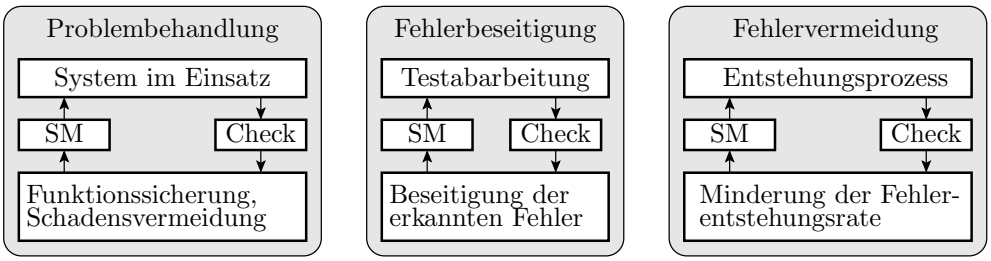
Test und Verlässlichkeit 5: Mehr zu Tests und Kontrollen

Prof. G. Kemnitz

6. Oktober 2025

Inhaltsverzeichnis

5.2 Gefährdungsabwendung (Folie 1.11)



Verlässlichkeit wird durch Problembeseitigung auf drei Ebenen gesichert:

- Überwachung und Problembehandlung während der Nutzung.
- Test und Fehlerbeseitigung vor und während der Nutzung.
- Fehlervermeidung durch Fehlerbeseitigung in den Entstehungsprozessen.

Mit der unterstellten Fehlerkultur, dass erkannte Probleme beseitigt werden, entscheiden die **Tests und Kontrollen** über die Verlässlichkeit.

5.3 Dieser Foliensatz

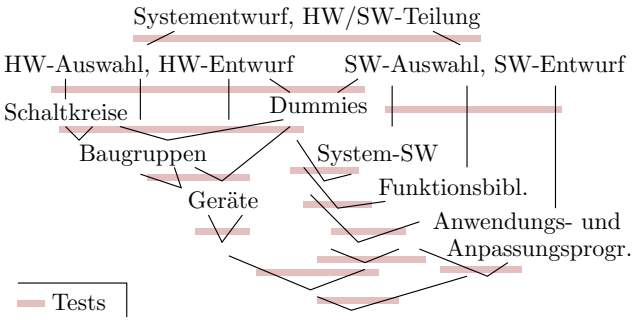
... beschäftigt sich weiter mit Tests und Kontrollen, speziell

- Inspektion: Gütemaße, Inspektionstechniken

- Dynamische Tests: Durchführung, Systemgestaltung, prüfgerechter Entwurf.
- Codebasierte Kontrollen: Fehlererkennende und fehlerkorrigierende Codes, Prüfkennzeichen, RAIDs.
- Kontrolle von Berechnungsergebnissen, insbesondere automatisiert generierbar Kontrollen für Testausgaben, und
- Syntax nicht nur für die Generierung von Kontrollautomaten, sondern auch für die Generierung von Würfelfunktionen für Testeingaben.

1 Inspektion

5.4 Vielfalt der Test (Abschn. 2.1.2)



In Entstehungsprozessen für IT-Systeme erfolgen in der Regel eine Vielzahl verschiedener Test:

- während und nach jeder Entwurfsphase,
- hierarchisch aufsteigend Teilsysteme dann Zusammenwirken, ...

In den ersten Entwurfsphasen Inspektion entstandener Dokumente.

5.5 Inspektion (Review)

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

- Dokumentationen (Spezifikation, Nutzerdokumentation, ...),
- Programmcode, Testausgaben,
- Schaltungsbeschreibungen, Konstruktionspläne, ...
- Auch Testausgaben, bevor Sollwerte festgelegt oder Kontrollen programmiert sind.

Eigenenschaften von Kontrollen durch Inspektion:

- Für alle aufgezeichneten Ergebnisse (Entwürfe, Daten, ...) nutzbar.
- Großer manueller Arbeitsaufwand.
- Geringere Güte als automatisierte Kontrollen.

Positive Zusatzeffkte:

- Nachweis auch nicht funktionaler Fehler,
- Know-How-Weitergabe.

Nicht funktionale Fehler (Antipattern): Beschreibungselemente, die die Übersichtlichkeit beeinträchtigen, die Kontrolle erschweren und die Fehlerentstehung begünstigen.

1.1 Kenngrößen

5.6 Kenngrößen einer Inspektion

Kenngrößen wie bei jedem anderen Test:

- Fehlerabdeckung:

  (2.1) $$FC = \frac{\#DF}{\#F}\Big|_{\text{ACR}}$$

- Phantomfehlerrate:

  (2.2) $$\zeta_{\text{PF}} = \frac{\#PM}{N}\Big|_{\text{ACR}}$$

Weitere Kenngrößen zur Bewertung von Inspektionsprozessen [3]:

- Effizienz (EFC): Gefundene Fehler pro Mitarbeiterstunde.
- Effektivität (EFT): Gefundene Fehler je 1000 NLOC.

Kenngrößenschätzung getrennt für funktionale und andere Fehler.

FC Fehlerabdeckung (fault coverage), Anteil der nachweisbaren Fehler.

$\#F$, $\#DF$ Fehleranzahl, Anzahl der davon nachweisbaren Fehler.
ζ_{PF} Phantomfehlerrate des Tests.
N, $\#PM$ Testanzahl, Anzahl der Phantomfehler.
ACR Geeignete Zählwertgrößen, typ. 100 ... 1000 ein- und nicht eingetretene Zählereignisse.
EFC Effizienz, gefundene Fehler pro Mitarbeiterstunde.
EFT Effektivität, gefundenen Fehler je 1000 NLOC.
NLOC Netto Lines of Code, Anzahl der Code-Zeilen ohne Kommentar und Leerzeilen.

Beispiel 5.1 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:

a) *Fehlerabdeckung FC?*

b) *Effizienz und Effektivität?*

	gesamt	funktionale Fehler	sonstige Fehler
$FC = \frac{\#DF}{\#F}$	$\frac{228}{300}$	$\frac{156}{200}$	$\frac{72}{100}$
Effizienz (EFC)	$\frac{228\text{ Fehler}}{200\text{ h}}$	$\frac{156\text{ Fehler}}{200\text{ h}}$	$\frac{72\text{ Fehler}}{200\text{ h}}$
Effektivität (EFT)	$\frac{228\text{ Fehler}}{10.000\text{ NLOC}}$	$\frac{156\text{ Fehler}}{10.000\text{ NLOC}}$	$\frac{72\text{ Fehler}}{10.000\text{ NLOC}}$

- Effizienz und Effektivität bewerten den Inspektionsprozess und berechnen sich aus tatsächlich zählbaren Werten.
- Die Fehlerabdeckung FC hängt von der nicht zähl-, sondern nur abschätzbaren Gesamtfehleranzahl $\#F$ ab.

Wie lässt sich die nicht zählbare Gesamtfehleranzahl schätzen?

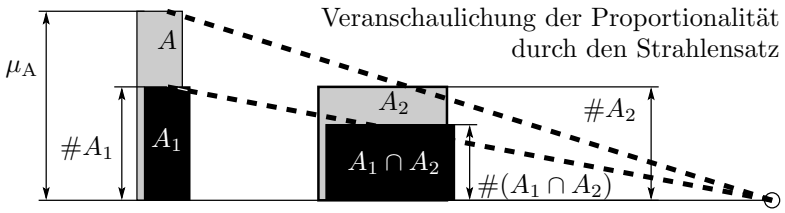
NLOC Netto Lines of Code, Anzahl der Code-Zeilen ohne Kommentar und Leerzeilen.

5.8 Capture-Recapture

Abgeleitet von einem Schätzer für die Größe von Tierpopulationen (z.B. von Vögeln in einem Gebiet) [2, 5, 4].

- Aus einer Menge A unbekannter Größe wird eine Menge A_1 von Tieren eingefangen, gekennzeichnet und freigelassen.
- Nach Vermischung der Population wird eine Menge A_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:



$$\frac{\hat{\mu}_A}{\#A_1} = \left.\frac{\#A_2}{\#(A_1 \cap A_2)}\right|_{ACR}$$

Zu erwartende Größe der Tierpopulation:

$$\hat{\mu}_A = \left.\frac{\#A_1 \cdot \#A_2}{\#(A_1 \cap A_2)}\right|_{ACR}$$

$\hat{\mu}_A$	Zu erwartende Anzahl aller Tiere.
A_1, A_2	Menge der beim ersten bzw. zweiten mal eingefangenen Tiere.
$A_1 \cap A_2$	Menge der Tiere, die beide Male eingefangen werden.
$\#...$	Anzahl der Elemente der Mengen.
ACR	Brauchbare Schätzwerte nur bei geeigneten Zählwertgrößen.

