Test und Verlässlichkeit Foliensatz 7: Software

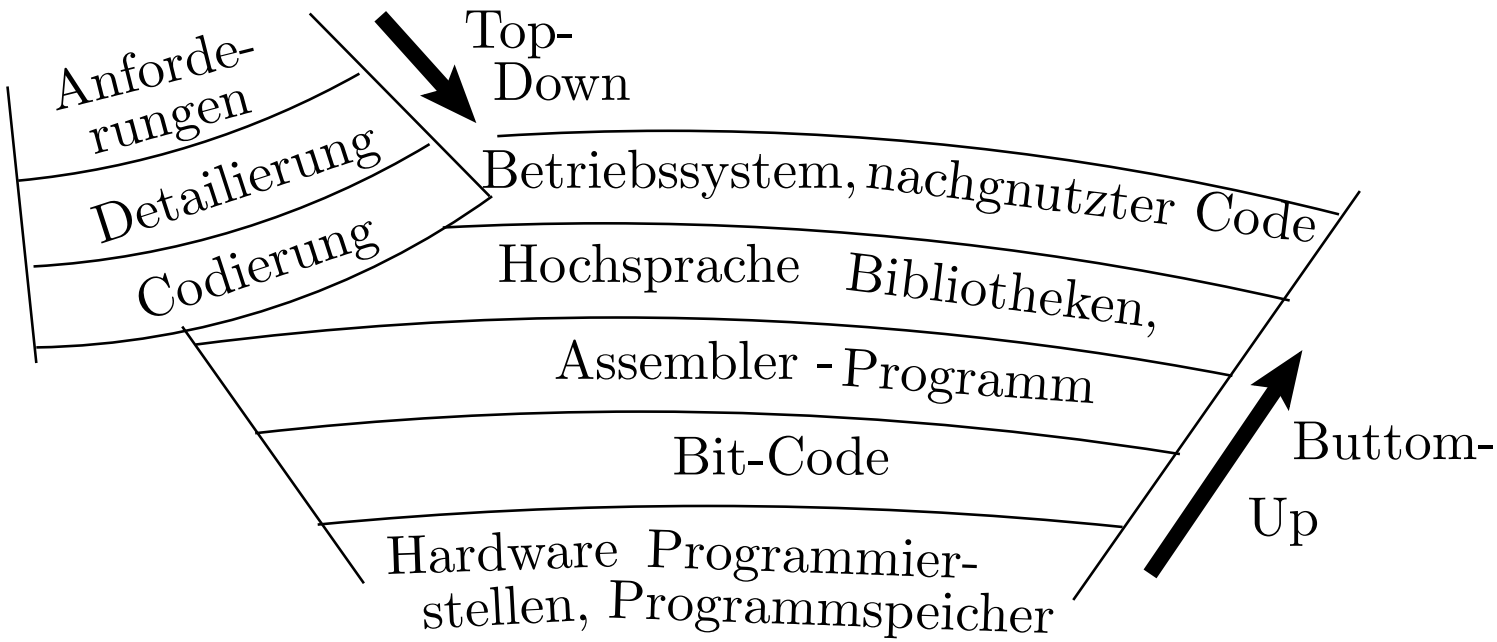
Prof. G. Kemnitz

Institut für Informatik, TU Clausthal (TV_F7)

5. August 2020

Inhalt TV_F7: Software

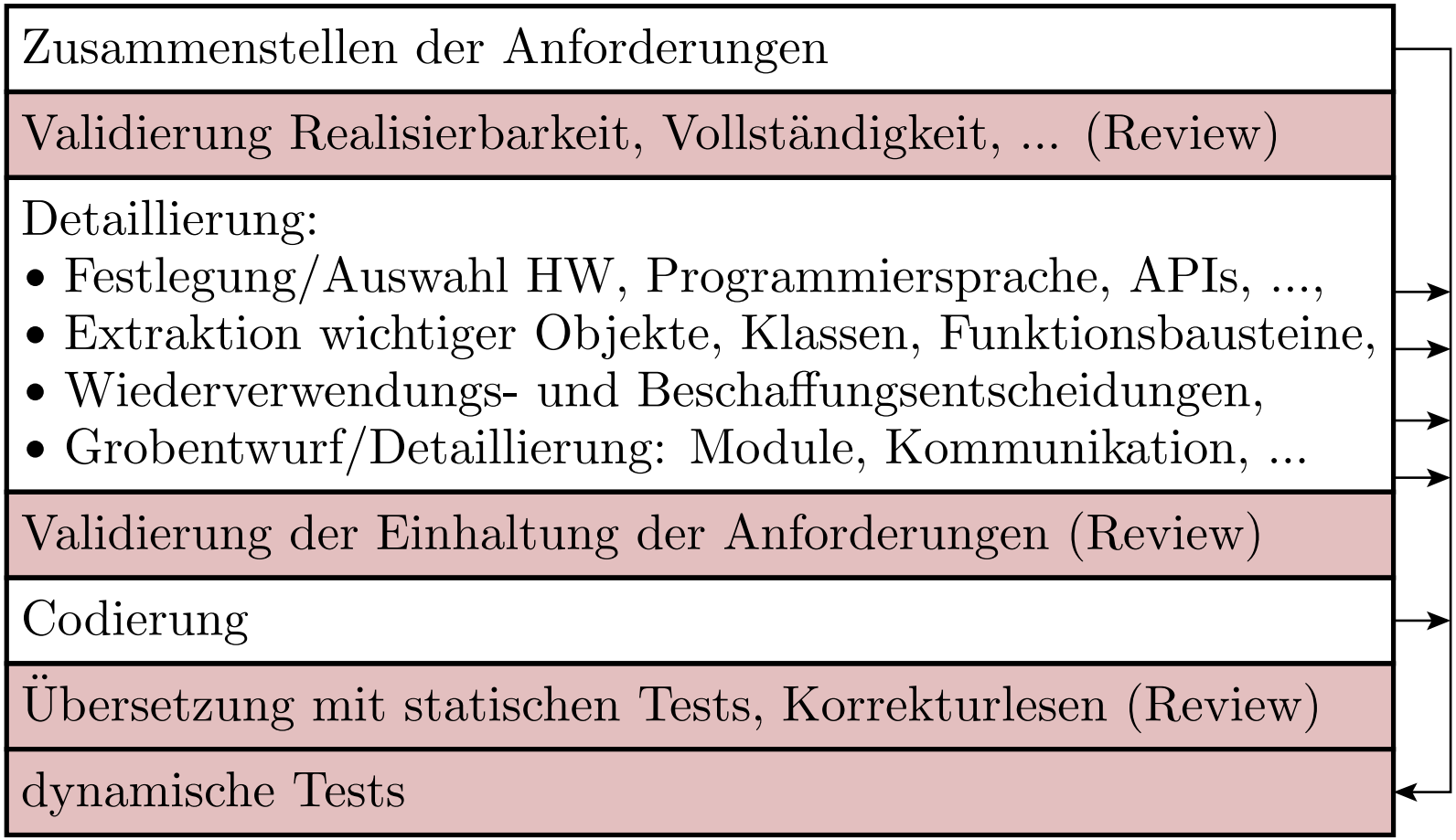
Software



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

Fehlervermeid., Test

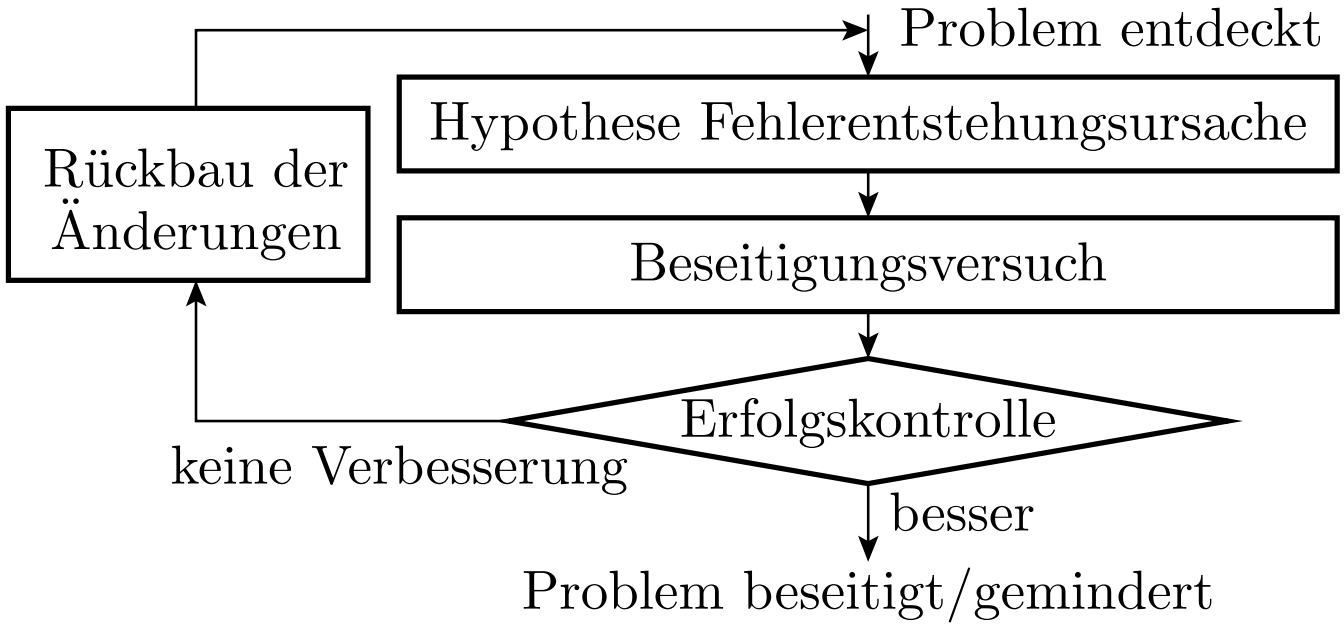
Der SW-Entwurf erfolgt in mehreren Stufen



Jede Stufe liefert Informationen über, was zu prüfen ist, und zu erwartende Fehler für die abschließenden dynamischen Tests.

Fehlervermeidung

Fehlervermeidung in Entwurfsprozessen



Nach FS 1, Absch. 5 »Fehlevermeidung« sind Fehler FF von Entstehungs-SL. Fehlervermeidung gleichbedeutend mit

- Beseitigung von Fehlern im Entstehungsprozess,
- Minderung der Störanfälligkeit von Entstehungs-SL

nach dem Prinzip der experimentellen Reparatur. Entwurfsprojekten fehlt die Wiederholung gleicher Entstehungs-SL zur Erfolgskontrolle ...

Vorgehensmodelle

Vereinheitlichung des Vorgehens für große Klassen von Projekten

- zur Aufwandsminimierung, besseren Vorhersagbarkeit und
- zur Fehlervermeidung durch »Lernen aus Fehlern«.

Vorgehensmodelle verlangen Kompromisse zwischen Kreativität und Reproduzierbarkeit.

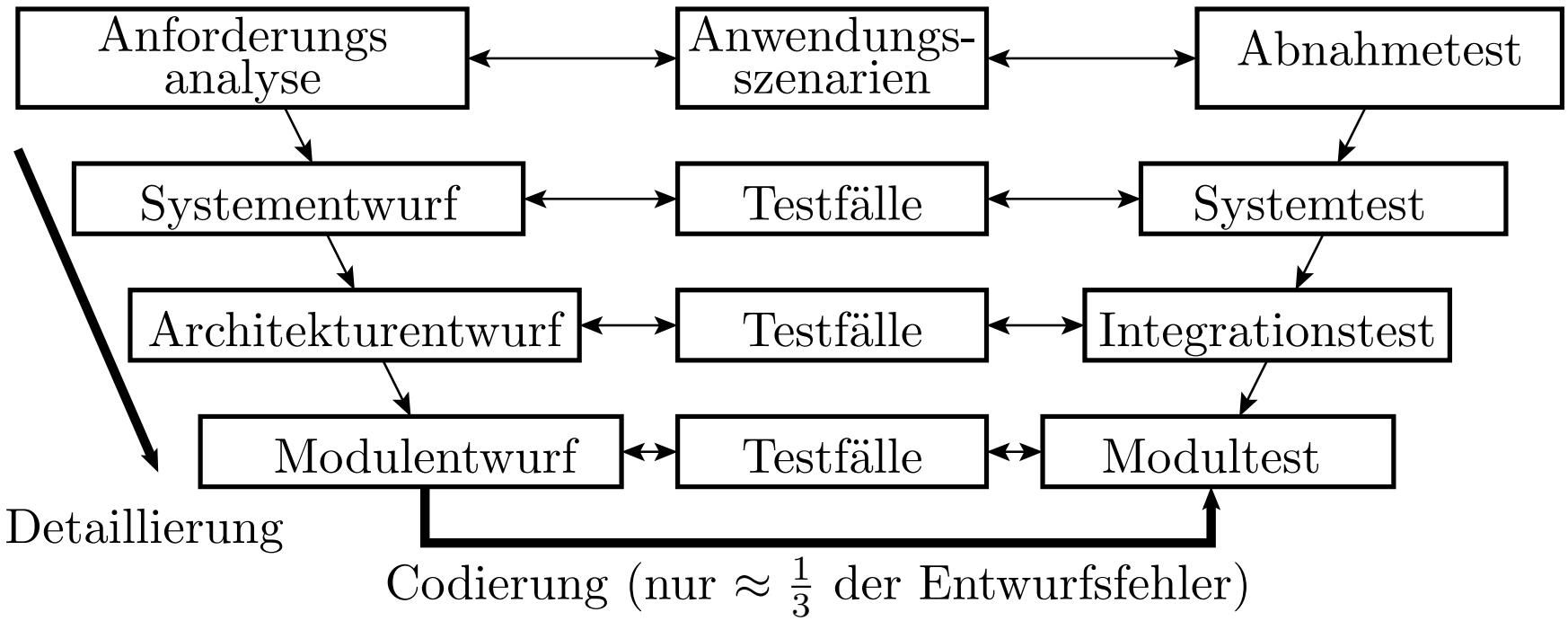
Fehlervermeidung verlangt für die Erfolgkontrolle:

- eine hohe Wiederholrate gleicher oder ähnlicher Tätigkeiten,
- einzuhaltende Arbeitsabläufe,
- Protokollierung aller Unregelmäßigkeiten und Probleme, ...

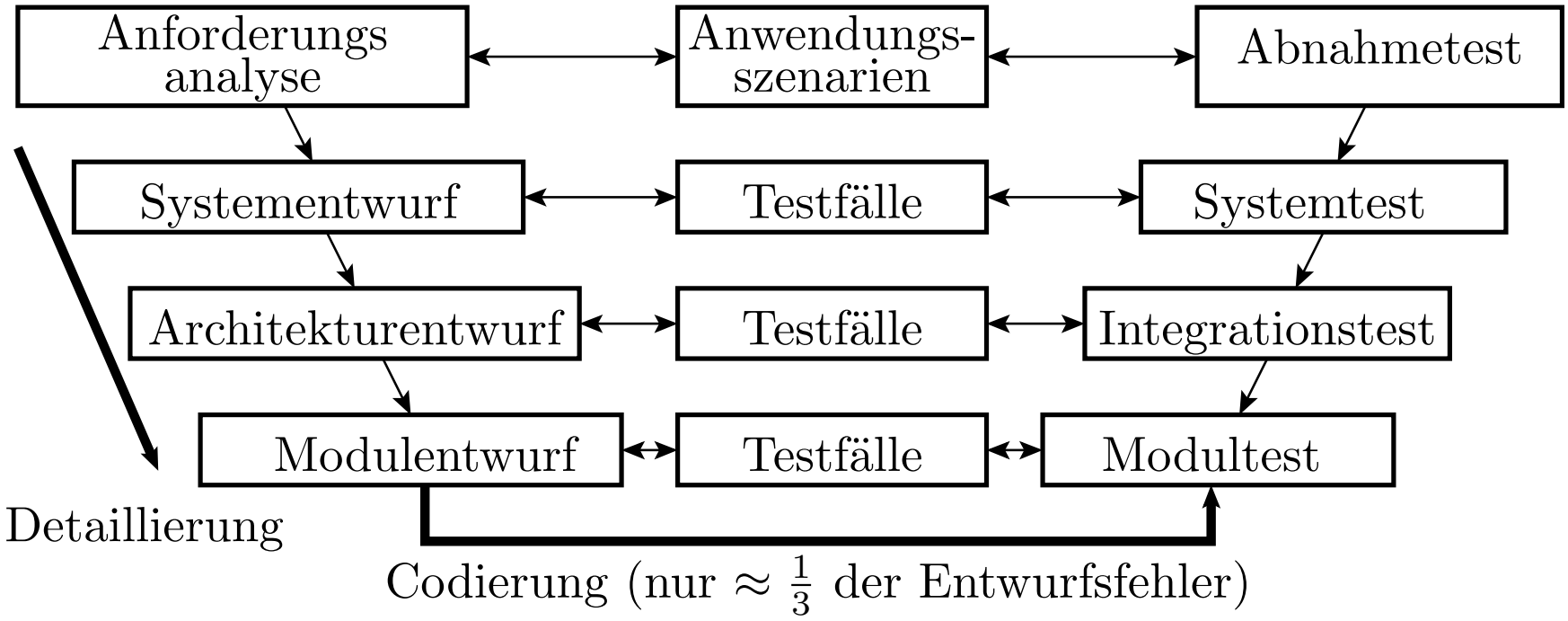
Kreativität verlangt »Einzigartigkeit«:

- Einbringen neuer Konzepte,
- Ausprobieren neuer Lösungswege,
- flexible Anpassung an sich ändernde Anforderungen.

Das V-Modell für Entwurfs- und Testabläufe



Das V-Modell ist ein Stufenmodell, in dem die Schrittfolge Anforderungszusammenstellung, Detaillierung und Codierung auch anderes oder in feinere Stufen unterteilt sein kann. In Ergänzung zu einem einfachen Stufenmodell definiert das V-Modell, dass in jeder Stufe Testszenarien oder Testfälle zu definieren und im aufsteigenden V-Ast abzuarbeiten sind.

