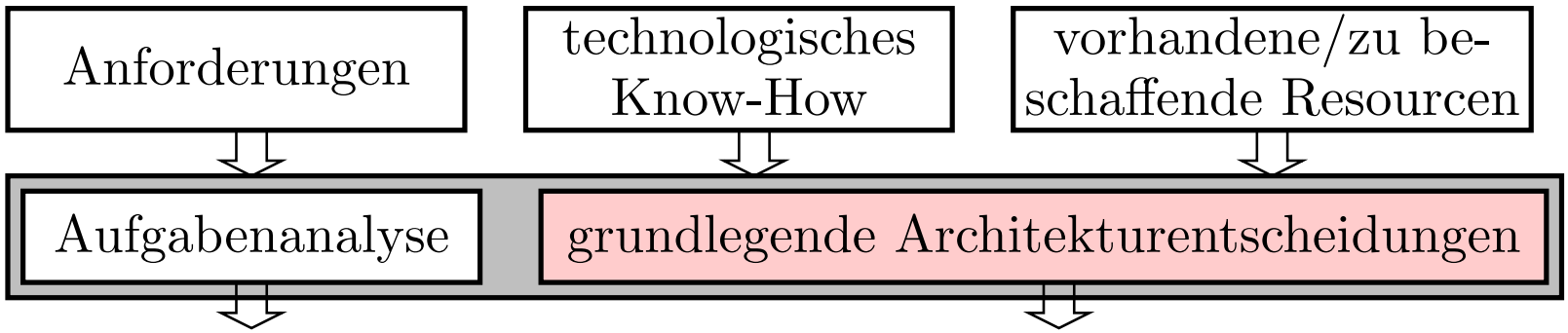
Technologisches Know-How:

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

vorhanden/zu beschaffen:

- Rechner, Software, Personal, ...

Grundsätzliche Architekturentscheidungen

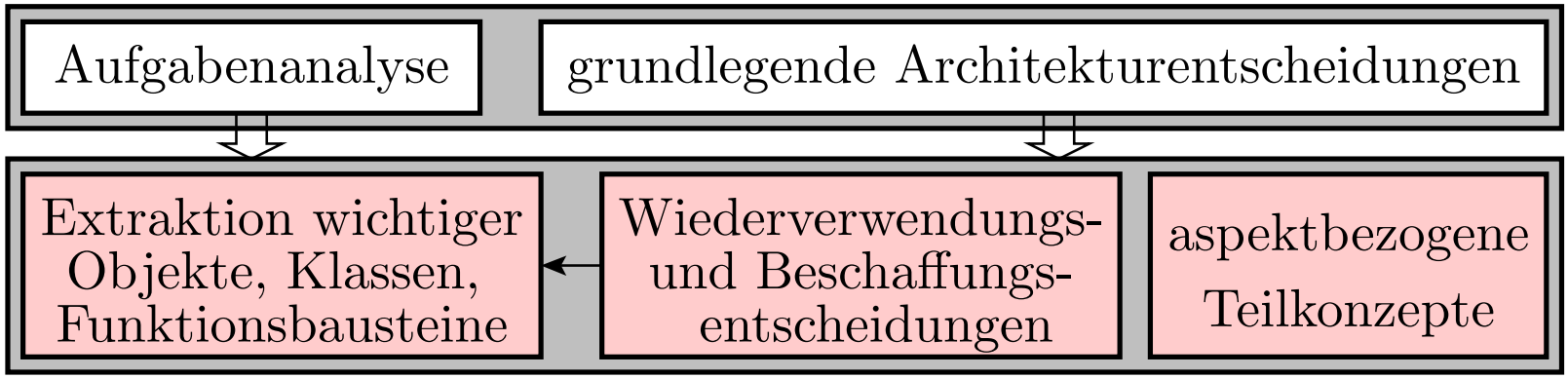


Nach den Analysen folgen grundlegende Entscheidungen:

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

Spätere Nachbesserungen dieser Grundsatzentscheidungen verursachen hohen Änderungsaufwand und entsprechend viele Fehler.

Detaillierung der Entscheidungen



Wiederverwendung und Beschaffung:

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

Aspektbezogene Teilkonzepte, über die zu Beginn zu entscheiden ist:

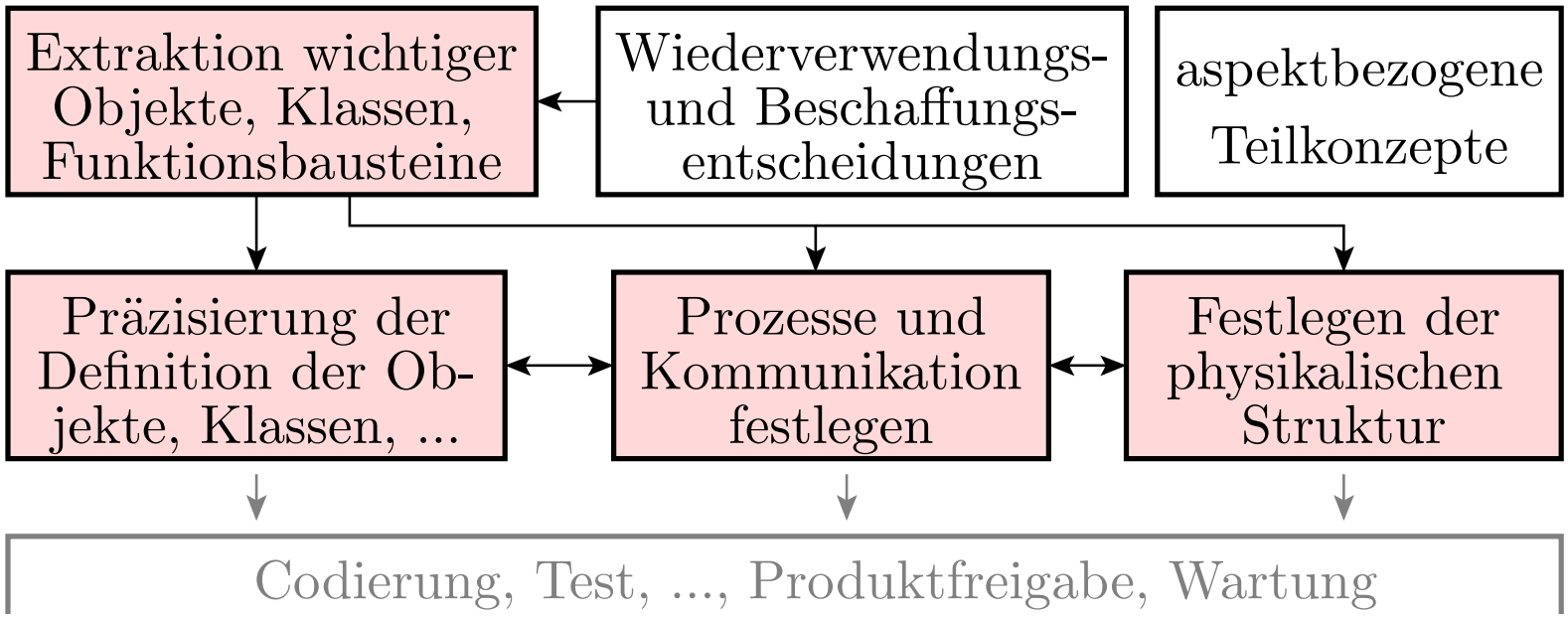
- Datenhaltung, Benutzerschnittstellen,
- Fehlerbehandlung, Fehlertoleranz, Sicherheit, ...

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. 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, ... [Bec03], [Sma14], [EAD14]

[DD16] J. Davis and R. Daniels. Effective DevOps - Building a Culture of Collaboration, Affinity, and Tooling at Scale. Sebastopol: O'Reilly Media, Inc., 2016. isbn: 978-1-491-92642-0.

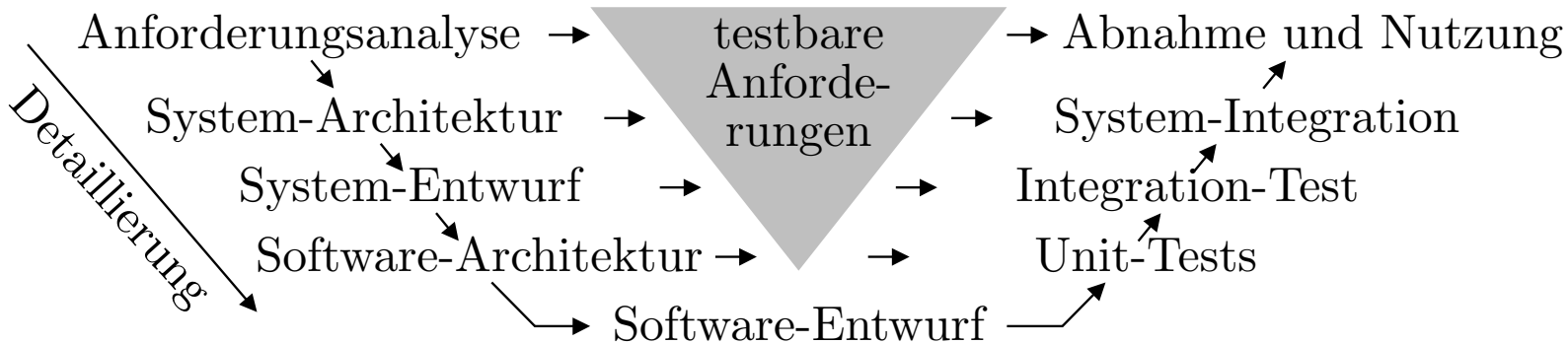
[Bec03] K. Beck. Test-driven Development - By Example. Boston: Addison-Wesley Professional, 2003. isbn: 978-0-321-14653- 3.

[Sma14] J. F. Smart. BDD in Action - Behavior-driven development for the whole software lifecycle. Birmingham: Manning Publications, 2014. isbn: 978-1-617-29165-4.

[EAD14] F. Erich et al. Report: DevOps Literature Review. In: (Oct. 2014). doi: 10.13140/2.1.5125.1201.

Testbare Anforderungen

Testbare Anforderungen



Abschn. 6.2.3: Testbare Anforderungen.

Beschreibung der Zielfunktionen für

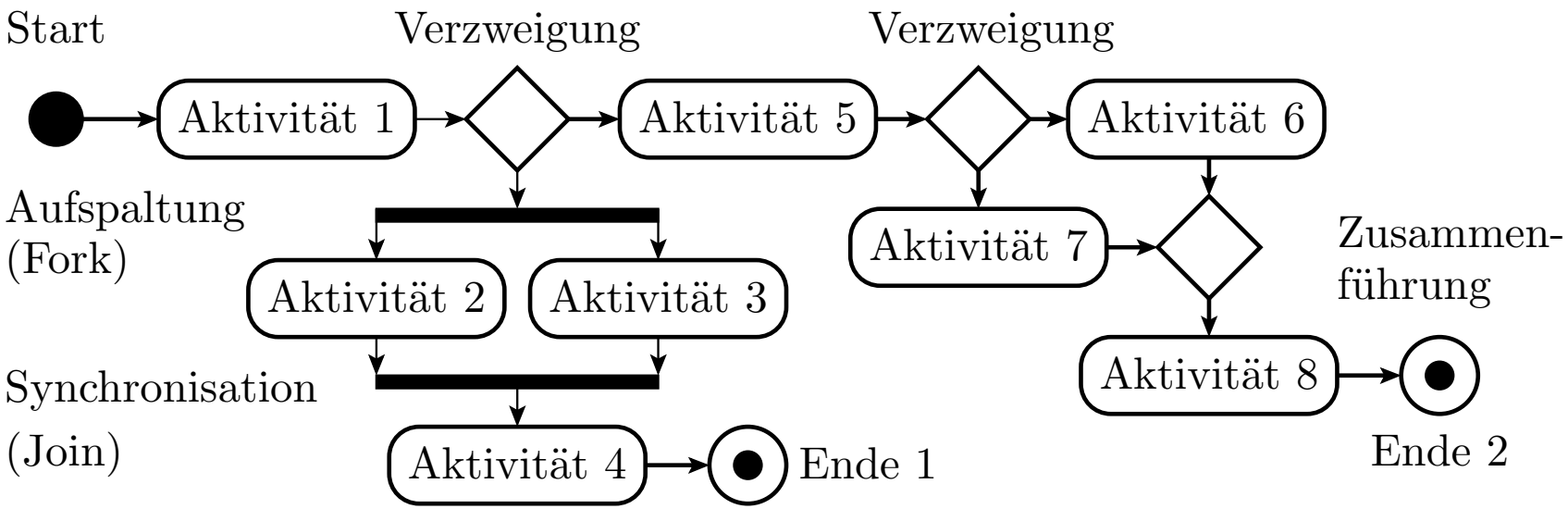
- des Gesamtsystem und
- der einzelnen Module, Schichten, Server, Clients, ...

als Sammung testbarere Anforderungen für die Tests im aufsteigenden Ast des V-Modells.

Modellierungssprache UML. Beschreibungsmittel:

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

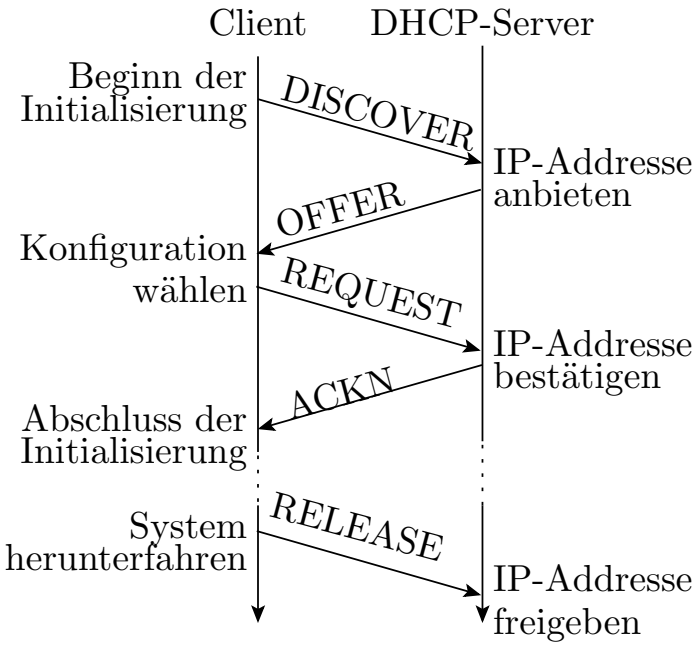
Aktivitätsdiagramm



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

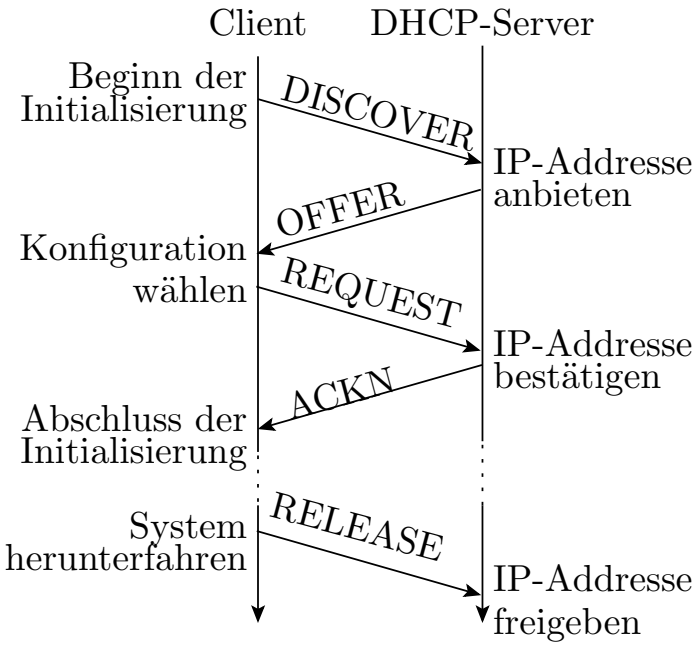
- Start, A1, A2||A3, A4, Ende 1
- Start, A1, A5, A7, A8, Ende 2
- Start, A1, A5, A6, A8, Ende 2

Sequenzdiagramm



Nachricht	Beschreibung
DISCOVER	Broadcast Clients, sucht Server
OFFER	Serverantwort mit Konfigurationsvorschlag
REQUEST	Braodcast Clients an 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.

