Test und Verlässlichkeit Foliensatz 7: Software: Beschreibungsmittel, Vorgehen und Testauswahl

Prof. G. Kemnitz

22. Januar 2025

Inhaltsverzeichnis

7.2 Dieser Foliensatz ...

untersucht software-spezifischen Besonderheiten

- Fehlerentstehung, Fehlervermeidung,
- Test, ...

Schwerpunkt auf drei Themen:

1. Programmiersprache: Schnittstelle zwischen dem manuellen und automatisierten Entwurf. Bestimmt erheblich Arbeitsaufwand und Fehlerentstehung bei der Codierung und die statischen Test, die am eingegebenen manuellen Entwurfsprodukt möglich sind.
2. Vorgehen: Fokus auf Techniken, die Arbeitsaufwand und Fehlerentstehung der Schritte vor der Codierung mindern und Kontroll- und Testmöglichkeiten liefern.
3. Testauswahl: Etablierte Techniken für die Auswahl dynamischer Tests und Vergleich mit dem, was sich für Hardware bewährt bzw. nicht bewährt hat.

7.3 Software

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

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

Wir wollen den Begriff SW erweitern um alle manuell und automatisch erzeugten Beschreibungen von intellektuellem Knowhow, in denen sich Fehler manifestieren, gesucht, vermieden und beseitigt werden:

- Anforderungen, Realisierungsideen,
- Absprachen, ...

Zu unterscheiden ist zwischen manuellen und automatisierten Entstehungsschritte. Manuelle SW-Entstehungsschritte:

- oft kreativ*, nicht deterministisch,
- große Fehlerentstehungsraten (Abschn. 2.3.2).

Die wirksamste Art der Fehlervermeidung ist eine möglichst weitgehende Automatisierung der Entwurfsprozesse.

* Routinearbeiten werden zunehmend automatisiert.

.

Automatisierte Entstehungsprozesse:

- Compilieren aus Quellcode incl. Syntax- und anderer statischer Tests.
- Codegenerierung z.B. Parser aus der Sprachdefinition, Oberflächen aus graphischen Eingaben, Steuer-SW aus Simulationsprogrammen, ...
- Konfigurationsgesteuertes Compilieren von Tests, ...

Rechnerunterstützung für Entwurf, Test und Fehlersuche:

- Templates, Versionsverwaltung, Refractioning, ...
- Code-Analyse, Änderungsverfolgung,
- Eliminierung von totem (ungenutztem) Code,
- Bestimmen von Codeabdeckungen, Fehlersimulation,
- ...

Tendenziell nimmt die Automatisierung immer weiter zu.

1 Programmiersprache

7.5 Programmiersprache und Verlässlichkeit

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

Die Wahl der Programmiersprache bestimmt sehr wesentlich

- den Programmieraufwand,
- mögliche und typische Fehler,
- Fehlerentstehungsrate,
- Fehlerfunktionsbehandlung, ...

Unterteilung der Fehler- und MF-Arten nach typ. Umgang damit:

ST	Ausschluss durch **s**tatische **T**ests zur Compile-Zeit.
DT	Suche mit **d**ynamischen **T**ests.
Tol	**Tol**erierung durch Check und Fehlfunktionskorrektur.
CT	Schadensvermeidung durch **C**heck + **T**erminierung.

Als Beispiel und, um Unterschiede zu erörtern, wird Rust betrachtet, eine neue Sprache, die versucht »alles richtig zu machen«.

7.6 Rust*

- Außergewöhnlich viel Fehlerausschluss durch statische Tests.
- Automatisch einprogrammierte Kontrollen in Testübersetzungen.
- Außergewöhnlich gute Unterstützung für die MF-Behandlung.
- Innovative Beschreibung und Verwaltung von Tests.

Insbesondere rutschen einige verbreitete »bösartige« Fehlerarten wie Speicherlecks, Deadlocks, ... aus den Kategorien:

- DT (Suche mit **d**ynamischen **T**ests)
- Tol (**Tol**erierung durch Check und Fehlfunktionskorrektur)
- CT (Schadensvermeidung durch Check + Terminierung)

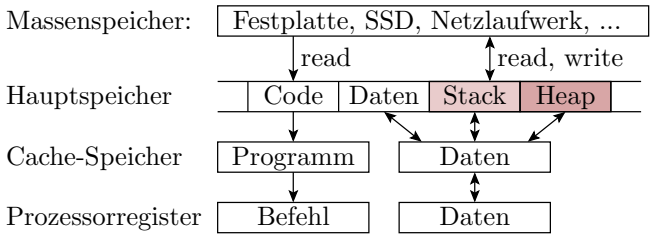
in die Kategorie »statisch nachweisbar«. Dadurch in übersetzten Programmen ausschließbar und somit unproblematisch.

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

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

1.1 Speicherlecks, ...

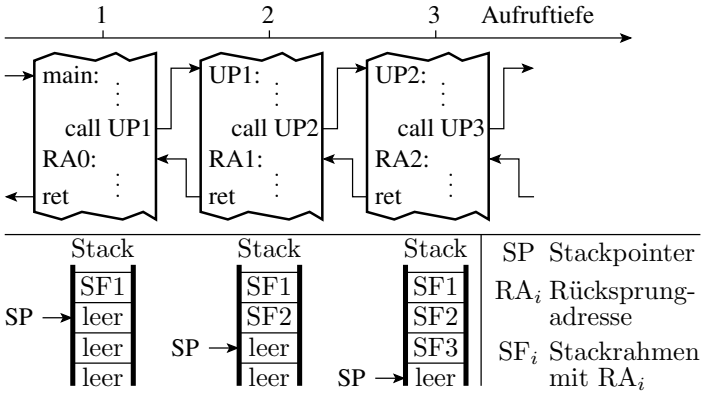
7.7 Speicherlecks, hängender Zeiger, ...



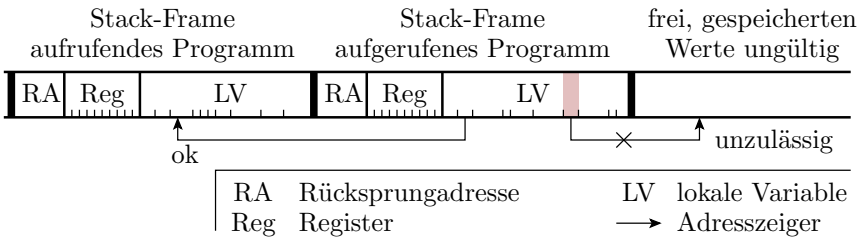
- Kompletten Daten als Dateien in Massenspeichern. Für aktive Programme werden Code-Kopien im Hauptspeicher gehalten und Platz für Daten reserviert.
- Von genutzten Code- und Datenbereichen werden Kopien im schneller zugreifbaren Caches gehalten, auf die der Prozessor zugreift.

Speicherlecks und hängende Zeiger hängen mit den Funktionsprinzipien bzw. der Nutzung von Stack und Heap zusammen.

7.8 Das Stack-Prinzip



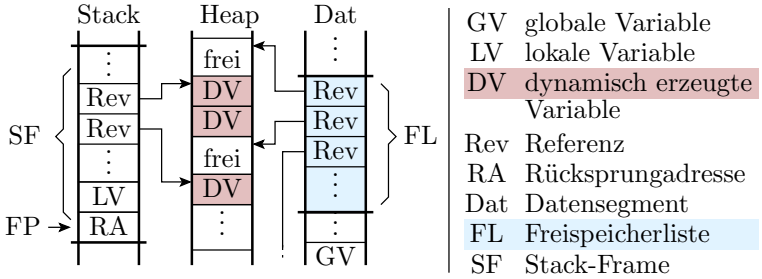
Programme sind Sammlungen von Funktionen, die aufgerufen werden und nach Beendigung zum Aufrufpunkt zurückspringen. Rückkehradresse werden auf einem Stapelspeicher (Stack) abgelegt. Das erlaubt eine große Aufruftiefe und sogar rekursive Aufrufe.



- In dem Stackrahmen kopiert das Programmm auch Werte von Registern, die das aufrufende und das aufgerufene Programm beide benötigen, und reserviert Platz für lokale Variablen.
- Lokale Variablen werden immer relativ zum Stackrahmen adressiert und haben bei jedem Aufruf andere absolute Adressen.
- Nur Variablen von Programmen auf dem Stapel belegen Speicher.
- Nach Rücksprung werden alle lokalen Variablen ungültig.

Hängende Zeiger sind ungültige Adresswerte in gültigen Variablen. Im Normalfall ist es Aufgabe des Programmierers, das zu verhindern.

7.10 Dynamische Daten auf dem Heap



- Von der SW verwalteter Datenbereich,
- Zugriff über Zeiger im Datensegment, Stack oder Heap.

Vereinfacht das Programmieren, ist aber eine böse Fehlerquelle:

- Hängende Zeiger (siehe Folie zuvor).
- Speicherleck: Auf dem Heap sich ansammelnde Speicherbereiche, auf die es im Programm keine Zeiger mehr gibt.
- Fragmentierung: Freispeicherzerstücklung in immer kleinere Teile.

Für Speicherlecks und hängende Zeiger bieten die meisten Programmiersprachen nur

- Suche mit dynamischen Tests (DT) und
- Programmabruch, wenn während der Laufzeit erkannt (CT).

Kein ausreichend großer zusammenhängender freier Speicher bei Anforderung durch Heap-Fragmentierung:

- durch sogfältigen Umgang mit Freispeicher minderbar,
- mit dynamischen Tests schwer ausschließbar,
- zur MF-Behandlung kann versucht werden, kleine freie Speicherbereiche zu größeren zusammenzufassen oder Abbruch, Heap-Neuinitialisierung und Wiederholung.

In sicherheitskritischen Anwendungen Nutzung dynamischer Speicherverwaltung zum Teil verboten.

Rust hat ein neuartiges Beschreibungskonzept, zur Vermeidung hängende Zeiger und Speicherlecks (ST).

ST	Ausschluss durch statische Tests zur Compile-Zeit.
CT	Schadensvermeidung durch Check und Terminierung.
FT	Fehlertoleranz durch Check + MF-Korrektur.
DT	Ausschluss durch dynamische Tests.