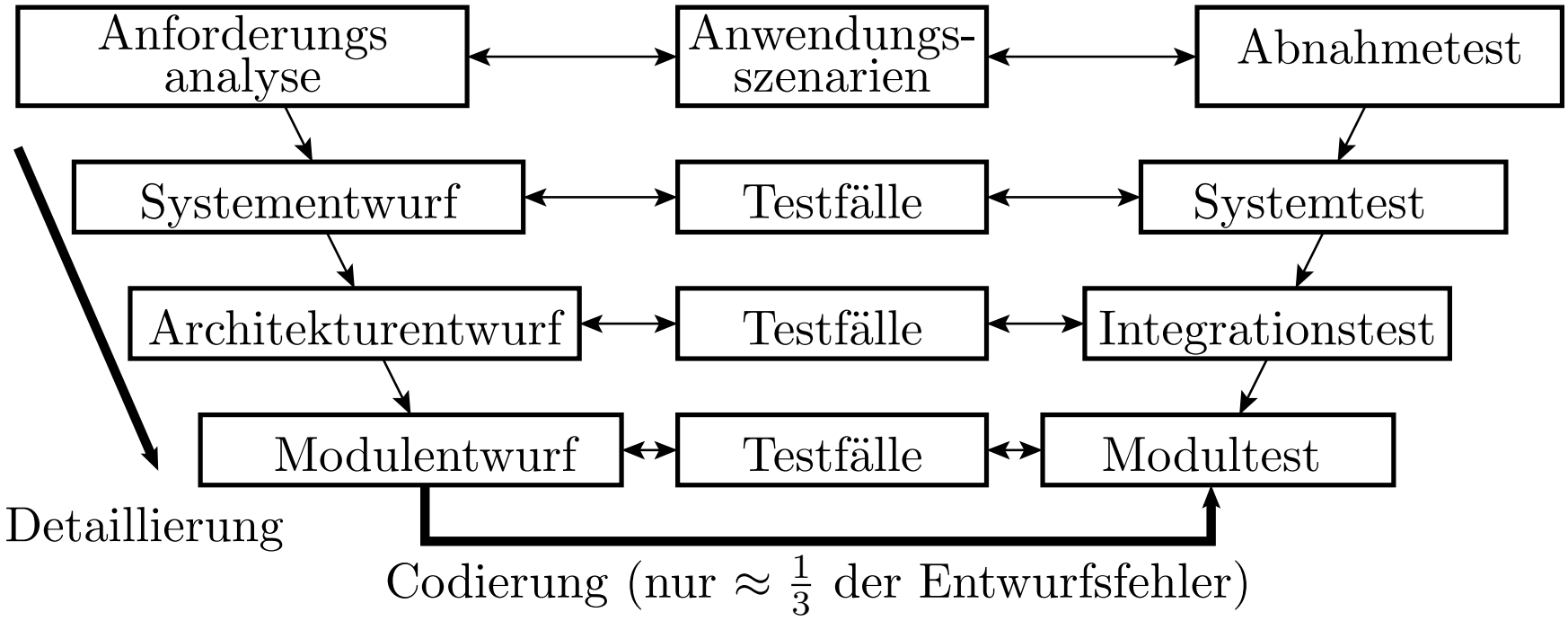


Das V-Modell-XT ergänzt formale Beschreibungsmittel, um

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

festzuschreiben.

Fehlervermeidung besteht praktisch in der Verbesserung der Vorgehensbeschreibung, z.B. in der V-Modell-XT-Notation.



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

Beispiel 1

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

Testbare Anforderungen

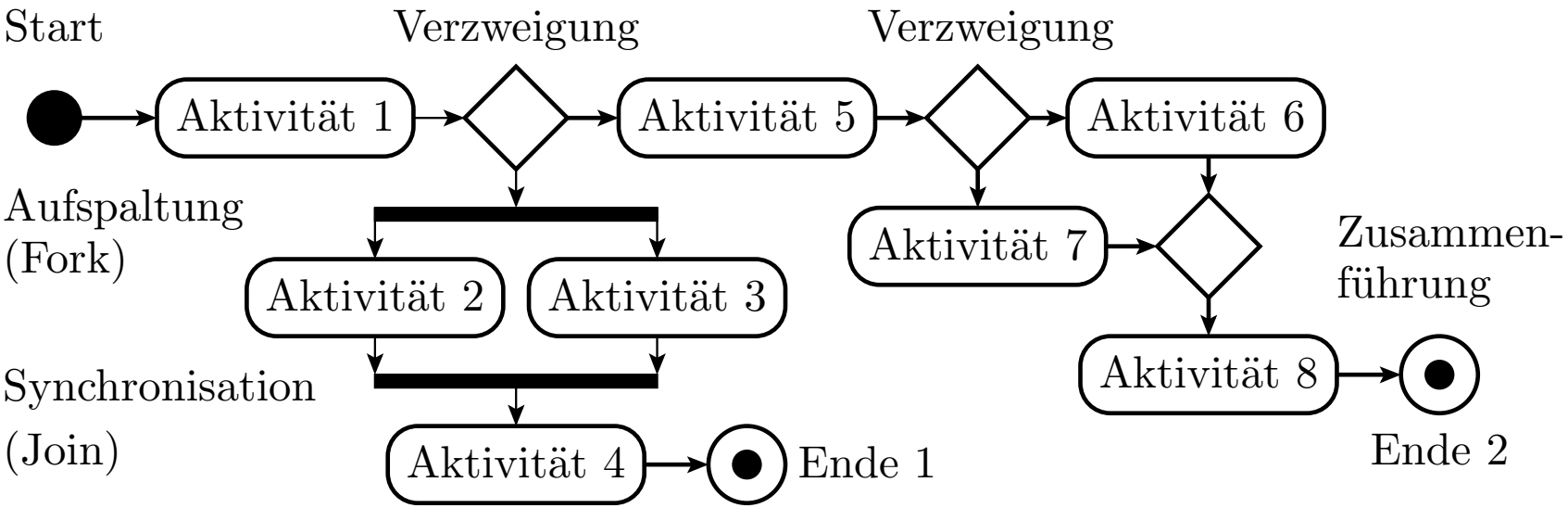
Testbare Anforderungen

Ein Grundprinzip zur Vermeidung und frühen Erkennung von Fehlern und Ungereimheiten bei der Zusammenstellung der Anforderungen ist, diese in einer überprüfbaren Form zu beschreiben:

- für Reviews (Korrekturlesen einzeln oder im Team, siehe später Abschn. #) als Abhaklisten,
- für andere statische Tests als zuzusichernde Bedingungen und
- für dynamische Tests so, dass Testbeispiele und Kontrollbedingungen ableitbar sind.
- für die Projektablauf einen Projektplan mit Meilensteinen und Zwischenkontrollen.
- Produktabnahme bzw. Anwendungsfreigabe erforderliche Kritärien und Kontrollen.

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

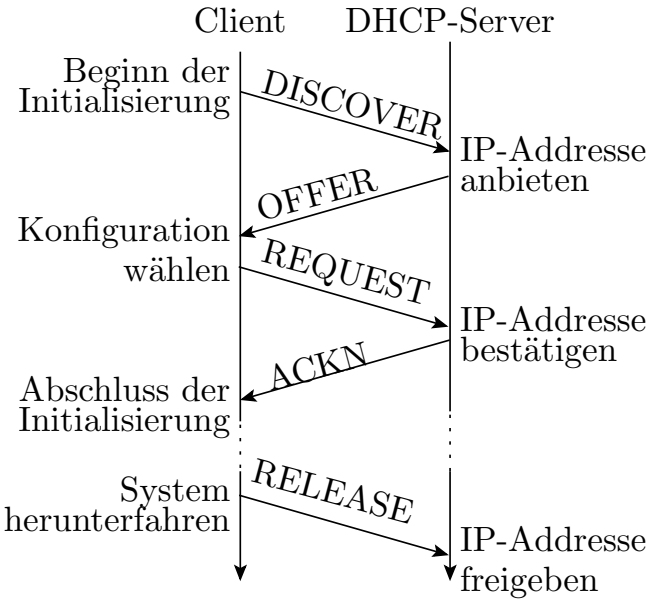
Aktivitätsdiagramm



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

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

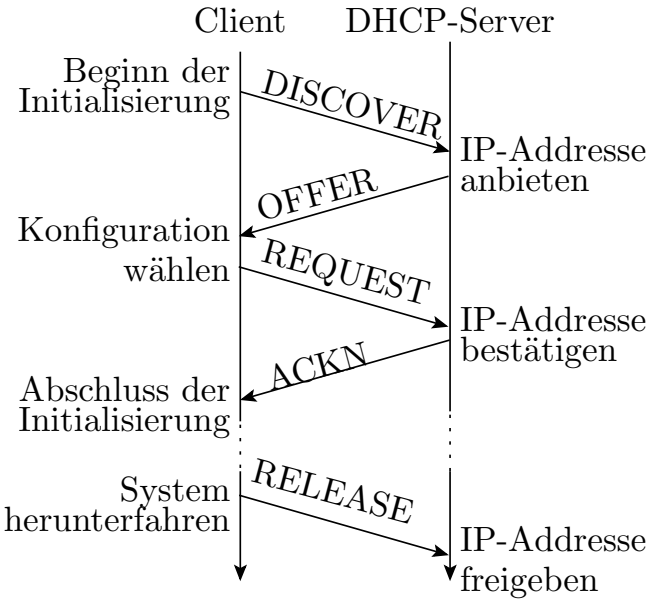
Sequenzdiagramm



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

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

Sequenzdiagramm

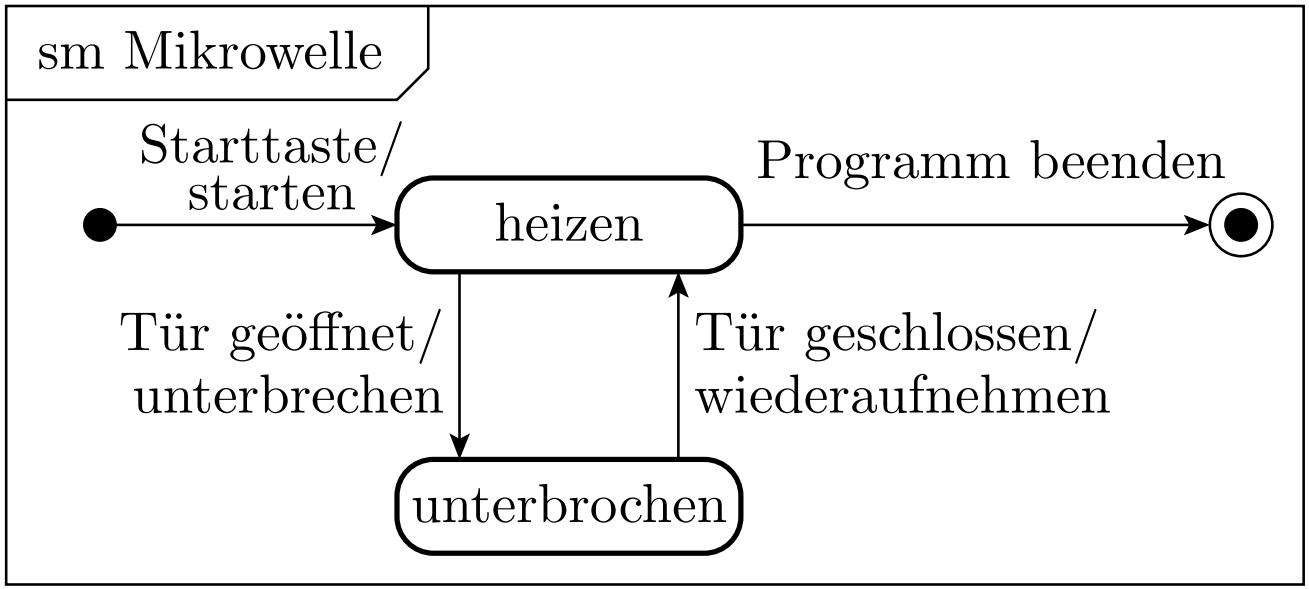


Nachricht	Beschreibung
DISCOVER	Broadcast eines Clients, der einen Server sucht
OFFER	Antwort eines Servers mitKonfigurationsvorschlag
REQUEST	Braodcast des Clients an den bevorzugten Server Ablehnung aller anderen Server
ACKN	Server liefertIP-Adresse
NAK	Der Server lehnt die IP-Adresse ab
DECLINE	Der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

Ableitbare Tests:

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

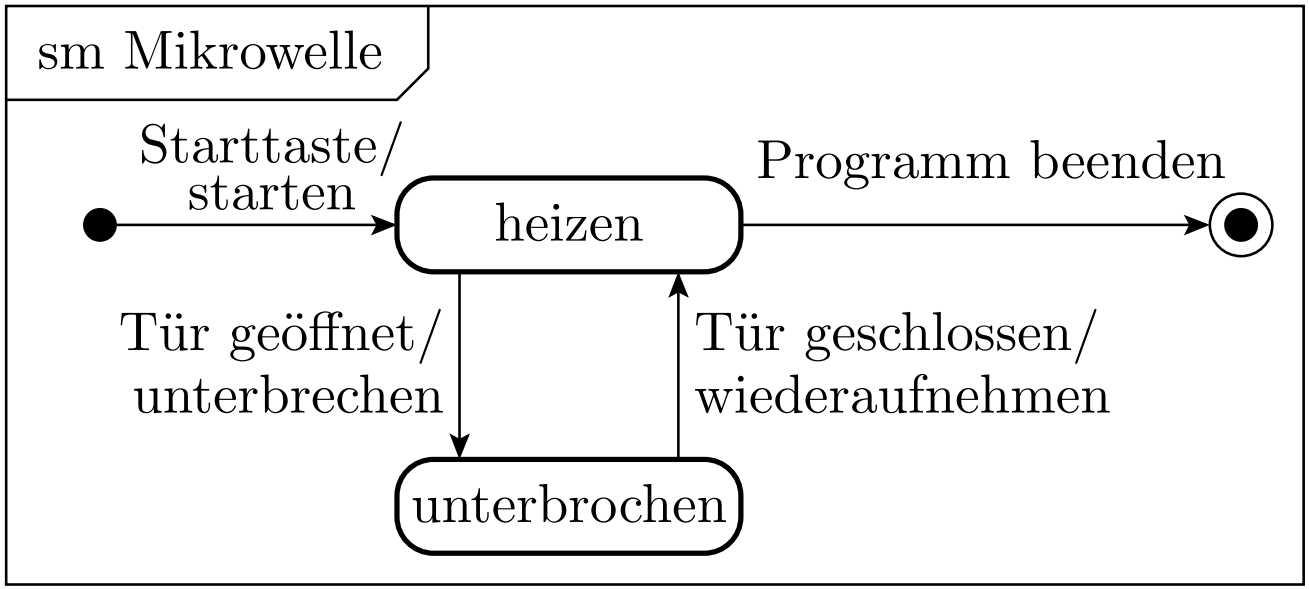
Zustandsdiagramm



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

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

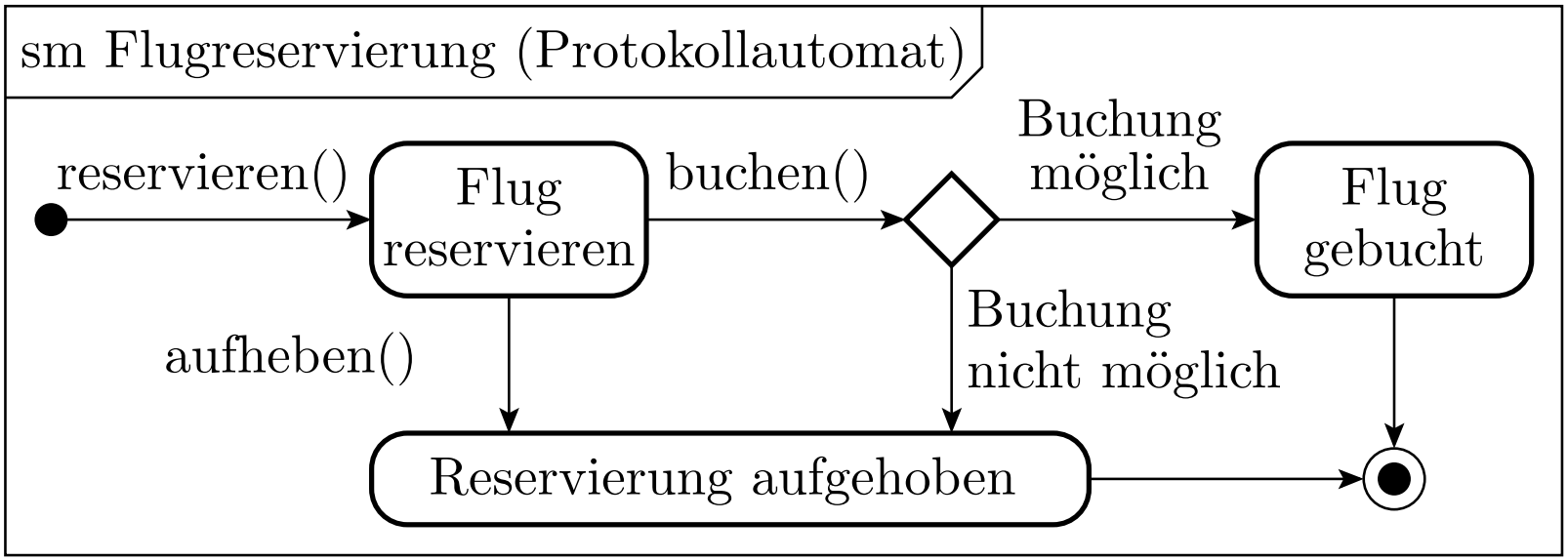
Zustandsdiagramm



Ableitbare Tests:

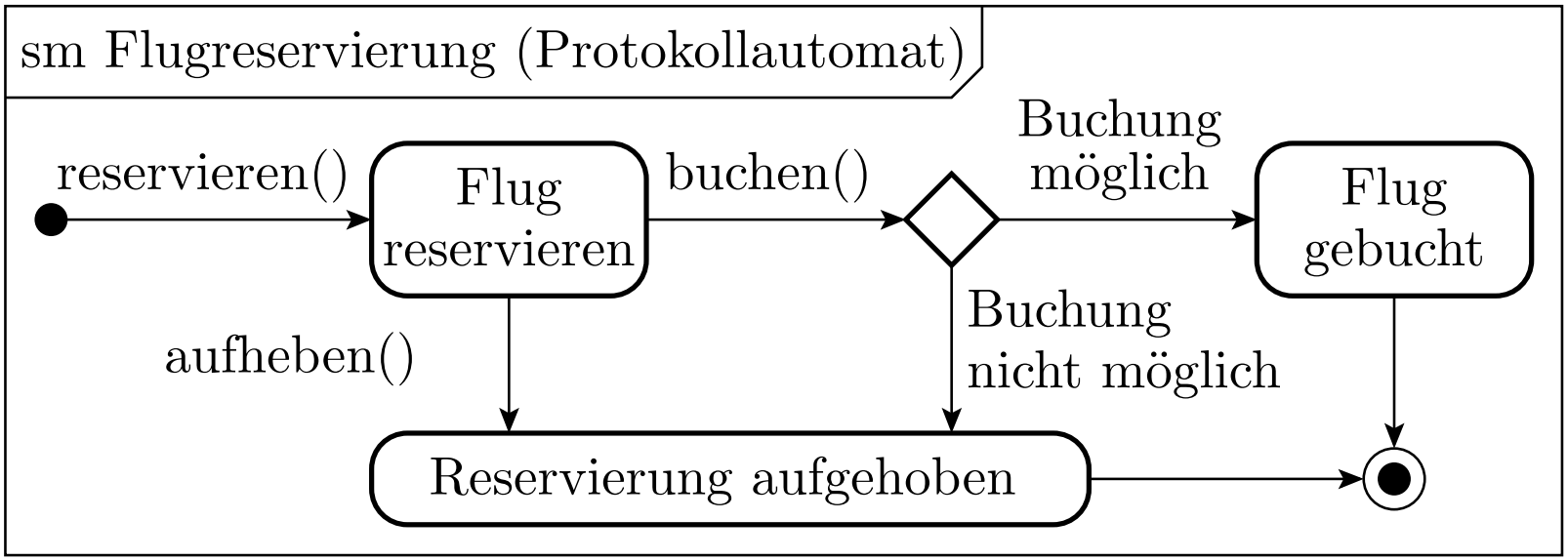
- Abläufe, die alle Knoten abdecken.
- Abläufe, die alle Kanten abdecken.
- Abläufe bis zu allen Knoten und Test der Reaktion auf nicht spezifizierte Übergangsbedingungen.