5.9 Fehler statt Tiere

Zwei Inspekteure $i \in \{1, 2\}$ finden jeweils eine Menge von F_i Fehlern, drunter $F_1 \cap F_2$ gleiche Fehler:

$$\hat{\mu}_F = \left.\frac{\#F_1 \cdot \#F_2}{\#(F_1 \cap F_2)}\right|_{ACR} \tag{5.1}$$

Die zu erwartende Fehlerabdeckung ist das Verhältnis der Anzahl der insgesamt von beiden Inspekteuren gefundenen Fehler $\#(F_1 \cup F_2)$ zur zu erwartenden Geamtfehleranzahl μ_F:

$$\hat{\mu}_{FC} = \left.\frac{\#(F_1 \cup F_2)}{\hat{\mu}_F}\right|_{ACR} = \left.\frac{\#(F_1 \cap F_2) \cdot \#(F_1 \cup F_2)}{\#F_1 \cdot \#F_2}\right|_{ACR} \tag{5.2}$$

$\hat{\mu}_F$	Geschätzter Erwartungswerte der Gesamtfehleranzahl.
$\#F_1, \#F_2$	Anzahl der von Inspekteur 1 bzw. Inspekteur 2 gefundenen Fehler.
$\#(F_1 \cap F_2)$	Anzahl von beiden Inspekteuren gefundenen Fehler.
$\hat{\mu}_{FC}$	Schätzwert der zu erwartenden Inspektionsfehlerabdeckung.

Beispiel 5.2 Capture-Recapture

- Inspekteur 1: 228 gefundene Fehler.
- Inspekteur 2: 237 gefundene Fehler.
- Übereinstimmend: 105 Fehler.

Wie groß ist die zu erwartende Gesamtfehleranzahl und die Inspektionsfehlerabdeckung?

(5.1)

(5.2)

Schätzwert der zu erwarte

Schätzwert der zu erwarte

$\hat{\mu}_F$ Geschätzter Erwartungswerte der Gesamtfehleranzahl.

$\hat{\mu}_{FC}$ Schätzwert der zu erwartenden Inspektionsfehlerabdeckung.

$\#F_1, \#F_2$ Anzahl der von Inspekteur 1 bzw. Inspekteur 2 gefundenen Fehler.

$\#(F_1 \cap F_2)$ Anzahl von beiden Inspekteuren gefundenen Fehler.

5.11 Schätzgenauigkeit

Die zu erwartende Gesamtfehleranzahl nach (Gl. 5.1) ist das Produkt zweier Zählwerte geteilt durch einen dritten. Für grobe Abschätzungen addieren sich bei Multiplikation und Division die Quadrate der Varianzkoeffizienten. Das bedeutet, dass die drei Zählwerte für dieselbe Schätzgenauigkeit dreimal so groß wie Zählwerte zur Schätzung von Eintrittswahrscheinlichkeiten nach (Gl. 4.74) und (Gl. 4.75) sein müssen:

$$x_{AV} \geq 3 \cdot \frac{\kappa \cdot \left(\Phi^{-1}\left(1-\frac{\alpha}{2}\right)\right)^2}{\varepsilon_r^2} \cdot (1-\hat{p}) \text{ für } \hat{p} \leq 50\%$$

$$n - x_{AV} \geq 3 \cdot \frac{\kappa \cdot \left(\Phi^{-1}\left(1-\frac{\alpha}{2}\right)\right)^2}{\varepsilon_{\bar{r}}^2} \cdot \hat{p} \text{ für } \hat{p} > 50\%$$

Zählwerte um hundert reichen für Schätzgenauigkeiten von $\mp 30\%$ der zu erwartenden Fehleranzahl und der Nichteintrittswahrscheinlichkeit.

ε_r, $\varepsilon_{\bar{r}}$ Intervallradius realtiv zum erwarteten Eintritts- bzw. Nichteintritts-Zählwert.

κ Varianzerhöhung durch Abhängigkeiten, für unabhängige Zählwerte $\kappa \leq 1$.

$\Phi^{-1}(..)$ Inverse Funktion zur Verteilungsfunktion der standardisierten Normalverteilung.

n, x_{AV} Anzahl der Zählversuche, Experimentell bestimmter Ist-Zählwert.

$\hat{p}$ Schätzwert der Eintrittswahrscheinlichkeit.

5.12 Systematische Schätzfehler

Zu den zufälligen kommen systematische Schätzfehler:

- Capture-Recaptur unterstellt für alle Fehler dieselbe »Einfangwahrscheinlichkeit«. In der Praxis reicht diese aber von *»Fehler kaum übersehbar«* bis *»Fehler fast nicht erkennbar«*.
- Capture-Recaptur verbietet Informationsaustausch zwischen den Inspekteuren. Falls es doch einen Informationsaustausch gibt, vergrößert der die Menge der von beiden Inspekteuren gefundenen Fehler $F_1 \cap F_2$ gegenüber einer unabhängigen Suche.
- Wenn die Inspekteure ihre Fehlerlisten voneinander abschreiben

$$\#F_1 = \#F_2 = \#\left(F_1 \cap F_2\right) = \#\left(F_1 \cup F_2\right)$$

$$\hat{\mu}_{FC} = \frac{\#(F_1 \cap F_2) \cdot \#(F_1 \cup F_2)}{\#F_1 \cdot \#F_2} = 1$$

  Ein völlig unsinniger Schätzwert.

$\#F_1, \#F_2$ Anzahl der von Inspekteur 1 bzw. Inspekteur 2 gefundenen Fehler.

$\#(F_1 \cap F_2)$ Anzahl von beiden Inspekteuren gefundenen Fehler.

$\hat{\mu}_{FC}$ Schätzwert der zu erwartenden Inspektionsfehlerabdeckung.

5.13 Kontrollfehler

Einbau von Kontrollfehlern (Mutationen) in das zu inspizierende Datenmaterial und Abschätzung der zu erwartenden Fehlerabdeckung aus dem Anteil der gefundenen Kontrollfehler. Wie bei einem Zufallstests tendiert die zu erwartende Modellfehlerabdeckung gegen die Fehlerabdeckung des c_{MF}-fachen Inspektionsaufwands (Gl. 2.42):

$$\mu_{\mathrm{FC}}(t) = \mu_{\mathrm{FCM}}(c_{\mathrm{MF}} \cdot t)$$

Mit typisch zu findenen Fehlern als Kontrollfehler (vergleichbare mittlere Fehlernachweiszeit) $c_{\mathrm{MF}} \approx 1$:

$$\hat{\mu}_{\mathrm{FC}} = \left.\frac{\#F_{\mathrm{DM}}}{\#F_{\mathrm{M}}}\right|_{\mathrm{ACR}} \tag{5.3}$$

Geschätzte Gesamtfehleranzahl:

$$\hat{\mu}_{\mathrm{F}} = \left.\frac{\#F_{\mathrm{D}}}{\hat{\mu}_{\mathrm{FC}}}\right|_{\mathrm{ACR}} = \#F_{\mathrm{D}} \cdot \left.\frac{\#F_{\mathrm{M}}}{\#F_{\mathrm{DM}}}\right|_{\mathrm{ACR}} \tag{5.4}$$

μ_{FCM}	Zu erwartende Mutationsabdeckung.
μ_{FC}	Zu erwartende Fehlerabdeckung.
c_{MF}, t	Mutationsspezifische Skalierung des Inspektionszeit, Inspektionszeit.
$\#F_{\mathrm{M}}$, $\#F_{\mathrm{DM}}$	Anzahl der untersuchten Mutationen, Anzahl der davon erkannten Mutationen.
μ_{F}, $\#F_{\mathrm{D}}$	Zu erwartende Gesamtfehleranzahl, Anzahl der erkannten Fehler.