7.12 Datenobjekte in Rust

Konstanten: benutzte Zahlenwert, Zeichen, Auszählungswerte:

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

Variablen:

- global (static), feste Adresse im Code-Segment,

```
static x: u16 = 3136; // globale, unveränderbar
```

- lokal, Stack, feste Adresse im Stack-Frame der Funktion

```
let mut yyy: u32 = 0; // feste Größe, änderbar
let yyy = yyy + 1254; // Wertänderung
```

- dynamisch, Adresszuordnung zur Laufzeit auf dem Heap.

```
let v = vec![1, 2, 3];// änderbare Größe
```

Besonderheiten von Rust:

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

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

- Besitzweitergabe:

```
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
```

Ausschluss hängende Zeiger und Speicherlecks durch Löschen der Datenobjekte bei Beendigung des Besitzers.

7.14 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 weitere Referenz zulässig.
- Löschen der Objekte mit der letzten Referenz.

Ausschluss hängende Zeiger, Speicherlecks, Read-After-Write-Hazzards, ... Programmierung gewöhnungsbedürftig. Bestimmte Konstrukte (Listen, Semaphore, ...) nur »unsafe« programmierbar.

1.2 Typ- und WB-Checks

7.15 Datentypen und Operationen

Elementare Typen fester Größe (wie alle Programmierhochsprachen):

- 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 (CT*)

Standardmäßige Zuweisung als »Konstanten« erlaubt mehr ST.

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

7.16 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 Elementeauswahltyp (ST),
- unzulässige Index-Konstante (ST),
- unzulässiger berechneter Index (CT),
- unzulässige Zuweisungen (ST*, CT)...

ST Ausschluss durch statische Tests zur Compile-Zeit.

CT Schadensvermeidung durch Check und Terminierung.

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

7.17 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.

Rust bietet komplexe Beschreibungsmittel wie die Verwaltung von Listen, Zeichenkettenverarbeitung incl. Parse-Funktionen, Hash-Tabellen z.B. für Wörterbücher, ...

- Verringert Quelltextgröße und -fehleranzahl.
- Die Beschreibungsmittel sind in der Regel für ein Maximum an statischen Tests und Kontrollen zur Laufzeit konstruiert.

7.18 Aufzählungen in Rust

Aufzählung von Werten, in der jedem Wert ein Tupel von Datenobjekten 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
}
```

Option<T>, Result<T>, erlauben eine einfache Übergabe der MF-Behandlung an die aufrufende Funktion.

7.19 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:

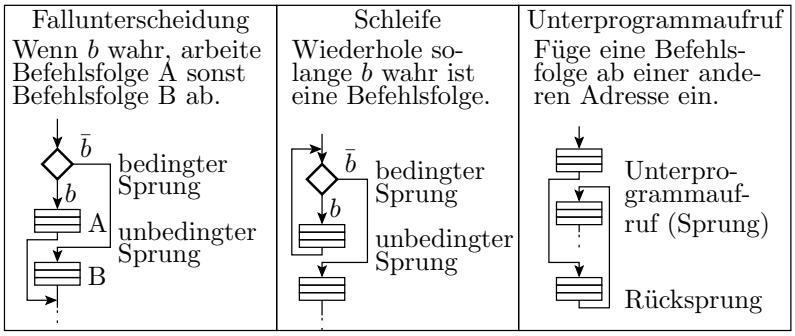
```
fn plus_one(x: Option<i32>) -> Option<i32> {
  match x {
      None => None,          // für x=None Ausdruck None und
      Some(i) => Some(i + 1),// für x=Some(i) wird i verarbeitet
  }
}
```

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

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

1.3 Kontrollfluss

7.20 Steuerung des Kontrollflusses



Nach jedem Befehl wird ein Folgebefehl abgearbeitet. Ausnahme Sprünge:

- Sprüge unbedingt, bedingt,
- Unterprogrammaufruf, Rücksprung.

Die Programmierung von Kontrollflüssen mit Sprüngen oder »goto« ist sehr fehlerträchtig und nach Möglichkeit zu vermeiden.

7.21 Fehlervermeidung in Hochsprachen

Alle Hochsprachen unterstützen »strukturierte Programmierung, d.h. ein Fokusierung auf robuste (weniger fehleranfällige) Ablaufstrukturen

- Fallunterscheidungen, Schleifen,

- Unterprogramme, ...

Robuste Konstrukte für Nebenläufigkeit

- Threads, Semaphore, ...

Rust geht hier auch über das übliche hinaus:

- stärkere Restriktionen für mehr statische Kontrollen,
- mächtige Beschreibungsmittel für häufig genutzte komplexe Abläufe, insbesondere auch für die Fehlfunktionsbehandlung.

7.22 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 };
```

Erlaubt als zusätzliche Kontrollen (ST):

- Abdeckung aller Möglichkeiten.
- Gleicher Typ aller Ergebnisausdrücke und
- Typkontrollen bei Verarbeitung und Zuweisung wie für Ausdrücke.

7.23 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}");
}
```

1.4 MF-Behandlung

7.24 MF-Behandlung

Nach jeder Kontrolle muss eine MF-Behandlung abzweigen:

- kontrollierter Abbruch
  - Retten von Daten,
  - 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.

7.25 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-Werte bewirken immer einen Panic-Abbruch:

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

7.26 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
```

7.27 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.

In Hochsprachen ohne diese Beschreibungsmittel entfällt auf eine gründliche MF-Behandlung schnell die Hälfte des gesamten manuellen Programmieraufwands.

1.5 Test

7.28 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
```

7.30 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.

Die einfachen Beschreibungsmöglichkeiten für Tests und die Automatisierung der Testdurchführung motivieren zum gründlichen Testen.

Zusammenfassung

7.31 Zusammenfassung

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

- Programmieraufwand, mögliche und wahrscheinliche Fehler,
- eingebaute Kontroll- und Fehlerbehandlung.

Rust hat gegenüber vergleichbaren Programmiersprachen

- Beschreibungsmittel für den Ausschluss hängender Zeiger und Speicherlecks,
- mehr Möglichkeiten auch für statische Tests.
- eine innovative Testunterstützung,
- Fail-Fast für den Test und Fail-Slow für den Einsatz, ...

Vorteile werden natürlich mit Nachteilen erkauft:

- Deutlich komplizierterer Übersetzungsprozess,
- Gewöhnungsbedürftige Programmierung (Lernaufwand), ...

Rust zeigt die Richtung der Weiterentwicklung der Programmiersprachen, wird aber wahrscheinlich nicht die letzte Lösung für alles sein.

2 Vorgehen

7.33 Vorgehen von der Idee bis zur Codierung

Das Vorgehen von der Idee über Festlegung der Zielfunktionen (Aufteilung in Arbeitsschritte, HW/SW-Teilung, Nachzunutzung vorhandener Bausteine, ...) bestimmt maßgeblich:

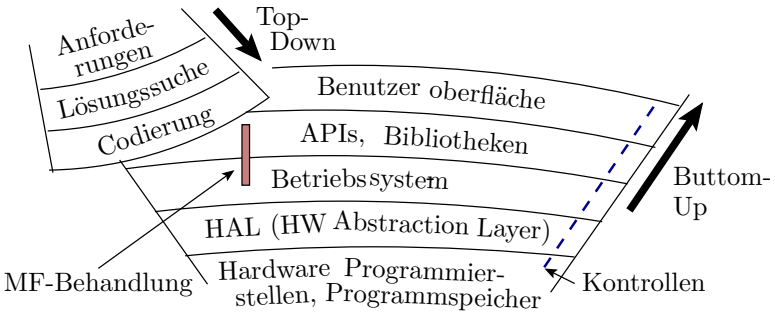
- Arbeitsaufwand, Fehlerentstehungsrate und Fehleranzahl,
- Zwischenkontrollen, ableitbare Tests für später,
- die Fähigkeit des Entstehungsprozesses zu reifen und

und hat auch Verbindungen zu anderen verlässlichkeitssichernden Maßnahmen (Fehlerkultur, prüfgerechter Entwurf, ...).

Fokus des Abschnitts auf:

- Software-Architektur: Rahmen für Aufteilung des Gesamtsystems in Teilbausteine und Gestaltung der Schnittstellen.
- Entwurfsablauf: Technologie mit definierten Stufen, Zwischenkontrollen, ... als Voraussetzung für das Lernen aus Fehlern.
- Testbare Anforderungen: Formulierung aller Entwurfsziele und -entscheidungen idealerweise so, dass sich daraus Tests ableiten.
- Programmierstil: Good Practice, Antipattern.

7.34 IT-Struktur und Entwurfsvorgehen

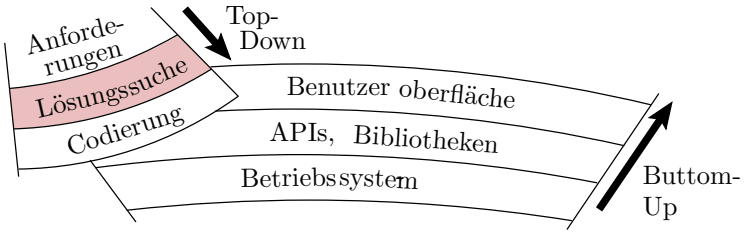


- Software legt funktionale Schichten über die Hardware,
- ist selbst in Schichten organisiert.

Der Entwurf ist ein Prozess zunehmender Detaillierung:

- Sammlung von Anforderungen, Use-Cases, Testbeispielen,
- Entscheidungen für die Lösungssuche: Auswahl HW, nachnutzbare SW, SW-Architektur, ...

7.35 Grundlegende Entscheidungen



Lösungsfindung beginnt mit grundlegenden Entscheidungen:

- vorhandene und zu beschaffende Ressouren (Personal, Technik, Knowhow, ...),
- grundlegende Architekturentscheidungen,
- Wiederverwendungsentscheidungen,
- Programmiersprache, Tools, ...
- Organisation der Arbeitsverteilung, Ablaufkontrollen,
- Aufteilung in zu entwerfende Programmodule, ...

bevor die eigentliche Entwurfsarbeit beginnt.

2.1 Software-Architektur

7.36 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, ...

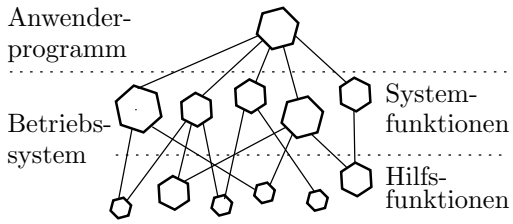
Grobeinteilung:

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

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

7.37 Prozedurensammlung

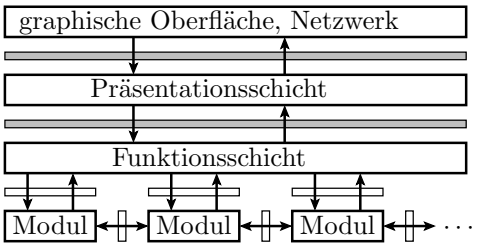
Die am wenigsten restriktive SW-Architektur ist die Aufteilung der Gesamtfunktion in eine Sammlung von Prozeduren (Funktionen):



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.

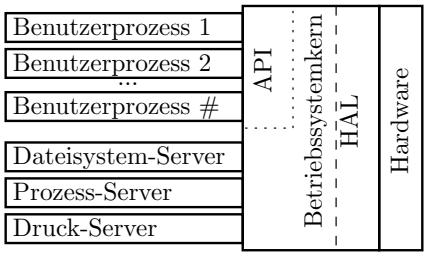
7.38 Schichtenmodell



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 wohl definierte Schnittstellen nutzen.
- Ein Kommunikationsprogramm kann nicht direkt auf den COM-Port zugreifen. Die Applikation stellt eine Anfrage an das Betriebssystem, ob COM-Port verfügbar, ...
- Vereinfachte Austauschbarkeit, Nachnutzbarkeit, ...

7.39 Client/Server-Modell



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

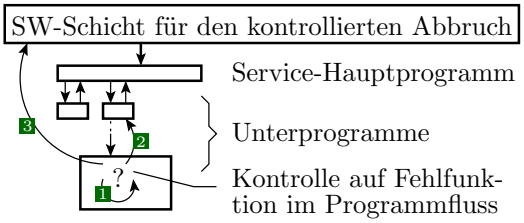
Beispiel Client/Server-Betriebssystem: Datei-, Prozess- und IO-Verwaltungen als Server. Kleiner Betriebssystemkern (Mikrokernel) für die Kommunikation zwischen den Servern, Clients und HAL.

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

API Programmiersschnittstelle.

HAL Hardwareabstraktionsschicht.

7.40 Fehlfunktionsbehandlung

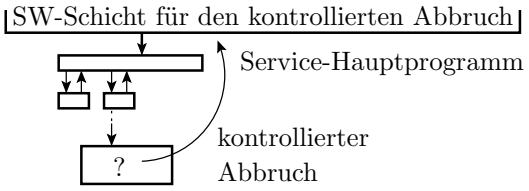


Kontrollen im laufenden Betrieb erfolgen an vielen Stellen im Kontrollfluss. Mögliche Orte der Fehlfunktionsbehandlung:

1. Im Kontrollfluss innerhalb der Prozedur, z.B. Ersatz nicht plausibler Werte durch Standardwerte, und Fortsetzung.

2. In einer aufrufenden Prozedur. Erfordert Rücksprung mit Fehlercode gegebenenfalls über mehrere Aufrufebenen, vergl. Rust-Datentyp Result<T, E> (Folie 7.26), .
3. Zentral in einer Schicht unterhalb des Service-Hauptprogramms. Kontrollierter Abbruch, in Rust mit dem Makro panic (Folie 7.25).

7.41 Kontrollierter Abbruch



Fehlfunktionen durch Fehler oder Störungen der Hardware verlangen in der Regel kontrollierter Abbruch mit einer Reihe von Aktionen:

- Fehlermeldung, Herstellung eines gefährdungsfreien Zustands,
- Sicherung schwer wieder zu beschaffender Daten,
- Sichern von Daten für eine eventuelle spätere Fehlerlokalisierung (Variablen, Aufrufstack, ...),
- Freigabe reservierter Ressourcen (Speicher, IO-Geräte, Dateien).
- ...

Das verlangt eine SW-Schicht unterhalb der Applikationen mit Fehlfunktionsbehandlung mit Zugriff auf alle Applikationsressourcen.

7.42 Test- und Überwachungsschnittstellen

Schichten sind auch Schnittstellen für:

- Überwachung: Trace-Aufzeichung, Stream-Monitor, ...
- 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')
```