Protokollautomat



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

Protokollautomat

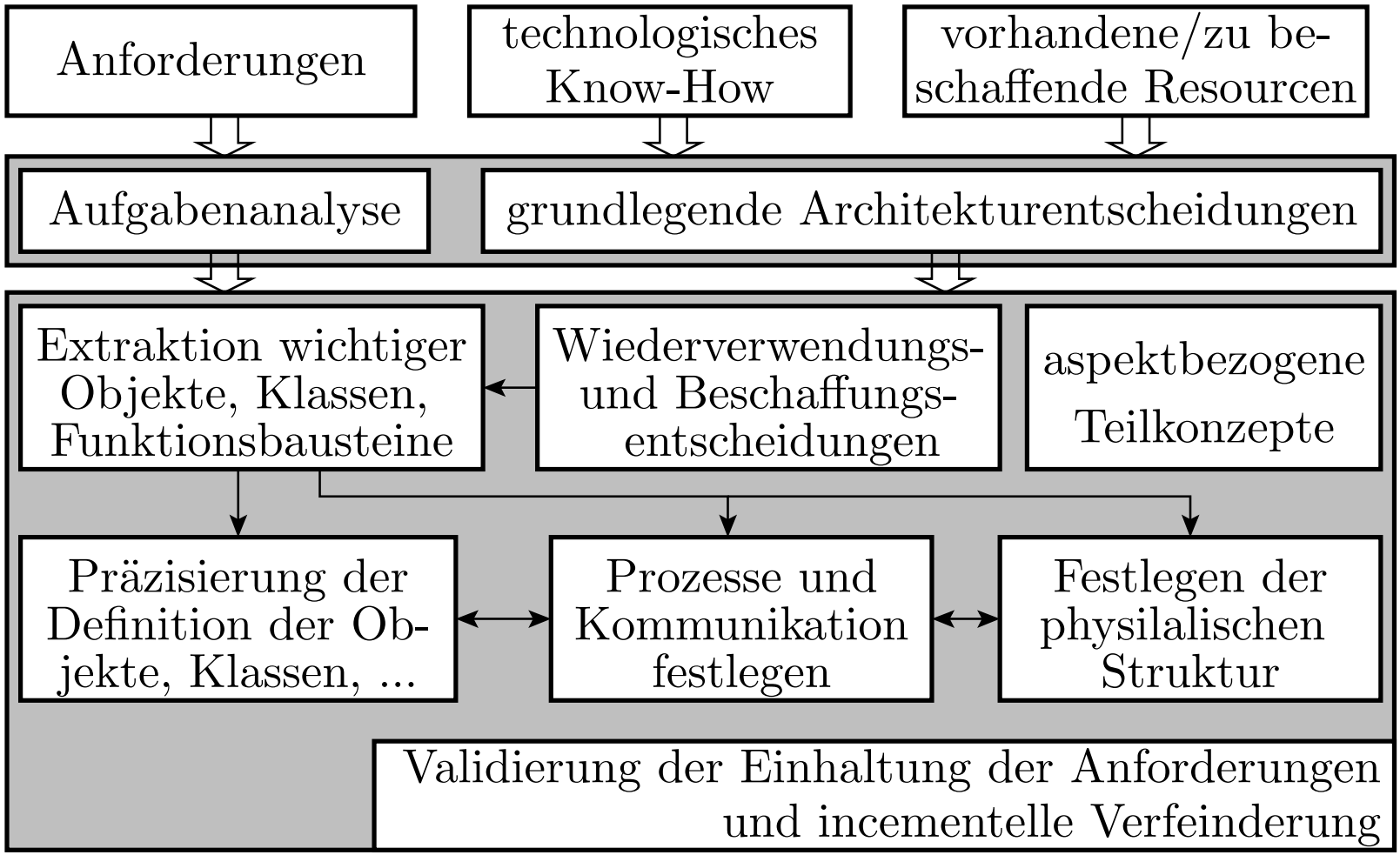


Kontrollautomaten können als Basis für Korrektheitsbeweise oder als Spezifikation für die Programmierung von Überwachungsautomaten genutzt werden.

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

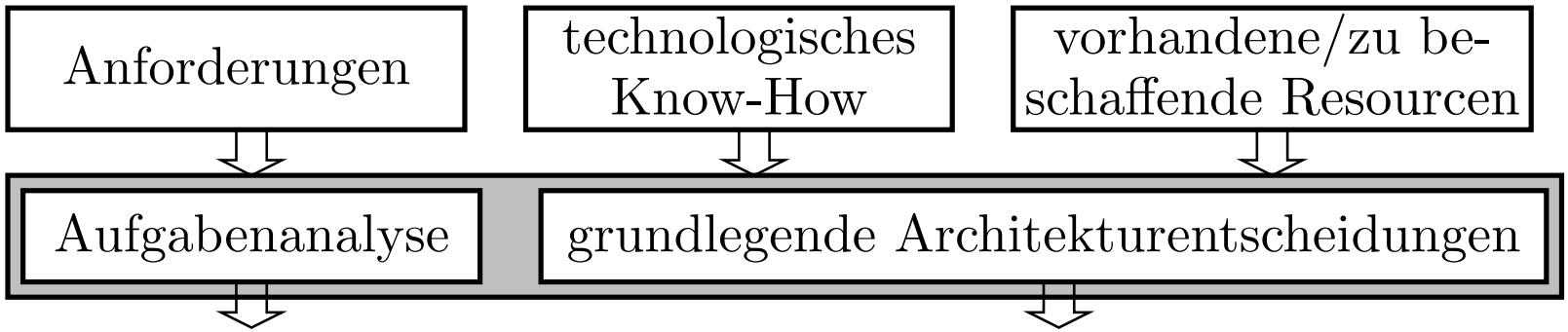
Detaillierung

System- und Architekturentwurf



Prozess zunehmender Detaillierung.

Vorgaben und Architekturentscheidungen



technologisches Know-How:

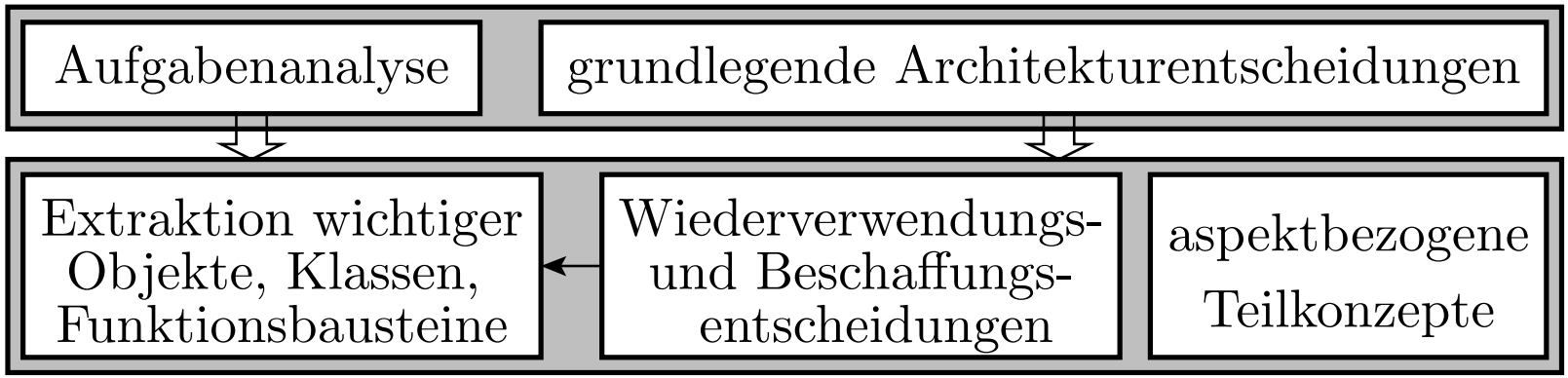
- Erfahrungen mit ähnlichen Projekten,
- nachnutzbare Software-Bausteine und Tests,
- alte Projektpläne, ...

vorhanden/zu beschaffen: Rechner, Software, Personal.

grundlegende Architekturentscheidungen:

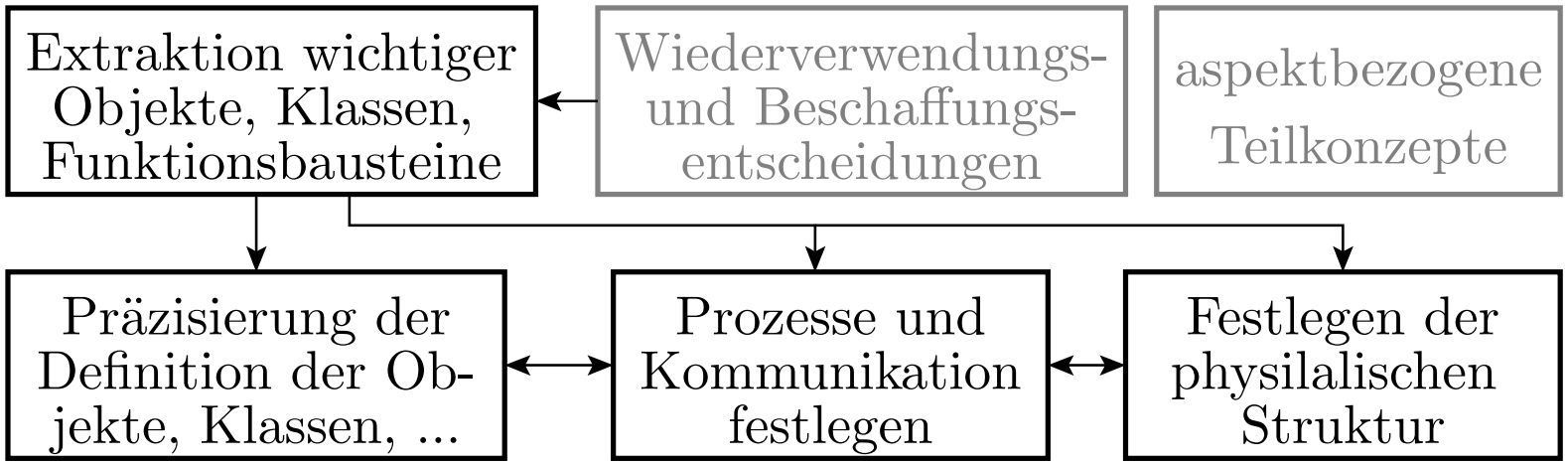
- Prozedurensammlung, Client-Server-Architektur, ...
- File-System oder Datenbank, ...
- Wiederverwendung, Vergabe von Unteraufträgen.
- Benutzerschnittstellen, Fehlerbehandlung, Fehlertoleranz, ...

Entwurf des Systemkonzepts



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

Schrittweise Verfeinerung



Intiale Festlegungen

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

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

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

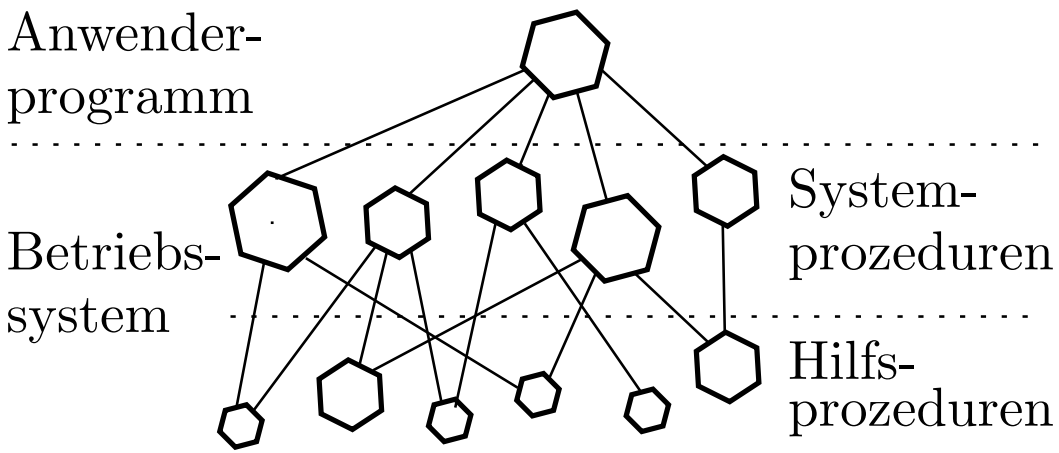
Software-Architektur

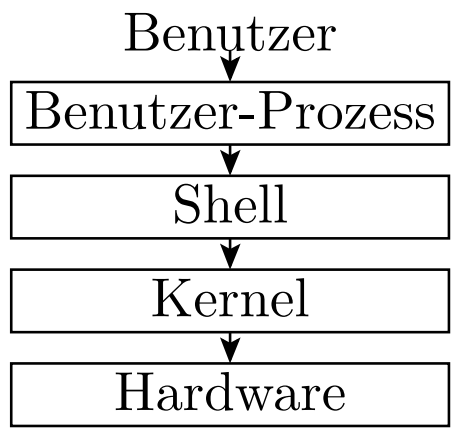
Software-Architektur

Je komplexer, desto wichtiger ein klare Struktur als Voraussetzung

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

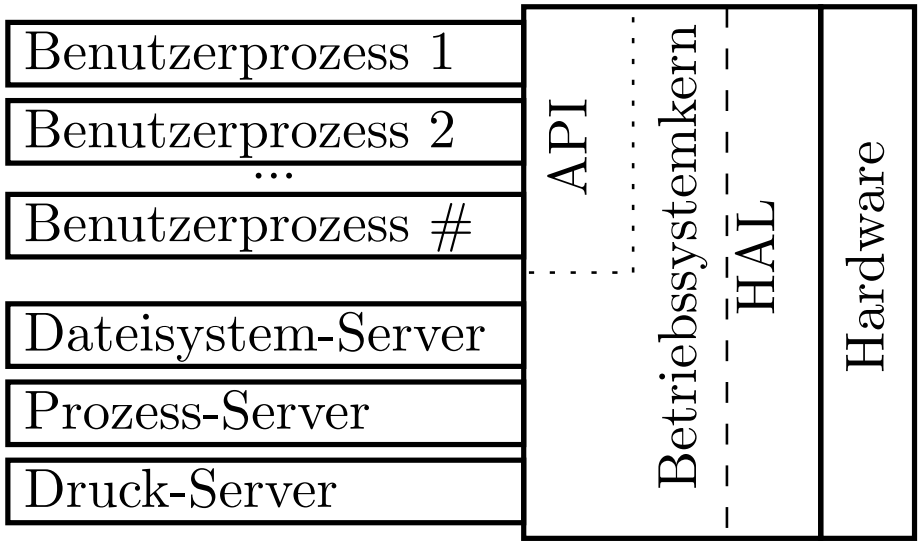
Prozedurensammlung und Schichten





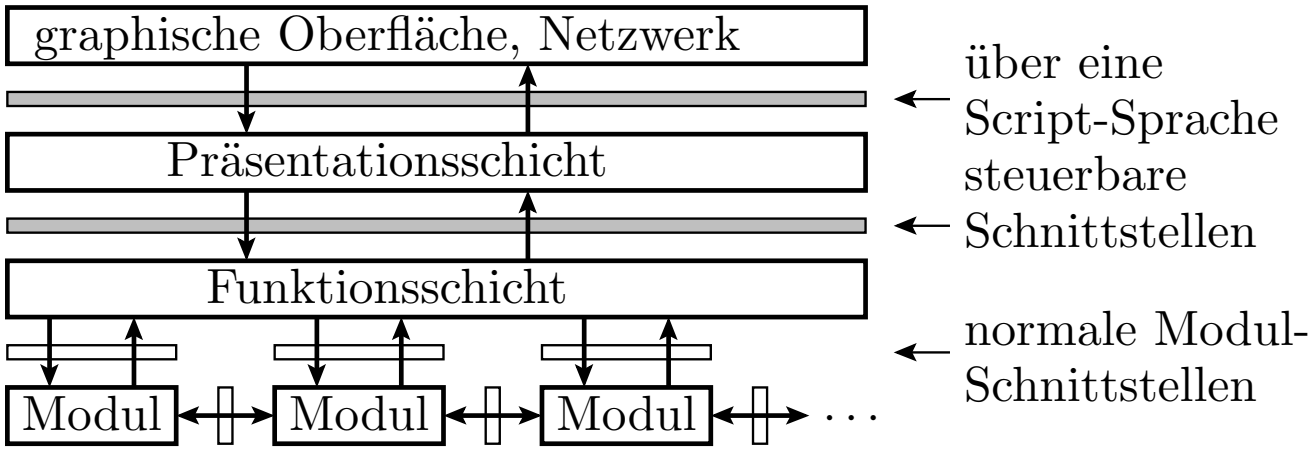
- Eine Prozedurensammlung bietet Schnittstellen, aber ein Programmierer muss sich nicht an diese Schnittstellen halten.
- Schichtenmodell: Ein Zugriff von einer anderen Schicht - beispielsweise einer Benutzeranwendung wie Excel oder Word - auf eine andere Schicht ist nur über eine definierte Schnittstelle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

Client/Server-Modell



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

Schichten als Testschnittstelle



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

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

Mit diesen Scriptsprachen dienen auch für den Test.

Codierung

Programmiersprache, Entwurfsumgebung, ...

Auswahlkriterien Sprache, Betriebssystem, ...:

- Möglichste einfache und übersichtliche Beschreibung der benötigten Funktionalität.
- Nutzbare fertige Klassen/Bausteine: Listen, Wörterbücher, Parser, Dateien, Fehlerbehandlung, ...
- Aufteilung der Fehlermöglichkeiten, um die sich der Compiler kümmert oder um die sich der Programmierer kümmern muss.
- Flexibilität & Performace vs. schwer zu findender Fehlermöglichkeiten.

Hilfreiche Funktionen einer Entwurfsumgebung:

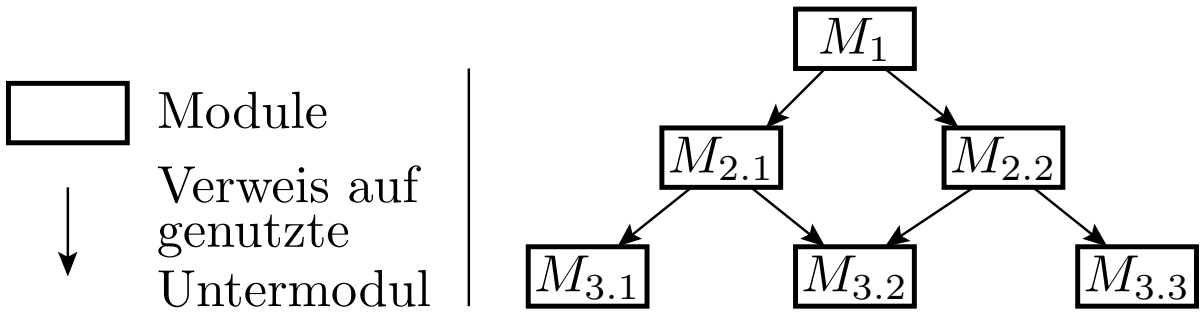
- Statische Kontrollen bei der Übersetzung.
- Eincompilieren von Kontrollen und Fehlerbehandlung für unzulässige Aktivitäten (Division durch null, WB-Überläufe, ...).
- Fehlerisolation und Ausschluss nicht autorisierter Zugriffe auf fremde Daten.

Hilfreiche Funktionen einer Entwurfsumgebung (Fortsetzung):

- Unterstützung Beschreibung, Durchführung und Archivierung von Tests.
- Versionsverwaltung für Regressionstest und den Rückbau in Fehlerbeseitigungsiterationen.
- Debugger mit Haltepunkten, Schrittbetrieb und Lese-/Schreibzugriff auf die Daten.
- Trace- und Event-Aufzeichung. Auffinden von totem Code, ...
- Unterstützung bei der Bestimmung von Code- und Fehlerüberdeckungen,
- Refractionig (Änderung von Bezeichnern).
- Unterstützung bei der Erstellung von Dokumentationen, auch für Reviews, Änderungen, ...