5.14 Vergleich mit Capture-Recapture

Zufällige Schätzfehler:

- Für $\hat{p} = \hat{\mu}_{\mathrm{FC}}$ nur ein zufälliger Zählwert, erforderlicher Zählwert:

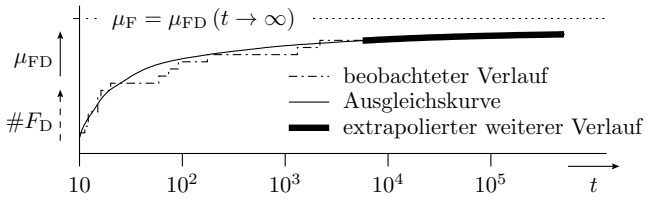
(4.74)
$$x_{\mathrm{AV}} \geq \kappa \cdot \left(\Phi^{-1}\left(1-\tfrac{\alpha}{2}\right)\right)^2 \cdot \varepsilon_{\mathrm{r}}^{-2} \cdot (1-\hat{p}) \quad \text{für } \hat{p} \leq 0{,}5 \tag{4.75}$$
$$n - x_{\mathrm{AV}} \geq \kappa \cdot \left(\Phi^{-1}\left(1-\tfrac{\alpha}{2}\right)\right)^2 \cdot \varepsilon_{\bar{\mathrm{r}}}^{-2} \cdot \hat{p} \quad \text{für } \hat{p} > 0{,}5$$

Auch geringere systematische Schätzfehler:

- Entfall der unzutreffenden Annahme gleichgroßer Nachweiswahrscheinlichkeiten und der daraus resultierenden systematischen Schätzfehler.
- Informationsaustausch zwischen Fehlereinbau und Inspektion besser unterbindbar als zwischen zwei Inspekteuren.

ε_{r}, $\varepsilon_{\bar{\mathrm{r}}}$	Intervallradius realtiv zum erwarteten Eintritts- bzw. Nichteintritts-Zählwert.
κ	Varianzerhöhung durch Abhängigkeiten, für unabhängige Zählwerte $\kappa \leq 1$.
$\Phi^{-1}(..)$	Inverse Funktion zur Verteilungsfunktion der standardisierten Normalverteilung.
n, x_{AV}	Anzahl der Zählversuche, Experimentell bestimmter Ist-Zählwert.
$\hat{p}$	Schätzwert der Eintrittswahrscheinlichkeit.

5.15 Extrapolation der Inspektionszeit



Auch für Inspektionen gilt in der Regel das Pareto-Prinzip. Mit einem kleinen Teil des Inspektionszeit t wird die Mehrheit der Fehler gefunden. Deutet auf Pareto-Verteilung der Nachweiszeit (Gl. 4.94):

$$F_X(t) = \mathbb{P}[X \leq t] = \mu_{\mathrm{FC}}(t) = \begin{cases} 0 & t \leq t_{\min} \\ 1 - \left(\frac{t_{\min}}{t}\right)^K & \text{sonst} \end{cases} \tag{5.5}$$

Schätzung $\mu_{\mathrm{F}}(t \to \infty)$ aus Verlauf für kleine t wie im Bild unsicher.

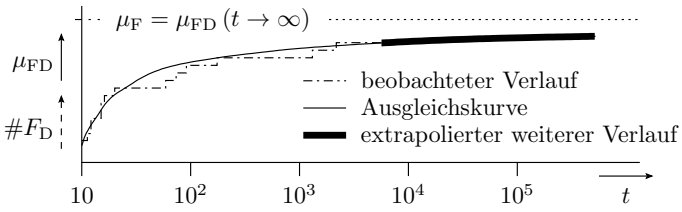
$\#F_{\mathrm{D}}$, μ_{FD}	Anzahl der erkannten Fehler, zu erwartenden Anzahl der erkennbaren Fehler.
t, $t_{\min}$	Inspektionszeit, Skalenparameter der Pareto-Verteilung.
X	Zufallsgröße der Inspektionszeit, um einen Fehler zu finden.
$K > 0$	Formfaktor der Pareto-Verteilung.

1.2 Inspektionstechniken

5.16 Inspektionstechniken

Inspektionen, die hunderte von Fehlern erkennen, kosten hunderte und mehr Mitarbeiterstunden. Außer der Fehlerüberdeckung spielt vor allem auch die Effizienz in gefundenen Fehlern pro Mitarbeiterstunde eine wichtige Rolle.

5.17 Inspektionszeitverteilung und Effizienz



Die zu erwartende Effizienz nimmt proportional mit dem Anstieg der Verteilungsfunktion, d.h. mit der Dichte der Inspektionszeit ab:

$$\frac{\mu_{\mathrm{EFC}}(t)}{\mu_{\mathrm{F}} \cdot 1\,\mathrm{h}} = f_X(N) = \frac{\mathrm{d}F_X(t)}{\mathrm{d}t} = \frac{K \cdot t_{\min}^K}{t^{K+1}} \tag{5.6}$$

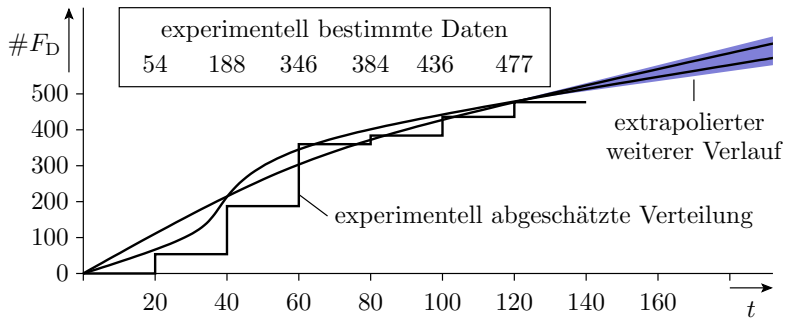
Bei pareto-verteilter Inspektionszeit mehr als umgekehrt proportionale Effizienzabnahme mit der Inspektionszeit.

μ_{EFC}	Zu erwartende Effizienz in gefundene Fehler pro Mitarbeiterstunde.
$f_X(t)$	Dichtefunktion der Inspektionszeit.
μ_{F}	Zu erwartende Anzahl der vorhandenen Fehler.
1 h	Eine Mitarbeiterstunde.

5.18 Experiment mit einem Inspekteur

Inspektion eines Buchmanuskripts* plus Beispielprogramme:

- Anzahl der gefunden Fehler in Abhängigkeit von der Inspektionszeit.



$\#F_D$ Anzahl der gefundenen Fehler (Number of detectable faults).
t Inspektionszeit in Mitarbeiterstunden.
* Bachelor-Arbeit von Yu Hong.

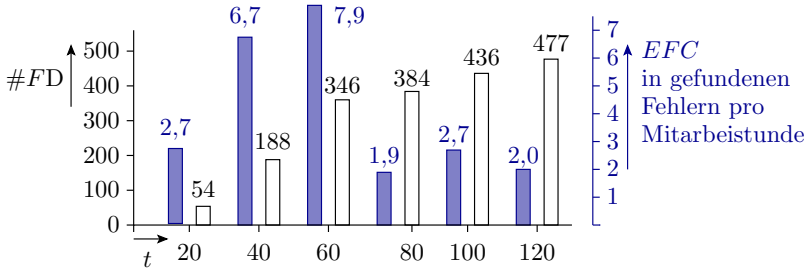
Die Breite des wahrscheinlichen Bereich von Zählwerten überschlagsweise proportional zur Wurzel aus dem Wert. Geschätzte Erwartungswertverläufe im Bereich der untersuchten Inspektionszeit unsicher und Extrapolation für längere Inspektionszeiten noch viel unsicherer. Im Experiment ist nicht einmal die mehr als umgekehrt proportionale Effizienzabnahme mit Inspektionszeit t, die für eine pareto-verteilte Nachweiszeit sprechen würde, zu erkennen.

Die nächste Folie untersucht den Zusammenhang zwischen Inspektionszeit und Effienz deshalb genauer.

$\#F_D$ Anzahl der gefundenen Fehler (Number of detectable faults).
t Inspektionszeit in Mitarbeiterstunden.

5.20 Untschiede Inspektion und Zufallstest

Die Effizienz wurde für jeweils 20 Inspektionsstunden aus dem Zuwachs der Anzahl der gefundenen Fehler geschätzt. Sie wuchst zu Beginn und nahm nach 60 bis 80 Stunden deutlich ab, obwohl erst die Hälfte der Fehler erkannt war.



Es gibt offenbar eine *»Anlernphase«*, mit zunehmender Effizienz, eine effiziente Phase und eine Ermüdungsphase mit geringer Effizienz.