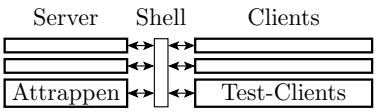
- Windows Low-Level Benutzerinteraktion:

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

- In IDEs (integrierten Entwicklungsumgebungen) und andern großen Sytemen sind einzelne Tool oft alternativ zur graphischen Eingabe über Konsolenbefehle steuerbar.

Schichtenstruktur vergrößert und verlangsamt Programme, aber vereinfacht Test, MF-Behandlung, Nachnutzung und Wartung.

7.43 Client-Server-Systeme



Clients und Server kommunizieren über Nachrichten. Nachrichten:

- implementationsunabhängig.
- in der Regel gut kontrollierbaren Formatmerkmale.
- gut geeignet für die Beschreibung Use-Cases und Tests.
- gut tracebar für die Fehlersuche.

Weitere Vorteile der Strukturierung in Clients und Server:

- Verteilbar auf unterschiedliche und räumlich entfernte Hardware,
- vermindertes Risiko von Common-Cause-Problemen (gemeinsamer Ausfall, Absturz, ...).
- Test-Clients für den Server-Test und Server-Attrappen, Client-Test,
- ...

7.44 Suche einer geeigneten Software-Architektur

Die grundsätzliche Entscheidung

- Prozedurensammlung, Schichtenmodell oder
- Client-Server-Struktur

hängt ab von der Systemgröße, den Anforderungen an die MF-Behandlung, die Testbarkeit, Änderbarkeit, ... sowie nachnutzbarer und nachzunutzender Bausteine ab. Danach sind noch viele Details zu präzisieren:

- wichtige Kernfunktionen, die Algorithmen dafür,
- Datenhaltung, Netzwerkunterstützung,
- Details der MF-Behandlung, z.B. Fehlerdatenübermittlung an den Hersteller (Reifeprozesse), ...

bis hin zu allg. Regeln für z.B. für die Vergabe von Bezeichnern und die Kommentare im Quellcode.

Je komplexer die Systeme, desto mehr Erfahrung und Aufwand erfordert die Suche einer geeigneten Software-Architektur.

2.2 Entwurfsablauf

7.45 Entwurfsablauf

Vorgehensmodell: Vereinheitlichung des Vorgehens, damit

- ähnlich oder vergleichbare Abläufe oft wiederholt werden, um
- dabei aus erkannten Fehlern zu lernen (Abschn. 2.3.4).

Good Practice (bewährte Techniken):

- Stufenmodell mit Zwischenkontrollen,

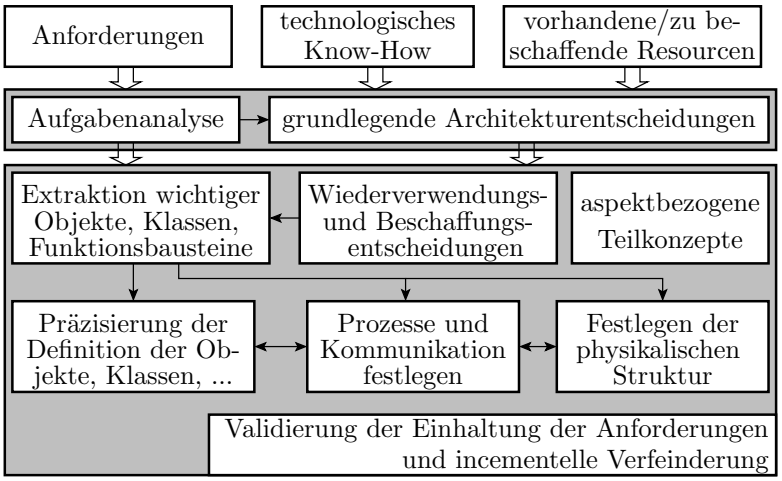
- Begrenzung/Minderung der Rückgriffhäufigkeit, ...

Stellschrauben:

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

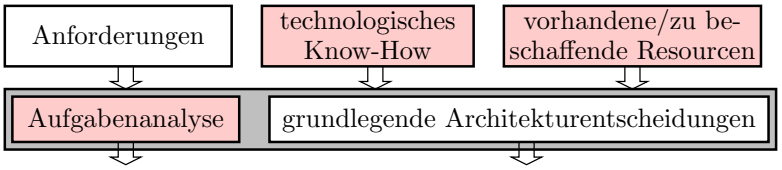
Wichtiges technologisches Knowhow eines SW-Unternehmens.

7.46 Beispielaufteilung in Entwurfsschritte



Ein realer Entwurf lässt sich nicht eine lineare Schrittfolge pressen.

7.47 Aufgabe und Ressourcen analysieren



Aufgabenanalyse:

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

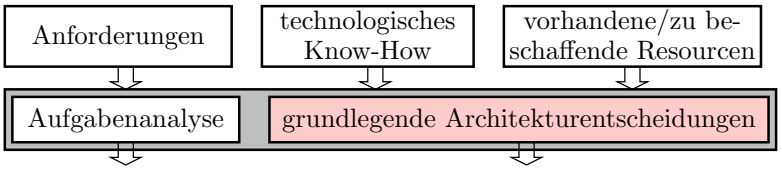
Technologisches Know-How:

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

vorhanden/zu beschaffen:

- Rechner, Software, Personal, ...

7.48 Grundsätzliche Architekturentscheidungen

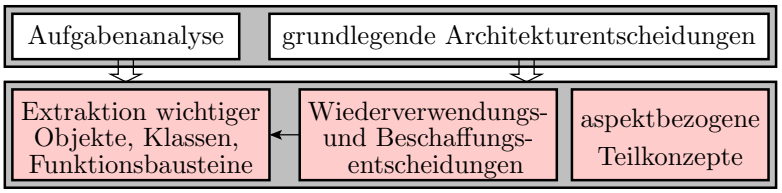


Nach den Analysen folgen grundlegende Entscheidungen:

- 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.