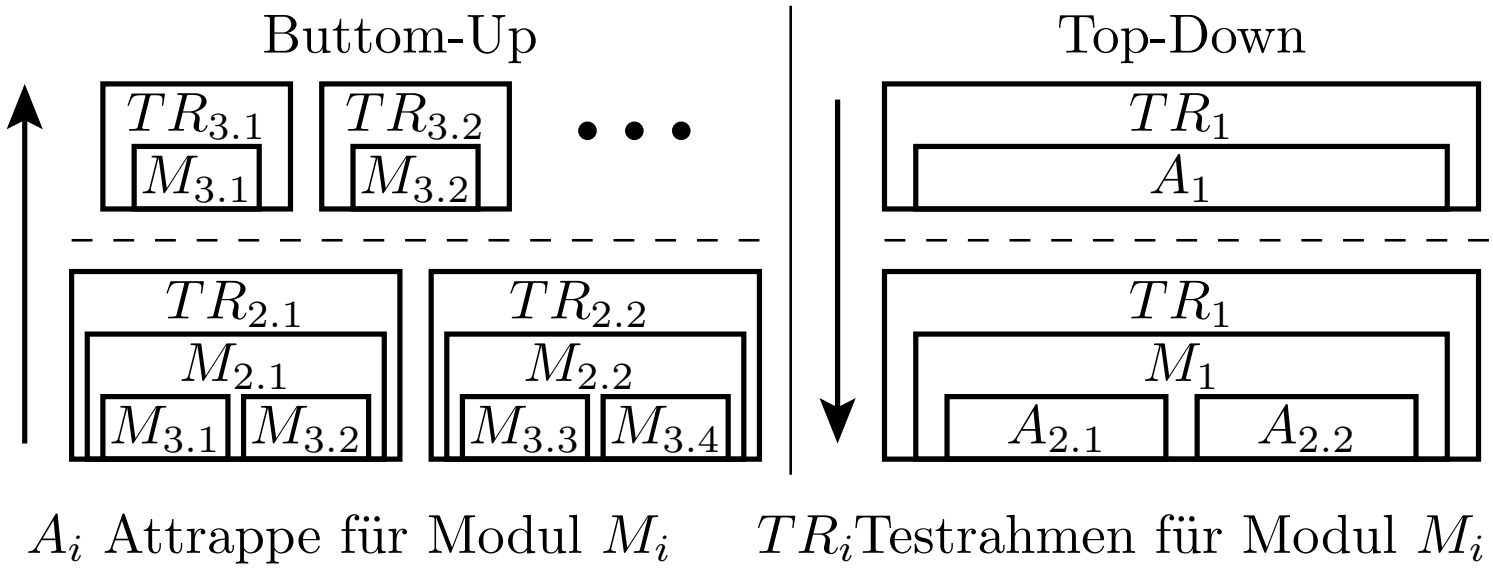
Codierung und Test

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.

Regeln für »Good Practice«

- Einfach, ohne überflüssige Schnörkel. Gut testbar. Gut änderbar.
- Verzicht auf Code für eventuelle künftige Erweiterungen, weil das voraussichtlich toter Code wird.
- Ausnahme Schnittstellen, weil nachträgliche Schnittstellenänderungen viel Nacharbeit mit hohem Fehlerentstehungsrate bedeuten.
- Wenn man das dritte mal dasselbe Stück Code schreibt, ist es Zeit für die Auslagerung in eine Hilffunktion, weil dann etwa klar ist, wie diese aussehen muss.
- Tests immer nach dem Prinzip »Fail Fast« programmieren, d.h. mit strengen Kontrollen und Abbruch bei FF.
- Sorgfäliger Entwurf externer Schnittstellen auch mit Rücksicht auf künftige Verwendung.

- Größenbegrenzungen: Funktionen ≤ 30 NLOC, Modul ≤ 500 NLOC, je schlechter testbar (z.B. nicht im Schrittbetrieb) um so kleiner und übersichtlicher.
- Fokus zuerst auf Korrektheit, dann erst auf Schnelligkeit.
- Codierung nur der benötigten Funktion statt Universallösungen mit einer Komplexität, die nicht erforderlich ist.
- Wenn ein Test versagt, zugrundeliegende Fehler sofort suchen beseitigen.
- Zum Test der Tests sollte jeder Test einmal mit einem wohlüberlegten Bug im Testobjekt zum versagen gebracht werden.
- ...

»Anti-Patter«

Das sollte man vermeiden:

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

Statische Tests

Statische Tests

Zusammenstellen der Anforderungen
Validierung Realisierbarkeit, Vollständigkeit, ... (Review)
Detaillierung
Validierung der Einhaltung der Anforderungen (Review)
Codierung
Review und Übersetzung mit statischen Tests

Während und nach den ersten Entstehungsschritten, bevor die Systemfunktion in einer maschinell abarbeitbaren Form vorliegt, sind die Kontrollmöglichkeiten begrenzt auf:

- Reviews: Korrekturlesen durch Kollegen oder im Team,
- Checklisten durchgehen,
- Demonstratoren für Teilaspekte zum Ausprobieren.

Weitere statische Kontrollmöglichkeiten für abarbeitbaren Code:

- Syntax, Schnittstellen, Typen, Wertebereichsgrenzen,
- Einhaltung von Codierungsregeln, ...

Inspektion

Inspektion (Review)

Inspektion, Sichtprüfungen (von lat. inspicere = besichtigen, betrachten). Anwendbar auf:

- Dokumentationen (Spezifikation, Nutzerdokumentation, ...),
- Programmcode, Testausgaben,
- Schaltungsbeschreibungen, Konstruktionspläne, ...

Einordnung, Merkmale und Besonderheiten:

- wenn manuell, arbeitsaufwändig,
- zufälliger Fehlernachweis mit subjektiv geprägter Güte,
- Nachweis nicht funktionaler Fehler (Standardverletzungen, unsichere Beschreibungsmittel, ...),
- für frühe Entwurfsphasen geeignet,
- Know-How-Weitergabe.

Automatisierung anstrebenswert.

Kenngrößen einer Inspektion

Inspektionsfehlerüberdeckung:

$$I\hat{F}C = \frac{\#EF}{\#F}$$

($\#EF$ – Anzahl der nachweisbaren; $\#F$ – Anzahl aller (entstandenen) Fehler). Insgesamt oder getrennt für funktionale und sonstige Fehler.

Abschätzmöglichkeiten:

- Capture-Recapture-Verfahren (klassischer Ansatz).
- Modell Zufallstest.

Weitere Bewertungsgrößen für Inspektionen nach [4]:

- Effizienz: Gefundene Abweichungen pro Mitarbeiterstunde.
- Effektivität: Gefundene Abweichungen je 1000 NLOC[1].

[1] NLOC: **N**etto **L**ines **o**f **C**ode. Anzahl der Code-Zeilen ohne Kommentar- und Leerzeilen.

Beispiel 2

Zähl- und Zeitwerte zur Bewertung einer Inspektion: Programmgröße: 10.000 NLOC, Arbeitsaufwand: 200 Stunden, 228 gefundene Fehler, davon 156 funktionale. Geschätzte Gesamtfehleranzahl (vor der Inspektion): 300, davon 200 funktionale. Wie groß sind

1. die Inspektionsfehlerüberdeckung,
2. die Effizienz und
3. die Effektivität

der Inspektion?

	gesamt	funktionale Fehler	sonstige Fehler
$I\hat{F}C = \frac{\#EF}{\#F}$	$\frac{228}{300}$	$\frac{156}{200}$	$\frac{72}{100}$
Effizienz	$\frac{228\text{ Fehler}}{200\text{ h}}$	$\frac{156\text{ Fehler}}{200\text{ h}}$	$\frac{72\text{ Fehler}}{200\text{ h}}$
Effektivität	$\frac{228\text{ Fehler}}{10.000\text{ NLOC}}$	$\frac{156\text{ Fehler}}{10.000\text{ NLOC}}$	$\frac{72\text{ Fehler}}{10.000\text{ NLOC}}$

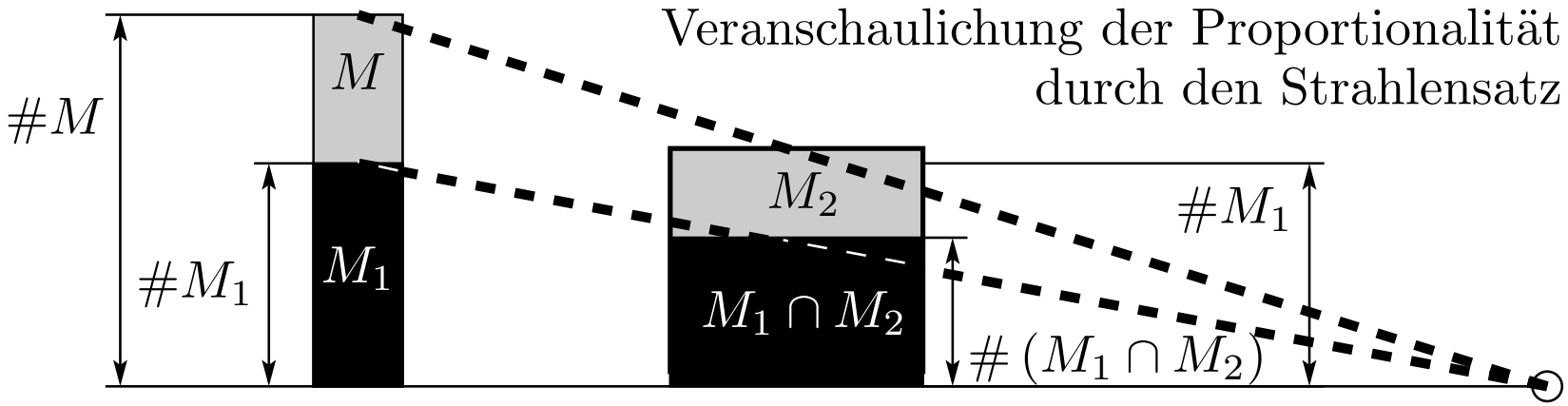
Effizienz und Effektivität ergeben sich ausschließlich aus Zähl- und gemessenen Zeitwerten. IFC basieren auf schlecht überprüfbaren Schätzwerten für die Gesamtfehleranzahl.

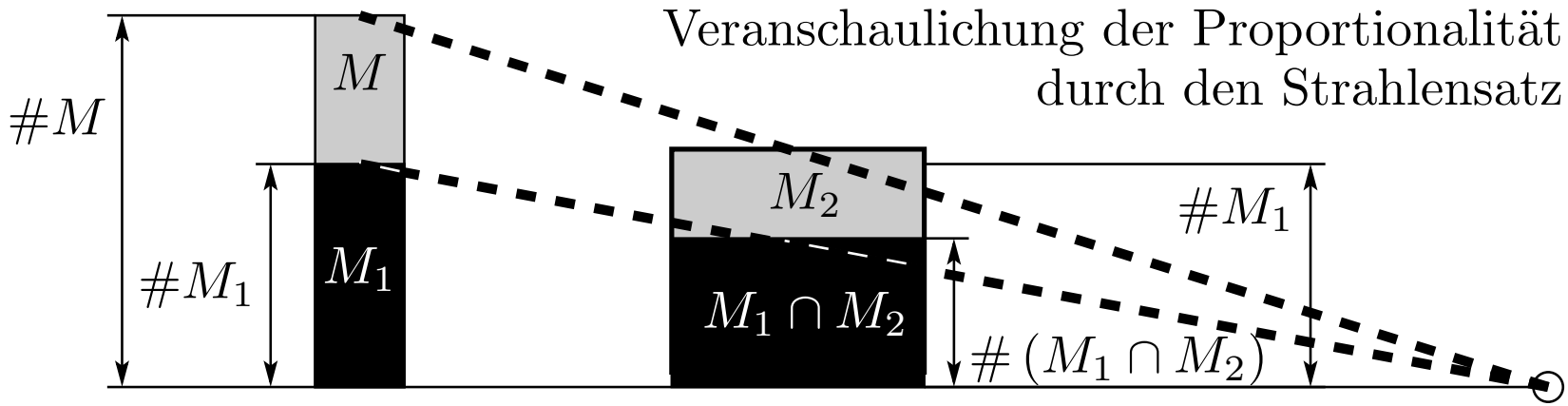
Capture-Recapture-Verfahren

Abgeleitet von einem Schätzer für die Größe von Tierpopulationen (z.B. von Vögeln in einem Gebiet) [2, 6, 5].

- Aus einer Menge M unbekannter Größe wird eine Menge M_1 von Tieren eingefangen, gekennzeichnet und freigelassen.
- Nach Vermischung der Population Menge M_2 von Tieren einfangen. Gekennzeichnete Tiere werden gezählt.

Bei tierunabhängiger Einfangwahrscheinlichkeit ergibt sich der Anteil der Tiere, die beim zweiten Einfangen gekennzeichnet sind, über den Strahlensatz:





$$\frac{\#M_1}{\#M} \approx \frac{\#(M_1 \cap M_2)}{\#M_2}$$

(#... – Größe der Mengen, hier Anzahl der Tiere; M – Menge aller Tiere, M_1, M_2 – beim ersten bzw. zweiten mal eingefangene Tiere; $M_1 \cap M_2$ – Menge der beide Male eingefangenen Tiere). Geschätzte Größe der Tierpopulation:

$$\#M \approx \frac{\#M_1 \cdot \#M_2}{\#(M_1 \cap M_2)}$$

Fehler statt Tiere

Zwei Inspekteure i finden jeweils eine Menge von M_i Fehlern:

$$\#M \approx \frac{\#M_1 \cdot \#M_2}{\#(M_1 \cap M_2)}$$

($\#(M_1 \cap M_2)$ – Anzahl der von beiden Inspekteuren unabhängig voneinander gefundenen gleichen Fehler; $\#M$ – geschätzte Anzahl der vorhandenen Fehler). Die geschätzte Fehlerüberdeckung ist das Verhältnis der Anzahl der insgesamt von beiden Inspekteuren gefundenen Fehler $\#(M_1 \cup M_2)$ zur geschätzten Geamtfehleranzahl $\#M$:

$$IFC \approx \frac{\#(M_1 \cup M_2)}{\#M} \approx \frac{\#(M_1 \cap M_2) \cdot \#(M_1 \cup M_2)}{\#M_1 \cdot \#M_2}$$

Gebunden an die Annahmen:

- Inspekteure erkennen die Fehler unabhängig voneinander.
- Alle Fehler haben dieselbe Erkennungswahrscheinlichkeit.

Beispiel 3

Inspektionsergebnisse für ein Programm:

- Inspekteur 1: 228 gefundene Fehler, davon 156 funktionale.
- Inspekteur 2: 237 gefundene Fehler, davon 163 funktionale.

Schnittmenge: 105 Fehler, davon 73 funktionale.
Welche Schätzwerte ergeben sich nach dem Capture-Recapture-Verfahren für

1. die Gesamtfehleranzahl,
2. die Inspektionsfehlerüberdeckung?

1. Gesamtfehleranzahl:

$$\#M \approx \frac{\#M_1 \cdot \#M_2}{\#\left(M_1 \cap M_2\right)}$$

2. Inspektionsfehlerüberdeckung:

$$IFC \approx \frac{\#\left(M_1 \cap M_2\right) \cdot \#\left(M_1 \cup M_2\right)}{\#M_1 \cdot \#M_2} \tag{1}$$

1. Gesamtfehleranzahl:

$$\#M \approx \frac{\#M_1 \cdot \#M_2}{\#(M_1 \cap M_2)}$$

2. Inspektionsfehlerüberdeckung:

$$IFC \approx \frac{\#(M_1 \cap M_2) \cdot \#(M_1 \cup M_2)}{\#M_1 \cdot \#M_2} \quad (1)$$

Fehler	$\#M_1$	$\#M_2$	$\#(M_1 \cap M_2)$	$\varphi = \#M$	IFC
alle	228	237	105	515	70%
funktional	156	163	73	348	71%
sonstige	72	74	32	166	68%

Vertrauenswürdigkeit der Schätzung

Zufälliger Fehler:

1. Als Richtwert ist die Intervallbreite, um die im Mittel Zählwerte von ihren Schätzwerten abweichen, das zwei bis fünffache der Wurzel aus dem Zählwert (FS3, Abschn. Bereichsschätzung). Für einen Schätzwert 100 nicht erkannte Fehler beträgt das Intervall für den Erwartungswert $[80, 120]$ bis $[50, 150]$.
2. Bei Multiplikationen und Divisionen addieren sich die relativen Schätzfehler. In Gl. 1

$$IFC \approx \frac{\#\left(M_1 \cap M_2\right) \cdot \#\left(M_1 \cup M_2\right)}{\#M_1 \cdot \#M_2}$$

   von vier Zählwerten, d.h. man benötigt 4^2 mal so große Zählwerte um die IFC mit derselben Genauigkeit/Irrtumswahrscheinlichkeit zu schätzen.

Die Zählwerte für die Fehleranzahlen müssten bei 10^3 bis 10^4 liegen, was sie in der Praxis nicht tun werden. Hinzu kommen systematische Fehler ...

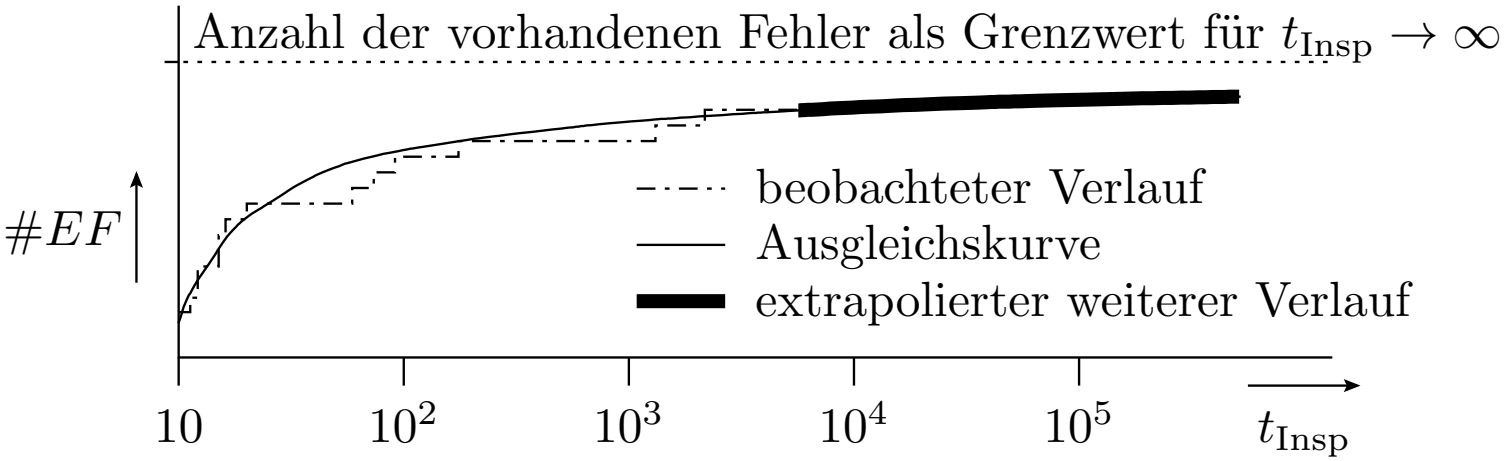
Systematische Fehler:

- Capture-Recaptur unterstellt für alle Fehler gleiche Erkennungswahrscheinlichkeit. Wenn gut erkennbare Fehler viel besser als schlecht erkennbare nachgewiesen werden, gelten die Schätzwerte nur für die gut erkennbaren.
- Capture-Recaptur verbietet Informationsaustausch zwischen den Inspekteuren. Falls es doch einen Informationsaustausch gibt, vergrößert der die Menge der gleichen gefundenen Fehler $M_1 \cap M_2$ gegenüber einer unabhängen Suche.
- Wenn die Inspekteure ihre Fehlerlisten voneinander abschreiben $M_1 = M_2$, ergibt sich als Schätzwert für die Inspektionsfehlerüberdeckung 100%.