$\#DF$ Anzahl der nachweisbaren Fehler.
EFC Effizienz, gefundene Fehler pro Mitarbeiterstunde.

- 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 erst etwa die Hälfte der Fehler 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, ...

These

Ein gute Inspektionstechnologie minimiert ineffiziente Anlernphasen bzw. nutzt sie zu Weiterbildung und vermeidet ineffiziente Ermüdungsphasen.

5.22 Inspektionstechniken

Arbeit *geschickt* auf mehrere Inspekteure mit unterschiedlichen Rollen verteilen. Diversität ausnutzen: »*Inspekteur ungleich Autor*«, »Vier Augen sehen mehr als zwei«, ...

Einteilung der Inspektionstechniken

- Review in Kommentartechnik: Dokumente Korrekturlesen und 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, ...

Stellschrauben einer Inspektionstechnologie:

- Lesegeschwindigkeit, Rollendisziplin,
- Vermeidung Langeweile und überhitzter Emotionen,
- Gruppennormen, ...

1.3 Zusammefassung

5.23 Anwendung, Güteabschätzung

Inspektion bedeutet in der Regel manuelle Sichtkontrolle von Dokumentationen (Entwurfergebnisse, Testausgaben, ...) und wird eingesetzt, wenn keine automatisierten Kontrollen verfügbar.

Anzahl der nicht nachweisbaren Fehler und der Fehlerabdeckung:

- Capture-Recapture: Zählen der von zwei Inspekteueren einzeln und gemeinsam gefundenen Fehler. Schätzer:

$$\hat{\mu}_\mathrm{F} = \left.\frac{\#F_1 \cdot \#F_2}{\#(F_1 \cap F_2)}\right|_\mathrm{ACR} \tag{5.1}$$

$$\hat{\mu}_\mathrm{FC} = \left.\frac{\#(F_1 \cup F_2)}{\hat{\mu}_\mathrm{F}}\right|_\mathrm{ACR} = \left.\frac{\#(F_1 \cap F_2) \cdot \#(F_1 \cup F_2)}{\#F_1 \cdot \#F_2}\right|_\mathrm{ACR} \tag{5.2}$$

- Inspektionsmaterial mit eingebauten Kontrollfehlern. Fehlerabdeckung etwa Kontrollfehlerabdeckung. Fehleranzahl:

$$\hat{\mu}_\mathrm{FC} = \left.\frac{\#F_\mathrm{DM}}{\#F_\mathrm{M}}\right|_\mathrm{ACR} \tag{5.3}$$

$$\hat{\mu}_\mathrm{F} = \left.\frac{\#F_\mathrm{D}}{\hat{\mu}_\mathrm{FC}}\right|_\mathrm{ACR} = \#F_\mathrm{D} \cdot \left.\frac{\#F_\mathrm{M}}{\#F_\mathrm{DM}}\right|_\mathrm{ACR} \tag{5.4}$$

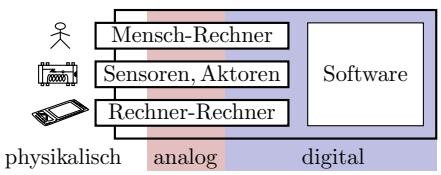
»Kontrollfehler« weniger Schätzfehler als »Capture-Recapture«, aber auch weniger populär.

5.24 Inspektionstechniken

Inspektion hat ineffiziente Einarbeitungs- und Ermüdungsphasen. Eine gute Inspektionstechnik vermeidet diese durch geschickte Abeitsorganisation.

2 Testdurchführung

5.25 Systemstruktur und Test

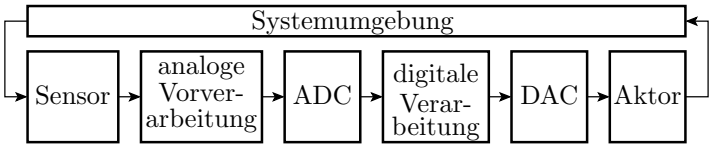


Ein IT-System kommuniziert mit der Welt über analoge Signale, arbeitet aber intern überwiegend digital. Getestet werden idealerweise alle Funktionsbausteine einzeln und hierarchisch aufsteigend die daraus zusammengesetzten Funktionsblöcke bis zum Gesamtsystem.

Einteilung der Tests nach der Art der Ein- und Ausgabe:

- physikalische Signale (Weg, Druck, Kraft, ...),
- elektrische analoge Signale (Spannung oder Strom),
- digitale Signale (Bit- und Bitvektorfolgen),
- Daten ohne physikalische Repräsentation (Software).

5.26 Verarbeitungskette und Aufgabenteilung



Entlang des Verarbeitungsflusses physikalisch, analog, digital, Software werden Entwurf und Testdurchführung immer einfacher. Dafür werden die realisierten Funktionen immer vielfältiger und komplexer.

- Physikalische und analoge elektrische Größen werden praktisch nur vorverarbeitet, gewandelt und ausgegeben.
- Die Informationsverknüpfung und Speicherung erfolgt digital.
- Komplizierte Funktione werden in Software realisiert.

Alle Systembestandteile sind einzeln und im Verbund zu testen. Dazu sind bei der Systemgestaltung alle Voraussetzungen zu schaffen:

- Prüftechnik anzuschließen, Testbeispiele aufzustellen,
- Testeingaben einzuspeisen, ...

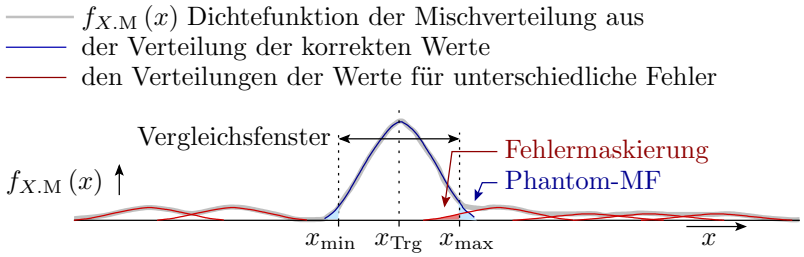
2.1 Physikalisch

5.27 Test mit physikalischen Ein- und Ausgaben

1. Die Bereitstellung und Messung physikalischer Ein- und Ausgaben (Weg, Kraft, Beschleunigung, Temperatur, auch elektrische Signale, ...) verlangt spezielle, oft teure Prüftechnik.
2. Diese Prüftechnik hat begrenzte Funktionalität: Wertebereich, Signalanzahl, Bandbreite, Signalformen, Genauigkeit, ...
3. Werteverfälschungen der Ein- und Ausgaben durch die Prüftechnik und den Messaufbau (Verzögerung, Übersprechen, Rauschen, Reflexionen, ...).
4. Systeme mit physikalischen Ein- und Ausgaben dienen of zur Steuerung oder Reglung und müssen auch im Zusammenwirken mit ihrer Umgebung getestet werden.

Es gibt für ausgewählte physikalische und analoge Prüf- und Messaufgaben gute Lösungen, für andere nicht. Beschränkung auf gut testbare Komponenten. Das sind in der Regel nur Vorverarbeitung und Wandlung, Stromversorgung, ...

5.28 Kontrolle von analogen Merkmalen

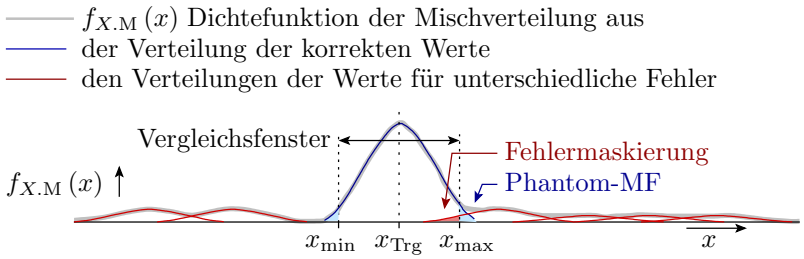


Aus Messwerten werden zu kontrollierende Merkmale gebildet:

- Verstärkung, Klirrfaktor (Maß der Linearität),
- Rauschen, Bandbreite, Sprungantwort, ...

und die Merkmalswerte einem Fenstervergleich unterzogen.