7.49 Detaillierung der Entscheidungen



Wiederverwendung und Beschaffung:

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

Aspektbezogene Teilkonzepte, über die zu Beginn zu entscheiden ist

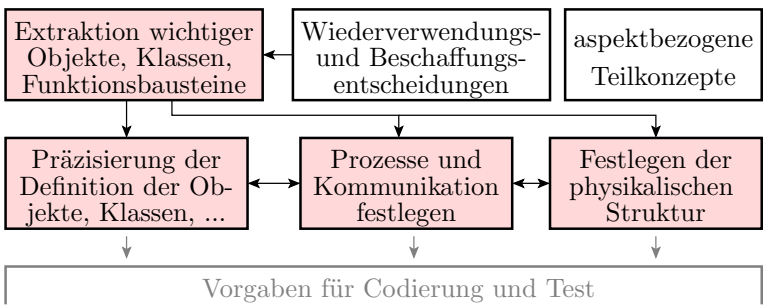
- Datenhaltung, Benutzerschnittstellen,
- Fehlerbehandlung, Fehlertoleranz, Sicherheit, ...

implizieren Vorgaben die bei der Extrahierung

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

zu berücksichtigen sind.

7.50 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
- Testvorgaben für die Module einzelnen und ihr Zusammenwirken.

7.51 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.

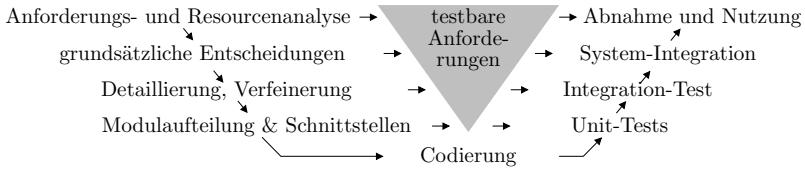
[Bec03] K. Beck. Test-driven Development - By Example. Boston: Addison-Wesley Professional, 2003. isbn: 978-0-321-14653-3.

[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.

2.3 Testbare Anforderungen

7.53 Testbare Anforderungen



Beschreibung der Zielfunktionen für

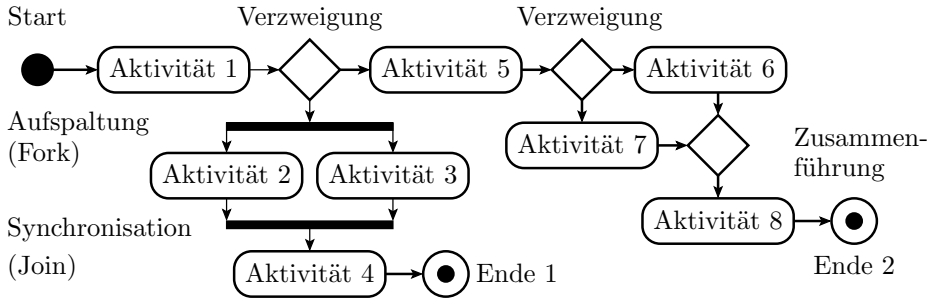
- des Gesamtsystem und
- der einzelnen Module, Schichten, Server, Clients, ...

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

Modellierungssprache UML. Beschreibungsmittel:

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

7.54 Aktivitätsdiagramm

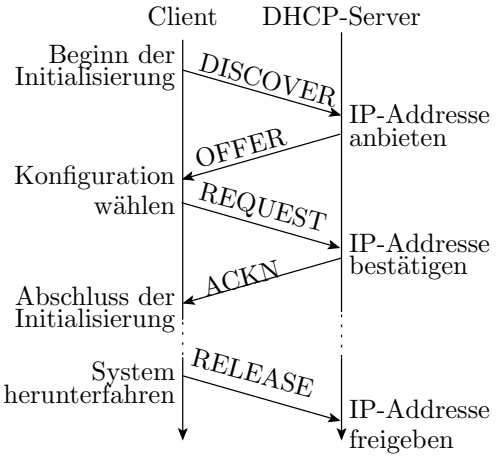


Aktivitätsdiagramme beschreiben Ablaufmöglichkeiten aus Aktivitäten (Schritten), Transaktion, Verzweigung, Synchronisation, Signale senden und empfangen. Aus dem Beispiel ableitbare Testfälle*:

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

* Symbolische Tests, für die später konkrete Tests zu suchen sind.

7.55 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

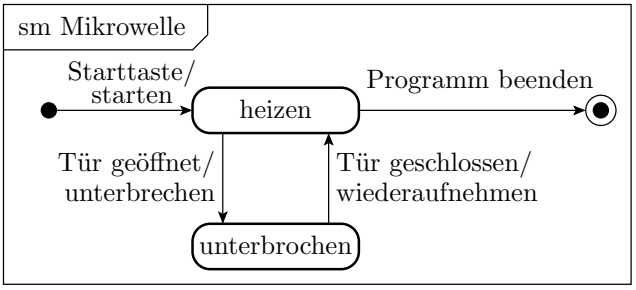
Sequenzdiagramme sind Interaktionsdiagramme und zeigen den zeitlichen Ablauf einer Reihe von Nachrichten (Methodenaufrufen) zwischen Objekten, Threads, Rechnern, ... in einer zeitlich begrenzten Situation. Dabei kann auch das Erzeugen und Entfernen von Objekten enthalten sein.

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 (Abschn. 7.3.2 *Def-Use-Ketten*).

* Symbolische Tests, für die später konkrete Tests zu suchen sind.

7.56 Zustandsdiagramm



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

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

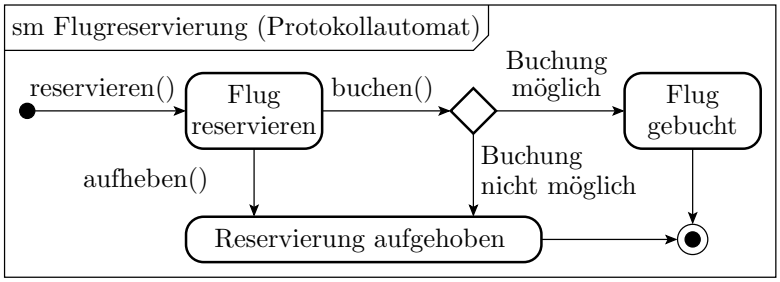
Ableitbare Tests*:

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

(Abschn. 7.3.5 *Automaten*).

* Symbolische Tests, für die später konkrete Tests zu suchen sind.

7.57 Protokollautomat

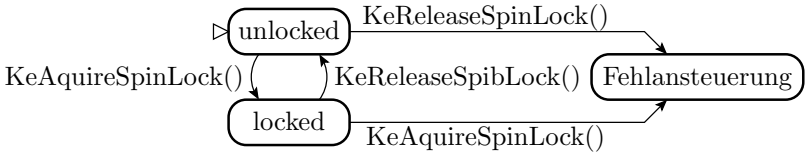


Beschreibung zulässiger Aktionsreihenfolgen. Mögliche Aktionen im Beispiel sind die Methodenaufrufe »reservieren()«, »aufheben()« und »buchen()«. Aus dem Protokollautomat im Beispiel geht hervor, dass ein Flug nur nach erfolgreicher Reservierung gebucht und dass ein einmal gebuchter Flug nicht gestrichen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

7.58 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.

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

7.59 Treiberbeispiel

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.