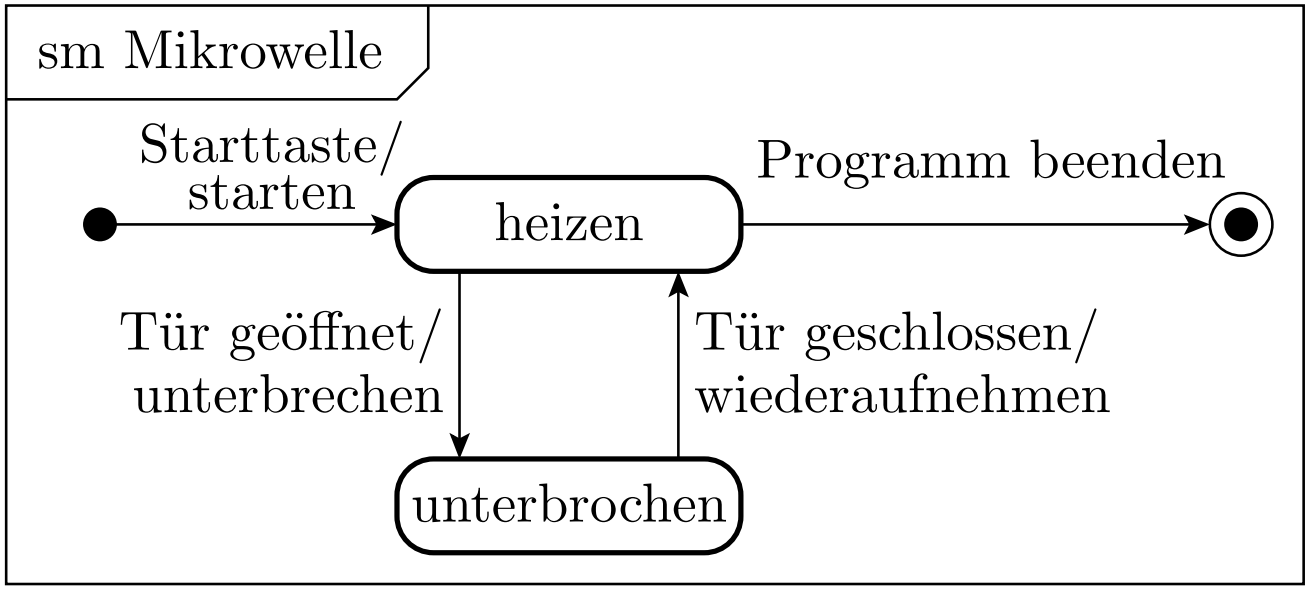


Nachricht	Beschreibung
DISCOVER	Broadcast Clients, sucht Server
OFFER	Serverantwort mit Konfigurationsvorschlag
REQUEST	Braodcast Clients an bevorzugten Server Ablehnung aller anderen Server
ACKN	Server liefertIP-Adresse
NAK	Der Server lehnt die IP-Adresse ab
DECLINE	Der Server hat ein Problem mit der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

Ableitbare Tests:

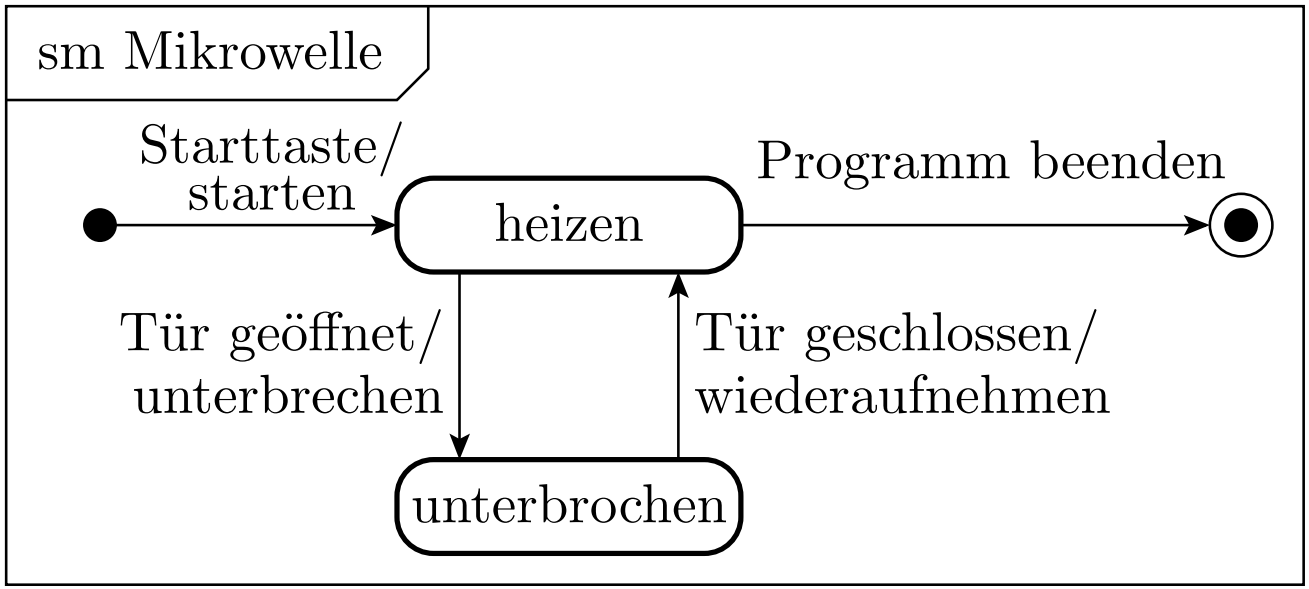
- Korrekte Abläufe mit korrekten und unzulässigen Daten.
- Korrekte Reihenfolge mit Zeitüberschreitungen.
- Unzulässige Reihenfolge der Nachrichten.
- Ursache-Wirkungsgraph für Server und Client für die Testauswahl (siehe Abschn. 6.3.6 *Ursache-Wirkungs-analyse*).

Zustandsdiagramm



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

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

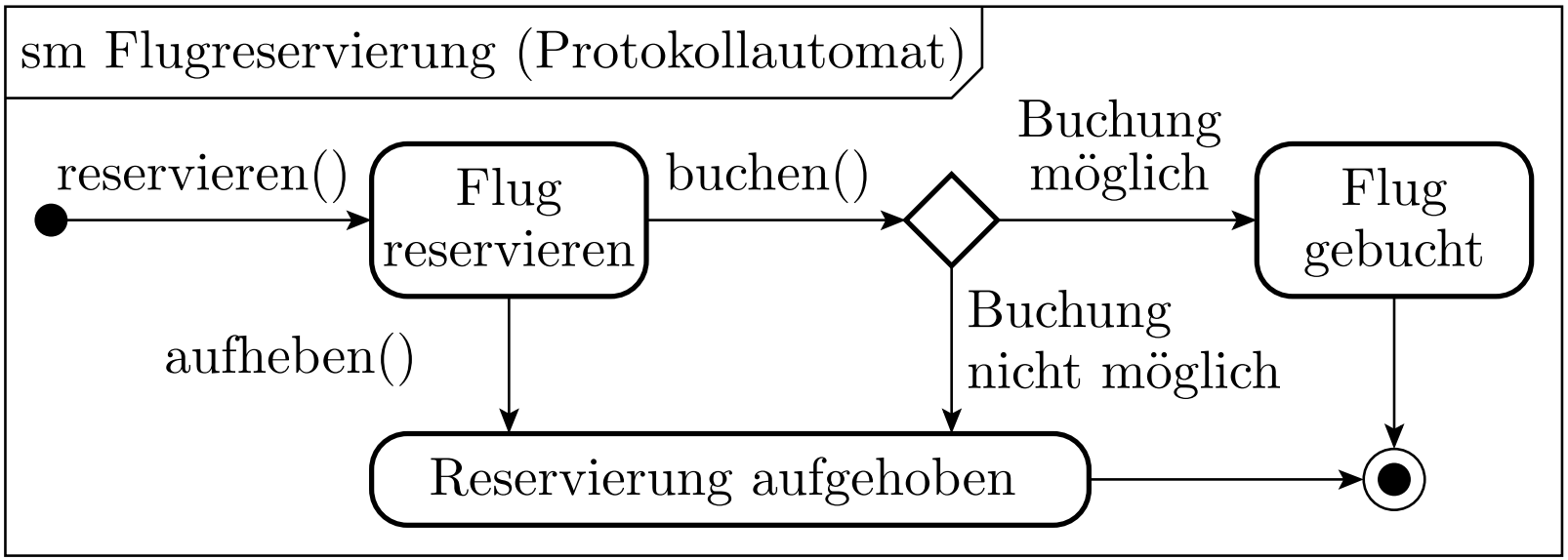


Ableitbare Tests:

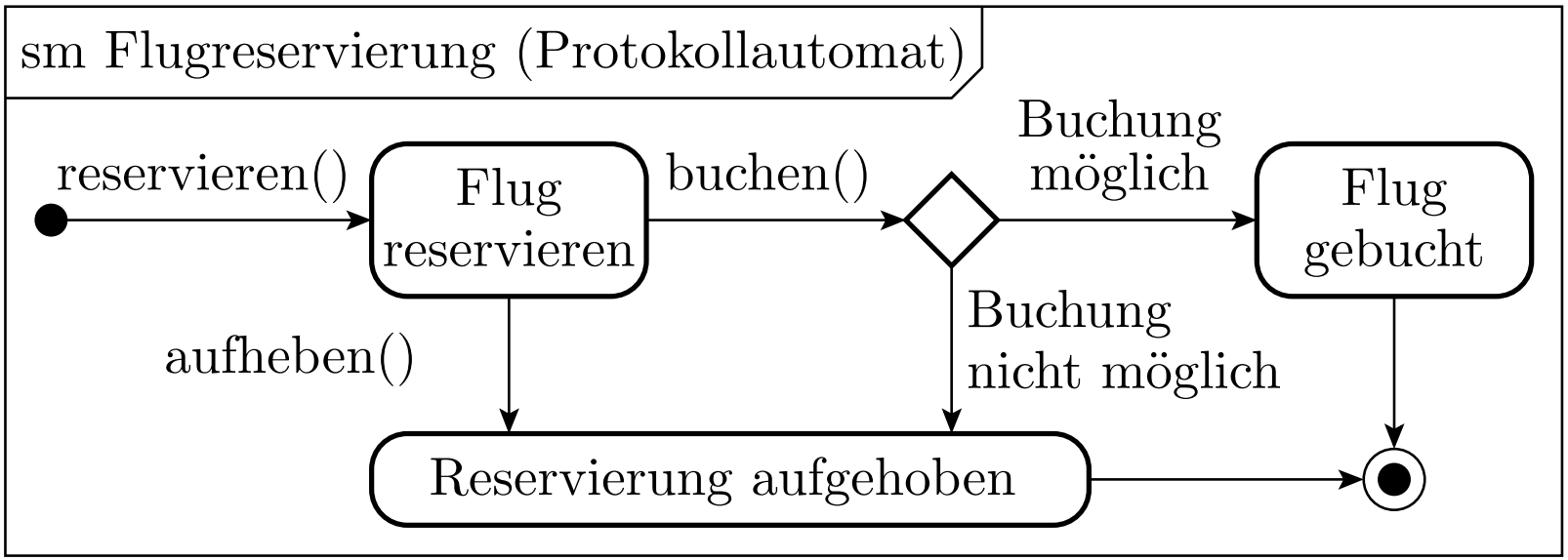
- Abläufe, die alle Knoten abdecken.
- Abläufe, die alle Kanten abdecken.
- Abläufe bis zu allen Knoten und Test der Reaktion auf nicht spezifizierte Übergangsbedingungen.

(siehe Abschn. 6.3.6 *Automaten*).

Protokollautomat



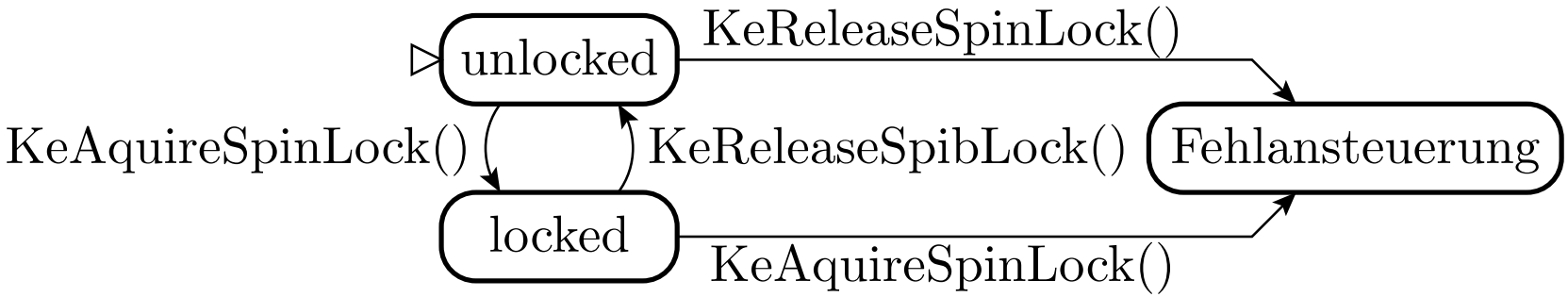
Beschreibung zulässiger Aktionsreihenfolgen. Mögliche Aktionen im Beispiel sind die Methodenaufrufe »reservieren()«, »aufheben()« und »buchen()«. Aus dem Protokollautomat im Beispiel geht hervor, dass ein Flug nur nach erfolgreicher Reservierung gebucht und dass ein einmal gebuchter Flug nicht gestrichen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

Eine zu testende Treiberfunktion

Eine Treiberfunktion ruft »KeAquire..« und »KeRelease...« u.U. mehrfach auf, in Fall-unterscheidungen, 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();
}
```

Codierung und Test

Codierung

- Bei der Codierung entstehen ca. 30% der Fehler (?), die restlichen 70% in den Entwurfsphase davor (?).
- Fehlerenstehungsrate manueller Code-Entwicklung ist ca. 10 bis 100 Fehler je 1000 NLOC.
- Fehlergenerierungsrate automatisierte Codegenerierung vergleichsweise vernachlässigbar.

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 MF.
- 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, const char *src)
```

beim Kopieren von Eingabezeichenketten in einen Puffer auf dem Stack. Wegen der fehlenden Längenkontrolle lassen sich damit auf der Stack hinter dem Puffer Variablenwerte und Rücksprungadressen gezielt mit Eingabezeichen überschreiben.