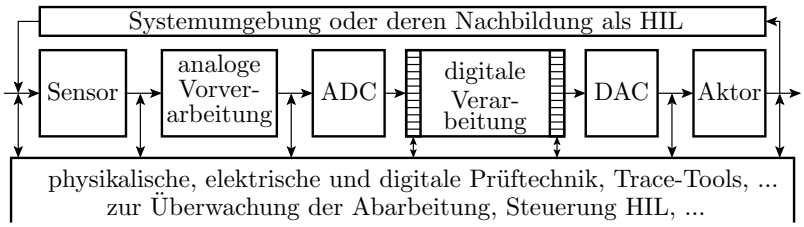
Systematische Fehler durch Messtechnik und Messaufbau ändern die Sollwerte, zufällige die erforderliche Vergleichsfensterbreite bei gleicher Phantomfehlfunktionsrate.



Die Varianzen zufälliger Einflüsse der einzelnen Komponenten der Testdatenerzeugungs-, Übertragungs- und Auswertungsketten addieren sich und die Standardabweichungen addieren sich nach Pythagoras. Die einzelnen zufälligen Einflüsse dürfen nur gering sein im Vergleich zur Vergleichsfensterbreite und damit zum Toleranzbereich der betrachteten Parameter.

Prüftechnik muss viel genauer arbeiten als das zu testende System.

5.30 Möglichkeiten der Testdurchführung



Ganzheitlicher Test mit Eingabevorgabe ohne Systemumgebung:

- Signalgeneratoren für Eingabe, Ausgabeaufzeichung, ...

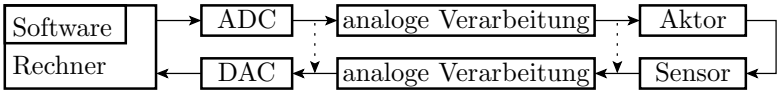
Ganzheitlicher Test in der Systemumgebung:

- Überwachter Betrieb in der Systemumgebung,
- HIL (Hardware in the Loop), Attrappe oder Simulation der Systemumgebung.

Isolierter Test der physikalischen bzw. analogen Ein- und Ausgabe:

- spezielle Testmodi mit Anschluss externer Prüftechnik,
- Loop-Test (testspezifische Signalflussumschaltung).

5.31 Loop-Test



- Messung der Aktorausgaben mit den Sensoren des Systems bzw. der analogen Ausgaben über die analogen Eingänge.
- Isolierter Test der Übertragungsfunktionen digital ⇒ analog ⇒ [physikalisch ⇒ analog] ⇒ digital.
- Fenstervergleich der Ergebnisswerte mit den Vorgabewerten,
- Einbindbar in Selbsttests.

Für die Nutzungs als Messtechnik müssen die Wandler, Sensoren und Aktoren genauer arbeiten als das zu testende System, von dem sie ein Teil sind (höhere Bitauflösung, höhere Bandbreite, ...).

5.32 Hardware-in-the-Loop (HIL)

Die Systemumgebung

- Reglungsstrecke,
- physikalischer Prozess,
- Verhaltes eines Autos oder eines Flugzeugs, ...

werden für den Test durch eine Attrappe oder Simulation ersetzt.

Erlaubt gründlichere Untersuchung des Systemsverhaltens, insbesondere gefährlicher Situationen, z.B. für das Verhalten von Flugzeugen in provozierten Fehlersituationen.

5.33 Prüfgerechter Entwurf

- Die externe Prüftechnik muss anschließbar sein, mechanisch (Stecker, Kontaktflächen) und elektrisch (Eingangswiderstand, ...),
- Triggersignal für Aufzeichungsbeginn, ...
- Testmodie z.B. für Loop-Tests müssen steuerbar sein, ...
- Schnittstelle für den HIL, der auch eine Simulation sein kann, die über eine Datenschnittstelle statt über Signalverlverläufe kommuniziert.
- Isolationsmöglichkeit für isoliert zu testende Bausteine.

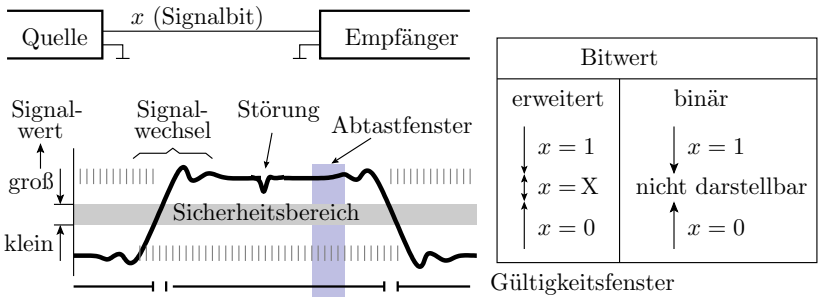
Man kann sich viel verbauen. Überlegungen zum Test müssen den gesamten Entwurfsprozess ab der Spezifilation begleiten.

Prüfgerechter Entwurf als Leitfaden zur vorbeugenden Vermeidung von Testbarkeitsproblemen umfasst

- Sammlungen gut funktionierender Lösungen,
- typische Probleme und Workarounds,
- Checklisten, was dabei nicht vergessen werden darf, ...

2.2 Digitale Bausteine

5.34 Digitale Verarbeitung (Folie 1.107)



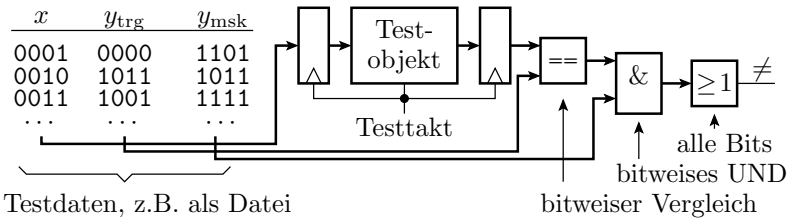
Informationsweitergabe durch Bits:

- Werteunterteilung in groß, klein und ungültig,
- Abtastung im Gültigkeitsfenster, ...

Immun gegen Störungen und Messfehler. Einfacherere Prüftechnik, ...

0, 1, X Logische Signalwerte für klein, groß und ungültig.

5.35 Test mit digitalen Ein und Ausgaben



Wiederhole für jeden Taktschritt (Test):

- Bereitstellung logischer Eingabewerte und
- Abtasten und Auswertung der vorherigen Ausgaben.

Auswertung vorzugsweise Soll/Ist-Vergleich. Ausmaskierung unvorhersagbarer Vergleichsergebnisse (Ist- oder Sollwert X).

x Testeingaben.
y_{trg} Sollwerte der Testausgaben.
y_{msk} Maskenwerte zum Ausschluss von Testausgaben vom Soll-Ist-Vergleich.
$\neq$ Vergleichsfehler.

Abwandlungen und Ergänzungen der Testanordnung:

- Kontrolle der Signalverzögungen durch Ergebnisabtastung mit mehrfacher Aufzeichnungsfrequenz.
- Tester auch als Kombination Signalgenerator Logikanalysator.

Regeln des prüfgerechten Entwurfs:

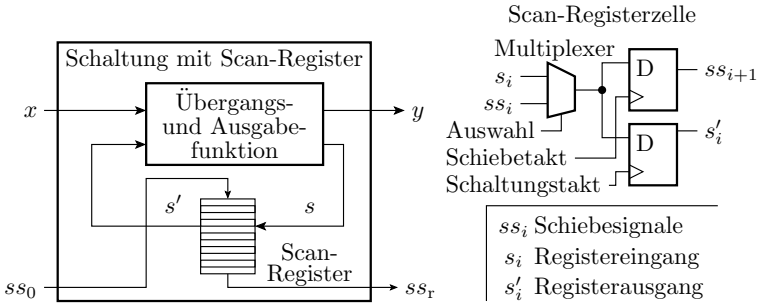
- Initialisierungsmöglichkeit für Speicherzellen.
- Isolierter Test komplexer Funktionsbausteine, insbesondere von großen Speicherblöcken (Abschn. 6.2.7), ...

Kostenfaktoren für externe Prüftechnik:

- Anschlussanzahl, Größe der Testdatenspeicher,
- zeitliche Genauigkeit und Testgeschwindigkeit,
- Kontaktierung (Stecker, Nadeladapter, ...).

Alternative zu externer Prüftechnik ist Selbsttest (Abschn. 6.3).