Mit dieser Sorte statischer Tests wurden 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();
}
```

2.4 Programmierstil

- 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.
- Die Beachtung einiger Regeln »of Good Practice« helfen, die Fehlerentstehungsraten bei der Codierung gering zu halten.

7.61 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 Hilfsfunktion, 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.
- ...

»7.62 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. ...

7.63 MISRA-Standard

Die Programmiersprache C erlaubt sehr maschienennahe Programmierung und darüber schnelle und effiziente Programme. Preis »böse« Fehlermöglichkeiten, z.B. bei Nutzung Goto, Pointern, Heap, ...

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 zusammenfassen.
- 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, ...

7.64 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 die gleichwertige Funktion mit Textlängenbegrenzung auf die Puffergröße

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

  n – Puffergröße.

- ...

Zusammenfassung

7.65 Zusammenfassung

Der SW-Entwurf ist ein Prozess zunehmender Detailierung:

- Zusammstellen und Konkretisierung der Anforderungen,
- grundsätzliche Entscheidungen: HW/SW-Teilung, Nachnutzung, SW-Architektur, ...
- Modulaufteilung und Schnittstellendefinitionen,
- Codierung und Test.

Das Vorgehen und die grundsätzlichen Entscheidungen dabei bestimmen maßgeblich den Arbeitsaufwand, die zu erwartende Anzahl und Art der entstehenden Fehler, ...

Behandelte Teilaspekte:

- Die grundlegenden Software-Architekturen,
- ein Vorgehensbeispiel für den Entwurf bis vor die Codierung,
- UML-Beschreibungen für testbare Anforderungen und
- einige Tips für die Codierung.

7.66 Software-Architektur

Eine gute Architektur ist die Basis für gut test-, überwach- und wartbare, flexible und verständliche Systeme. Betrachtet wurden:

- Prozedurensammlung: Am wenigsten restriktive SW-Architektur.
- Schichtenmodell: Beschränkung der nutzbaren Prozeduren auf die der Schicht darunter. Übersichtlicher, besser änderbar.
- Client/Server-Modell: Server sind autonom arbeitende Dienstprogramme, Clients sind Applikationen, die Dienste der Server nutzen.

Fehlerfunktionsbehandlung benötigt mindestens eine Schicht für den kontrollierten Abbruch. Darüberliegende Schichten bietet bessere Schnittstellen für Test und Überwachung bis hin zu Script-Sprachen für die Programmierung von Tests und die Erzeugung von Trace-Ausgaben.

Strukturierung in in Clients und Server erlaubt Dezentralisierung, Minderung von Common-Cause-Problemen, bietet gut zu handhabende Überwachungs- und Testschnittstellen, ...

Je komplexer die Systeme, desto mehr »Struktur« empfehlenswert.

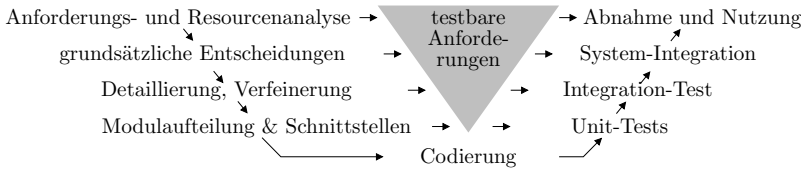
7.67 Testbare Anforderungen

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:

- Aktivitätsdiagramm: Beschreibt Ablaufmöglichkeiten als Graph aus Aktivitäten, Transaktionen (Übergängen), Verzweigungen und Synchronisation.
- Sequenzdiagramm: Zeitlicher Ablauf von Nachrichten zwischen Objekten, Treads, Rechnern.
- Zustandsdiagramm: Verhaltensbeschreibung durch Zustände für Aktivitäten und Kanten mit Übergangsbedingungen.
- Protokollautomat: Automat zur Beschreibung zulässiger Aktionsreihenfolgen, auch geeignet für Generierung statischer Tests und die Programmierung von Überwachungsfunktionen.

3 Testauswahl

7.68 Testauswahl für Software



Fehler entstehen in allen Entstehungsphasen z.B. durch

- Fehlinterpretation von Anforderungen, HW-Funktion, ...
- logische Denkfehler, Versehen, ...

Das V-Modell symbolisiert, dass in jeder Entwurfsphase Testanforderungen abzuleiten und nach der Codierung in umgekehrter Reihenfolge abzuarbeiten sind.

Testanforderungen aus frühen Entwurfsphasen sind symbolische Tests, für erst in späteren Entwurfsphasen ausführbare Tests entwurfen oder erzeugt werden.

7.69 Symbolische Tests

Symbolische Tests beschreiben, was wie gründlich zu testen ist:

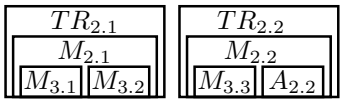
- Funktionen, Ausnahmebehandlungen, Standarkonformität,
- typische erwartete Fehler und Fehlverhalten,
- ...

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

Die automatische Erzeugung von ausführbaren aus symbolischen Tests verlangt vom Rechner verarbeitbare Beschreibungen:

- Ein- und Ausgabegabeformat (Abschn. 1.2.2),
- Testziele,
- Operationsprofil (Abschn. 3.2.4),
- testbare Anforderungen (Abschn. 7.2.3),
- Kontrollkriterien,
- ...

7.70 Typische Reihenfolge der Testauswahl



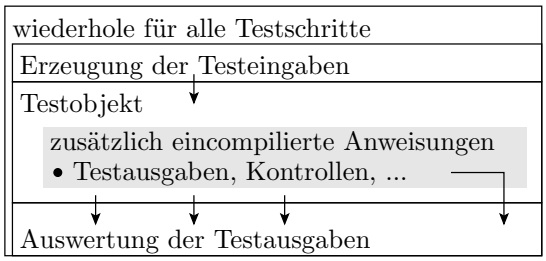
A_i Attrappe für Modul M_i
TR_i Testrahmen für Modul M_i

Nach (oder diversitär zur) Implementierung der zu testenden Einheiten (Testrahmen mit Modul(en) und Dummies):

1. Wenige typische Testeingaben, manuelle Kontrolle. Beseitigung erkannter Abweichungen vom Sollverhalten,
2. Komplettierung der Tests um automatische Kontrollen.
3. Geziele Ergänzung von Tests für Grenzfälle und unzulässige Eingaben zur Kontrolle der MF-Behandlung.
4. Erweiterung um eine größere Anzahl zufälliger Tests.
5. Ausführbare Tests zu den symbolischen Tests aus früher Entwurfsphasen.

Die manuelle Programmierung der Tests ist deutlich aufwändiger als die Implementierung der Funktion.

7.71 Instrumentierung



Ergänzen von Computerprogrammen (Quellcode, Binärcode, Zwischencode) um speziellen Code:

- Testausgaben, Kontrollen,
- Zähler für Anweisungs-, Zweig- und Bedingungsabdeckung, ...

jeweils angepasst an die damit zu lösenden Aufgaben:

- Untersuchung des Programms,
- zusätzliche Kontrollen während des Tests,
- Fehlersuche, ...

7.72 Fussing

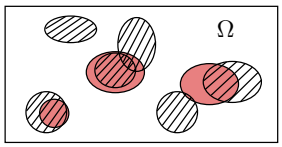
Fuzzing: Automatisierter Test mit einer großen Menge formatierter zufälliger Eingaben: Files, Tastatureingaben, Nachrichten, ...:

- Dump Fuzzer: Eingabeauswahl ohne Rücksicht auf Eingabemodell. Überwiegend ungültige nicht verarbeitbare Eingaben.
- Smart Fuzzer: Bevorzugung zulässiger und geringfügig falscher Eingaben über ein Eingabemodell [PB1630], grammatikbasiert [GKL??] oder protokollbasiert [BCF06].
- White-Box-Fuzzer: Smart Fuzzer mit einer gezielen Suche von Eingabebedingungen, um bestimmte Codebereiche zu erreichen.

- Grey-Box-Fuzzer: Nutzung von Instrumentierung statt statischer Code-Analyse zur Suche der Eingabebedingungen zur Befriedigung von Testkriterien.

[GKL??] Patrice Godefroid; Adam Kiezun; Michael Y. Levin. "Grammar-based Whitebox Fuzzing"(PDF). Microsoft Research. Literatur TV/pldi-kiezun.pdf.
[PB16] Van-Thuan Pham; Marcel Böhme; Abhik Roychoudhury. Model-based whitebox fuzzing for program binaries. ASE'16.
[BCF06] Greg Banks; Marco Cova; Viktoria Felmetsger; Kevin Almeroth; Richard Kemmerer; Giovanni Vigna. SNOOZE: Toward a Stateful NetwOrk prOtocol fuzZEr. ISC'06.

7.73 Fehlerabdeckung von Fuss-Tests



Ω Menge der Eingabewerte / Teilfolgen die einen Fehler nachweisen können
Erfüllungsmenge Überdeckungskrit.
Nachweismenge eines Fehlers

Bei einer zufällige Auswahl formatierter Eingaben

- für dasselbe Operationsprofil und
- gleicher Formfaktor der Dichte der MF-Raten

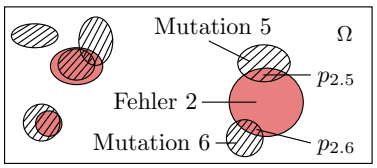
ist die zu erwartende Kriterienabdeckung etwa die der c_{MF}–fachen Testanzahl:

(2.37) $$N = c \cdot N_{\mathrm{MF}} \quad \text{für } \mu_{\mathrm{FC}}(N) = \mu_{\mathrm{FCM}}(N_{\mathrm{MF}})$$

Der Skalierungsfaktor c_{MF} hängt von den Kriterien ab, deren Abdeckung »gemessen« wird.

$\mu_{\mathrm{FCM}}(N)$ Zu erwartende Kriterienabdeckung in Abhängigkeit von der Testanzahl.
c_{MF} Kriterienspezifische Skalierung der effektiven Testanzahl.
$\mu_{\mathrm{FC}}(N)$ Zu erwartende Fehlerabdeckung in Abhängigkeit von der Testanzahl.

7.74 Das Problem mit der gezielten Testauswahl



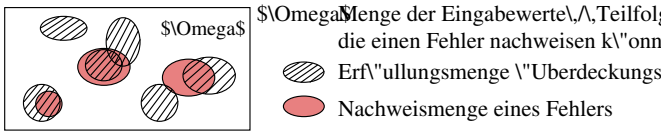
Nachweismenge Fehler i
Nachweismenge Kriterium j
$p_{i.j}$ Wahrscheinlichkeit, dass ein Test für Kriterium j Fehler i nachweist

Gezielte Testauswahl sucht Eingaben, die Kriterien einer Kriterienmenge, z.B. die Ausführung alle Anweisungen, erfüllen.

Nachweiswahrscheinlichkeit für Fehler i mit $v_i \geq 1$ ähnlich nachweisbaren Kriterien, für die je w Tests gesucht und gefunden werden:

(3.18) $$p_{\mathrm{FD}.i} = \mathbb{P}(D_i) = 1 - \prod_{j=1}^{v_i} (1 - (FC_{\mathrm{M}} \cdot (1 - (1 - p_{ij})^w)))$$

$p_{\mathrm{FD}.i}$ Nachweiswahrscheinlichkeit Fehler i.
v_i Anzahl der ähnlich nachweisbaren Krierien für Fehler i.
FC_{M} Kriterienabdeckung, Anteil der nachweisbaren Abdeckungskriterien.
w Anzahl der Tests, die für jedes Abdeckungskriterium gesucht werden.
p_{ij} Wahrscheinlichkeit, dass ein Test, der Kriterium j nachweist, auch Fehler i findet.



Für die meisten vergessenen Aspekt (Anweisungen, Fallunterscheidungen, ...) lassen sich aus der fehlerfreien Beschreibung keine ähnlich nachweisbaren Kriterien ableiten. Nachweiswahrscheinlichkeit $p_{\mathrm{FD}.i}$ ohne ähnlich nachweisbare Kriterien ($v_i = 0$) sehr klein. Zu erwartende Fehlerabdeckung kaum größer als der Anteil der Fehler mit ähnlich nachweisbaren Kriterien:

$$\mu_{\mathrm{FC}} \lesssim \mu_{\mathrm{FSDC}} \tag{7.1}$$

Gezielte Suche ohne Fusser für angestrebte $FC \gg 50\%$ ungeeignet.

---

v_i Anzahl der ähnlich nachweisbaren Krierien für Fehler i.
μ_{FC} Zu erwartende Fehlerabdeckung.
μ_{FSDC} Zu erwartende Anzahl der Fehler mit ähnlich nachweisbaren Abdeckungskriterien.

3.1 Kontrollflussabdeckung

7.76 Kontrollflussbasierte Testauswahl

Nach Standard DO-178 B genügen* für SW als Testabdeckung

- 100% Anweisungsabdeckung für nicht sicherheitskritische SW,
- 100% Zweigabdeckung für SW, die bedeutende Ausfälle verursachen kann und
- 100% Bedingungsabdeckung für flugkritische SW.

Alle drei Abdeckungsmaße

- 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 (Abschn. 6.1.2 *Praxistaugliche Fehlermodelle*).

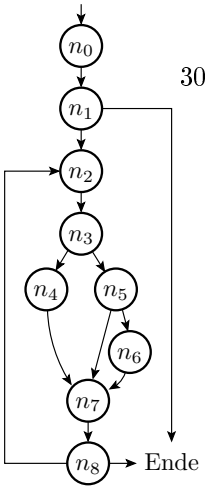
Beobachtungsbedingungen
RefSubsecCauseEffectL bleiben unberücksichtigt.

C_{I} Anweisungsabdeckung.
C_{B} Zweigabdeckung.
C_{C} Bedingungsabdeckung.
* Zur Abwendung von Produkthaftung für Schäden durch durch nicht erkannte Fehler.

7.77 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
}
```



Der Kontrollflussgraph lässt sich automatisch aus dem Maschinenprogramm berechnen.