Problemvermeidung durch statische Code-Analyse:

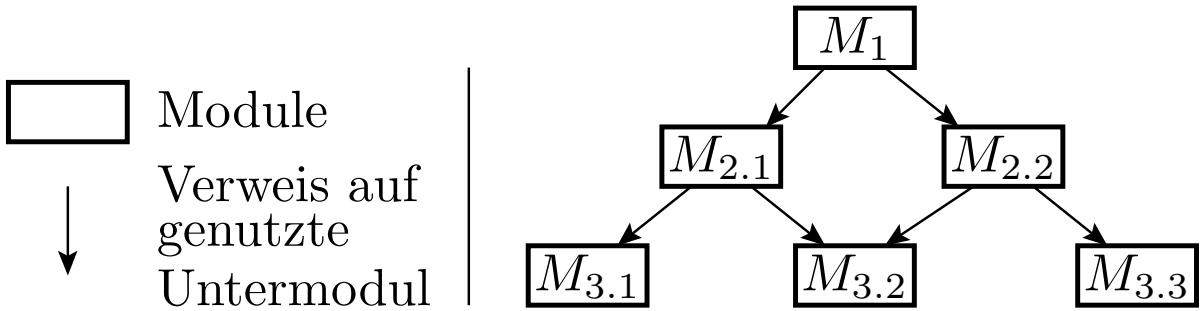
- Suche alle Aufrufe von strcpy (die Eingabedaten in Puffer kopieren).
- Ersatz durch

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

  n – Puffergröße.
- ...

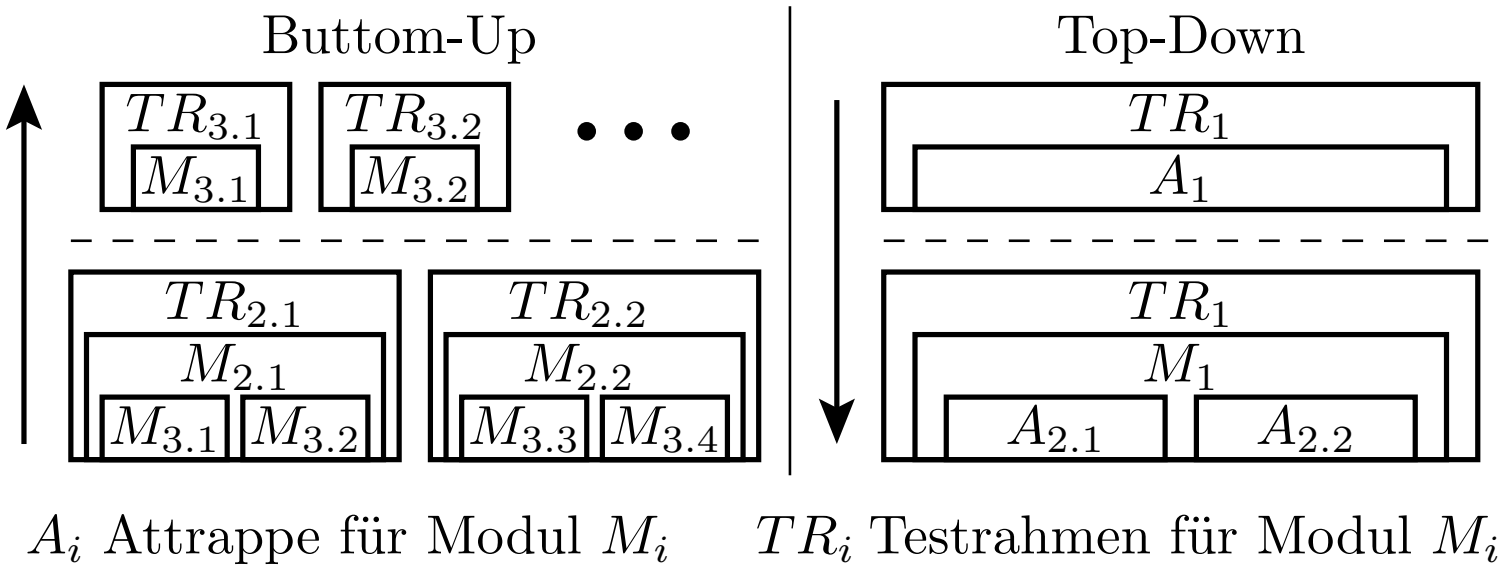
Modularisierung und Testrahmen

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.



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.

* Mit Fuzzifizierung meint man im SW-Betreich längere Zufallstets.

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, ...

Zusammenfassung

Software-Architektur

Eine gute Architektur ist die Basis für langfristig wartbare, flexible und verständliche Systeme.

- Prozedurensammlung: Am wenigsten restriktive SW-Architektur.
- Schichtenmodell: Beschränkung der nutzbaren Prozeduren auf die der Schicht darunter. Übersichtlicher, besser änderbar. Vorteile für das Testen.
- Client/Server-Modell: Server sind autonom arbeitende Dienstprogramme, Clients sind Applikationen, die Dienste der Server nutzen. Vereinfacht getrennte Entwicklung, Portierung auf andere Hardware, ...

Je komplexer die Systeme, desto mehr Restriktionen empfehlenswert.

Testbare Anforderungen

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 verhält sich, wie bereits auf dem ersten Foliensatz gezeigt, proportional

- zur Testanzahl durch die Anzahl der nicht beseitigten Fehler
- mal Anteil der nicht erkennbaren MF der MF-Behandlung im Einsatz.

Höhe Zuverlässigkeit:

- Fehler vermeiden durch
  - gereiftes Vorgehen,
  - Automatisieren, ...
- viel Testen und
- viel überwachen.

Codierung und Test

Für die praktische Programmierung gibt es allgemeine Empfehlungen, um den Aufwand und die Anzahl der entstehenden Fehler gering zu halten und Regeln, was man unbedingt vermeiden sollte.

Für sicherheitskritische Anwendungen gibt es dafür sogar Normen, wie den MISRA-Standard.

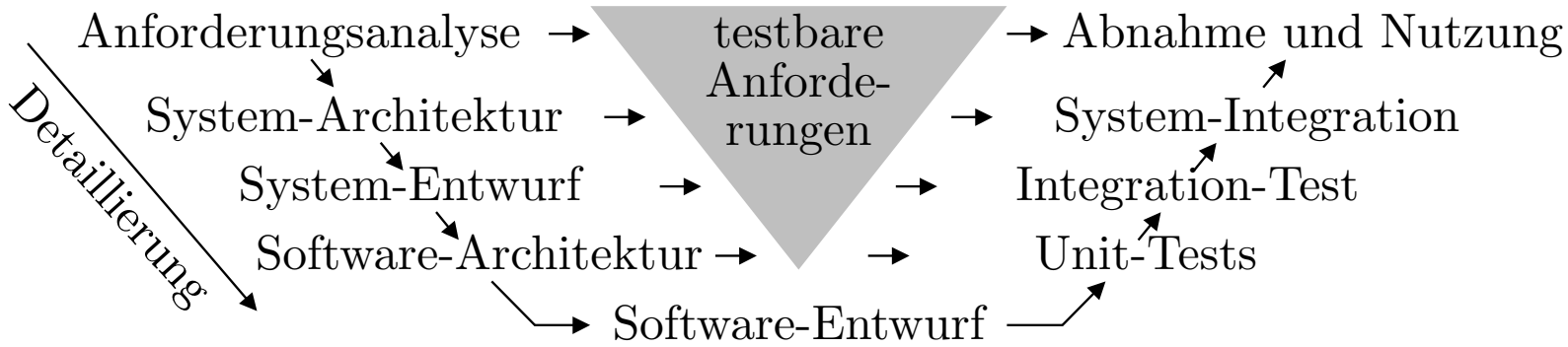
Jeder Code-Baustein muss ausprobiert werden:

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

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

Testauswahl

Testauswahl für Software



Abschn. 6.3: Testauswahl.

Testfälle sollten in allen Entwurfsphasen spezifiziert werden (V-Modell).

Für Software sind fehlerorientierte Testauswahl und die Bewertung durch Fehlerüberdeckung und Phantomfehlerrate (noch) unüblich.

Gegenüber Fertigungstests für HW gibt es in der Regel keine als fehlerfrei geltende Beschreibung des Sollverhaltens.

Testfälle aus frühen Entwurfsphasen sind genau wie Anforderungen und Zielfunktionen in der Regel nur informal spezifiziert durch

- einem Bezeichner,
- eine Beschreibung der Testabsicht, ...

Vom Testfall zum Test

Testabsicht, nachzuweisende Anforderung:

- Funktionalität, Fehlverhalten, Standardkonformität, ...
- testbare Anforderungen (siehe Abschn. 6.2.3) und
- andere Beschreibungen zur Ableitung symbolischer Tests: Äquivelenzklassen, CE-Beziehungen, ... (siehe später).

Mit der Definition der Schnittstellen und der Funktionalität dahinter schrittweise Vervollständigung zu ausführbaren Tests:

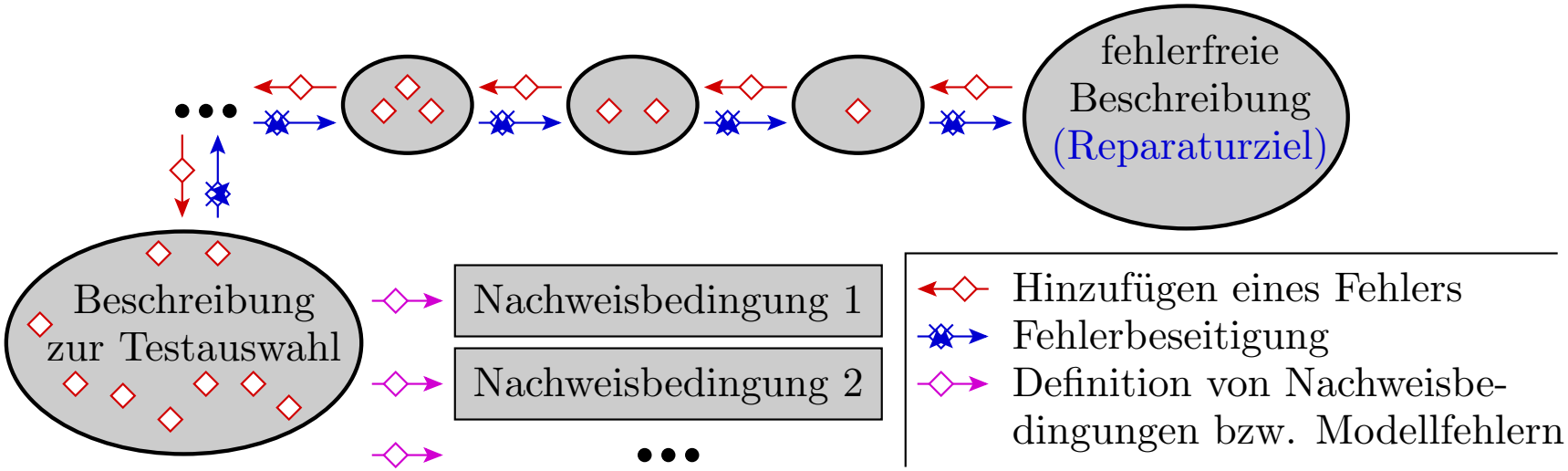
- Beispielwerte für Zustände und Eingaben und
- Kontrollkriterien für Ausgaben und Folgezustände.

Typische Reihenfolge der Testauswahl:

1. wenige typische Testeingaben, manuelle Kontrolle. Beseitigung der Abweichungen insbesondere durch Missverständnisse,
2. Komplettierung der Tests um automatische Kontrollen,
3. geziele Ergänzung um Tests für Grenzfälle,
4. unzulässige Eingaben zur Kontrolle der MF-Behandlung,
5. Zufallstests zur Kontrolle der Zuverlässigkeit (Fuzzifiezierung).

Mutationen

Mutationen statt Modellfehler



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

- Beschreibung: Programm in der Hochsprache oder übersetzt,
- Statt der Modellfehler lassen sich nur Mutationen der potentiell fehlerhaften Beschreibung konstruieren.
- Für vergessene Aspekt lassen sich keine ähnlich nachweisbaren Mutationen ableiten.
- keine fehlerfreie Beschreibung zur Sollwert-Bestimmung.

Beispiele für Mutationen

Mutationen sind geringe Verfälschungen des Testobjektbeschreibung:

- Verfälschung arithmetischer Ausdrücke (x=a+b $\Rightarrow$ x=a*b)
- Verfälschung boolescher Ausdrücke (if(a>b){} $\Rightarrow$ if(a<b){})
- Off-by-One-Fehler, z.B. Wertezuweisung (y=a+x $\Rightarrow$ y=a+x+1)
- Verfälschung der Adresszuweisung (ref=obj1 $\Rightarrow$ ref=obj2)
- Entfernen von Schlüsselworten (static int x=5 $\Rightarrow$ int x=5)

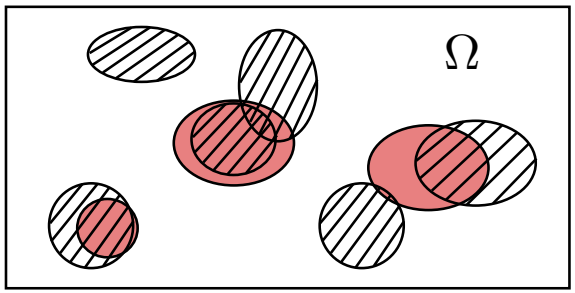
Bestimmung der Mutationsüberdeckung:

- Aufstellungstellung einer Mutationsmenge.
- Zusammenfassen identisch nachweisbarer und Streichen redundanter Mutationen, ... (siehe Abschn. 1.4.4 *Haftfehler*)

```
Wiederhole für jede Mutation
    Übersetze mutiertes Programm inc. Tests und Kontrollen
    Wiederhole für alle Test oder bis von w Test nachgewiesen
        Abarbeitung und Zählen, mit wie vielen Tests nachweisbar
```

w Anzahl der je Modellfehler gesuchten Tests. Gefunden werden alle oder keiner.