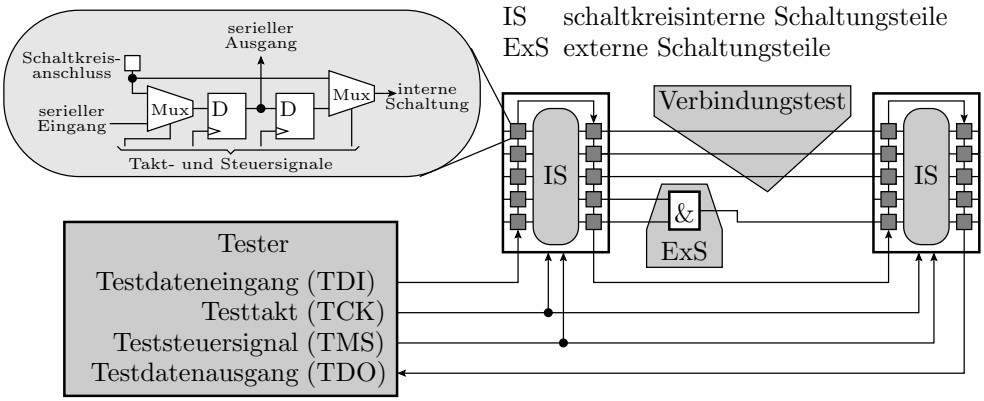
5.37 Scan-Verfahren



Fehlender Lese- und Schreibzugriff auf interne Signale von Schaltkreisen, insbesondere Zustandsbits, können die Erstellung von Tests erheblich erschweren. Problemumgehung mit Scan-Registern (Abschn. 6.2.6). Im Bild ist jede Speicherzelle um einen Multiplexer und eine Schiebezelle erweitert zur Bereitstellung der Funktionen:

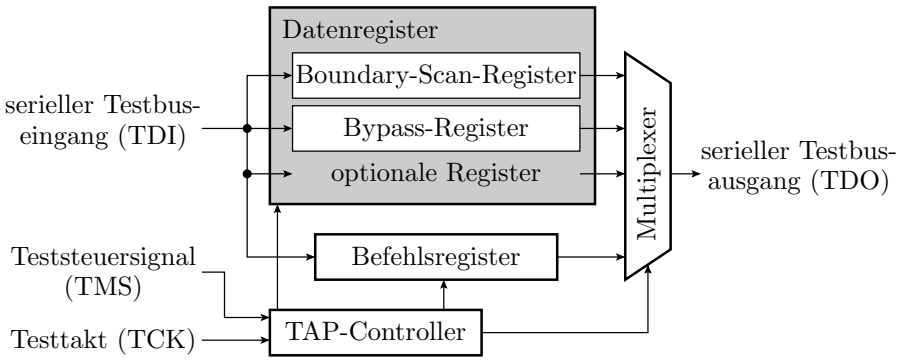
- Capture: Übernahme aus der Schaltung in die Schiebezellen,
- Shift: serielles Auslesen und neu beschreiben und
- Update: Übergabe seriell eingelesene Daten an Schaltung.

5.38 Boundary-Scan



Durch SMD-Bestückung, Packungsdichte, ... frühere Kontaktierung der Leitungen auf Baugruppen für den Test mit Nadeladaptern kaum noch möglich (Abschn. 6.4). Heute oft ersetzt durch Boundary Scan, einem Scan-Register an den Schaltkreisanschlüssen, mit denen die logischen Pegel an den Schaltkreisanschlüssen gesetzt und gelesen werden.

5.39 JTAG-Testbus



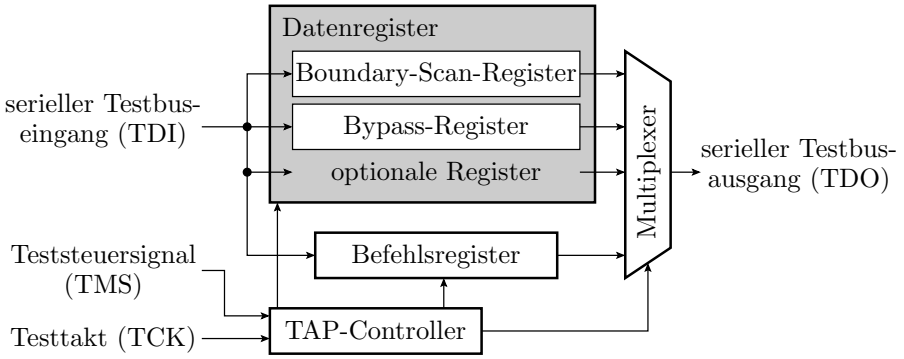
JTAG standardisiert einen Testbus für Boundary-Scan, der auch für andere Test-, Debugg- und Programmieraufgaben nutzbar ist. Eine JTAG-Testbusimplementierung umfasst:

- den TAP- (Test Access Port) Controller,
- ein Befehls- und mehrere Testdatenregister.

5.40 Busprotokoll und TAP

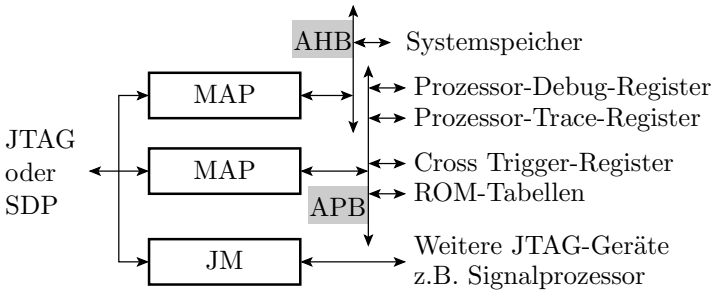
TAP	Testbussteuerung (Test access port).
TMS	Testoperationsauswahlsignal (Test mode select signal).
*	Zwei Bits des Befehlsregisters liefern beim Lesen null und eins.

5.41 Optionale Testregister und Funktionen



- Lesen einer Hersteller- und Bauteil-ID (Bestückungstest),
- Flashen von Programm- und Konfigurationsspeichern,
- Lese- und Schreibzugriff auf Programmspeicher,
- Debugger-Register für Schrittbetrieb, Haltepunkte, Trace-Steuerung, ...

5.42 ARM-Testbusarchitektur



Die ARM-Testbusarchitektur definiert zusätzlich über den AHB Funktionalität für Lese- und Schreibzugriff auf den kompletten Speicher und über den APB auf die Debug-Register, ROM-Tabellen, ...

ARM Verbreitetste Mikroprozessor-Architektur für Embedded-Systems (Smartphones, ..).
SDP Serieller Debug-Port.
JM JTAG-Master für weiterer Rechnerbausteine auf dem Chip, z.B. Co-Prozessoren.
MAP Speicherzugriffsport mit serieller Adress- und Datenübergabe.
AHB, APB Advanced High Performance Bus, Advanced Peripherial Bus.

5.43 Test- und Debug-Register (ARM)

- 32-Bit Debug-Control und Status-Register (DSCR).
- Instruction-Transfer-Register (ITR): 32 Befehlsbit + ein Statusbit, zur Ausführung von Prozessorbefehlen im Debug-Modus.
- Debug Communications Channel (DCC): 32-Bit-Datenwort + 2 Statusbits für den bidirektionalen Datentransfer mit Prozessorkern.
- Embedded Trace Module (ETM): 7 Adressbits + 32-Bit-Datenwort + 1 R/W-Bit zur Steuerung von Trace-Operationen*.
- Debug-Modul: 7 Adressbits + 32-Bit-Datenwort + 1 R/W Bit. Zugriff auf die Register für Hardware Breakpoints, Watchpoints etc..

ARM Befehle für Zusammenarbeit Programm und Debugger:

- HALT für den Übergang in Debug-Modus, in dem über das ITR Befehle eingefügt werden können und
- RESTART zum Verlassen des Debug-Modus.

---

* Trace-Aufzeichnung entweder in einen eingebetteten Trace-Buffer (ETB) auf dem Chip oder Ausgabe über einen High-Speed-Port.

2.3 Software

5.44 Digitale Verarbeitung vs. Software

Als digitale Schaltung lassen sich gut realisieren:

- Logische Verknüpfungen von Eingaben zu Ausgaben,
- Register-Transfer-Funktionen für die Ausführung in jedem Taktschritt,
- Blockspeicher, Addierer, Multiplizierer, ...

Damit werden auch programmierbare Strukturen realisiert:

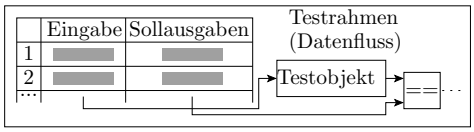
- speicherprogrammierbare Steuerungen,
- Universal- und Spezialprozessoren,
- programmierbare Logikschaltungen, ...