7.78 Kontrollflussbasierte Testvollständigkeitsmaße

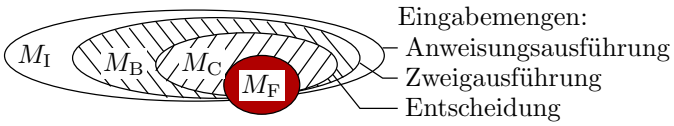
1. Anweisungsabdeckung: Jede Anweisung muss ausgeführt werden. Beispiel eines ausreichenden Tests: Start, n_0, n_1, n_2, n_3, n_4, n_7, n_8, n_2, n_3, n_5, n_6, n_7, n_8, Ende

2. Kantenabdeckung: 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. Entscheidungsabdeckung: Jede Entscheidung muss von jeder Bedingung abhängen.

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

7.79 Vergleich der Testvollständigkeitsmaße



$$M_C \subseteq M_B \subseteq M_I$$

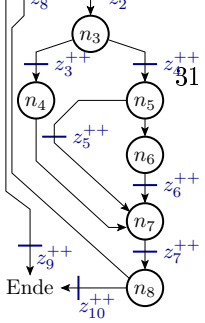
Anweisungen könnne über unterschiedliche Zweige und Zweige unter unterschiedlichen Bedingungen abgearbeitet werden. Tendentiell gilt bei gezielter Testsuche, wenn für alle Abdeckungskriterien dieselbe Anzahl w von Tests gesucht werden (vergl. Gl. 7.1):

$$\mu_{\text{FC.Instr}} < \mu_{\text{FC.Branch}} < \mu_{\text{FC.Cond}} \lesssim \mu_{\text{FSDC}}$$

M_F	Nachweismenge eines ähnlich nachweisbaren Fehlers.
$M_{I/B/C}$	Eingabemenge Kontrolle Anweisungs- / Zweig- / Bedingungsausführung.
$\mu_{FC...}$	Zu erwartende Fehlerabd. für 100% Anweisungs-, Zweig- bzw. Bedingungsabdd..
w	Anzahl der Tests, die für jedes Abdeckungskriterium gesucht werden.
μ_{FSDC}	Zu erwartende Anzahl der Fehler mit ähnlich nachweisbaren Abdeckungskriterien.

7.80 Instrumentierung von Kantenzählern

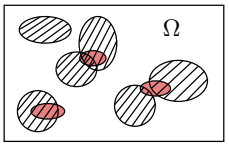
```
int z[11]={0,0,0,0, ...};
...
int ZZ(int Ct_max){char c;
n0: Ct_A=0;Ct_B=0;Ct_N=0;z(0)++;
n1: while (Ct_N<Ct_max){ z(1)++;
n2:  c=getchar(); z(2)++;
n3:  if (is_TypA(c)){
n4:   z(3)++; Ct_A++;}
```




```
        else {z(4)++;
n5:      if (is_TypB(c)){
n6:        Ct_B++; z(5)++;}
        else z(6)++;
        }
n7:     Ct_N++; z(7)++;
n8:     ...
}
```

Für einen aus dem Maschinenprogramm berechneten Kontrollflussgraph werden die Kantenzähler in das Maschienenprogramm eingefügt.

7.81 Fuzzer



Ω Menge aller formatierten Eingaben incl. fehlerhafter für den Test der Eingabekontrolle

Nachweismenge eines ähnlich nachweisbaren Überdeckungskriteriums

Nachweismenge eines Fehlers

Anweisungs-, Zweig- und Bedingungsausführung sind notwendige, aber nicht hinreichende Anregungungsbedingungen. Bedingte Wahrscheinlichkeit, dass ein Fehler i bei Erfüllung von Abdeckungskriterium j nachgewiesen wird, gering. Für Smart Fuzzer (Auswahl mit Eingabemodell, aber unabhängig von der Struktur) gilt abschätzungsweise

$$N = c \cdot N_{\mathrm{MF}} \quad \text{für } \mu_{\mathrm{FC}}(N) = \mu_{\mathrm{FCM}}(N_{\mathrm{MF}}) \tag{2.37}$$

mit $c_{\mathrm{MF}} \ll 1$. Grey-Box-Fuzzer versprechen nur für $FC \lesssim 50\%$ höhere tatsächliche Fehlerabdeckungen als Smart-Fuzzer bei gleichem N.

$\mu_{\mathrm{FCM}}(N)$ Zu erwartende Kriterienabdeckung in Abhängigkeit von der Testanzahl.

c_{MF} Kriterienspezifische Skalierung der effektiven Testanzahl.

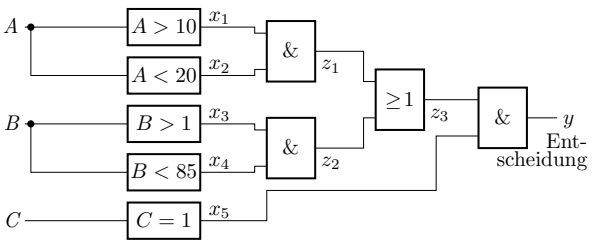
$\mu_{\mathrm{FC}}(N)$ Zu erwartende Fehlerabdeckung in Abhängigkeit von der Testanzahl.

7.82 Modellfehler statt Abdeckungsbedingungen

Ein logischer Ausdruck, z.B.

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

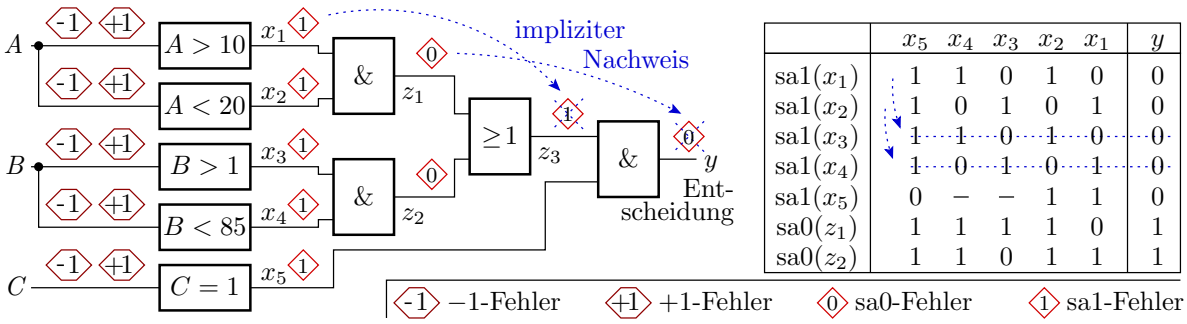
enthält Bedingungen für A, B und C und logische Verknüpfungen, auch nachbildbar durch eine Schaltung aus Gattern und Vergleichern:



Statt Bedingungsabdeckung Off-By-One- und sa- Fehler.

7.83 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

Instrumentierung Fehlernachweiskontrolle komplizierter als Zähler für Bedingungsausführung, z.B. Thread je angeregter Fehler bis Nachweis. Vorteile: man könnte Begriffe, Algorithmen und Erfahrungen von der Fehlersimulation für DIC übernehmen.

Ideen, die es wert sind, weiterverfolgt zu werden.

3.2 Def-Use-Ketten

7.84 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[1] «:

```
    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
...	...	...

7.85 Mengen von Def-Use-Tupeln und -Ketten

Alle Def-Use-Tupel:

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

Eine Beobachtungsmöglichkeit je »Def«:

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

Alle Beobachtungsmöglichkeit je »Def«:

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

Zunahme der Anzahl der Möglichkeiten problematisch (Folie 6.37
Pfadverzögerungsfehler).
```

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

7.86 Abdeckungskriterien und Fehlerabdeckung

Mögliche Mengen von Abdeckungskriterien für die *FC*-Schätzung:

1. Für alle »Defs« mindestens ein »Use«. Einfach bestimmbar, aussgekräftiger für μ_{FC} als Codeabdeckungskriterien ohne Berücksichtigung der Beobachtbarkeit.
2. Kombinationen aller »Defs« mit allen »Use«-Möglichkeiten. Stark überproportionale Zunahme Kriterienmenge mit Systemgröße, damit verbunden Zunahme Nachweisabhängigkeiten.
3. Für alle »Defs« mindestens eine Def-Use-Kette. Aussgekräftiger für μ_{FC} als (1), weil wirklichkeitsnäheres Beobachtbarkeitsmodell.
4. ...

Es wäre zu untersuchen, ob der Ansatz wirklich Vorteile gegenüber der Simulation einer Stichprobe von Off-By-One- und Stuck-At-Fehlern hat*.

* Off-By-One für arithmetische, Stuck-At für logische Operanden und Ergebnisse.