Mutationen und Fehler



Ω Menge der Eingabewerte / Teilfolgen die einen Fehler nachweisen können

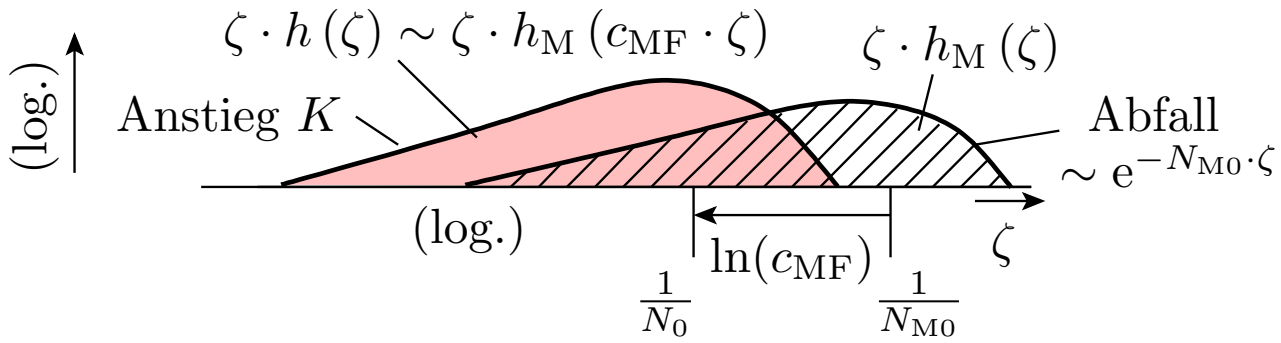
Nachweismenge einer Mutation

Nachweismenge eines Fehlers

Für jedes Programm gibt es unzählige Fehlermöglichkeiten, von denen nur ganz wenige vorhanden sind. Welche ist zum Zeitpunkt der Testauswahl unbekannt.
Mutationen sind zusätzliche kleine Verfälschungen des fehlerhaften Testobjekts, die sich idealerweise ähnlich wie die später zu findenen Fehler nachweisen lassen. Die zu fordernde Art der Ähnlichkeit hängt von der Testauswahl ab:

- zufällige Auswahl: ähnliche Dichte der MF-Rate,
- gezielte Testsuche: ähnlich nachweisbare Mutationen für einen möglichst großen Anteil der potentiellen Fehler.

Zufälliger Testauswahl



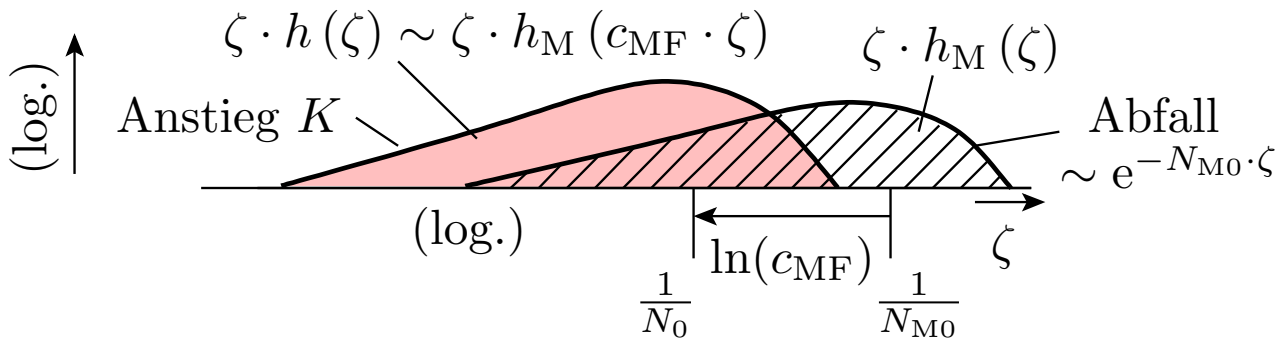
Die MF-Rate der Fehler ist abschätzungsweise gamma-verteilt

$$h(\zeta) = \frac{N_0^K}{\Gamma(K)} \cdot \zeta^{K-1} \cdot \mathrm{e}^{-N_0 \cdot \zeta} \tag{3.96}$$

mit demselben Formfaktor K für Mutationen und Fehler:

$$h_{\mathrm{M}}(\zeta) = \frac{N_{\mathrm{M0}}^K}{\Gamma(K)} \cdot \zeta^{K-1} \cdot \mathrm{e}^{-N_{\mathrm{M0}} \cdot \zeta}$$

$h(\zeta)$	Dichtefunktion der Fehlfunktionsrate.
$h_{\mathrm{M}}(\zeta)$	Dichtefunktion der Fehlfunktionsrate der Mutationsmenge.
K	Formfaktor der gamma-verteilten Fehlfunktionsrate $(0 < K < 1)$.
N_0	Skalenparameter der MF-Rate, effektive Anzahl bereits durchgeführter Tests.
N_{M0}	Skalenparameter der MF-Rate der Mutationsmenge.

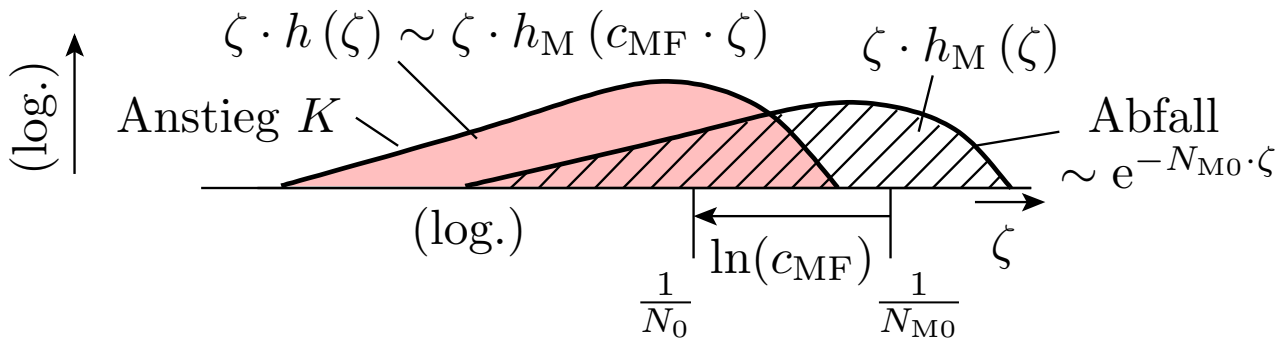


In Abhängigkeit von der Mutationsauswahl dürfen sich die Skalenparameter für Mutationen und Fehler auch um einen Skalierungsfaktor

$$c_{\mathrm{MF}} = \frac{N_0}{N_{\mathrm{M0}}} \approx 1$$

unterscheiden. Der Formfaktor K (log. Anstieg linker Kurventeil) hängt vom Testobjekt und den Fehlern ab und wird für Mutationen und potentielle Fehler als gleich angenommen.

$h(\zeta)$	Dichtefunktion der Fehlfunktionsrate.
$h_{\mathrm{M}}(\zeta)$	Dichtefunktion der Fehlfunktionsrate der Mutationsmenge.
K	Formfaktor der gamma-verteilten Fehlfunktionsrate $(0 < K < 1)$.
N_0	Skalenparameter der MF-Rate, effektive Anzahl bereits durchgeführter Tests.
N_{M0}	Skalenparameter der MF-Rate der Mutationsmenge.
c_{MF}	Mutationsspezifische Skalierung der effektiven Testanzahl.



Die Mutationsmenge wird praktisch nur für die Abschätzung bzw. Kontrolle der Annahmen über Formfaktor und Skalierung benötigt. Die Festlegung der erforderlichen Testsatzlänge erfolgt über:

$$1 - \mu_{\mathrm{FC}}(N) = \left(\frac{N}{N_0}\right)^{-K} \tag{1.57}$$

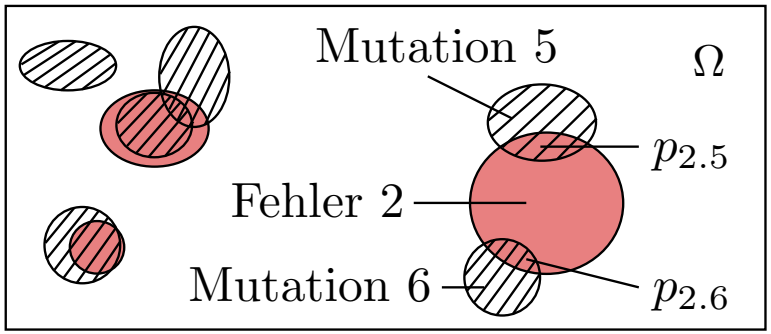
$$\zeta_{\mathrm{F}}(N) = \frac{\mu_{\mathrm{F}}(N) \cdot K}{N} \tag{1.60}$$

$$\zeta_{\mathrm{F}}(N_2) = \zeta_{\mathrm{F}}(N_1) \cdot \left(\frac{N_2}{N_1}\right)^{-(K+1)} \tag{1.61}$$

$$N_{\mathrm{Sim}} \approx c_{\mathrm{MF}} \cdot N \tag{2.11}$$

$\mu_{\mathrm{FC}}(N)$	Zu erwartende Fehlerüberdeckung in Abhängigkeit von der Testanzahl.
$\mu_{\mathrm{F}}(N)$	Zu erwartende Anzahl der Fehler, die nach N Tests nicht erkannt und beseitigt sind.
$\zeta_{\mathrm{F}}(N)$	Fehlfunktionsrate durch Fehler in Abhängigkeit von der Testanzahl.
$\mu_{\mathrm{FM}}(N)$	Zu erwartende Modellfehlerüberdeckung in Abhängigkeit von der Testanzahl.

Gezielte Testsuche



Nachweismenge Fehler i

Nachweismenge Mutation j

$p_{i.j}$ Wahrscheinlichkeit, dass ein Test für Mutation j Fehler i nachweist

Nachweiswahrscheinlichkeit für Fehler i mit $v_i \geq 1$ ähnlich nachweisbaren Mutationen

$$p_i = 1 - \prod_{j=1}^{v_i} (1 - (FC_{\mathrm{M}} \cdot (1 - (1 - p_{ij})^w))) \tag{2.10}$$

Für vergessene Aspekt (Anweisungen, Fallunterscheidungen, ...) aus der fehlerfreien Beschreibung keine ähnlich nachweisbaren Mutationen ableitbar. Gezielte Suche für angestrebte $FC \gg 0{,}5$ ungeeignet.

p_i	Nachweiswahrscheinlichkeit Fehler i.
v_i	Anzahl der ähnlich nachweisbaren Mutationen für Fehler i.
FC_{M}	Fehlerüberdeckung für Mutationen.
w	Anzahl der Tests für jede nachweisbare Mutation.
p_{ij}	Wahrscheinlichkeit, dass ein Test, der Mutation j nachweist, auch Fehler i findet.

Operationsprofil

- Art der Systemnutzung,
- Nutzungshäufigkeit unterschiedlicher SL und Eingabemuster.

Die MF-Raten der einzelnen Fehler und Mutationen im System hängen erheblich vom Operationsprofil ab (siehe Abschn. 2.2.1 *Nachweiswahrsch. ohne Gedächtnis*).

Zufallstests mit mehreren Operationsprofilen versprechen höhere Fehlerüberdeckung bei gleicher Testanzahl (siehe Abschn. 5.3.4 *Fehlerorientierte Wichtung*).

Auswahlmöglichkeiten für Operationsprofile:

- nach zu erwartender Systemnutzung für typische Anwendungen, auch als anwendungsspezifischer Zuverlässigkeitsnachweis,
- anhand spezifizierter Testfälle,
- mutationsbasiert.

Ideen, die es wert sind, weiterverfolgt zu werden.

Mutationsbasierte Operationsprofilauswahl

- Zusammenstellung Mutationsmenge. Streichen redundanter und Zusammenfassen identisch nachweisbarer Mutationen.

```
Wiederhole, bis genug Mutationen ausreichend of nachgewiesen werden
    Wahl Op.-Profil, das bisher kaum nachweisbare Mutationen bevorzugt
        Auswahl eines langen Zufallstestsatzes mit diesem Operationsprofil
        Wiederhoe für alle Tests dieses Testsatzes
            Increment der Nachweiszähler für alle nachweisbaren Mutationen
```

Viele Tests je Mutation erhöhen die Wahrscheinlichkeit, dass der Testsatz zu Mutationen ähnlich nachweisbare Fehler abdeckt.

Der zunehmend Fokus auf schlechter testbare Systemteile verbessert die Steuer- und Beobachtbarkeit für diese und darüber auch die Nachweiswahrscheinlichkeiten auch für Fehler ohne ähnlich nachweisbare Mutationen in diesen Systemteilen (vergessene Aspekte).

Ideen, die es wert sind, weiterverfolgt zu werden.

Kontrollfluss

Kontrollflussbasierte Testauswahl

Nach Standard DO-178 B genügen* für SW als Testüberdeckung:

- Nicht sicherheitskritische SW: $C_{\mathrm{I}} = 100\%$ (Anweisungsüberd.),
- SW, die bedeutende Ausfälle verursachen kann: $C_{\mathrm{B}} = 100\%$.
- Flugkritische SW: $C_{\mathrm{C}} = 100\%$ (Bedingsüberdeckung).

Alle drei Überdeckungsmaße

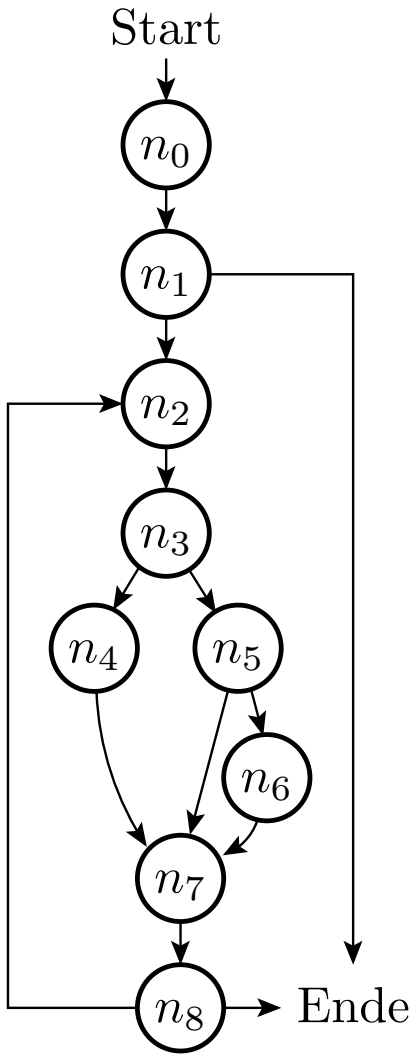
- sind am Kontrollfluss eines Programms definiert,
- lassen sich einfach bestimmen und
- vernachlässigen ähnlich wie der Toggle-Test für digitale Schaltungen die Beobachtbarkeit lokal nachweisbarer MF (siehe Abschn. 5.1.3 *Fehlermodelle für digitale Schaltkreise*).