Software benötigt programmierbare Hardware und kann viel komplexere Funktionen realisieren, nämlich alles, was beschreibar ist mit:

- Berechnungsfolgen, Fallunterscheidungen, Schleifen,
- Unterprogrammen, Rekursionen, Nebenläufigkeit, ...

Wichtigste Regeln des prüfgerechten Entwurfs: Modularität, Determinismus und eine Service-Struktur (zumindest fast aller Teilbausteine).

5.45 Testrahmen



Testrahmen: (Kontrollfluss)

Wiederhole für alle Tests
Bereitstellung der Eingaben
Testausführung
Kontrolle der Ausgaben

Softwaretest in der Regel ohne externe Prüftechnik. Zu testende Programmbausteine werden von einem Testrahmen ausgeführt, der Testeingaben bereitstellt und Testausgaben auswertet.

Der **Testrahmen** ist ein ausführbares Programm und das zu testende Programm ein **deterministischer Service**, der wenn fehlerfrei auf Anforderung aus gleich Eingaben dieselben Ausgaben erzeugt.

Wünschenswerte Unterstützung durch die Entwicklungsumgebung:

- Erstellung und Verwaltung der Testrahmen,
- Eingabeprofilbasierte Testeingabeauswahl (Abschn. 2.3.8), ...

5.46 Ein Testobjekt und sein Testrahmen

Beispieltestsobjekt: Unterprogramm zur Quadrierung:

```
uint32_t quad(int16_t a){ // Square calculation
  return (uint32_t)a * a; // Why with Typcast?
};
```

Testbeispiele sind Tupel aus Eingaben und Sollausgaben. Man kann dafür einen neuen Datentyp definieren:

```
typedef struct{
  int16_t x;              // Input
  uint32_t y;             // Target output
} test_t;
```

Ein »struct« ist eine Zusammenfassung aus bereits definierten Datentypen.

5.47 Testsatz

Eine Testsatz als Menge von Tests ist im einfachsten Fall ein initalisiertes Feld von Testbeispielen:

```
test_t testset[] ={{<Tupel1>},{...},...};
```

Testbeispiele für die Quadratberechnung:

```
test_t testset[] ={      // Input  Target output
  {0, 0},                // 0x0000  0x00000000
  {1, 1},                // 0x0001  0x00000001
  {9, 81},               // 0x0009  0x00000051
  {-5, 25},              // 0xFFFB  0x00000019
  {463, 214369},         // 0x01CF  0x00034561
  {0x7FFF, 1073676289}   // 0x7FFF  0x3FFF0001
};
```

Welche Testbeispiele signalisieren Fehler?*

* Vermutlich ergibt -5 konvertiert in uint32_t mal -5 keinen positiven Wert.

5.48 Testrahmen

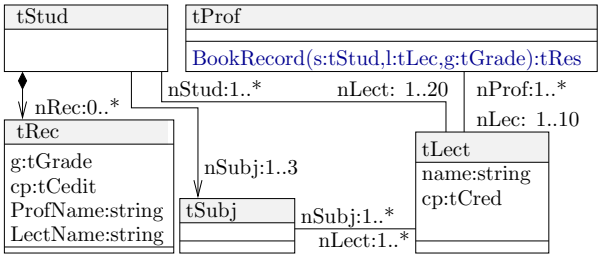
Programm, das in einer Schleife alle Testbeispiele abarbeitet und die Ergebnisse kontrolliert oder zur Kontrolle ausgibt:

```
int main(){
 uint8_t idx, err_ct=0;
 uint32_t erg;
 for (idx=0; idx<6;idx++){
 erg = quad(testsatz[idx].x);// Target output
 if (erg != testsatz[idx].y) // Comparison
  <Protokollierung des fehlgeschlagenen Tests>;
 }
}
```

Zur Untersuchung, welchere Test versagt und zur Fehlerlokalisierung:

- Protokollierung fehlgeschlagener Tests.
- Bei unklarer Ursache, Protokollierung zusätzlicher Werte.
- Trace-Aufzeichung von Zwischenwerten, z.B. in einem Ringpuffer,
- Setzen von Haltepunkten entgegen Berechnungsfluss (Folie 2.37),
- Harware-Unterstützung (Folie 5.42 *ARM-Testbusarchitektur*).

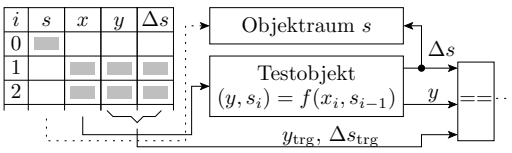
5.49 Test von Methoden in einem Objektraum



Das UML-Klassendiagramm für ein Prüfungssystem zeigt die wesentlichen Datenobjekte und deren Beziehungen zueinander für eine zu testende Beispielmethode BookRecord(...) für die Notenverbuchung durch einen Dozenten: t... – Datentyp/Klasse, tProf – Dozenten, tStud – Studenten, tLect – Lehrveranstaltungen, tSubj – Studiengänge.

Der Testrahmen muss einen Beispielobjektraum erzeugen und kann dann ein- oder mehrfach BookRecord(...) und andere Methoden mit Beispielwerten auf den Objektraum anwenden.

5.50 Testrahmen für Methoden



Der Test muss zusätzlich den Objektraum initialisieren und nach jedem Testschritt die Änderungen und Unveränderthеiten des Objektraums kontrollieren.

Die Sollwerte von Software-Ausgaben sind in der Regel aufgezeichnete, durch Inspektion und andere Kontrollen überprüfte Ist-Wert.

In Kombination mit dem Beispiel auf der Folie zuvor ist erkennbar, das Testrahmen und Testdaten umfangreich sind und eine weitgehende Automatisierung der Testerzeugung und -durchführung wünschenswert ist.

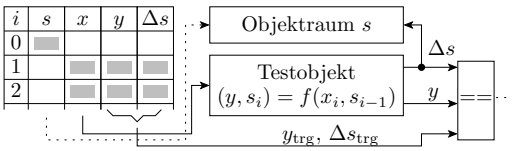
---

x, y Testeingaben, Testausgaben.

s, Δs Objektraum (bearbeitete gespeicherte Daten), Änderungen des Objektraums.

y_{trg}, Δs_{trg} Sollwerte der Testausgaben und der Objektraumänderung.

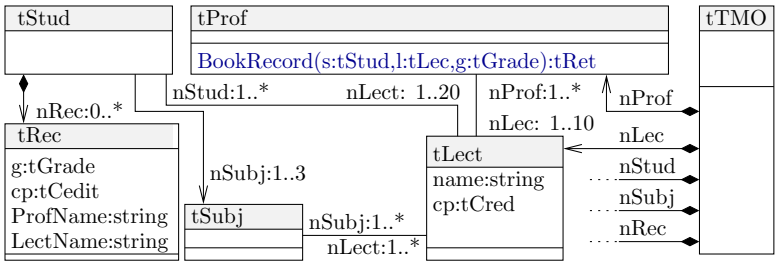
5.51 Testrahmengenerierung



Ein Testrahmen besteht vereinfach aus Tabelle und Ausführungsschleife. Die Tabellendaten sollen typische Werte, Grenzwerte und zufällige Werte mit Nutzungsprofilen und Testprofilen zur Abdeckungsverbesserung enthalten und benötigt zur Erzeugung

- alle verwendeten Datentypen und Klassen Textdarstellung für die Inspektion und Fehlersuche.
- Eine Beschreibung möglicher Objektstrukturen und Objektraumänderungen,
- »Würfelfunktion« für alle elementaren Datentypen und deren Zusammenfassungen zu Eingabe und Objekträumen.

5.52 Würfelfunktionen für Objekträume

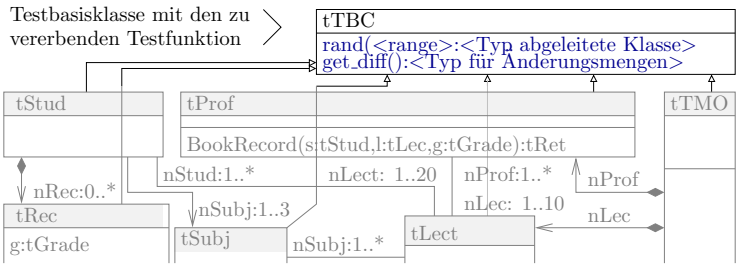


Unser Prüfungssystem benötigt Würfelfunktionen (rand(<range>) für

- die Objektanzahl der Professoren, Studenten, Studiengänge, , ...
- die Bedatung dieser Objekte und
- die Assoziationen zu anderen Objekten.