Inspektion als Zufallstest

Einbeziehung, dass Fehlernachweiswahrscheinlichkeiten auch bei einer Inspektion um Größenordnungen variieren.

- Aufzeichnung der Anzahl der gefundenen Fehler in Abhängigkeit von der Inspektionsdauer.
- Abschätzen des weiteren Verlaufs.
- Gesamtfehleranzahl ist der Grenzwert für eine unendliche Inspektionsdauer[2]:

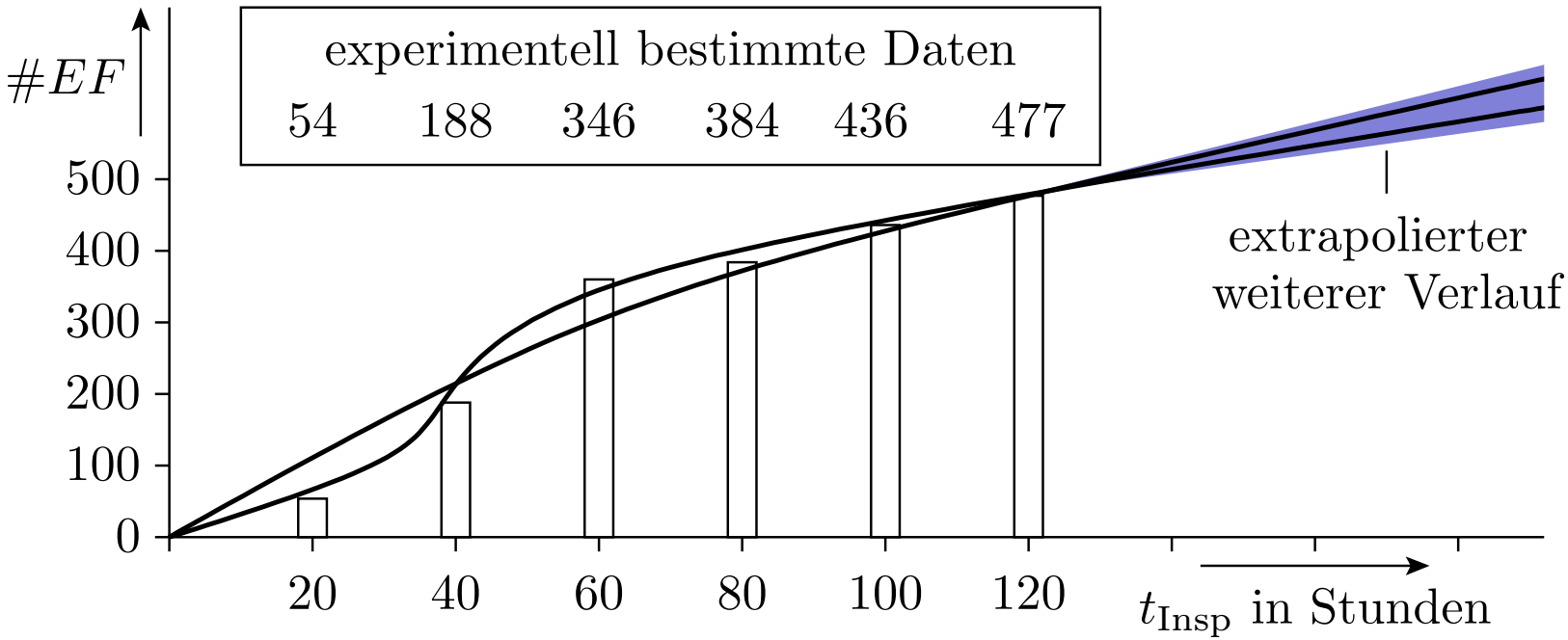


[2] Untersuchungen in dieser Richtung in der Literatur noch nicht gefunden.

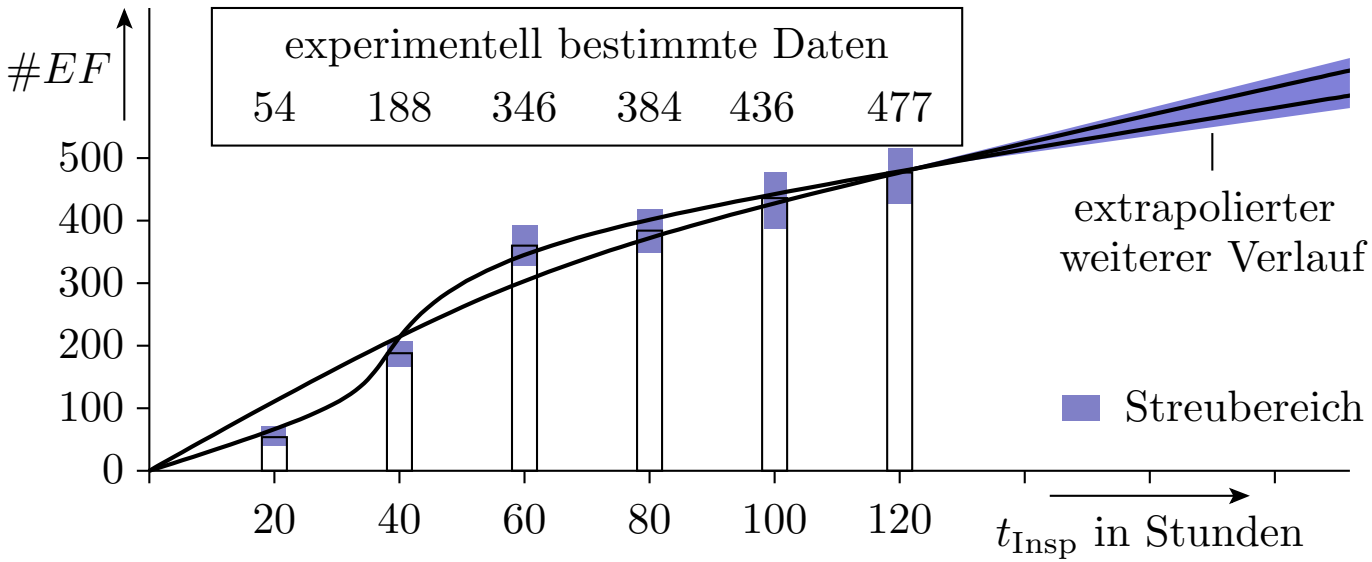
Experiment mit einem Inspekteur[3]

Inspektion des Buchmanuskripts [3] plus Beispielprogramme:

- Anzahl der gefunden Fehler in Abhängigkeit von der Inspektionsdauer.



[3] Bachelor-Arbeit von Yu Hong.



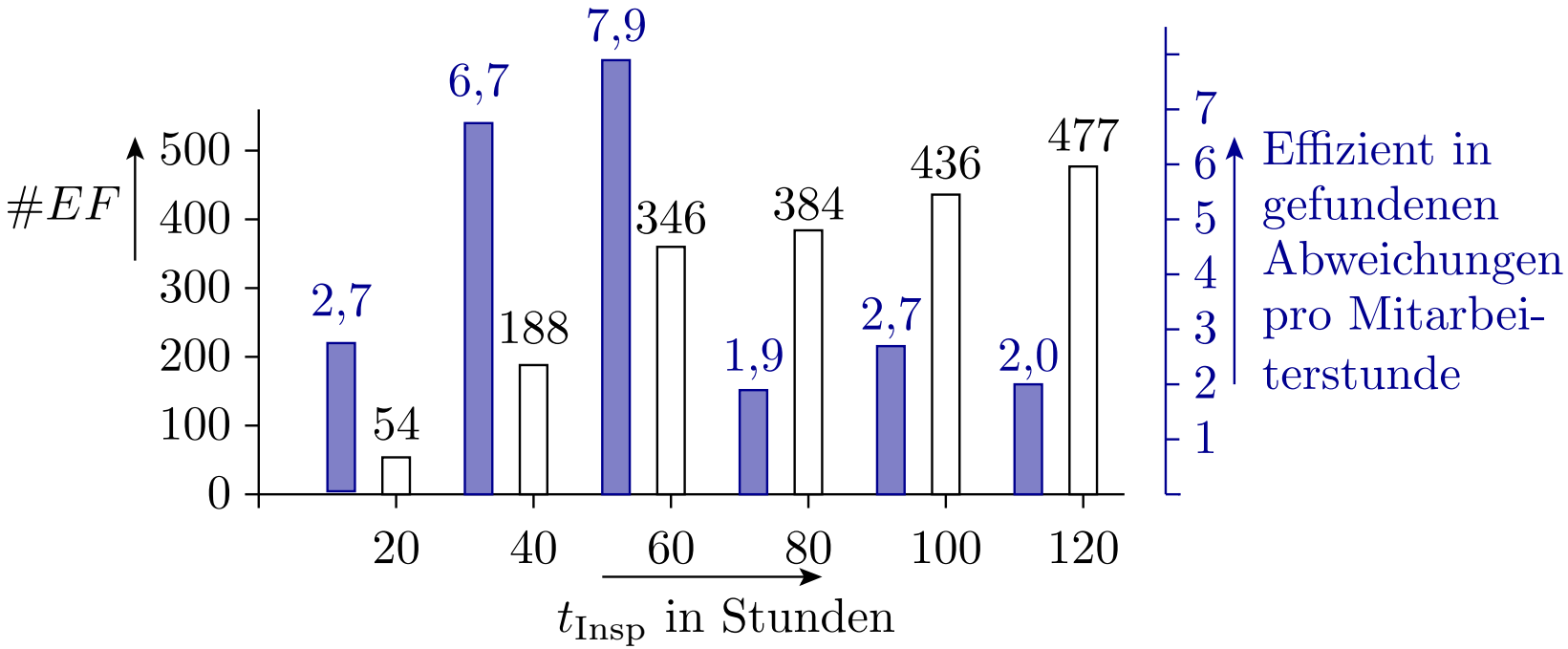
Unterschiedliche Approximationsmöglichkeiten für die weitere Abnahme der zu erwartenden Anzahl der nicht gefundenen Fehler, z.B. Abnahme der zu erwartenden Anzahl nicht gefundenen Fehler mit einem Exponenten $0 < k < 1$ (vergl. TV_F1):

$$\mathbb{E}\left[X\left(t_{\text{Insp}}\right)\right] = \mathbb{E}\left[X\left(t_0\right)\right] \cdot \left(\frac{t_{\text{Insp}}}{t_0}\right)^{-k}$$

(X – Anzahl der nicht nachweisbaren Fehler). Auch dieses Verfahren hat erhebliche Schätzfehler.

Untschiede zwischen Inspektion und Zufallstest

Bei einem Zufallstest nimmt die Effizienz (gefundene Abweichungen pro Mitarbeiterstunde) mit der Testdauer ab, weil nicht erkannte Fehler tendenziell schlechter als erkannte Fehler nachweisbar sind.



Die Beispielinspektion hatte offenbar eine »Anlernphase«, in der die Effizienz mit der Inspektionsdauer zugenommen hat.

- Beim dritten und vierten mal »Lesen des Buchs und der Aufgabentexte« nahm im Experiment nicht nur die Effizienz, sondern auch die Zeit dafür deutlich ab, obwohl ein erheblicher Anteil (ca. 25%) der Fehler noch nicht gefunden war.

Anzahl, wie oft gelesen	1	2	3	4
Anzahl der gefundenen Fehler	251	126	79	4
Zeitaufwand	50 h	70 h		

- Ein Mensch als Inspekteur ermüdet offenbar nach einiger Zeit und wird blind für Fehler, ...

Ein gute Inspektionstechnologie vermeidet die uneffizienten Einarbeitungs- und Ermüdungsphasen.

Inspektionstechniken

- Arbeit »geschickt« auf mehrere Inspekteure mit unterschiedlichen Rollen verteilen.
- Know-How-Weitergabe (Inspekteur ungleich Autor).
- Diversität ausnutzen »Vier Augen sehen mehr als zwei«.

Einteilung der Inspektionstechniken

- Review in Kommentartechnik: Korrekturlesen und Dokument mit Anmerkungen versehen.
- Informales Review in Sitzungstechnik: Lösungsbesprechung in der Gruppe, Vier-Augen-Prinzip. Nimmt die Monotonie, steigert die Aufmerksamkeit, fördert den Wissensaustausch.
- Formales Review in Sitzungstechnik: Festlegen von Rollen (Leser, Moderator, Autor, Inspekteure) und Abläufen, Inspektionstechnologie ...

Syntax, Korrektheit

Syntax-, Typ-, WB- und Korrektheitstests

Kontrollverfahren aus der theoretischen Informatik:

1. Syntaxtest: Kontrolle, dass eine Zeichenfolge ein Wort einer Sprache ist (vergl. FS5, Abschn. Syntax).
2. Typ- und WB-Kontrollen. Wertekontrollen für Konstanten.
3. Kontrollen auf unsichere Programmkonstrukte.
4. Test der partiellen Korrektheit: Nachweis, dass jede Eingabe einer Berechnung, die eine Vorbedingung P erfüllt, auch eine Nachbedingung Q erfüllt, falls die Berechnung terminiert.
5. Test der totalen Korrektheit: zusätzlicher Beweis, dass die Berechnung terminiert.

(1) bis (3) sind die üblichen Kontrollen bei der Eingabe von Programmen und Entwurfsbeschreibungen in den Rechner. Die Fehlererkennungsmöglichkeiten hängen von der Programmiersprache, Zusatzregeln, ... ab.

VHDL[4] – Sprache mit strenger Typkontrolle

VHDL definiert keine Datentypen, sondern Beschreibungsmittel, um Datentypen zu definieren. Beispiel seien die Zahlentypen.

- Ein Zahlentyp ist ein zusammenhängender Zahlenbereich:

```
type <Zahlentyp> is range <Bereich>;
```

- Bereiche können auf- oder absteigend geordnet sein:

```
type tWochentag is range 1 to 7;
type tBitnummer is range 3 downto 0;
```

$\text{Wochentag} \in \{1,2,3,4,5,6,7\}$; $\text{Bitnummer} \in \{3,2,1,0\}$

- ganzzahlige Bereichsgrenzen $\Rightarrow$ diskreter Zahlentyp
- Bereichsgrenzen mit Dezimalpunkt $\Rightarrow$ reeller Zahlentyp

```
type tWahrscheinlichkeit is range 0.0 to 1.0;
```

[4] Hardware-Beschreibungssprache mit für Kontrollen besonders gut geeignetem Typenkonzept.

Kontrollmöglichkeiten

- Kontrollen bei einer Variablenzuweisung

  Variablenname := *Ausdruck*;

  Typenübereinstimmung. Zulässiger Wert des Ausdrucks.

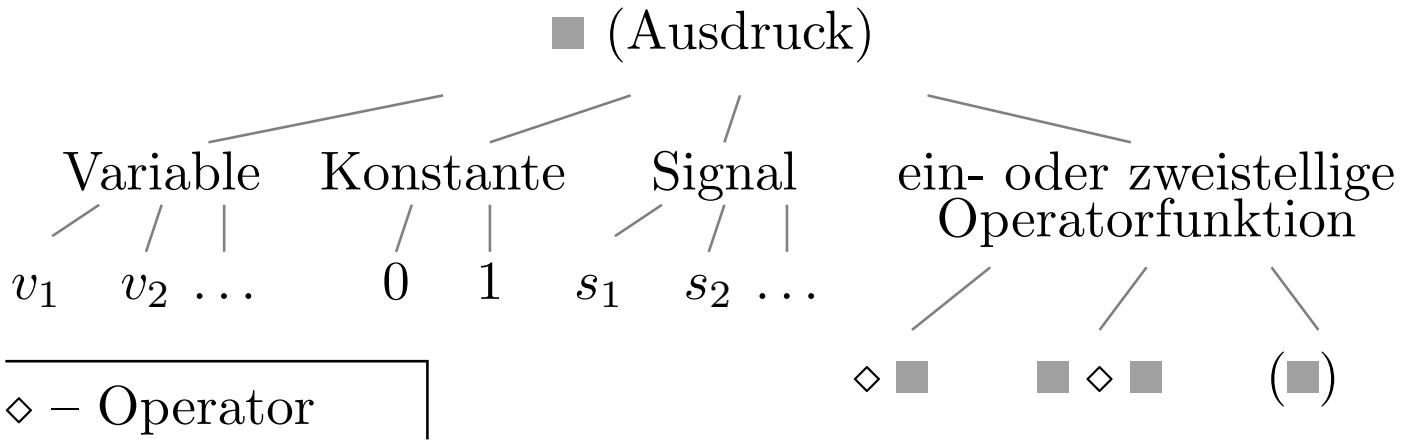
```
variable Wochentag: tWochentag;
variable Bitnummer: tBitnummer;
...
Wochentag := Bitnummer; -- Typ unzulaessig
Wochentag := 9;         -- Wert unzulaessig
```

- Kontrollen bei Signalzuweisungen

  Signalname <= W **[after** t_d**]{,** W **after** t_d**}**;

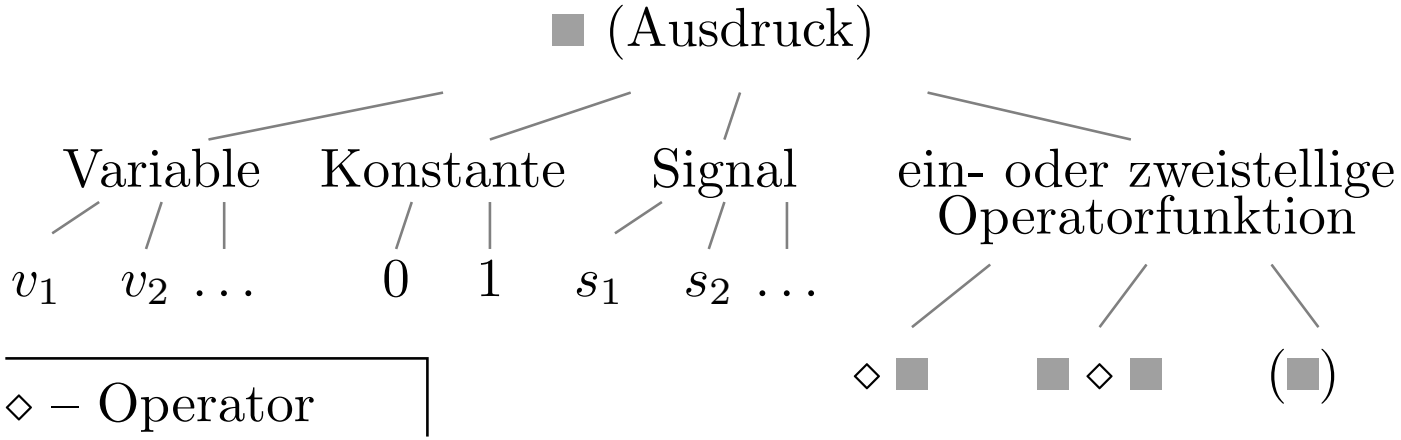
  Gleicher Typ des Ausdruck W und des Signal. Ausdruck t_d muss Typ TIME haben; $t_\mathrm{d} \geq 0$.

Kontrollen in Ausdrücken: Ein Ausdruck ist eine rekursive Beschreibung einer Funktion mit ein- und zweistelligen Operatorfunktionen.