Erweiterungsmöglichkeit um Beobachtungsbedingungen (siehe Abschn. 6.3.6 *Ursache-Wirkungs-analyse*) gelten nicht als zwingend.

C_{I} Anweisungsüberdeckung.
C_{B} Zweigüberdeckung.
C_{C} Bedingungsüberdeckung.
* Zur Abwendung von Produkthaftung für Schäden durch durch nicht erkannte Fehler.

Beispielprogramm und sein Kontrollflussgraph

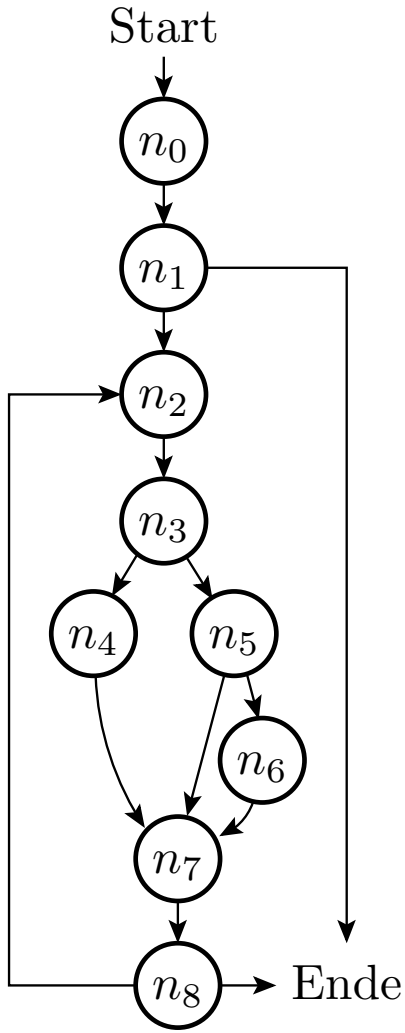
```
int Ct_A, Ct_B, Ct_N;
int ZZ(int Ct_max){
    char c;
n0: Ct_A=0; Ct_B=0; Ct_N=0;
n1: while (Ct_N<Ct_max){
n2:  c=getchar();
n3:  if (is_TypA(c))
n4:   Ct_A++;
n5:  else if (is_TypB(c))
n6:   Ct_B++;
n7:  Ct_N++;
n8: } //Test Abbruchbedingung
}
```



Kontrollflussgraphen automatisch erzeugbar.

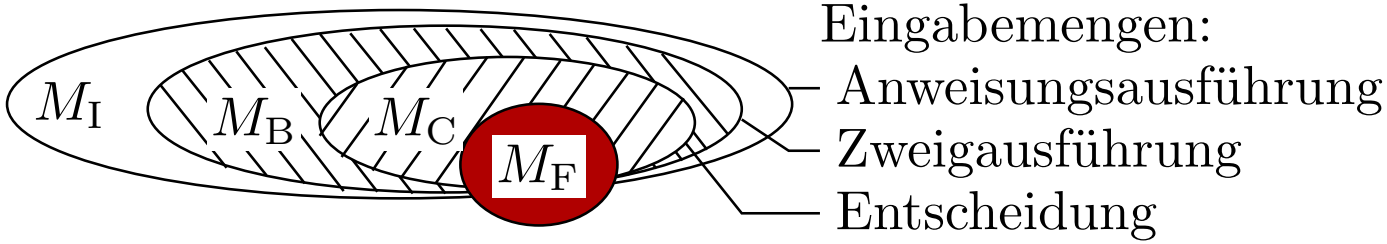
Kontrollflussbasierte Testvollständigkeitsmaße

1. Anweisungsüberdeckung: Jede Anweisung muss 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 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 von jeder Bedingung abhängen.



Gefundene Kontrollflussabläufe sind symbolische Tests, die über geeignete Eingaben zu steuern sind.

Testvollständigkeitsmaß und Fehlerüberdeckung



$$M_\mathrm{C} \subseteq M_\mathrm{B} \subseteq M_\mathrm{I}$$

- Anweisungen könnne über unterschiedliche Zweige und
- Zweige unter unterschiedlichen Bedingungen

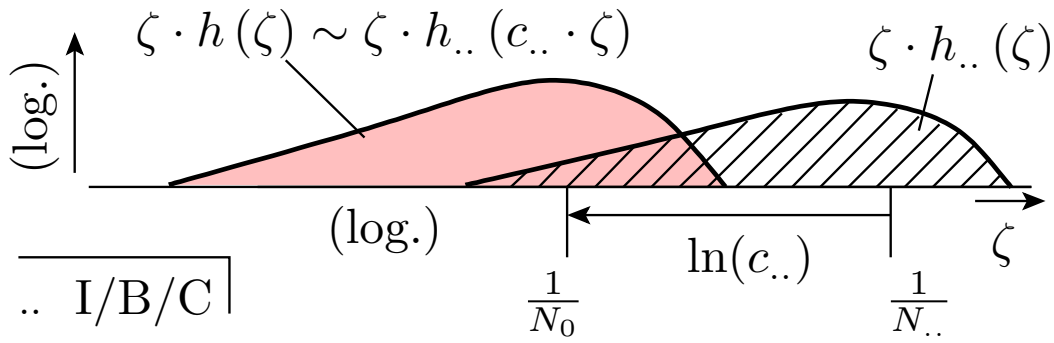
abgearbeitet werden. Testvollständigkeit:

- $C_\mathrm{B} = 100\%$ gründlicherer Test als $C_\mathrm{I} = 100\%$
- $C_\mathrm{C} = 100\%$ gründlicherer Test als $C_\mathrm{B} = 100\%$.

Der Nachweis eines Fehler i verlangt meist die Ausführung bestimmter Anweisungen, nicht unbedingt über einen bestimmten Zweig oder unter einer bestimmten Bedingung.

M_F	Nachweismenge eines ähnlich nachweisbaren Fehlers.
$M_{\mathrm{I/B/C}}$	Eingabemenge Kontrolle Anweisungs- / Zweig- / Bedingungsausführung.
$C_{\mathrm{I/B/C}}$	Anweisungs-, Zweig- bzw. Bedingungsüberdeckung.

Beobachtbarkeit nicht gefordert



Fehlernachweis verlangt zusätzlich zur Steuerung Ausführung Beobachtbarkkeit fehlerbedingter falscher Anweisungs-, Zeig- oder Bedingungsausführung. Die Testlängenskalierung

$$c_{..} = \frac{N_0}{N_{..0}} \quad \text{mit .. für I/B/C}$$

tendiert gegen die mittlere Wahrscheinlichkeit der Beobachtbarkeit, die erheblich von den einprogrammierten Kontrollen abhängt ...

$h_{..}(\zeta)$	Dichtefunktion der Ausführungsrate der Anweisungen, Zweige bzw. Bedingungen.
$N_{..}$	Skalenparameter der Ausführungsrate der Anweisungen, Zweige bzw. Bedingungen.
N_0	Skalenparameter der MF-Rate, effektive Anzahl bereits durchgeführter Tests.
$c_{\mathrm{I/B/C}}$	Skalierung der effektiven Testanzahl für Anweisungs- / Zweigs- / Bedingungsabdeckung.

... erheblich von den einprogrammierten Kontrollen abhängt

- für eine Abarbeitung im Schrittbetrieb und manuelle Inspektion der Anweisungsausführung $c_{\mathrm{C/B/I}}$ nahe 1,
- nur Formatkontrollen der Ausgabe $c_{\mathrm{C/B/I}} \ll 1$.

Die zu erwartende Fehlerüberdeckung tendiert nach ?????????? Gl. 2.11 gegen die Anweisungs-, Zweig- bzw. Bedingsüberdeckung der $c_{..}$-fachen Testsatzlänge:

$$\mu_{\mathrm{FC}}(N) = C_{..}(c_{..} \cdot N) \quad \text{mit .. für I/B/C} \tag{6.1}$$

Aus der Teilmengenbeziehung $M_{\mathrm{C}} \subseteq M_{\mathrm{B}} \subseteq M_{\mathrm{I}}$ folgt:

$$c_{\mathrm{I}} \leq c_{\mathrm{B}} \leq c_{\mathrm{C}}$$

Experimente zur Abschätzung der Größenordnung von c_{I}, c_{B} und c_{C} sind dem Autor nicht bekannt.

Ideen, die es wert sind, weiterverfolgt zu werden.

$\mu_{\mathrm{FC}}(N)$ Zu erwartende Fehlerüberdeckung in Abhängigkeit von der Testanzahl.
$C_{\mathrm{I/B/C}}$ Anweisungs-, Zweig- bzw. Bedingungsüberdeckung.
$c_{\mathrm{I/B/C}}$ Skalierung der effektiven Testanzahl für Anweisungs- / Zweigs- / Bedingungsabdeckung.
N Anzahl der Tests.

Überdeckung

Bestimmung der Überdeckungsmerkmale

Nach Standard DO-178 B ist für jedes Überdeckungskriterium (Bedingung, Zweig oder Anweisung) für den gegebenen Code zu kontrollieren, dass der Testsatz es mindestens einmal abdeckt.

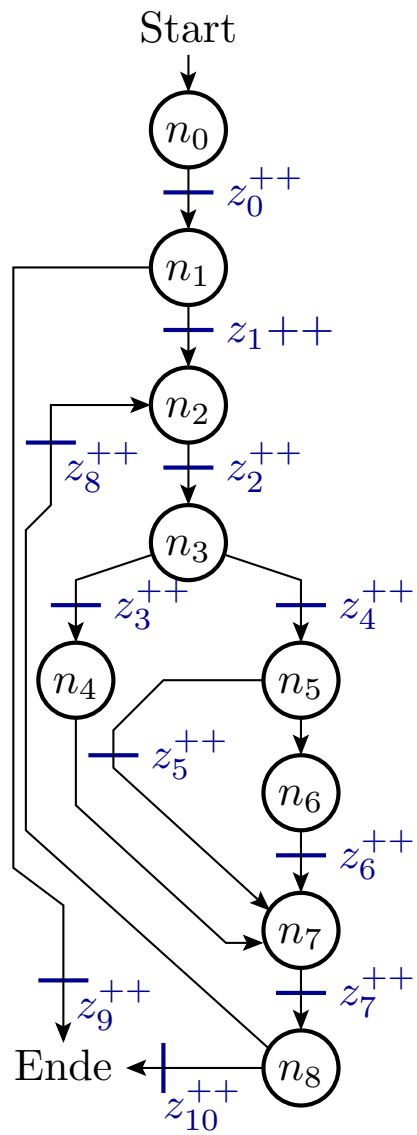
Bei einer Wahrscheinlichkeit der Beobachtbarkeit falscher Anweisungs-, Zweig- oder Bedingungsausführung $\ll 1$ ist es zweckmäßiger zu fordern, dass alle Überdeckungskriterien vom Testsatz $w \gg 1$ mal abgedeckt werden. Zähler statt Bits zum Abhaken für erfüllte Kriterien.

Nicht ausführbare Anweisungen, Zweige und Bedingungen sind redundant, zählen im Überdeckungsmaß nicht mit und sind nach den üblichen SW-Richtlinien möglichst aus dem Release-Code zu entfernen. Problematisch bei nachgenutztem Code.

Entfernung aus dem Release-Code auch zwingend für einprogrammierten Testhilfen, z.B. um Tests gezielt zum Versagen zu bringen, Hintertüren mit einprogrammiertem Universalpasswort, ...

Zähler für die Kantenausführung

```
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]}
}
```



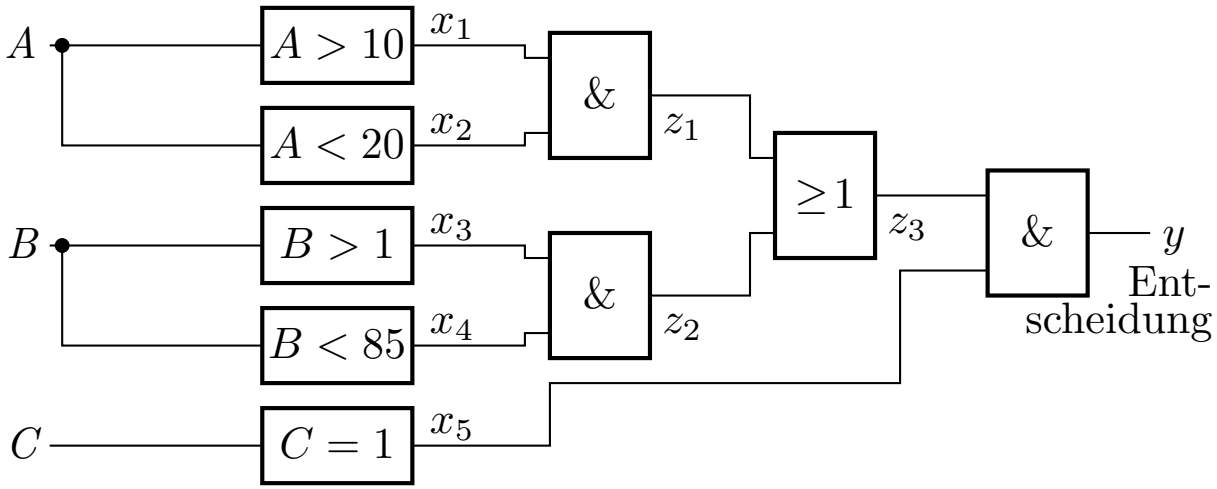
[1] Zur Unterbringung aller Zähler Schleife in Maschinenbefehle auflösen.

Off-By-One- und Haftfehler, statt Bedingungen

Ein logischer Ausdruck, z.B.

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

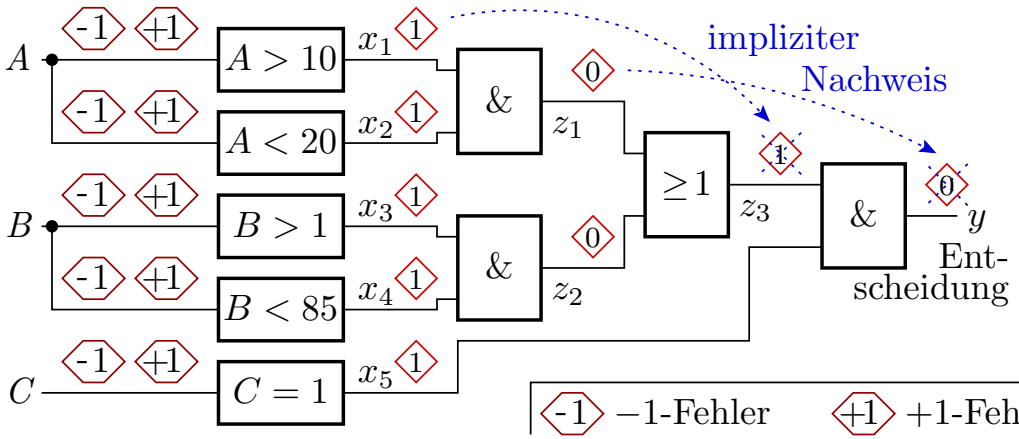
enthält Bedingungen für A, B und C und logische Verknüpfungen, auch nachbildbar durch eine Schaltung aus Gattern und Vergleichern:



Statt »Bedingungskontrollen« Off-By-One- und sa- Fehler.