7.87 Def-Use-Tupel für Code-Analyse

Suche nicht initialisierte Variablen:

```
Wiederhole für alle »Use«
    Streichen, wenn garantiert ein »Def«
```

Suche redundanter Berechnungen:

```
Wiederhole für alle »Defs«
    Streichen, wenn mindestes ein »Use«.
```

7.88 Def-Use-Tupel zur Fehlerlokalisierung

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

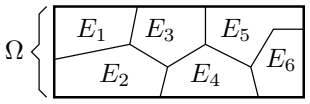
```
int ggt(int a, int b){
n0:   int c = a;
n1:   int d = b;
n2:   if(c == 0)
n3:     return d;
n4:   while(d != 0){
n5:     if(c > d)
n6:       c = c - d;
n7:     else
n8:       d = d - c;
      }
n9:   return c;
    }
```

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

Basis für ein interaktive Unterstützungs-Tools zur Fehlerlokalisierung durch Pfadrückverfolgung

3.3 Äquivalenzklassen

7.89 Äquivalenzklassen



Ω Eingaberaum
E_i Äquivalenzklasse

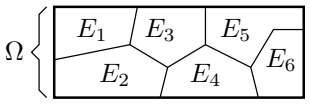
Äquivalenzklassen sind Eingabemengen ähnlich zu verarbeitender Daten, auch beschreibbar durch Fallunterscheidungen:

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

Eine solche 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");
}
```

7.90 Operationsprofil und Fehlermodellierung



Ω Eingaberaum
E_i Äquivalenzklasse

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

Eine Wenn-Dann-Beschreibung ist nutzbar für

- die Definition des Operationsprofils für den Fuzzer und die
- Definition von zählbarer Testvollständigkeitskriterien.

Als Testvollständigkeitskriterien eignen sich wie für fertige Programme

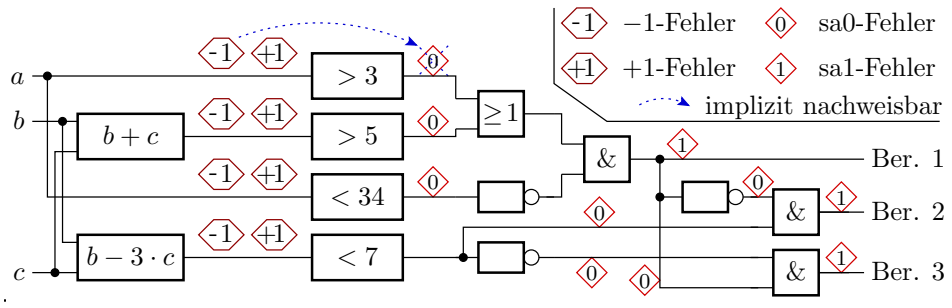
- Ausführung aller Zweige (hier Fälle) und Bedingungen,
- alternativ Mengen aus Off-By-One- und Haftfehlern.

Die Auswahl auf Basis einer (diversitären) Testspezifikationen aus einer frühen Entwurfsphasen hat den Vorteil, später vergessene Aspekte zu berücksichtigen, auf die das zu testende Programm keine Hinweise mehr enthält.

7.91 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.

3.4 CE-Analyse

7.92 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.

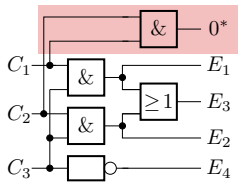
Ursache-Wirkungs-Graphen sind eine zur Beschreibung durch Äquvalenzklassen alternative Form einer Wenn-Dann-Beschreibung für die Spezifikation von Tests mit denselben Vorteilen:

- Sammeln von Testanforderungen vor oder parallel zur Codierung,
- Diversität zum zu testenden Programm,
- automatisierbar und
- Knowhow vom Test digitale Schaltungen nachnutzbar.

7.93 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.

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	

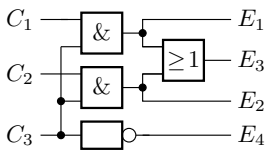
Eine Ursache-Wirkungs-Analyse deckt auch Mehrdeutigkeiten und Widersprüche auf.

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

7.94 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	

Erkennbare Ungereimtheiten:

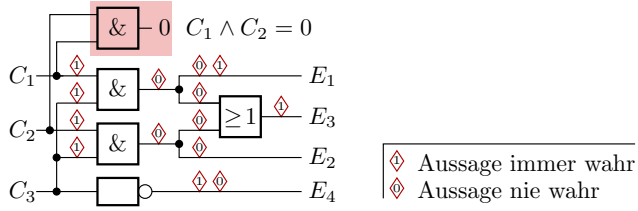
- Im CE-Modell können bei Ursache $C_3 = 0$ (max. Zählwert erreicht) Zeichen vom Type A oder B eingegeben werden, im Programm nicht. Wie lautet das gewünschte Sollverhalten?

7.95 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)	E_1	Ct_A++
C_2	Zeichen ist vom Typ B (Großbuchstabe)	E_2	Ct_B++
C_3	max. Zählwert nicht erreicht	E_3	Ct_N++
–	es wird kein Zeichen gelesen	E_4	Ende

7.96 CE-Ersatzschaltung mit Haftfehler



- Identisch nachweisbare Fehler zusammengefasst.
- Erkennbar redundante Haftfeher am oberen Gatter weggelassen.
- Geziele Testauswahl mit D-Algorthmus berechnet für die unterstellten Fehler Ursachenvektoren mit Bitwerten 0 und 1 (Ursache (nicht) eingetreten sowie X für Ursachen ohne Einfluss.
- Im Beispiel würde ein Test mit allen Kombinationen von Ursachen auch alle nachweisbaren Haftfehler erfassen.

C_i Ursache (cause) i.

E_i Wirkung (effect) i.

7.97 Grey-Box-Fuzzer mit Ursachenwichtung

Testvektor:		1	2	3	4	5	6	7	8	9	10	11	$g(x_i)$
Eingabebit	x_1	1	X	0	1	0	0	X	1	0	1	1	0,5
	x_2	1	1	X	1	1	X	1	X	X	X	X	1
	x_3	0	0	1	0	0	X	0	X	1	0	0	2^{-3}
	x_4	1	0	X	0	X	0	1	X	1	X	0	0,5
	x_5	1	1	1	X	0	1	1	1	X	0	1	$1-2^{-3}$

Ausgehend von dem Selbsstestkonzept für digitale Schaltkreise mit fehlerorientierter Wichtung aus Abschn. 6.3.4:

1. Langer Zufallstest und Abhaken aller nachweisbaren Modellfehler.
2. Testsuche für nicht abgehakte Modellfehler mit möglichst vielen Bitwerten »X« (kein Einfluss auf den Fehlernachweis).
3. Wichtungsanpassung zur Bevorzugung der gefundenen Tests.
4. Langer Zufallstest und Abhaken aller nachweisbaren Modellfehler.
5. Wenn nicht nachgewiesen Modellfehler übrig, Wiederholung ab 3.

Nach diesem Prinzip könnte man auch einen Grey-Box-Fuzzer programmieren.

Ursachenvektor:		1	2	3	4	5	6	7	8	9	10	11	Wichtung
Ursachenbit	C_1	1	X	0	1	0	0	X	1	0	1	1	50%
	C_2	1	1	X	1	1	X	1	X	X	X	X	1
	C_3	0	0	1	0	0	X	0	X	1	0	0	$\ll 50\%$
	C_4	1	0	X	0	X	0	1	X	1	X	0	50%
	C_5	1	1	1	X	0	1	1	1	X	0	1	$\gg 50\%$

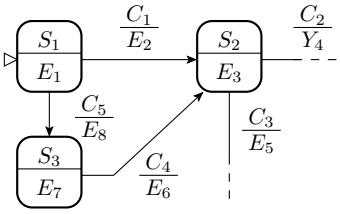
1. Langer Fuzz-Test. Abhaken der nachweisbaren CE-Haftfehler*.
2. Berechnung von Ursachenvektoren für nicht abgehakte CE-Haftfehler mit möglichst vielen »X« (Ursache ohne Einfluss).
3. Grobe Wichtungsanpassung der Ursachenbits zur Bevorzugung der gefundenen Ursachenvektoren.
4. Langer Fuzz-Test mit der gefundenen Ursachenwichtung, vielleicht mit genetischen Algorithmen durch Mutation, Auslese und Kombination günstiger Muster. Abhaken der nachweisbaren CE-Haftfehler.
5. Falls nicht alle CE-Haftfehler nachgewiesen, Wiederholung ab 3.

Ideen, die es wert sind, weiterverfolgt zu werden.

* Haftfehler in der CE-Ersatzschaltung.

3.5 Automaten

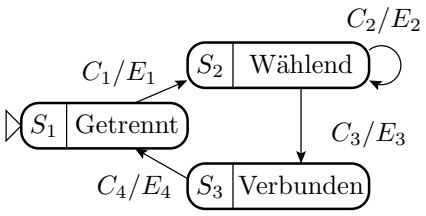
7.99 Zielfunktion als Automat



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.

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

7.100 Auf- und Abbau einer Telefonverbindung

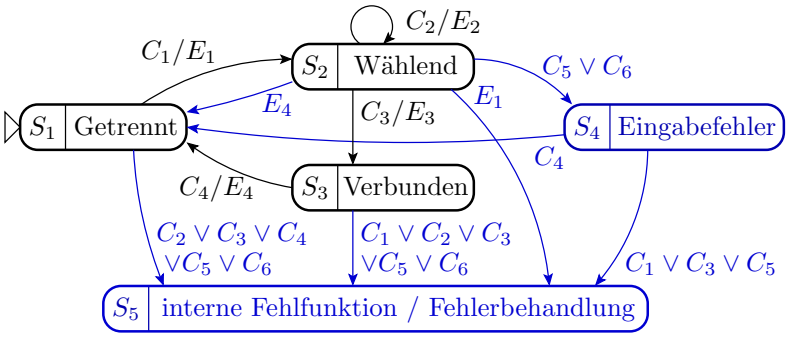


C_1	Abnehmen
C_2	Ziffer wählen
C_3	Rufnummer gültig
C_4	Auflegen
E_1	Rufnummer zurücksetzen
E_2	Ziffer zur Rufnummer hinzufügen
E_3	Verbindung aufbauen
E_4	Verbindung trennen

- 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

7.101 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		

7.102 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.

7.103 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).
- ...

7.104 Konkrete Tests

Für alle gefundenen symbolischen Tests, z.B. $C_1, C_2, C_2, \ldots, C_2, C_3, C_4$ sind konkrete Tests zu suchen, z.B.

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

7.105 Fuzzer

- Smart Fuzzer für Automaten wichtet die Eingaben je Zustand so, dass die Übergänge mit geeigneten Häufigkeiten stattfinden.
- Die angestrebten Häufigkeiten leiten sich aus Testanforderungen oder symbolischen Tests ab.
- Wichtung durch Instrumentierung und genetischen Algorithmen (Mutation, Auslese und Kombination günstiger Muster).
- Wichtungsänderung nach langen Fuzz-Tests, um bisher nicht erfüllte Nachweisbedingungen zu bevorzugen, gleichfalls denkbar.

3.6 Fehlerorientierte Testauswahl

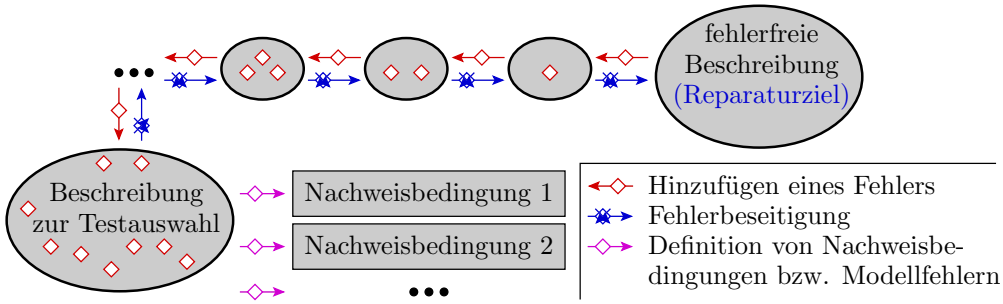
7.106 Fehlerorientierte Tests

Vorlesungsfazit: Gründliche dynamische Tests verlangen lange Zufallstests mit fehlerorientierter Bewertung und testzielorientierter Wichtung.

- Bei HW verlief vor Jahrzehnten die Entwicklung von »intuitiver« zu fehlerorientierter Testauswahl und weiter zu Selbsttests mit langen Zufallsfolgen, weil Testziele anders nicht mehr erreichbar.
- Für SW ist noch intuitive Testauswahl mit wenigen Test verbreitet.
- Bei erhöhten Verlässlichkeitsanforderungen Testbewertung mit Anweisungs-, Zweig- und Bedingungsabdeckung, d.h. ähnlich wie früherer HW-Toggle-Test.
- Fuzz-Tests, ursprünglich für Sicherheitsuntersuchung, zeigen, dass die Bedeutung langer Zufallstests den Praktikern bewusst ist.
- Fehlerinjektion zur Bewertung der Wirkung künstlich eingebauter Fehler wird hauptsächlich für die Bewertung von Maßnahmen der Fehlertoleranz und Sichersverbesserung genutzt.

SW-Bereich kann vermutlich noch einiges von der Entwicklung der fehlerorientierten Testauswahl im HW-Bereich lernen.

7.107 Mutationen statt Modellfehler



Alternativ zu den für SW etablierten und für HW unüblichen Testauswahlkriterien ist auch eine Auswahl auf Basis angenommener Fehler möglich. Entwurfsbeschreibungen enthalten die zu findenden Fehler:

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

Erschwerend ist auch das Fehlen der korrekten Beschreibung für den Soll-Ist-Vergleich zur Kontrolle der Testausgaben.

7.108 Beispiele für Programmmutationen

Mutationen sind geringe Verfälschungen der Testobjektbeschreibung:

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

Bestimmung der Mutationsabdeckung:

Zusammenstellung Mutationsmenge. Streichen redundanter und Zusammenfassung identisch nachweisbarer Mutationen
Wiederhole für jede Mutation
Übersetze mutiertes Programm inc. Tests und Kontrollen
Wiederhole für alle Test oder bis alle Mutationen nachgewiesen
Abarbeitung und Abhaken der nachweisbaren Mutationen

7.109 Mutationsbasierte Operationsprofilauswahl

In Anlehnung an den Selbsttest mit fehlerorientierte Wichtung (Abschn. 6.3.4 *Fehlerorientierte Wichtung*) mit der Verallgemeinerung »Operationsprofil statt Wichtung«:

Zusammenstellung Mutationsmenge. Streichen redundanter und Zusammenfassung identisch nachweisbarer Mutationen
Wiederhole, bis genug Mutationen ausreichend oft nachgewiesen werden
Wahl Op.-Profil, das bisher kaum nachweisbare Mutationen bevorzugt
Auswahl eines langen Zufallstestsatzes mit diesem Operationsprofil
Wiederhole für alle Tests dieses Testsatzes
Wiederhole für jede noch nicht nachweisbare Mutation
Bestimmung ob nachweisbar, wenn ja abhakten

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).

Zusammenfassung

7.110 Testauswahl für SW

Fehler entstehen in allen Entstehungsphasen. Aus jeder Entwurfsphase leiten sich Testanforderungen ab, die nach der Codierung in umgekehrter Reihenfolge abzuarbeiten sind.

Testanforderungen aus frühen Entwurfsphasen sind symbolische Tests, die beschreiben was wie gründlich zu testen ist.

Die automatische Erzeugung von ausführbaren aus symbolischen Tests verlangt vom Rechner verarbeitbare Beschreibungen:

- Ein- und Ausgabegabeformat, Testziele,
- testbare Anforderungen, Kontrollkriterien, ...

Typ. Reihenfolge der Auswahl dynamischer Tests:

- typischen Eingaben,
- gezielte Tests für Grenzfälle und MF-Behandlung,
- längere Zufallstests und
- Tests zu den symbolischen Tests früher Entwurfsphasen.

Instrumentierung. Zusatzcode im Testobjekt:

- Testausgaben, Kontrollen,
- Check von Abdeckungskriterien, ...

Fuzzing. Zufallstests mit formatierten Eingaben:

- Dump-Fuzzer: ohne Rücksicht auf Eingabemodell.
- Smart Fuzzer: Bevorzugung zulässiger und geringfügig falscher Eingaben.
- White-Box-Fuzzer: Geziele Eingabesuche, um bestimme Codebeteiche zu erreichen.
- Grey-Box-Fuzzer: Nutzung von Instrumentierung statt statischer Code-Analyse zur Eingabesuche.

Fehlerabdeckung:

- Gezielte Testauswahl: für Fehler ohne ähnlich nachweisbare Kriterien nicht möglich. Fehlerabdeckung etwa Anteil der ähnlich nachweisbaren Kriterien.
- Für Smart Fuzzer und gleiches Operationsprofil Fehlerabdeckung etwa Kriterienabdeckung der $c_{\mathrm{MF}}-$fachen Testanzahl.
- Skalierungsfaktor c_{MF} abhängig von den Abdeckungskriterien, für Codeabdeckungen vermulich deutlich kleiner eins, ...

7.112 Kontrollflussabdeckung

Aktuelle gelten 100% Kontrollflussabdeckung als ausreichend grünlicher Test:

- Anweisungsabdeckung für nicht sicherheitskritische SW,
- Zweigabdeckung, wenn erheblicher Schaden möglich und
- Bedingungsabdeckung, wenn sicherheitskritisch.

Extraktion Kontrollflussgraph und Instrumentierung von Abdeckungszählern automatisierbar.

Aus dem Kontrollflussgraph lassen sich symbolische Tests in Form zu testender Anweisungsfolgen ableiten, für die danach ausführbare Tests mit Eingabebereitstellung und Ausgabekontrolle zu entwerfen sind.

Kontrollflussabdeckung eignet sich für Testlängenabschätzung für Smart Fuzzer und FC-Verbesserung nach dem Grey-Box-Prinzip.

Die Modellierung von Bedingungsabdeckungskriterien durch Off-By-One- und Haftfehler könnte als Brücke dienen, Begiffe und Erfahrungen von der Fehlermodellierung und -simulation digitaler Schaltkreise zu übernehmen.

7.113 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.
Code-Analyse kann automatisch

- Def-Use-Tupel, Defs mit Use-Ketten,
- Use ohne Def (Gefahr der Nutzung nicht initialisierter Variablen),
- Def ohne Use (überflüssige Berechnungen)

bestimmen. Darauf lassen sich unterschiedliche Abdeckungsmaße definieren, auch wirklichkeitsnähere mit mehr Aussagekraft als Codeabdeckungsmaße.

Ob der Ansatz wirkliche Vorteile gegenüber der Bewertung mit einer Stichprobe von Off-By-One- und Haftfehlern hat, ist nicht untersucht.

Def-Use-Tupel auch nutzbar für die Fehlerlokalisierung durch Pfadrückverfolgung.

7.114 Äquivalenzklassen

Äquivalenzklasse sind Eingabemengen ähnlich zu verarbeitender Daten, beschreibbar durch Wenn-Dann-Beziehungen, auf als Testspezifikation in frühen Entwurfsphasen geeignet.

Eine Wenn-Dann-Beschreibung ist nutzbar für

- die Definition des Operationsprofils für den Fuzzer und die
- Definition von zählbarer Testvollständigkeitskriterien.

Als Testvollständigkeitskriterien wie für fertige Programme Ausführung aller Zweige (hier Fälle) oder Ausführung aller Bedingungen.

Testvollständigkeitsbedingungen auch durch Mengen von Off-By-One- und Haftfehlern ersetzbar.

Die Auswahl auf Basis einer (diversitären) Testspezifikationen aus einer frühen Entwurfsphasen hat den Vorteil, später vergessene Aspekte zu berücksichtigen, auf die das zu testende Programm keine Hinweise mehr enthält.

7.115 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).

Beides sind auch Wenn-Dann-Beschreibungen, und auch nutzbar für

- Ableitung symbolischer Tests,
- die Definition von Operationsprofilen für den Fuzzer,
- die Definition von zählbarer Testvollständigkeitskriterien, ...

Genau wie Äquvalenzklassen in der Regel diversitäre Beschreibungen zum Code, auch aus früheren Entwurfsphasen, auch mit zusätzlichen Informationen für zu testende Aspekte.

7.116 Testbare Anforderungen

In Abschn. 7.2.3 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 und aus diesen weiter

- symbolische Test,
- Testvollständigkeitskriterien,
- Operationsprofilen für den Fuzzer, ...

7.117 Fehlerorientierte Testauswahl

Lange Zufallstests mit fehlerorientierter Bewertung und testzielorientierter Wichtung sind erst ansatzweise im SW-Bereich angekommen:

- Code-Abdeckungen sind vergleichbar mit der früheren Toggle-Abdeckung für HW,
- Fuzz-Tests noch nicht die Regel,
- Fehlerinjektion als Bewertung mit einer representativen Stichprobe künstlich eingefügter Fehler für die Bewertung MF-Behandlung etc., aber kaum für die Testauswahl.

SW kann da vermutlich von der HW lernen, aber es gibt auch Unterschiede und zusätzliche Probleme:

- fehlende Soll-Beschreibung für Fehlermodellierung und Ergebnisvergleich,
- Mutationen statt Modellfehler,
- für fehlende Aspekte keine gezielte Testsuche möglich, ...

Fuzz-Test mit fehlerorientierter Bewertung und Wichtung zeichnen sich als Weg zu höherer Verlässlichkeit durch gründlicheres Testen ab.

7.118 Literatur

Literatur

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