- Jeder Operator ist nur für bestimmte Operandentypen definiert.
- Jeder Kombination aus Operator und Operandentypen ist ein Ergebnistyp zugeordnet[5].
- Aus dem Operanden-WB und dem Operator lässt sich der Ergebnis-WB bestimmen und auf Zulässigkeit prüfen.

[5] In VHDL sind die arithmetischen Operatoren (+, -, *, /) nur identische Operandentypen, z.B tWochentag, definiert.



Eine strenge Typenprüfung erkennt nicht nur, sondern vermeidet auch Fehler, indem sie den Programmierer zwingt, genauer über die beabsichtigte Zielfunktion nachzudenken.

Fehlervermeidung durch Gleitkomma-Darstellung:

- Zahlendarstellung durch Vorzeichen, Mantisse und Exponent,
- Sonderwerte für $\mp\infty$ und NaN (no a number).
- Vermeidung von WB-Überläufen, Minimierung von Rundungsfehlern.
- Preis ist ein höherer Rechenaufwand für arithmetische Operationen, oft in HW ausgelagert.

Statische Code-Analyse

Statische Code-Analyse

Untersuchung des Quellcodes auf Problemquellen:

- fehlerbegünstigend, Verständnis erschwerend,
- Uneindeutigkeiten,
- Programmkonstrukte, die Speicherlecks, Deadlocks, ... begünstigen,
- potentielle Sicherheitsrisiken,
- falsche Benutzung von Betriebssystemschnittstellen, ...

MISRA

Regeln für C-Programme:

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

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

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

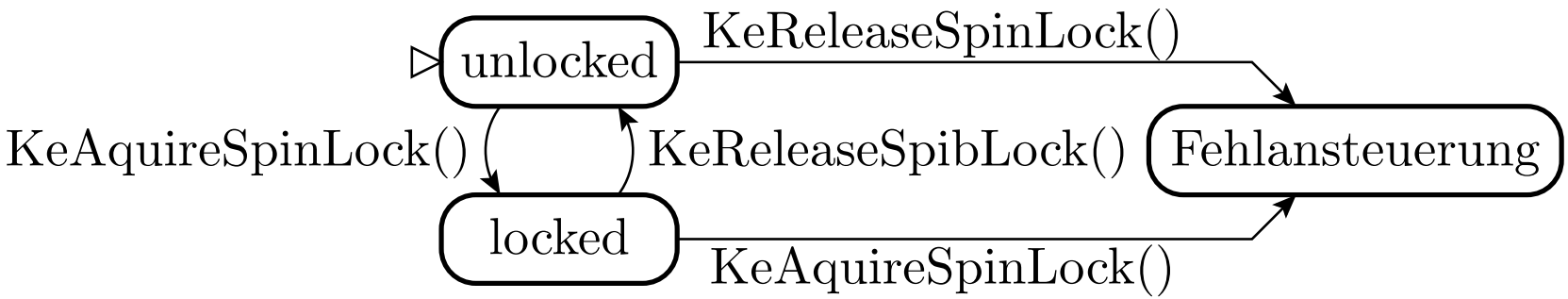
Insgesamt über 100 Regeln, zum Teil verpflichtend, zum Teil Empfehlungen.

API-Benutzungsregeln

Beispielregeln für die Benutzung der Windows-API aus [1]:

spinlock	Spinlocks müssen alternierend reserviert und freigegeben werden.
spinlocksafe	Vermeidung von Deadlocks mit Spinlocks.
criticalregions	Problemvermeidung im Zusammenhang mit der Nutzung kritischer Regionen.
...	

Kontrollautomat für Regel »spinlock«:



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.

```
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();
}
```

Sicherheitslücken

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

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

für Eingabezeichenketten ohne Längenkontrolle. Zu lange Zeichenketten überschreiben nachfolgende Variablen ...

Problemvermeidung durch statische Code-Analyse:

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

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

  mit der zusätzlichen Übergabe der Puffergröße n.

Wie lässt sich durch Überschreiben von Daten hinter einem Zeichenkettenpuffer der Programmablauf manipulieren?

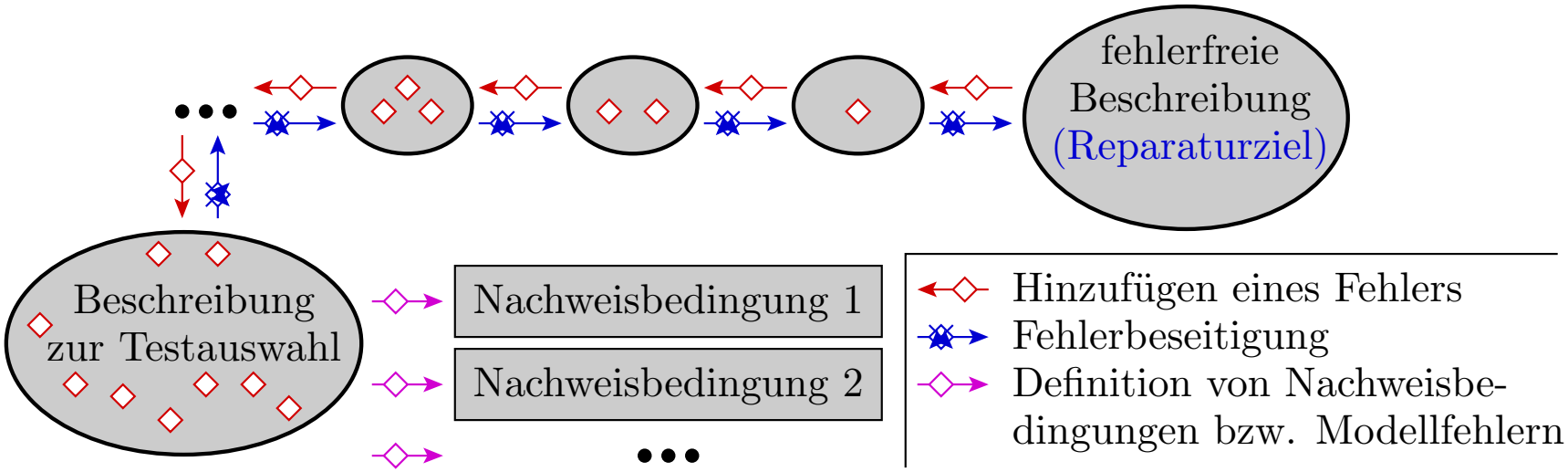
Testauswahl

Fehlermodellierung

Besonderheiten von Software für die Testauswahl

- Fehler entstehen im gesamten Entwurfsprozess (Spezifikation, Architekturentwurf, Programmierung) und auch bei der Fehlerbeseitigung.
- Es existiert in der Regel keine fehlerfreie Sollbeschreibung zur Generierung von Modellfehlermengen und die Bestimmung der Sollwerte für die Tests.
- Agiler Entwurf: Fortlaufende Weiterentwicklung, Fehlerbeseitigung und Fehlerentstehung während der Nutzung.
- Eine geziele Kontrolle und Veränderung beliebiger Zwischenergebnisse zur Fehlereingrenzung oder der Untersuchung einer hypothetischen Fehlerwirkung (außer bei Echtzeittests) unproblematisch.

Mutationen statt Modellfehler



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

Mutationen für Programme

Mutationen auf der Hochsprachenebene sind geringfügige Verfälschungen im Programmtext:

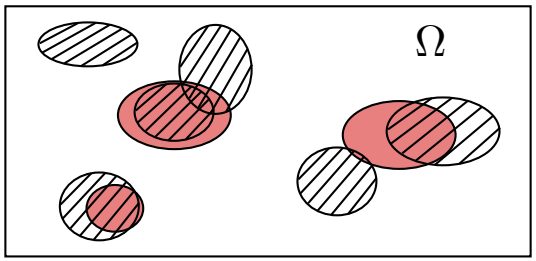
- Verfälschung arithmetischer Ausdrücke (x=a+b ⇒ x=a*b)
- Verfälschung boolescher Ausdrücke (if(a>b){} ⇒ if(a<b){})
- Verfälschung der Wertezuweisung (value=5 ⇒ value=50)
- Verfälschung der Adresszuweisung (ref=obj1 ⇒ ref=obj2)
- Entfernen von Schlüsselworten (static int x=5 ⇒ int x=5)

Bestimmung der Modellfehler- (Mutations-) überdeckung:

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

Kostet viel Rechenzeit, ist aber prinzipiell durchführbar.

Gezielte Testauswahl



Ω Menge der Eingabewerte / Teilfolgen die einen Fehler nachweisen können

Nachweismenge eines Modellfehlers

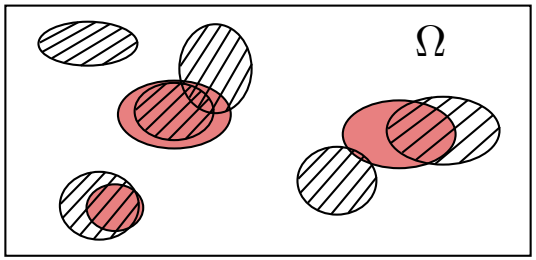
Nachweismenge eines tatsächlichen Fehlers

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

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

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

Zufällige Testauswahl (Fuzzing)



Ω Menge der Eingabewerte / Teilfolgen die einen Fehler nachweisen können

Nachweismenge eines Modellfehlers

Nachweismenge eines tatsächlichen Fehlers

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

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

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

Fehlende Sollfunktion

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

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

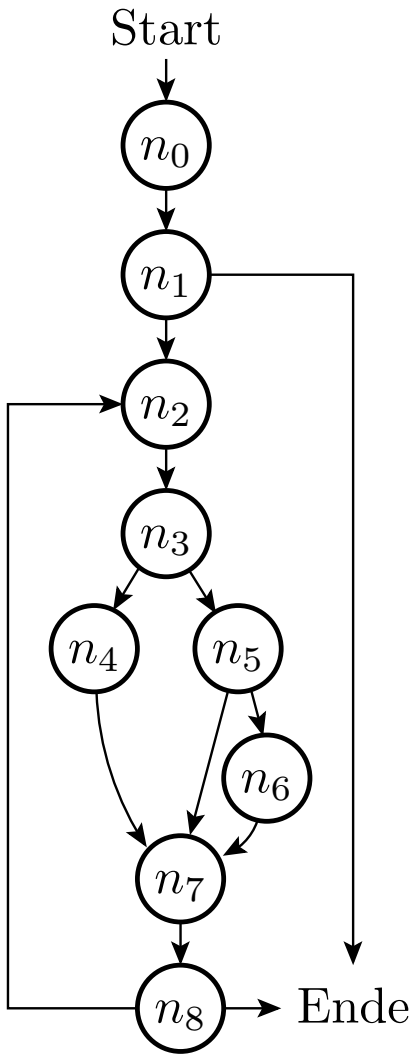
Konzeptionelle Ansätze:

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

Kontrollfluss

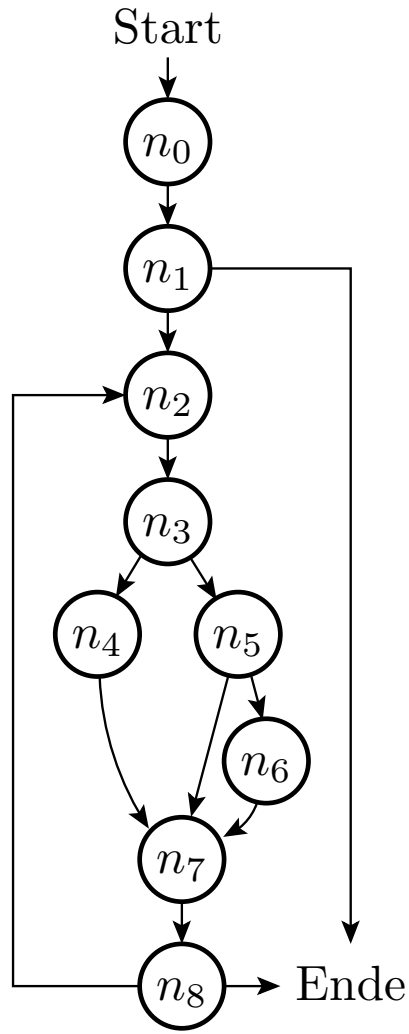
Ein Beispielprogramm und sein Kontrollflussgraph

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



Auswahlkriterien für Tests

1. Anweisungsüberdeckung: Jede Anweisung muss mindestens einmal ausgeführt werden. Beispiel:
   Start, n_0, n_1, n_2, n_3, n_4, n_7, n_8, n_2, n_3, n_5, n_6, n_7, n_8, Ende
2. Kantenüberdeckung: Jede Kante muss mindestens einmal durchlaufen werden. Beispiel:
   Start, n_0, n_1, n_2, n_3, n_4, n_7, n_8, n_2, n_3, n_5, n_6, n_7, n_8, n_2, n_3, n_5, n_7, n_8, Ende
3. Entscheidungsüberdeckung: Jede Entscheidung muss mindestens einmal von jeder Bedingung abhängen.



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

Bestimmung der Überdeckung bei manueller/zufälliger Testauwahl:

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

Kontrolle der Testergebnisse:

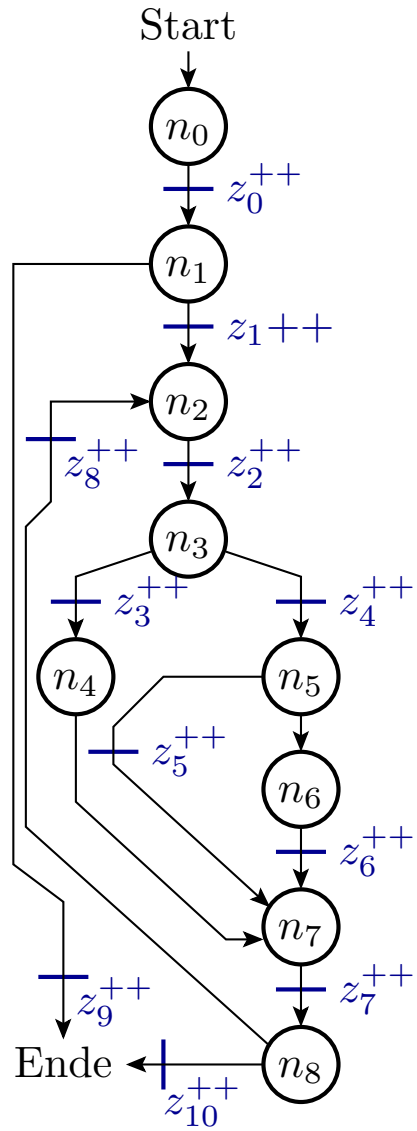
- einprogrammierte Überwachungsfunktionen,
- Trace-Aufzeichnung und Review aufgezeichneter Daten,
- Regressionstest.

Erweiterung der Kontrolle auf Anweisungsergebnisse möglich:

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

Bestimmung der Kantenüberdeckung

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



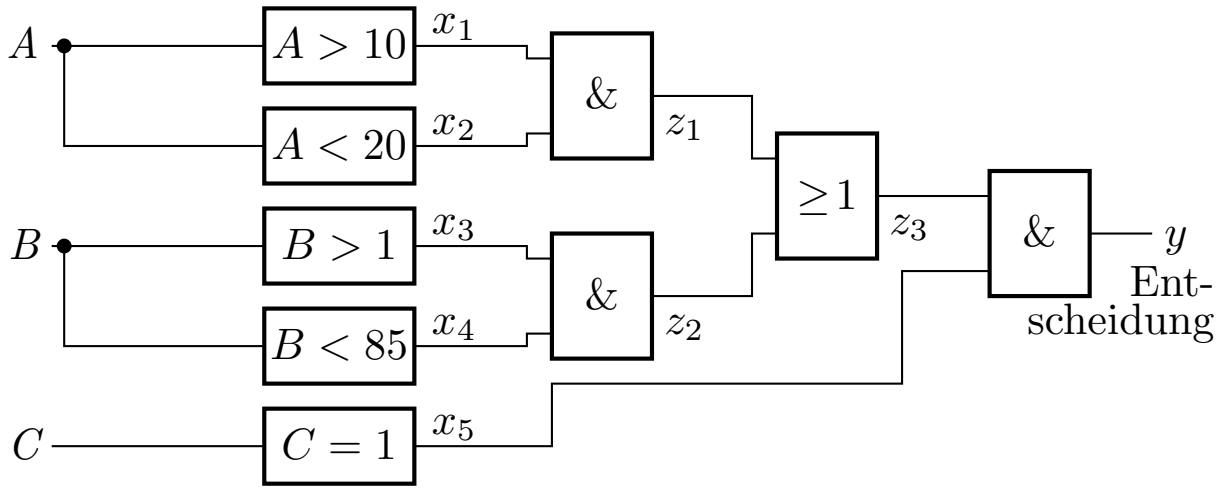
[5] Zur Unterbringung aller Zähler Schleife in Maschinenbefehle auflösen.

Bedingungsüberdeckung

Ein logischer Ausdruck, z.B.

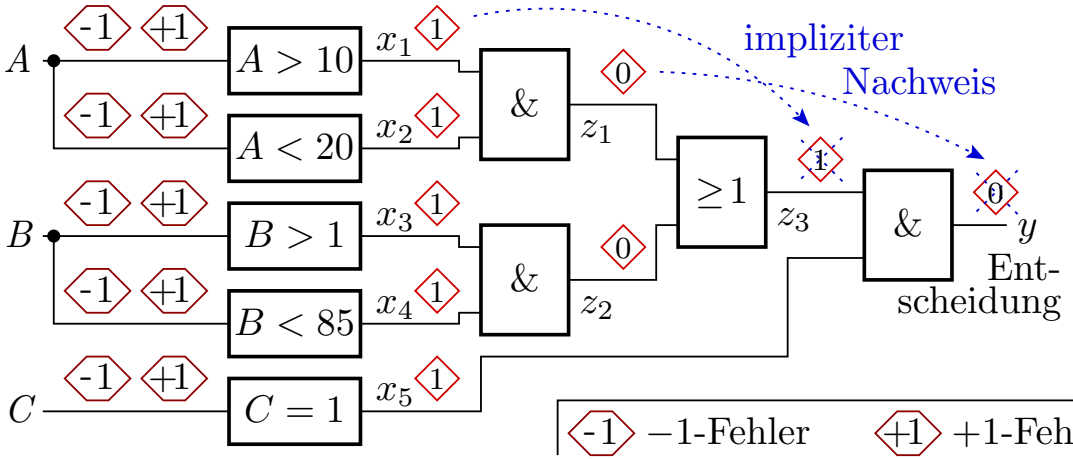
```
n1: if (((A>10) && (A<20)) || ((B>1) && (B<85))
          && (C==1)) {
n2:   ... }
    else {
n3: ... }
```

ist nachbildbar durch einen Schaltplan aus Gattern und Vergleichern:



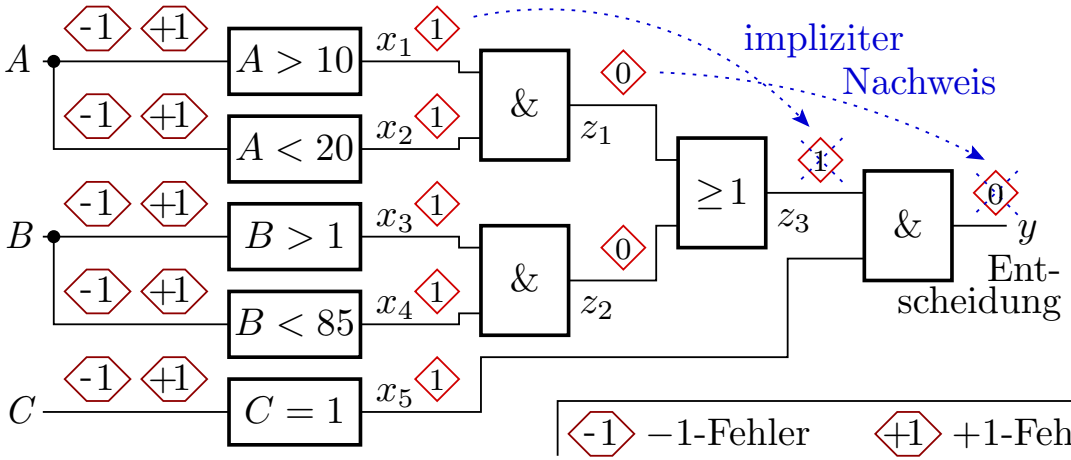
Berechnungsfluss mit eingezeichneten Fehlern

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



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

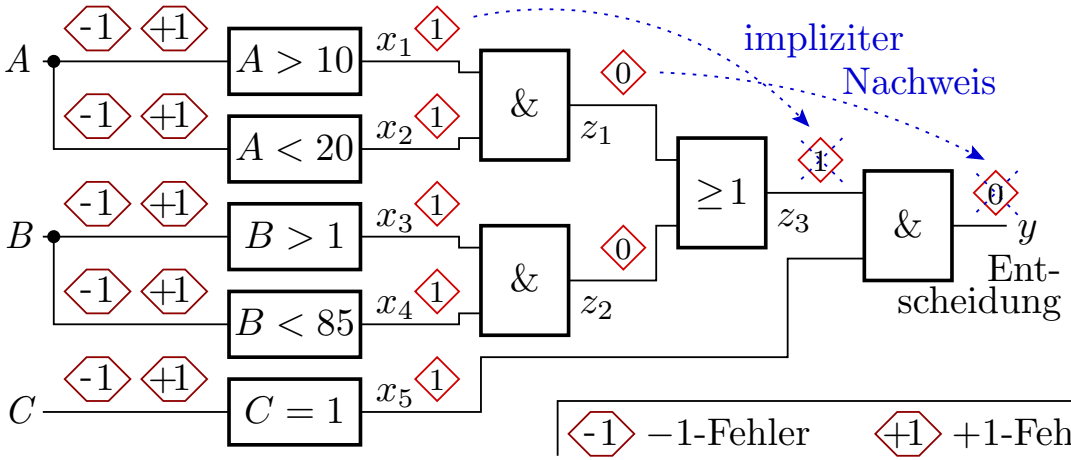
Weiter wie bei Haftfehlern



	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

- Zusammenfassen identischer Fehler. Streichen redundanter und implizit nachweisbarer Fehler (vergl. Foliensatz F2, Abschn. 1.3 und dieser Foliensatz, Abschn. 2.2).
- Die ∓ 1-Fehler implizieren den Nachweis aller Haftfehler nach den Vergleichsoperatoren, ...
- Eventuelle redundante Fehler deuten auf Möglichkeiten zur Programmvereinfachung.