Berechnungsfluss mit eingezeichneten Fehlern

Anfangsfehlermenge: je Vergleich zwei Off-By-One-Fehler und je Gatter zwei Haftfehler. Beseitigung redundanter und Zusammenfassen identischer Fehler, ...



	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

Fehlerannahmen statt zu überprüfender Bedingsmengen erlaubt:

- auch die Erweiterung auf Mutationsüberdeckungen und
- die Übernahme von Begriffe, Algorithmen Erfahrungen für die Fehlersimulation und Testauswahl für DIC.

Ideen, die es wert sind, weiterverfolgt zu werden.

Def-Use-Ketten

Def-Use-Ketten

Def-Use-Tupel: Paare aufeinanderfolgender Schreib- und Lesezugriffe auf eine Variable. Modell für notwendige, aber nicht hinreichende Beobachtungsbedingungen.

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.

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.

Für hohe Korrelation zwischen Fehler- und Kriterienüberdeckung:

»Anweisungsauführung mit Def-Use-Kette zu einer Kontrolle«.

Notwändige, aber nicht hinreichende Bedingung für den Nachweis einer MF, die eine Anweisungsergebnis verfälscht.

Algorithmus zum Zählen dieser Überdeckungskriterien:

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

Ideen, die es wert sind, weiterverfolgt zu werden.

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 MF zur Entstehungsursache im Def-Use-Graphen.

Verwendung zur Fehlerlokalisierung

Rückverfolgung des Def-Use-Graphen zur Entstehungsursache der MF am Beispiel »größter gemeinsamer Teiler«:

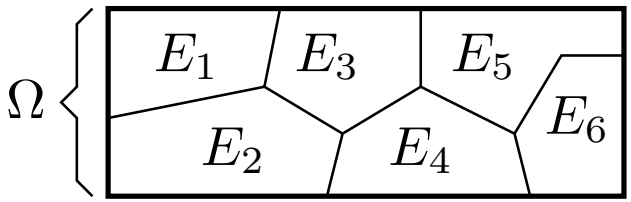
```
int ggt(int a, int b){
n0:   int c = a;
n1:   int d = b;
n2:   if(c == 0)
n3:     return d;
n4:   while(d != 0){
n5:     if(c > d)
n6:       c = c - d;
n7:     else
n8:       d = d - c;
      }
```

Wenn »n9« MF, dann sind die möglichen »Defs«, nach denen Testausgaben vor dem nächsten Testdurchlauf einzuprogrammieren sind, »n0« und »n6«

Ideen: interaktives Unterstützungsprogramm für die Fehlersuche durch Pfadrückverfolgung entgegen dem Berechnungsfluss (siehe Folie 1.120 *Rückverfolgung von Datenverfälschungen*).

Äquivalenzklassen

Äquivalenzklassen



Ω Eingaberaum
E_i Äquivalenzklasse

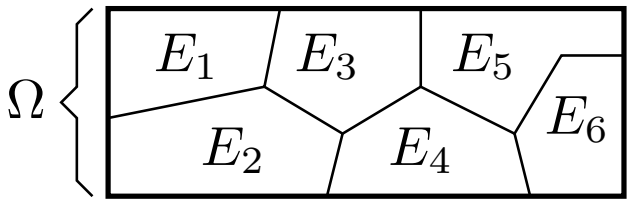
Äquivalenzklassen sind Eingabemengen ähnlich zu verarbeitender Daten, beschreibbar durch Fallunterscheidungen:

```
wenn       <Bedingung 1> dann <Berechnung E1>;
sonst wenn <Bedingung 2> dann <Berechnung E2>;
sonst ...
```

So eine »Wenn-Dann-Beschreibungen« kann eine Funktions- oder Testspezifikationen aus einer frühen Entwurfsphasen z.B. der Anforderungsanalyse oder auch ein halb fertiges Programm sein:

```
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");
}
```

Äquivalenzklassen und Kontrollfluss



Ω Eingaberaum
E_i Äquivalenzklasse

```
wenn       <Bedingung 1> dann <Berechnung E1>;
sonst wenn <Bedingung 2> dann <Berechnung E2>;
sonst ...
```

Ausgehend von der »Wenn-Dann-Beschreibungen:

- Auswahl einer Teststichprobe für alle Äquivalenzklassen entspricht »Anweisungsüberdeckung«,
- Tests an den Grenzen zwischen Äqivalenzklassen entspricht »Bedingungsüberdeckung«.

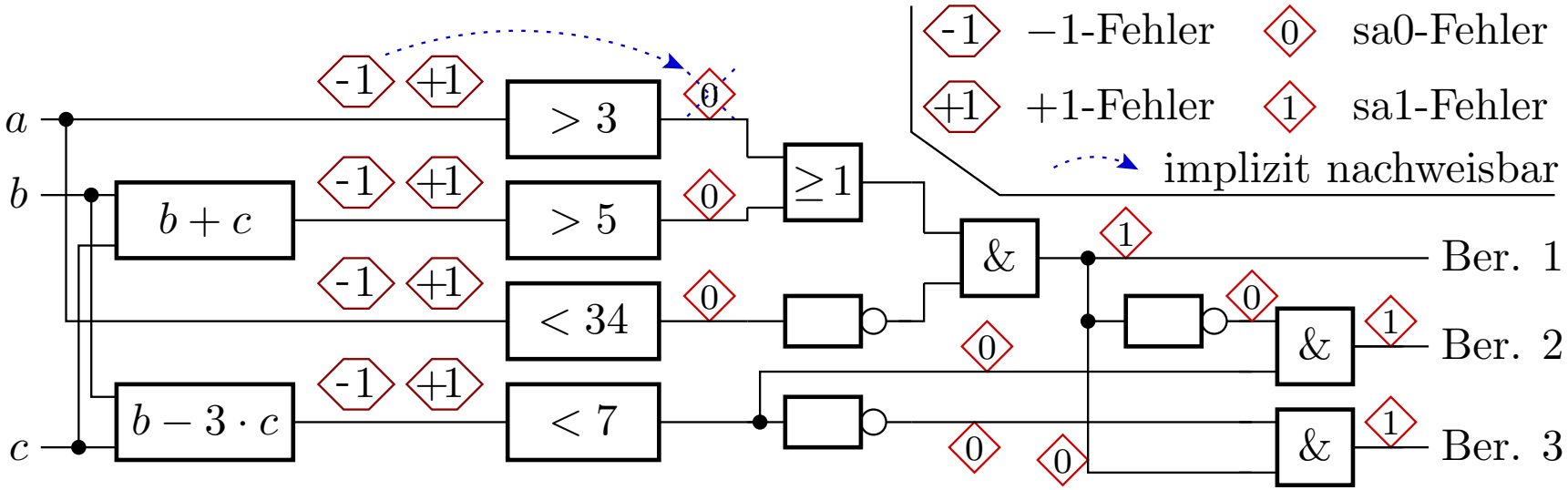
Vorteile gegenüber Auswahl / Bewertung am fertigen Programm:

- Testerstellung vor oder parallel zum Programmentwurf möglich,
- Diversität der »Wenn-Dann-Beschreibung« zum zu testenden Programm.

Fehlerannahmen statt Bedingsmengen

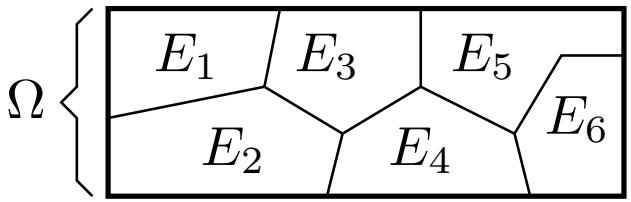
```
int fkt(int a, int b, int c){
 if((a>3)||(b+c>5))&& !(a<34))printf("Berechn. 1");
 else if(b-3*c<7)             printf("Berechn. 2");
 else                         printf("Berechn. 3");
}
```

Wenn-Dann-Berechnungsfluss mit Off-By-One- und Haftfehlern:



Ideen, die es wert sind, weiterverfolgt zu werden.

Auswahl von Operationsprofilen für den Test



Ω Eingaberaum
E_i Äquivalenzklasse

Wichtung der Testanzahl nach:

- der erwarteten Nutzungshäufigkeit für unterschiedliche Einsatzzenarien,
- der zu erwartendene Anzahl der Fehler in den den Äquivalenzklassen zugeordeneten Realisierungen,
- den Kosten durch Fehler in den zugeordneten Realisierungen,
- ...

Ideen, die es wert sind, weiterverfolgt zu werden.

CE-Analyse

Ursache-Wirkungs-Analyse

Abschn. 6.3.6: Ursache-Wirkungs-analyse.

- Ursachen (cause, wenn): Auslöser von Aktionen.
- Wirkung (effect, dann): auszulösende Aktionen.

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

Zwischen den Ursachen und Wirkungen werden logische Verknüpfungen formuliert.Abschn. 6.3.6: Ursache-Wirkungs-analyse.

Ursache-Wirkungs-Graphen sind eine zur Äquvalenzklassenbeschreibung alternative Form einer Wenn-Dann-Beschreibung für die Spezifikation von Tests mit denselben Vorteilen:

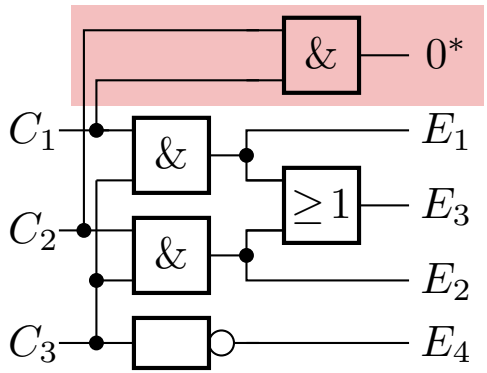
- Testerstellung vor oder parallel zum Programmentwurf möglich,
- Diversität zum zu testenden Programm.
- Rückführbar auf die Testauswahl für digitale Schaltungen.

Beispiel »Zähle Zeichen«

- Wirkungen:
  E_1: Anzahl_TypA* +1
  E_2: Anzahl_TypB* +1
  E_3: Gesamtzahl +1
  E_4: Programm beenden
- Ursachen:
  C_1: Zeichen ist vom Typ A
  C_2: Zeichen ist vom Typ B
  C_3: Zeichenanzahl < Maximalwert
- Sich ausschließende Ursachen:
  UND-Verknüpfung muss »0« sein.

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

CE-Ersatzschaltung



* Eingabe kann nicht gleichzeitig Typ A und Typ B sein

Test mit allen einstellbaren Ursachenkombinationen

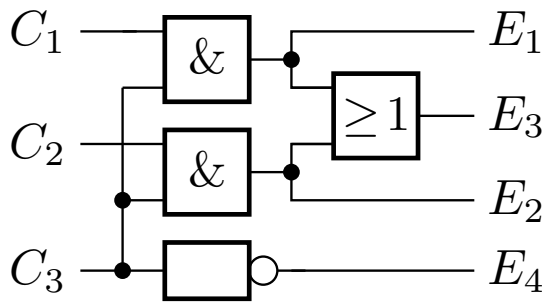
C_1	0	1	0	1	0	1	0	1
C_2	0	0	1	1	0	0	1	1
C_3	0	0	0	0	1	1	1	1
E_1	0	0	0		0	1	0	
E_2	0	0	0		0	0	1	
E_3	0	0	0		0	1	1	
E_4	1	1	1		0	0	0	

C_i Ursache (cause) i.
E_i Wirkung (effect) i.
* Zeichen vom vom Typ A sind Ziffern und vom Typ B Großbuchstaben.