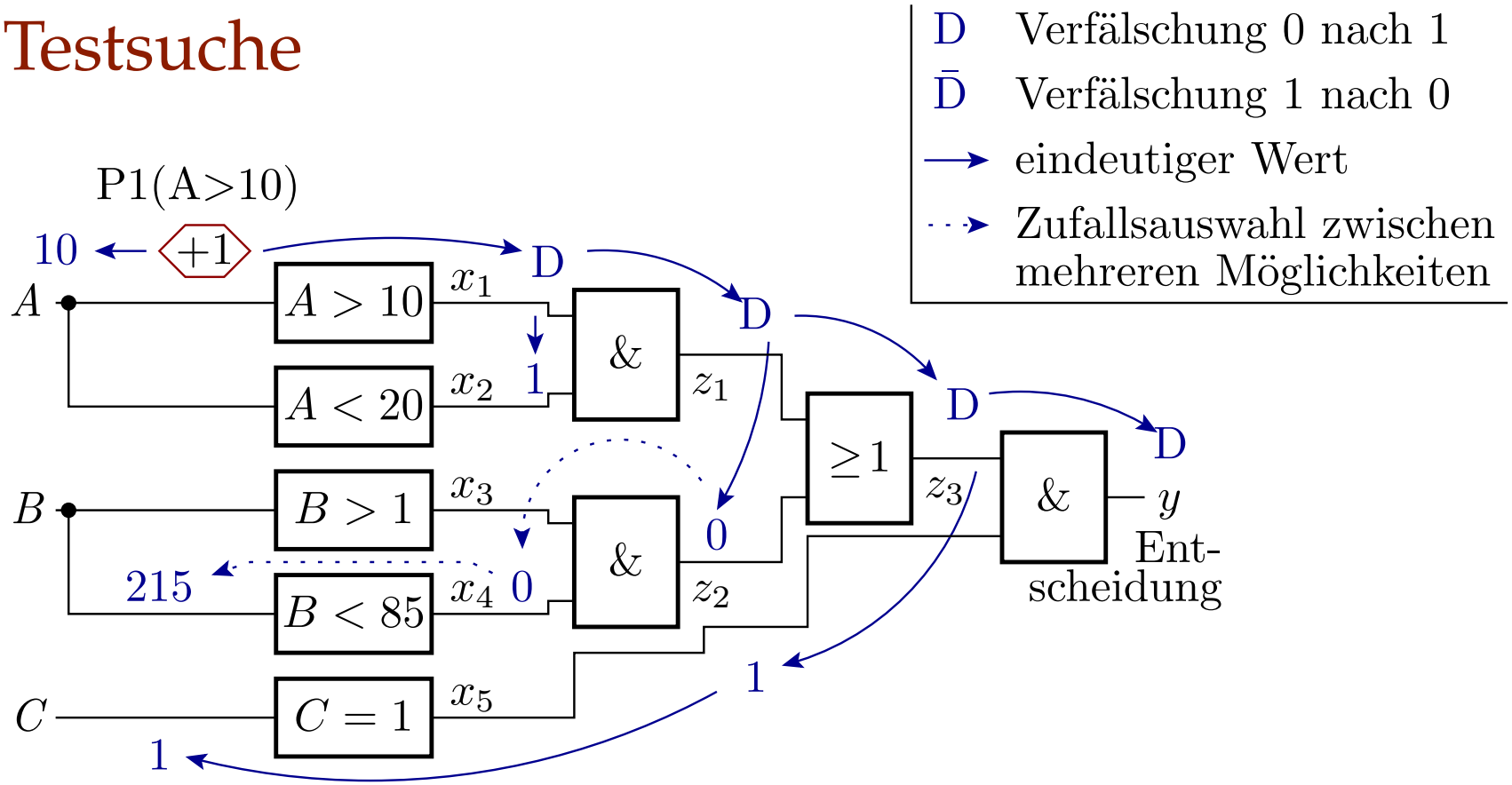
Weiter wie bei Haftfehlern



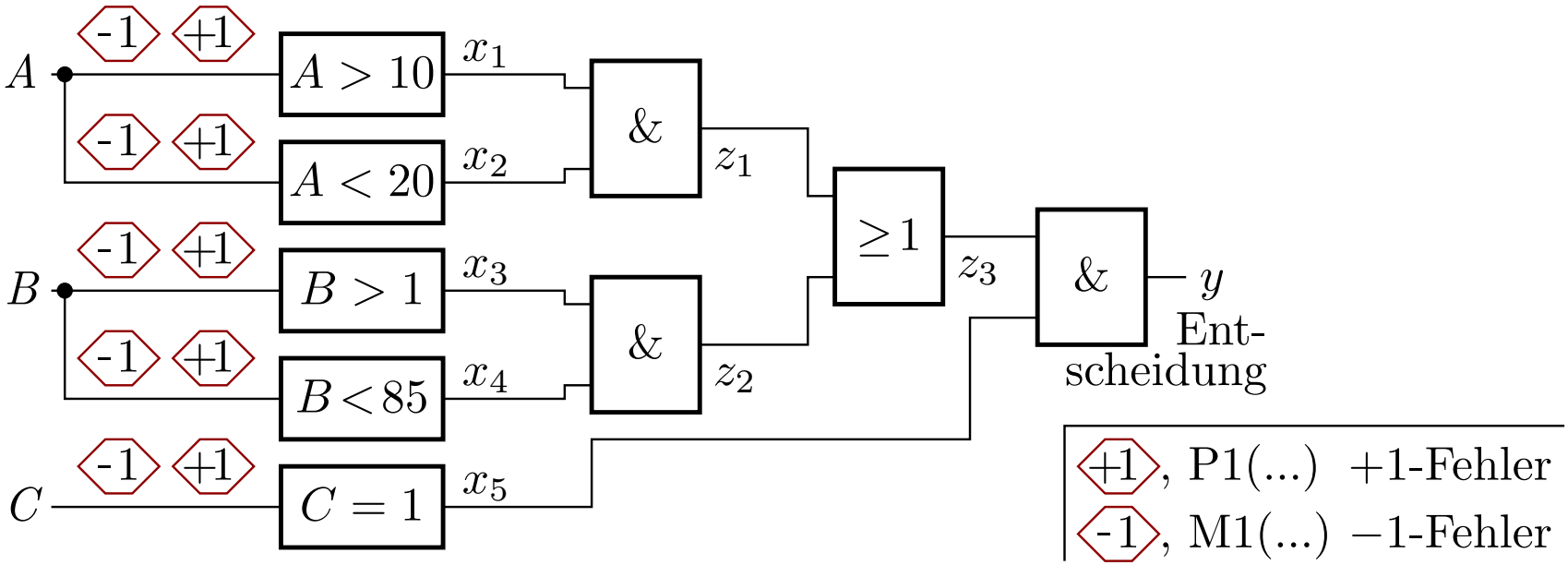
	x_5	x_4	x_3	x_2	x_1	y
sa1(x_1)	1	1	0	1	0	0
sa1(x_2)	1	0	1	0	1	0
sa1(x_3)	1	1	0	1	0	0
sa1(x_4)	1	0	1	0	1	0
sa1(x_5)	0	–	–	1	1	0
sa0(z_1)	1	1	1	1	0	1
sa0(z_2)	1	1	0	1	1	1

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

Testsuche

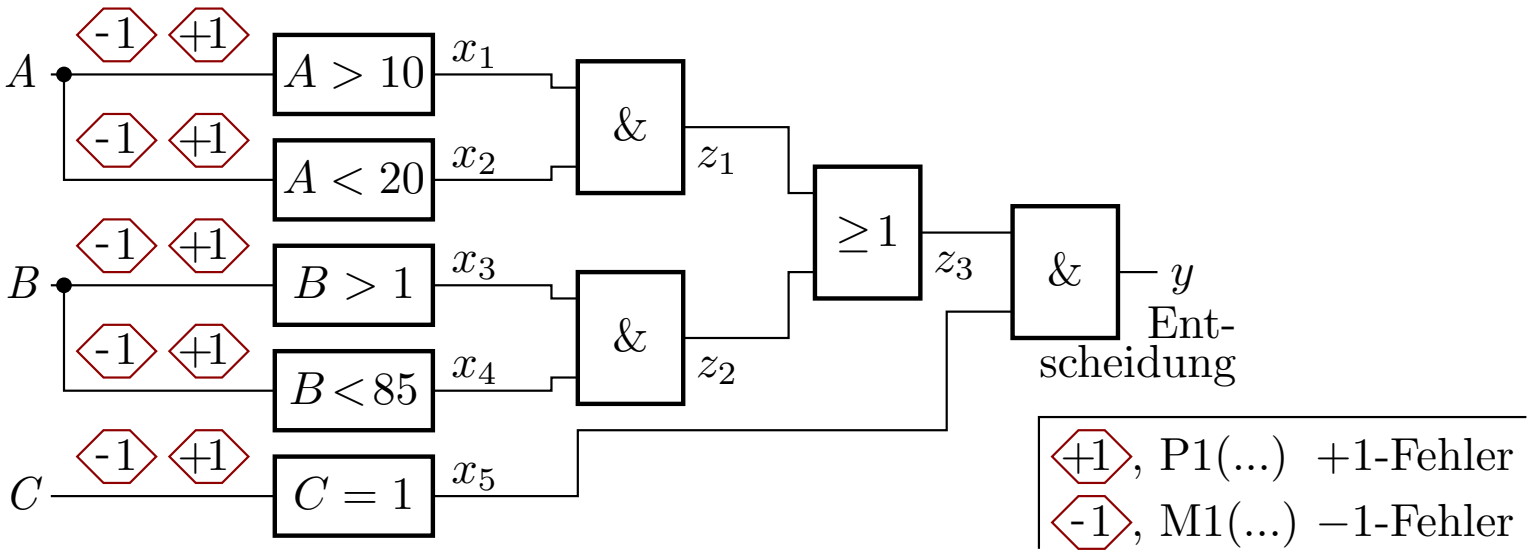


Für den »+1«-Fehler ist am Fehlerort ein Wert einzusetzen, bei dem sich Erhöhnung um eins das Vergleichsergebnis ändert. Vom Vergleichsergebnis wird ein D-Pfad zum Entscheidungsausgang durch zurücktreiben von Steuerpfaden zu Eingängen sensibilisiert.



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

Lösung



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

Testanforderungen für heutige Software

Nach Standard DO-178 B gilt als ausreichend[6]:

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

Nach Stand der Technik noch nicht gefordert:

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

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

Def-Use-Ketten

Def-Use-Ketten

Def-Use-Tupel: Datenstruktur, die aufeinanderfolgende Paare von Schreib- und Lesezugriffen einer Variable beschreibt.
Programmbeispiel »größter gemeinsamer Teiler[7] «:

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

Berechnung und Verwendung von Def-Use-Ketten

Berechnung aller Def-Use-Tupel:

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

Verwendung als Testvollständigkeitskriterien:

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

Statische Code-Analyse:

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

Fehlerlokalisierung:

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

Beispiel: Rückverfolgung in »größter gemeinsamer Teiler«:

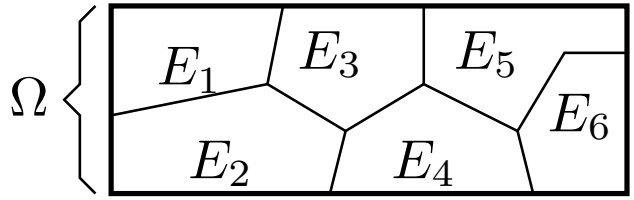
```
int ggt(int a, int b){
n0:   int c = a;
n1:   int d = b;
n2:   if(c == 0)
n3:     return d;
n4:   while(d != 0){
n5:     if(c > d)
n6:       c = c - d;
n7:     else
n8:       d = d - c;
      }
n9:   return c;
    }
```

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

Äquivalenzklassen

Testauswahl mit Äquivalenzklassen

- Äquivalenzklasse: Eingabemenge ähnlich zu verarbeitender Daten.
- Fehlerannahme A: Fehler in der Verarbeitung werden mit jedem Beispiel der Klasse mit hoher Wahrscheinlichkeit nachgewiesen.



Ω Eingaberaum
E_i Äquivalenzklasse

- Fehlerannahme B: Spezifikations- und Implementierungsfehler sind oft falsch gesetzte Bereichsgrenzen.

Testauswahl / Basis:

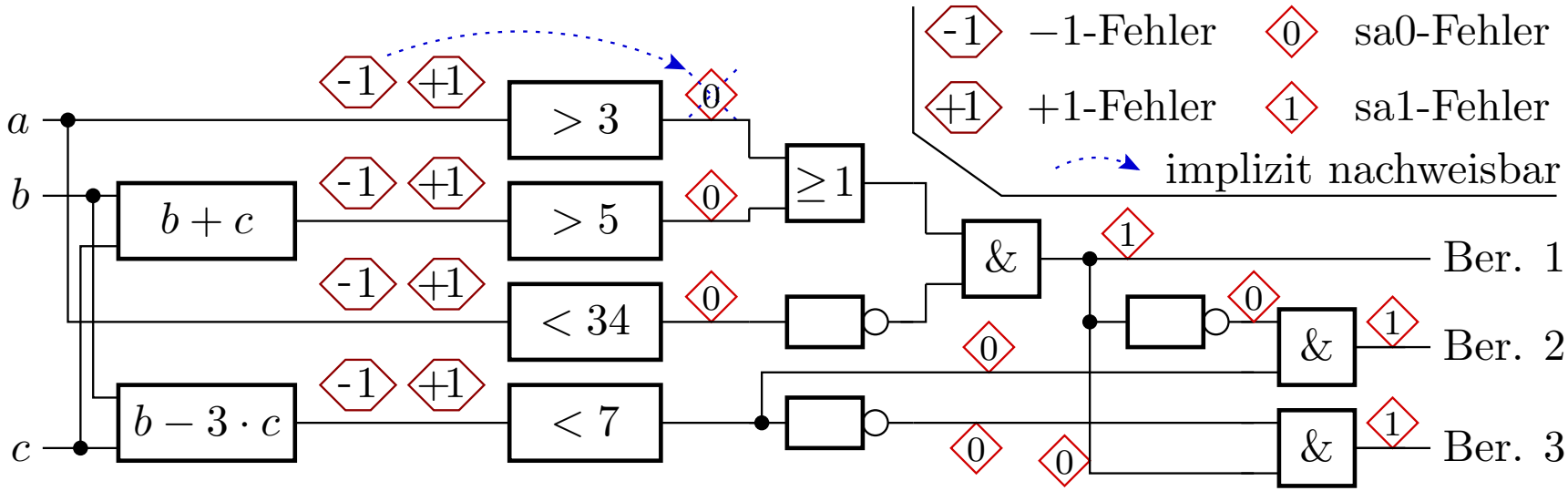
- Fertiges Programm: Testauswahl vergleichbar mit der für »Bedingungsüberdeckung« (siehe Folie 83).
- Spezifikation: Erstellen einer zum Testobjekt diversitären Fallbeschreibung. Weiter wie »Bedingungsüberdeckung«.

Spezifikationsbasierte Testauswahl

1. Zusammenstellung der Eingabedaten, Ausgabedaten, Berechnungsvorschriften und Bedingungen, die bei der Berechnung zu unterscheiden sind.
2. Bildung von Äquivalenzklassen durch Unterteilung der Eingabewertebereiche, beschreibbar durch ein Programm mit Fallunterscheidungen und Dummy-Funktionen für die Ausführung.
3. Konstruktion eines Tests mit 100% Anweisungs-, Zweig- oder Bedingungsüberdeckung für die so entstandene Programmbeschreibung.
4. (Manuelle) Sollwertbestimmung entsprechend Eingabebereich und zugeordneter Sollfunktion.

Beispiel einer aus der Spezifikation gewonnen Äquivalenzklassenbeschreibung

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



Testauswahl / Überdeckungskontrolle weiter wie »Bedingungsüberdeckung« ab Folie 83.

UW-Analyse

Ursache-Wirkungs-Analyse

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

Der empirische Ansatz ist anders.

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

- Auslösern für Aktionen (Ursachen) und
- ausgelösten Aktionen (Wirkungen).

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

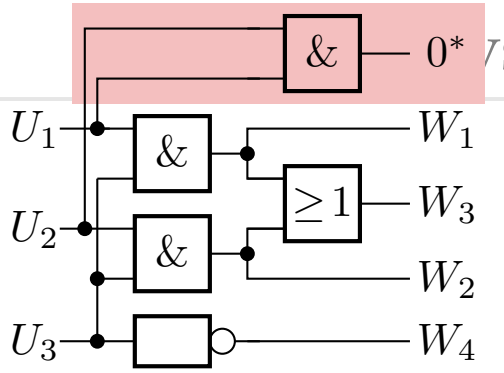
Wirkungen sind einzeln spezifizierte Zielfunktionen, ergänzte selbstverständliche Funktionen und Fehlerbehandlungen.

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

Zwischen den Ursachen und Wirkungen werden logische Verknüpfungen formuliert.

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

Beispiel »Zähle Zeichen«



* Eingabe kann nicht gleichzeitig Typ A und B sein

- Wirkungen:
  W_1: Anzahl_TypA +1[8]
  W_2: Anzahl_TypB +1
  W_3: Gesamtzahl +1
  W_4: Programm beenden
- Ursachen:
  U_1: Zeichen ist vom Typ A
  U_2: Zeichen ist vom Typ B
  U_3: Zeichenanzahl < Maximalwert
- Sich ausschließende Ursachen:
  UND-Verknüpfung muss »0« sein.
- Eine Ursache-Wirkungs-Analyse deckt Mehrdeutigkeiten und Widersprüche in der Spezifikation auf.

Test mit allen einstellbaren Ursachen

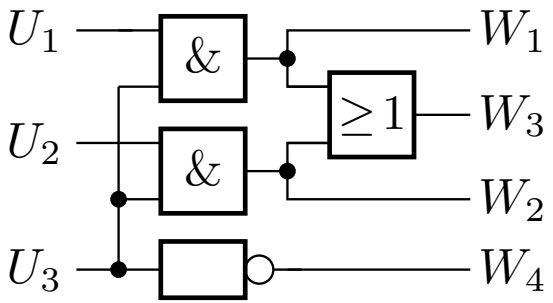
U_1	0	1	0	1	0	1	0	1
U_2	0	0	1	1	0	0	1	1
U_3	0	0	0	0	1	1	1	1
W_1	0	0	0		0	1	0	
W_2	0	0	0		0	0	1	
W_3	0	0	0		0	1	1	
W_4	1	1	1		0	0	0	

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

Beispielimplementierung als C-Funktion

```
      int Ct_A, Ct_B, Ct_N;

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



Test mit allen einstellbaren Ursachen

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

Testbeispiel konkret /symbolisch

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

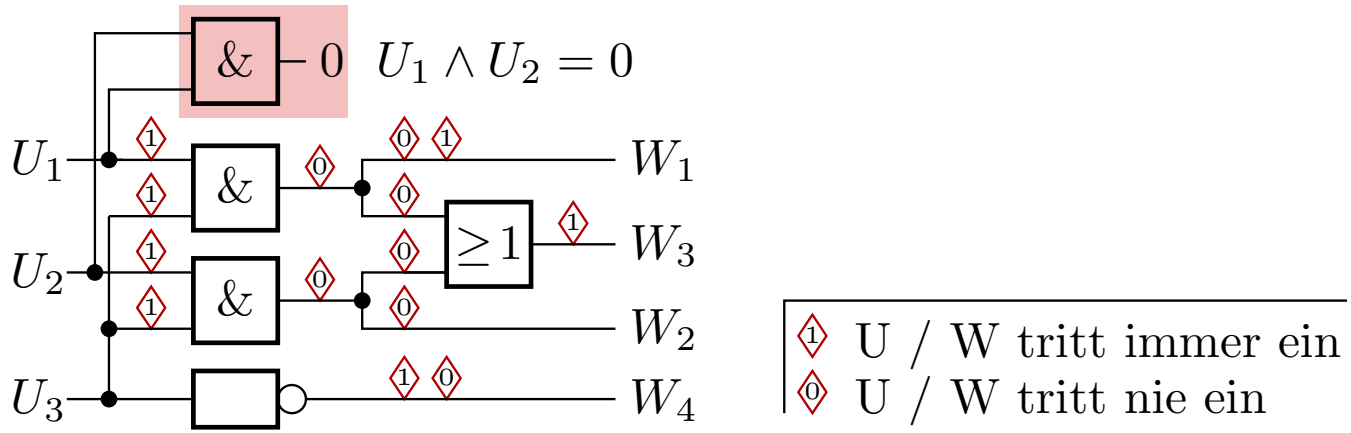
U_1	Zeichen ist vom Typ A (Ziffer)	W_1	Ct_A++
U_2	Zeichen ist vom Typ B (Großbuchstabe)	W_2	Ct_B++
U_3	max. Zählwert nicht erreicht	W_3	Ct_N++
–	es wird kein Zeichen gelesen	W_4	Ende

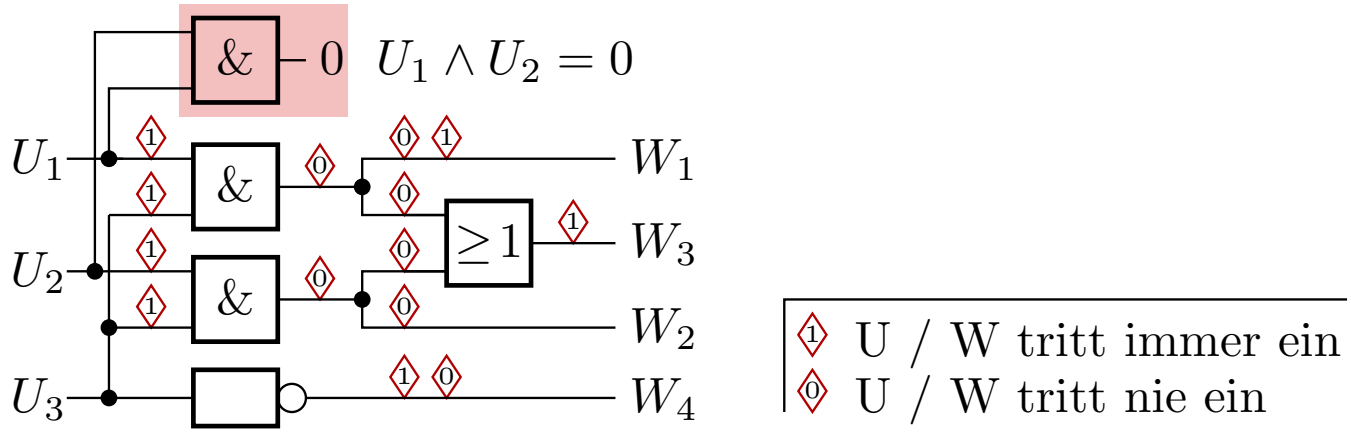
Ungereimtheiten / Haftfehler

Erkennbare Ungereimtheiten:

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

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

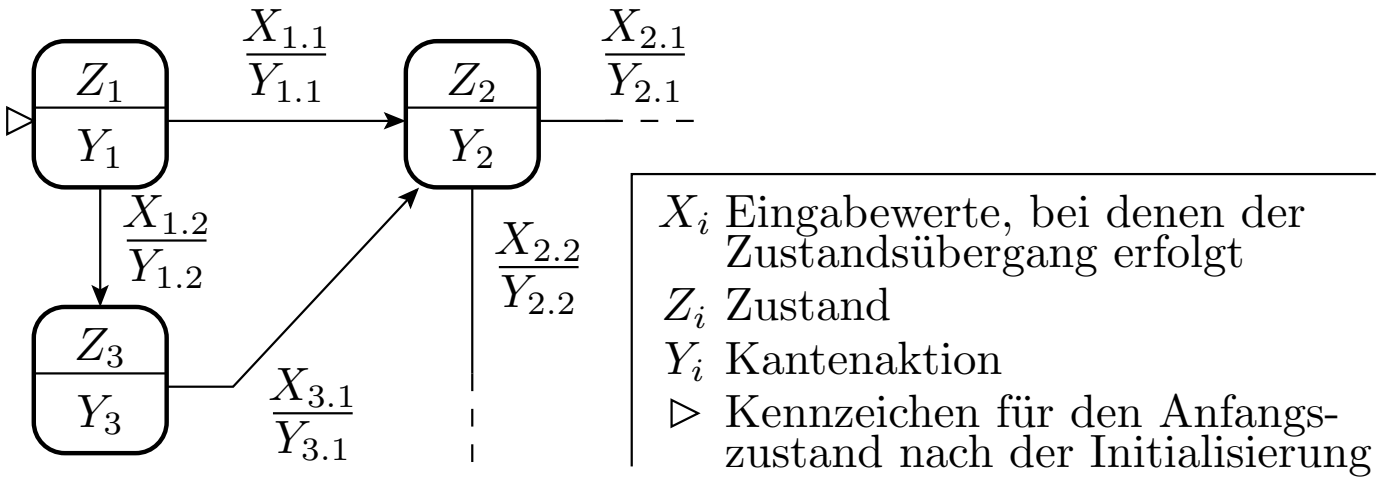




- Im Beispiel würde ein Test mit allen Kombinationen von Ursachen auch alle nachweisbaren Haftfehler erfassen.
- Für eine große Anzahl von Ursachen kann die Anzahl der Haftfehler auch wesentlich kleiner als die Anzahl der Ursachenkombinationen sein.
- Nach Berechnung der gleichzeitig zu (de-) aktivierenden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

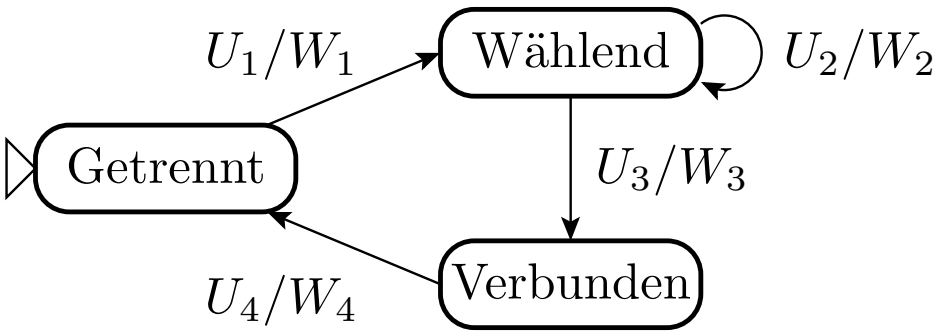
Automaten

Zielfunktion als Automat



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

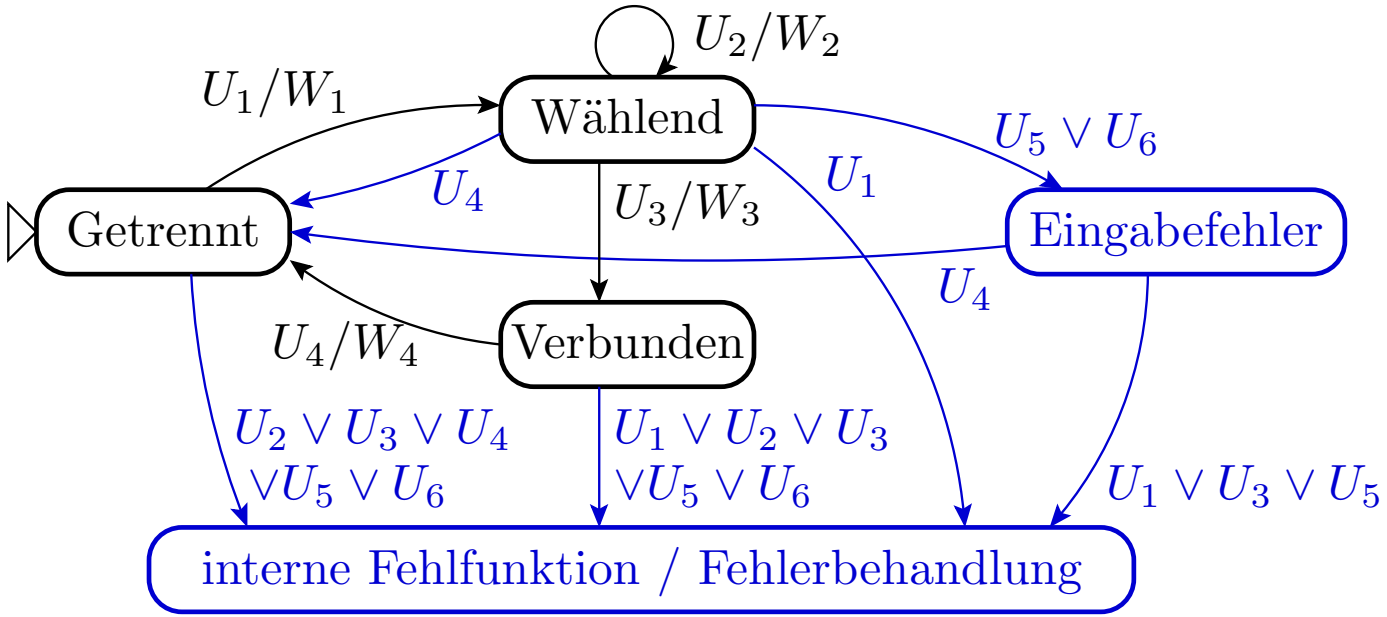
Verbindungsaufbau und -abbau beim Telefonieren



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

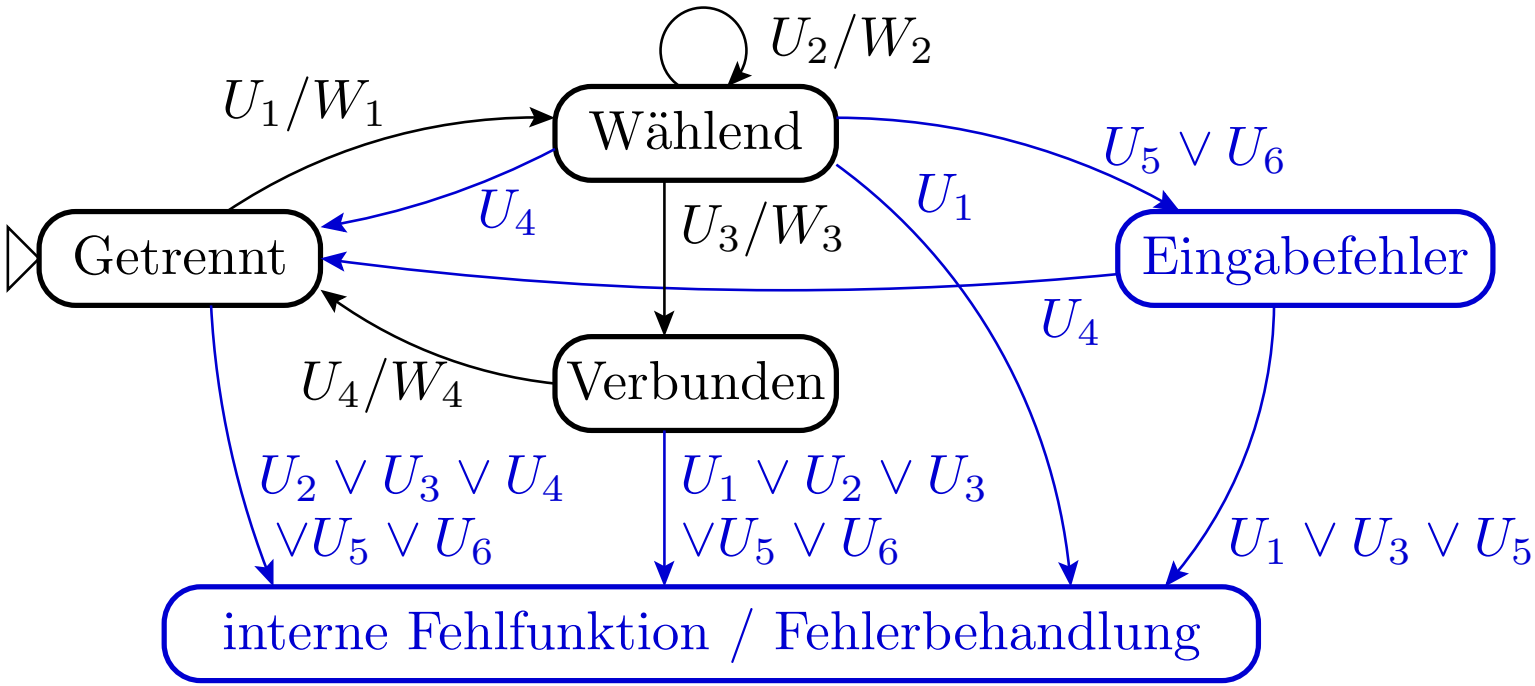
- Test der Sollfunktion: $U_1 \rightarrow U_2 \rightarrow \ldots \rightarrow U_2 \rightarrow U_3 \rightarrow U_4$
- Verhalten für andere Eingabefolgen?
  - Abnehmen, Wählen, Auflegen ($U_1 \rightarrow U_2 \rightarrow U_4$)
  - Abnehmen, Wählen, Wählen, falsche Nummer)
  - ...

  $\Rightarrow$ Ablaufgraph ist noch unvollständig



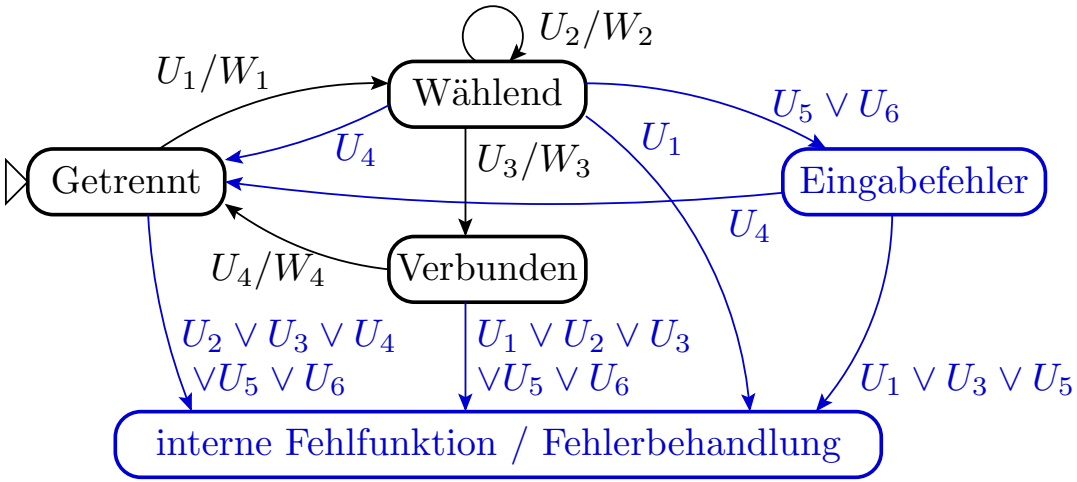
U_1	Abnehmen	W_1	Rufnummer zurücksetzen
U_2	Ziffer wählen	W_2	Ziffer zur Rufnummer hinzufügen
U_3	Rufnummer gültig	W_3	Verbindung aufbauen
U_4	Auflegen	W_4	Verbindung trennen
U_5	Rufnummer ungültig		
U_6	Timeout		

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



Test aller Zustandsübergänge, Wirkungen, ...

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



Test der Reaktion auf interne Fehlfunktionen

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

Auswahlregeln sind wie bei der kontrollflussorientierten Auswahl:

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

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

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

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

[2] Nader B. Ebrahimi.
On the statistical analysis of the number of errors remaining in a software design document after inspection.
IEEE Transactions on Software Engineering, 23(8):529–532, 1997.

[3] Günter Kemnitz.
Technische Informatik 2: Entwurf digitaler Schaltungen.
Springer, 2011.

[4] Peter Liggesmeyer.
Software-Qualität: Testen, Analysieren und Verifizieren von Software.
Spectrum, 2002.

[5] Frank Padberg, Thomas Ragg, and Ralf Schoknecht.
Using machine learning for estimating the defect content after an inspection.
IEEE Transactions on Software Engineering, 30(1):17–28, 2004.

[6] Qinbao Song, Martin Shepperd, Michelle Cartwright, and Carolyn Mair.
Software defect association mining and defect correction effort prediction.
IEEE Transactions on Software Engineering, 32(2):69–82, 2006.