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;
C3:   while (Ct_N<Ct_max){
        c=getchar();
C1:     if (is_TypA(c))
E1:       Ct_A++;
C2:     else if (is_TypB(c))
E2:       Ct_B++;
E3:     Ct_N++;
E4:     }
      }
```



Test mit allen einstellbaren Ursachenkombinationen

C_1	0	1	0	1	0	1	0	1
C_2	0	0	1	1	0	0	1	1
C_3	0	0	0	0	1	1	1	1
E_1	0	0	0		0	1	0	
E_2	0	0	0		0	0	1	
E_3	0	0	0		0	1	1	
E_4	1	1	1		0	0	0	

Testbeispiel konkret /symbolisch

Funktions-aufruf	Eingabe	Sollzähl-werte	Ursachen C_1	C_2	C_3	Wirkungen E_1	E_2	E_3	E_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

C_1 Zeichen ist vom Typ A (Ziffer)
C_2 Zeichen ist vom Typ B (Großbuchstabe)
C_3 max. Zählwert nicht erreicht
– es wird kein Zeichen gelesen

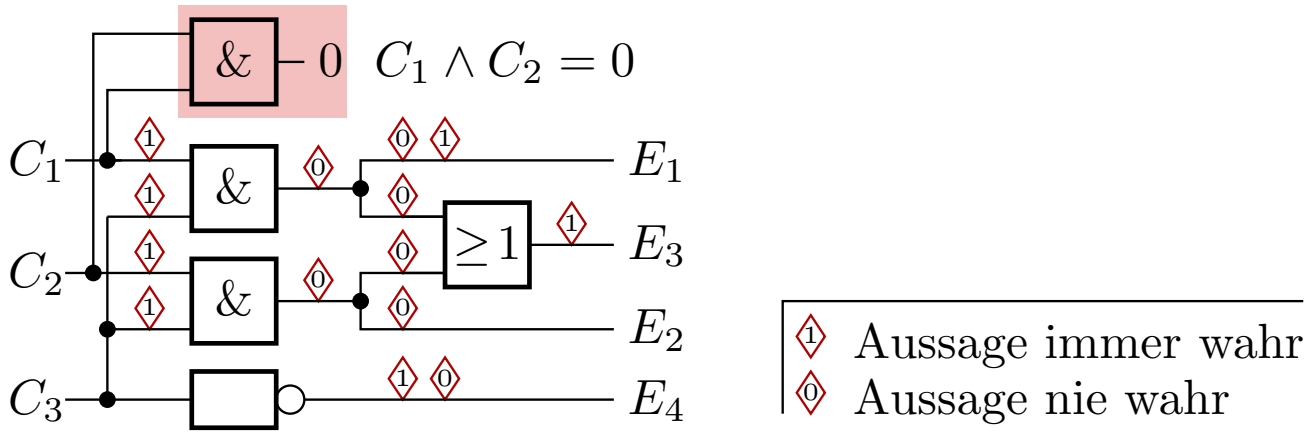
E_1 Ct_A++
E_2 Ct_B++
E_3 Ct_N++
E_4 Ende

Ungereimtheiten / Haftfehler

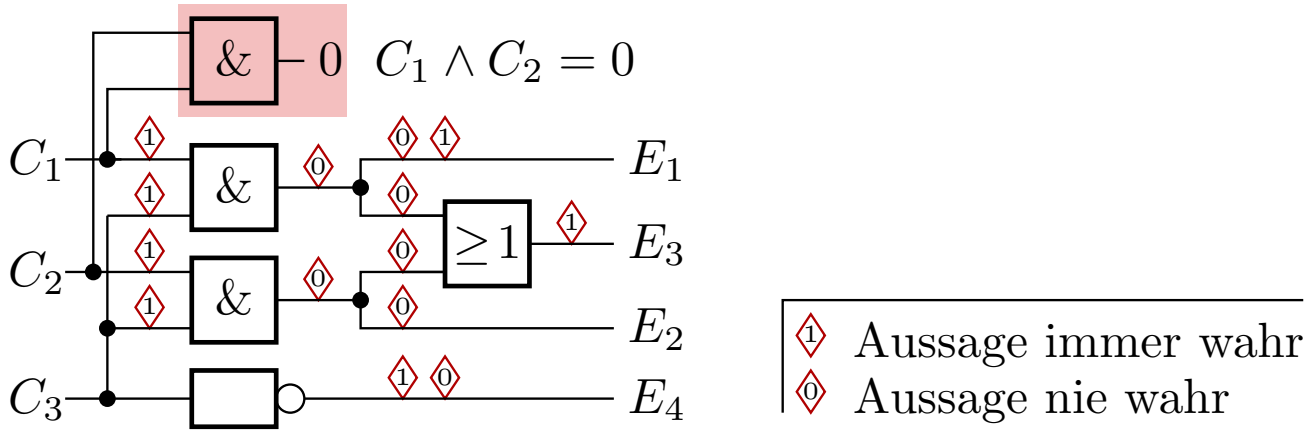
Erkennbare Ungereimtheiten:

- Im CE-Modell können bei Ursache $C_3 = 0$ (max. Zählwert erreicht) Zeichen vom Type A oder B eingegeben werden, im Programm nicht. Wie lautet das gewünschte Sollverhalten?

Haftfehler in der CE-Ersatzschaltung (identisch nachweisbare Fehler zusammengefasst):



C_i	Ursache (cause) i.
E_i	Wirkung (effect) i.

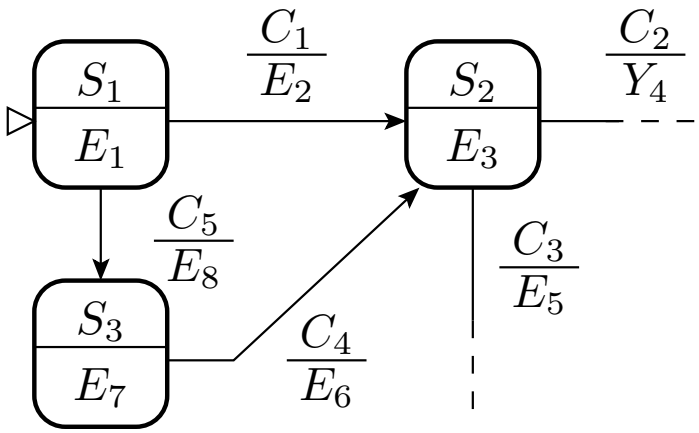


- Im Beispiel würde ein Test mit allen Kombinationen von Ursachen auch alle nachweisbaren Haftfehler erfassen.
- Für eine große Anzahl von Ursachen kann die Anzahl der Haftfehler auch wesentlich kleiner als die Anzahl der Ursachenkombinationen sein.
- Nach Berechnung der gleichzeitig zu (de-) aktivierenden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

Ideen, die es wert sind, weiterverfolgt zu werden.

Automaten

Zielfunktion als Automat

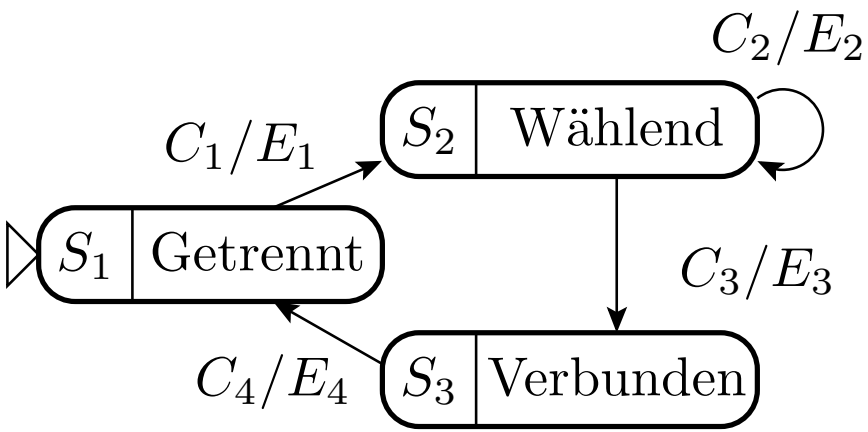


Beschreibung der Zielfunktion durch Mengen von Eingaben, Ausgaben, Zuständen und Zustandsübergänge mit Übergangsbedingungen. Die Zustände bzw. die Zustandsübergängen steuern Aktionen. Wie im CE-Modell werden bei Automaten für die Testauswahl die Ursachen (Bedingungen für Zustandsübergänge) und die Wirkungen (gesteuerte Aktionen) binarisiert.

Abschn. 6.3.6: Automaten.

S_i	Zustand i.
$\triangleright$	Startzustand.
C_j	Übergangsbedingung j.
E_k	Berechnung (effect) k, wahlweise einem Zustand oder einer Kante zugeordnete.

Auf- und Abbau einer Telefonverbindung

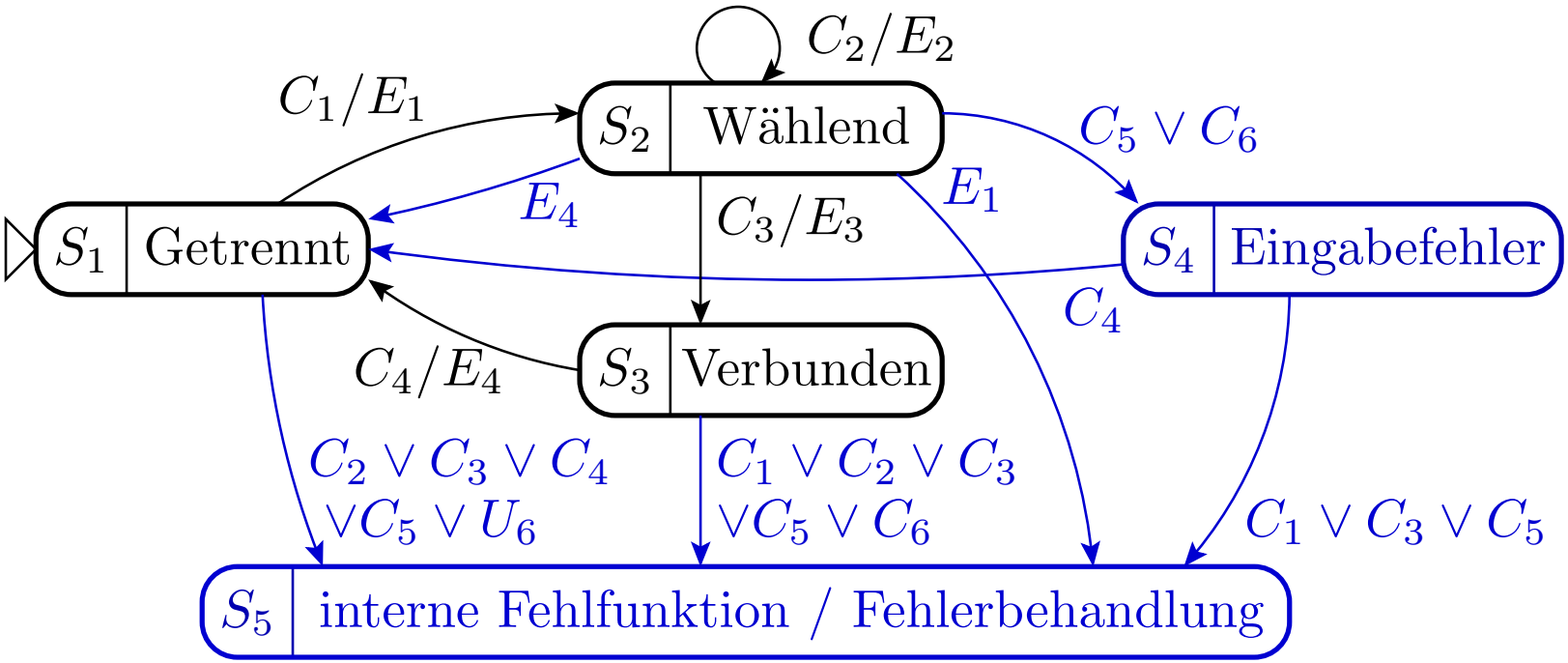


C_1 Abnehmen
C_2 Ziffer wählen
C_3 Rufnummer gültig
C_4 Auflegen
E_1 Rufnummer zurücksetzen
E_2 Ziffer zur Rufnummer hinzufügen
E_3 Verbindung aufbauen
E_4 Verbindung trennen

- Test der Sollfunktion: $C_1 \rightarrow C_2 \rightarrow \ldots \rightarrow C_3 \rightarrow C_4$
- Verhalten für andere Eingabefolgen?
  - Abnehmen, Wählen, Auflegen ($C_1 \rightarrow C_2 \rightarrow C_4$)
  - Abnehmen, Wählen, Wählen, falsche Nummer)
  - ...

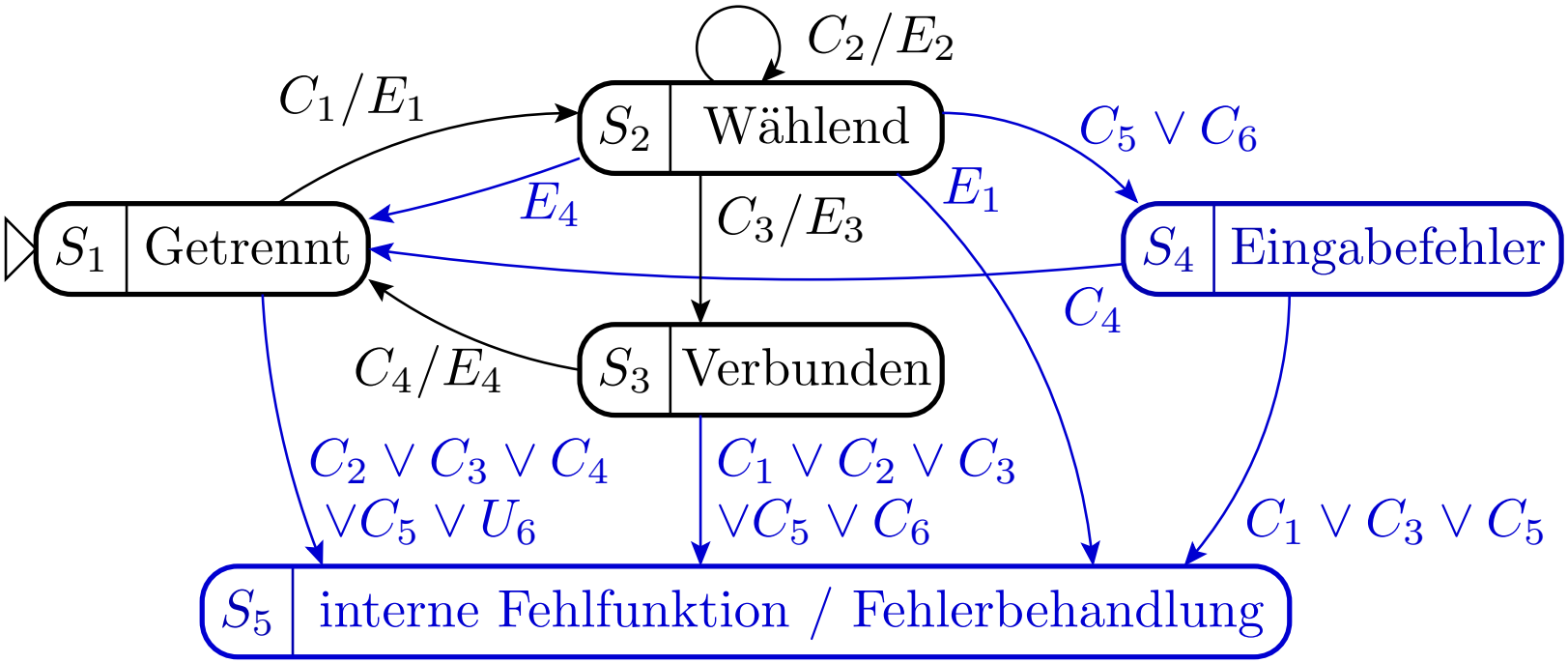
$\Rightarrow$ Ablaufgraph ist noch unvollständig

Ergänzen von Fehlerzuständen



C_1	Abnehmen	E_1	Rufnummer zurücksetzen
C_2	Ziffer wählen	E_2	Ziffer zur Rufnummer hinzufügen
C_3	Rufnummer gültig	E_3	Verbindung aufbauen
C_4	Auflegen	E_4	Verbindung trennen
C_5	Rufnummer ungültig		
C_6	Timeout		

Testauswahl

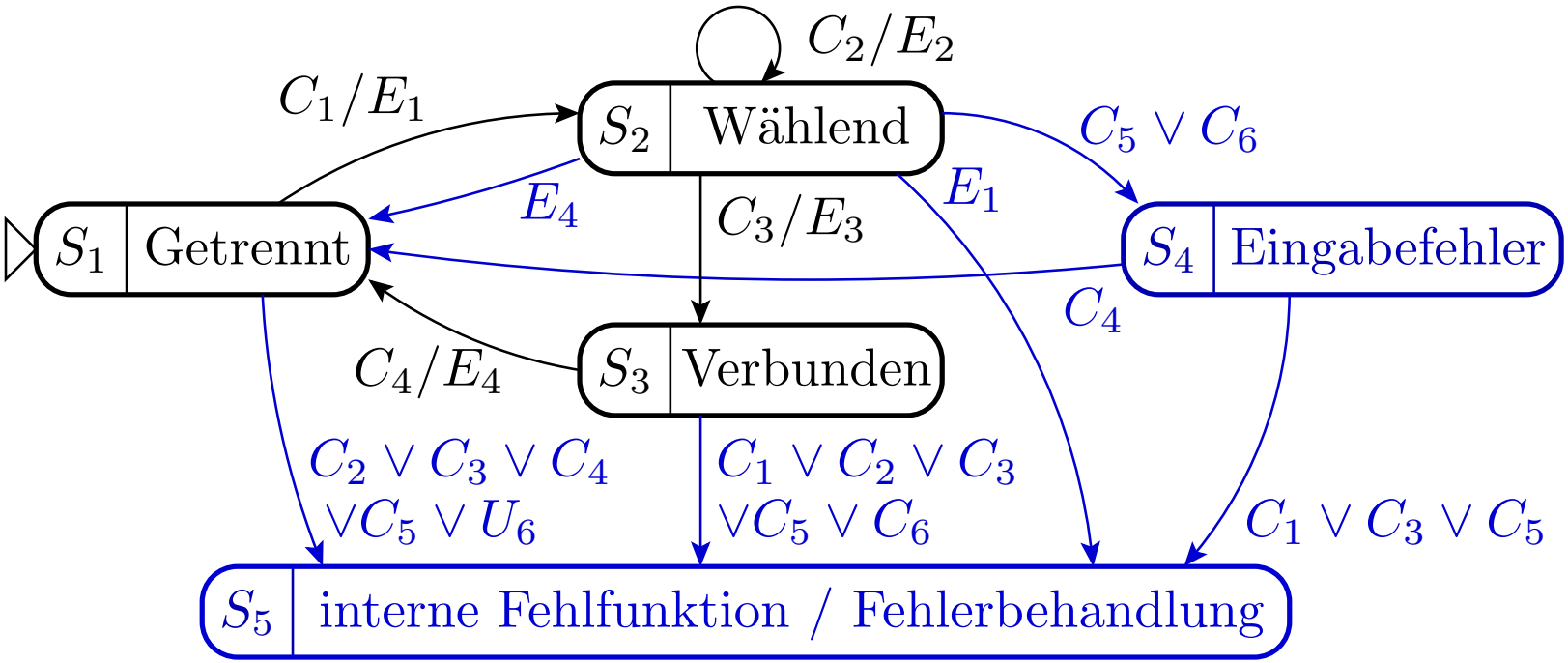


Auswahl von Abläufen mit denen:

- alle Zustände,
- alle Kanten oder
- alle Kantenbedingungen

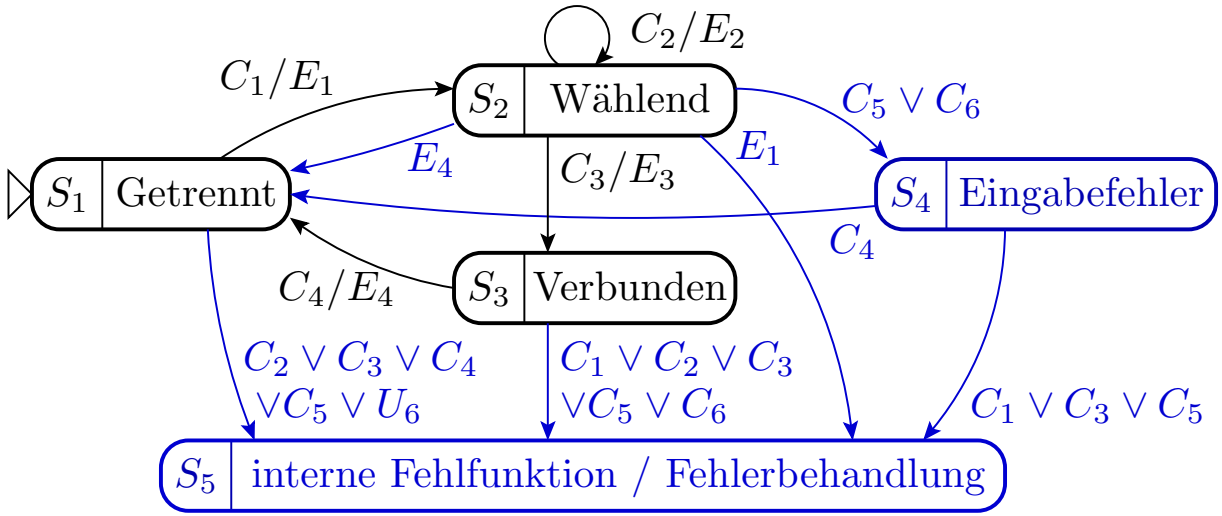
abgedeckt werden. Das Ergebnis der Testauswahl sind symbolische Tests in Form von Folgen auszulösender Ursachen.

Symbolische Tests



- Abheben (C_1), Wählen (C_2), Nummer gültig (C_3), Auflegen (C_4).
- Abheben (C_1), Wählen (C_2), Wählen (C_2), Auflegen (C_4).
- Abheben (C_1), Wählen (C_2), Timeout (C_6), Auflegen (C_4).
- Abheb. (C_1), Wählen (C_2), Nummer ungültig (C_5), Auflegen (C_4).
- ...

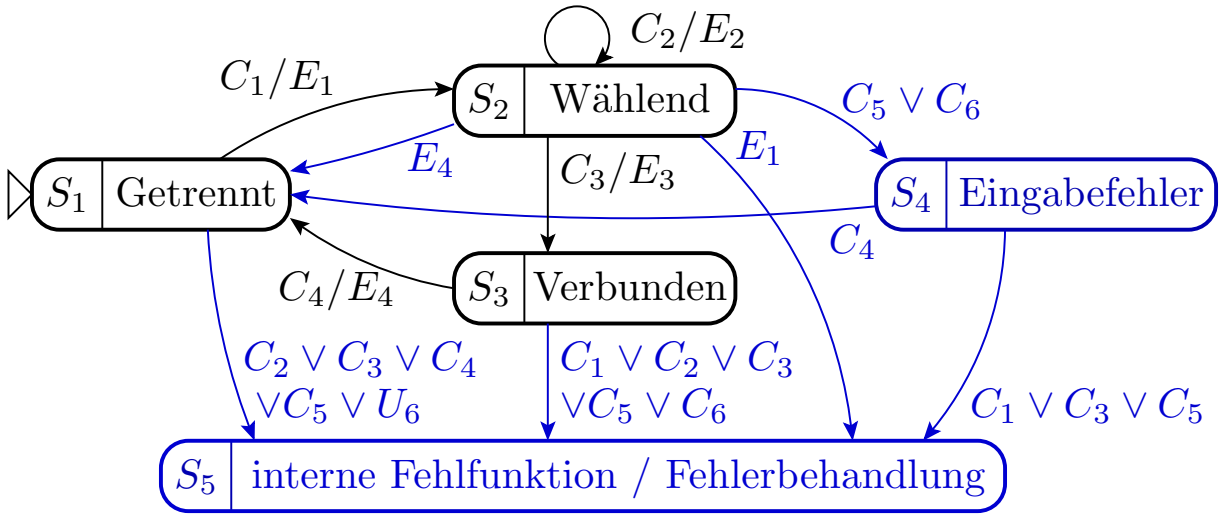
Konkrete Tests



Für alle gefundenen symbolischen Tests, z.B. $C_1, C_2, C_2, \ldots, C_2, C_3, C_4$ müssen konkrete Eingaben gefunden werden, im Beispiel:

- Abheben,
- Kontrolle Wahlbereitschaft,
- Wahl einer zulässigen Nummer,
- Kontrolle, das die Verbindung hergestellt ist,
- Verbindung trennen,
- Kontrolle, dass die Verbindung getrennt ist.

Symbolische Test, Use-Case, Operationsprofile



Ein symbolischer Test, z.B. $C_1, C_2 \{, C_2, \}, C_3, C_4$ repräsentiert i.allg. ein riesige Menge konkreter Tests, aus der zufällig ein oder mehrere Beispiele gewählt werden. Alternativ können symbolische Tests als

- bestimmte Art der Systemnutzung (Use-Case) betrachtet,
- zur Definition von Operationsprofilen (relative Nutzungsrate von Use-Cases) genutzt und
- zur Generierung von anwendungsnahen oder fehlerorientiert gewichteten Zufallstests verwendet werden.

Zusammenfassung

Testauswahl für SW

Für SW gibt es i.allg. keine fehlerfrei Sollbeschreibung, sondern ein Entwicklungsprozess in dem

- Anforderungen und Randbedingungen gesammelt,
- Lösungsideen entstehen,
- Entscheidungen getroffen werden,
- Code geschrieben, getestet und nachgebessert wird.

Die sich ansammelnden Informationen sollen auch zur Testauswahl genutzt werden. Unterschiede zur Testauswahl für DIC-Fertigungsfehler:

- Mutationen statt Modellfehler,
- Statt Soll-/Ist-Vergleich Kontrolle der Testergebnisse durch manuelle Inspektion sowie durch einprogrammierte Format- und andere Kontrollen.
- Die meisten Infos über Testaspekte erlauben nur die Ableitung symbolischer Tests.
- Auf vergessene Aspekte enthalten fehlerhafte Beschreibungen kaum Hinweise, d.h. nur zufälliger Nachweis möglich.

Gezielte oder zufällige Auswahl

Bei zufälliger Testauswahl und Beseitigung aller erkennbaren Fehler gelten auch für SW die allgemeinen Gesetze:

- unterprop. Abnahme der Fehleranzahl $\sim N^{-K}$ $(0 < K < 1)$ und
- überproportionale Abnahme der MF-Rate $\sim N^{-(K+1)}$.

Zufallstests mit mehreren unterschiedlichen Operationsprofilen versprechen höhere Fehlerüberdeckung bei gleicher Testanzahl.

Bei gezielter Testauswahl ist die erzielbare Fehlerüberdeckung begrenzt auf den Anteil der Fehler, für die sich aus der Beschreibung ähnlich nachweisbare Mutationen ableiten lassen und hängt für diesen Anteil hauptsächlich vom der Anzahl der Tests je Mutation ab.

Automatisierte Testerzeugung und -durchführung benötigen

- ausführbaren oder simulierbaren Programm-Code und
- für Sollwerte (diversitäre) Beschreibungen der Zielfunktion.

N	Anzahl der Tests.
K	Formfaktor der Verteilung der Fehlfunktionsrate $(0 < K < 1)$.

Kontrollflussbasierte Testauswahl

Die derzeit verbreitetsten Testvollständigkeitsmaße basieren auf dem Kontrollfluss:Für den Kontrollfluss:

- 100% Anweisungsüberdeckung,
- 100% Zweigüberdeckung und
- 100% Bedingungsüberdeckung.

bestimmbar durch einprogrammierte Zähler für Zweige oder erfüllte Bedingungen.

Bei zufälliger Testauswahl ist eine $c_{\mathrm{I/B/C}} \gg 1$ fache Testanzahl wie für ausreichende »Kriterienüberdeckung« erforderlich, wobei $c_{\mathrm{I/B/C}}$ vor allem davon abhängt, mit welcher Wahrscheinlichkeit Verfälschungen im Kontrollfluss beobachtbar sind, also ob z.B. Soll-Ist-Kontrolle Abarbeitungsfluss oder nur Kontrolle auf Abstürze während des Tests.

Eine interessante Alternative mit mehr Aussagewert über die tatsächliche Fehlerüberdeckung, aber auch mehr Rechenaufwand, wäre eine Fehlersimulation oder Testberechnung mit Off-By-One- und sa- Fehlern statt der einprogrammierten Kriterienzähler.

Def-Use-Ketten

Def-Use-Tupel sind Paare aufeinanderfolgender Schreib- und Lesezugriffe auf eine Variable. Nutzbar zur Erweiterung der Anweisungs-, Zweig- oder Bedingungsausführung um Beobachtungskriterien, z.B.

Anweisungsauführung mit Def-Use-Kette zu einer Kontrolle.

Weitere Anwendungsmöglichkeiten:

- statische Code-Analyse
  - »Use«« ohne »Def« Initialisierungsfehler,
  - »Defs« ohne »Use« redundanter Code.
- Fehlerlokalisierung durch Rückverfolgung verfälsche »Use« zu möglichen verfälschten »Defs«.

Äquivalenzklassen

Äquivalenzklasse sind Eingabemengen ähnlich zu verarbeitender Daten, beschreibbar durch Wenn-Dann-Beziehungen, aus denen sich für alle »Wenns« und »Danns«

- symbolische Tests,
- daraus Stichproben konkreter Tests,
- Operationsprofile für den Test oder
- Mutationsmengen aus Off-By-One- und Haftfehlern

ableiten lassen.

Eine äquivalenzklassenbasierte Wenn-Dann-Beschreibung lässt sich auch diversität zum Code

- z.B. von einem anderen Programmierer oder
- bereits in frühen Entwurfsphase als Testspezifikation

mit anderen Fehlern* und anderen Hinweisen auf mögliche Fehler* erstellen.

* insbesondere vergessene Aspekte.

CE-Modell und Automaten

Ein CE-Modell spezifiziert eine Zielfunktion durch

- Ursachen, Auslöser von Aktionen (conditions, wenn),
- Wirkung, ausgelöste Aktionen (effects, dann) und
- die logischen Beziehungen zwischen diesen.

Ein Automat beschreibt eine Zielfunktion durch

- Zustände,
- Übergangsbedingungen zwischen Zuständen (wenn) und
- den Zuständen oder Übergängen zugeordnete Aktionen (dann).

Aus den resultierenden Wenn-Dann-Beschreibungen lassen wie für »Äquivalenzklassen« symbolische Tests, konkrete Tests, Mutationsmengen und Operationsprofile ableiten.

Die Wenn-Dann-Beschreibungen können auch diversitär zum Code erstellt werden

- z.B. von einem anderen Programmierer oder
- bereits in frühen Entwurfsphase als Testspezifikation

mit anderen Fehlern und Hinweisen auf mögliche Fehler.

Rückblick »Testbare Anforderungen«

In Abschn. 6.2.3 wurden zur Beschreibung testbarer Anforderungen in UML vorgestellt

- Aktivitätsdiagramm, Sequenzdiagramm,
- Zustandsdiagramm und Protokollautomat.

Aus diesen graphischen Beschreibungen von Anforderungen lassen sich im Grunde auch Wenn-Dann-Beschreibungen für die Testauswahl ableiten, aus denen sich symbolische Test, weiter konkrete Tests, Mutationsmengen und Operationsprofile ableiten lassen.

Der Weg zu verlässlicherer Software

- Viel Testen, Zufallsauswahl, unterschiedliche Operationsprofile,
- an der Fehlerüberdeckung orientierte Testvollständigkeitsmaße,
- Nutzung diveristärer Beschreibungen aus Entwurfsprozess,
- Werzeugunterstützung, ...

Die über Jahrzehnte mit HW-Tests gesammelten Erfahrungen sind für die künftige Entwicklung im Bereich SW-Test ein guter Wegweiser.

Literatur

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