Das Auswürfeln einer zufälligen Objektanzahl, Erzeugung zufällig bedateter Objekt, deren Löschung, Änderungskontrolle nach jedem Testschritt, ... verlangt ein zentrales Testverwaltungsobjekt (TMO).

5.53 Testbasisklasse (tTBC)



Funktionen für alle Objekttypen werden in der objektorientierten Programmierung durch Vererbung beschrieben, im Bild

- rand(<range>): Erzeugung eines Zufallswerts,
- get_diff(): Bestimmung der Objektraumänderungen, ...

Für komplexe Software sind allein die Programmierung von Würfelfunktionen für geeignete zufällige Eingaben und Objektrauminitialisierungen, ... ein Aufwand, der Automatisierung verlangt.

Zusammenfassung

5.54 Physikalische Ein- und Ausgaben

Die Durchführbarkeit ausreichend gründlicher Tests am fertigen System und seinen Komponenten verlangt entwurfsbegleitende Testentscheidung und Testbarkeitskontrollen praktisch ab der Spezifikation.

Das Zauberwort heißt prüfgerechter Entwurf, eine Sammlung von Regeln, um potentielle Probleme zu umschiffen.

Die meisten Vorüberlegungen erfordern die Tests des Gesamtsystems bzw. die isolierten Tests der Ein- und Ausgabeeinheiten mit physikalischen oder analogen elektrischen Ein- und Ausgabesignalen:

- Geeignete Prüftechnik? Anschließbar? Deterministisch?
- Messgenauigkeit bzw. Umgang mit Messfehlern?

Test des Zusammenwirkens mit der physikalischen Umgebung:

- Prüfstand, Simulation, HIL, Ausprobieren erst durch Nutzer?
- Auch hier Technik verfügbar, ausreichend genau, anschließbar?
- ...

5.55 Test digitaler Systeme

Für digitale Teilsysteme ist die Testdurchführung – Bereitstellung der Eingabebits und Soll-Ist-Vergleich für Ausgabebits – vergleichsweise einfach. Dafür sind die zu testenden System viel komplizierter

- viel mehr Eingänge,
- große Zustandsspeicher,
- viel schnellere Verarbeitung.

Die erforderlich Kontaktierung der großen Zahl von Schaltungspunkten für Testeingaben und -ausgaben auf Baugruppen und in Schaltkreisen wird heute überwiegend mit Scan-Verfahren und Testbussen umgangen. Das demonstriert, welcher Hardware-Aufwand heute zur Umgehung von Testproblemen akzeptiert ist.

Weitere Probleme, Lösungen und Maßnahmen des prüfgerechter Entwurf für Schaltkreise und Baugruppen siehe später Foliensatz 6.

5.56 Software-Test

Für Software ist die Testdurchführung noch viel einfacher. Das Testobjekt wird in ein Rahmenprogramm eingebettet, das die Eingaben bereitstellt und die Ausgaben auswertet, übersetzt und ausgeführt.

Die testenden Funktionen sind jedoch viel komplexer.

Für eine einfache Funktion ist die Programmierung der Testrahmen eine Routineaufgabe: Definition einiger Datentypen und -objekten, Zusammenstellung von Beispieleingaben und -ausgaben, ... Ähnlicher Code-Umfang und Arbeitsaufwand wie für die Testobjekte.

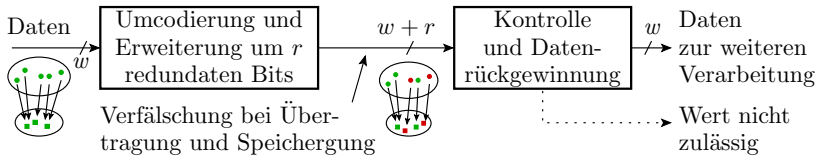
Für den Test von Methoden in einem Objektraum kommt die Initialisierung dazu. Der Testrahmen benötigt objektspezifische Pseudo-Zufallszahlengeneratoren, Kontrollfunktionen. Programmieraufwand $\Uparrow$.

Ausreichender Operations- und Testprofilabdeckung übersteigen die Möglichkeiten händischer Testprogrammierung (siehe später).

Insgesamt sind auch bei Software viele Probleme zu umschiffen, überwiegend andere als für Hardware. Prüfgerechter Entwurf, Testsuche, Testautomatisierung, ... für Software siehe später Foliensatz 7.

3 Datenüberwachung

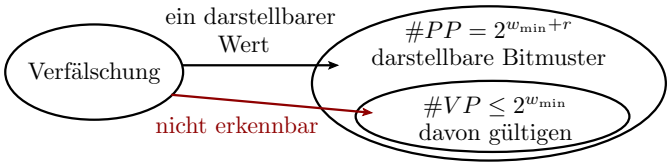
5.57 Nachweis und Korrektur von Datenfehlern



Erweiterung zu übertragender oder zu speichernder Daten um r redundante Bits für den Nachweis und die Korrektur von Verfälschungen. Wir behandeln

- fehlererkennende Codes für zufällige Verfälschungen mit Schwerpunkt auf zyklisch redundante Codes,
- Prüfkennzeichen, Prüfsummen, CRC-Kennzeichen,
- Hamming-Codes mit Schwerpunkt auf Paritätstests und fehlerkorrigierende Codes für Einzelbit- und Burstverfälschungen.

5.58 Mathematische Grundlagen



- Informationsredundanz: Erweiterung der Datenobjekte vor der Speicherung oder Weitergabe um r redundante Bits.
- Pseudo-zufällige Verteilung der zulässigen Datenworte in der Menge der darstellbaren Datenworte. Idealerweise Ausschluss, dass bevorzugt zulässige Datenworte entstehen können.
- Linearität. Nachweisbarkeit unabhängig vom Sollwert:

$$f(c \oplus \Delta c) = f(c) \oplus f(\Delta c)$$

$\#VP, \#PP$ Anzahl der gültigen Bitmuster, Anzahl der darstellbaren Bitmuster.
$w_{\min}$ Mindestbitanzahl zur Darstellung der zulässigen Werte, Anzahl der redundanten Bits.
$c, \Delta c$ Bitdarstellung der Daten, bitweise Verfälschung.

3.1 Fehlererkennende Codes

5.59 Prinzip der fehlererkennenden Codes

Umkehrbar eindeutige pseudo-zufällige Abbildung zulässiger Werte auf darstellbare Werte.

Das macht es für Verfälschungen unmöglich, zulässige oder unzulässige Werte zu bevorzugen und garantiert die Gültigkeit von:

(1.33)
$$\mu_{\mathrm{MC}} = 1 - \frac{\#VP}{\#PP}$$

Beispiel: Multiplikation der zu überwachenden Werte mit einer großen Konstanten C:

$$y = C \cdot x$$

Verfälschungen werden bei der Decodierung am Divisionsrest $y\%C \neq 0$ erkannt. Erkennungswahrscheinlichkeit:

$$\mu_{\mathrm{MC}} = 1 - \frac{\#x}{\#x \cdot C} = 1 - \frac{1}{C} \tag{5.7}$$

C Große ganzzahlige Konstante, bevorzugt eine Primzahl.
x, y Zu verschlüsselndes Datenwort, Verschlüsseltes Datenwort.
$\#x$, % Anzahl der darstellbaren Code-Worte, Divisionsrest.

5.61 Polynom-Multiplikation und -division

Codierung durch die Multiplikation mit einer Konstanten, allerdings nicht arithmetisch, sondern modulo-2. In Hard- oder Software einfacher als arithmetische Multiplikation:

Codierung

```
        10010101101
⊙           10011
-----------------
⊕     100[1]0101101
⊕      10010101101
⊕     000000000000
⊕    0000000000000
⊕   10010101101
-----------------
  10001110[0]100111
```

Decodierung

```
 100011100100111 : 10011 = 10010101101
⊕10011
-------
  10110
 ⊕10011
 -------
   10101
   10011
   -----
    11000
   ⊕10011
   -------
    10111
   ⊕ 10011
   -------
     10011
    ⊕10011
    -------
Rest: 0000        (unverfälscht)
```

5.62 Warum Polynommultiplikation?

Die zu multiplizierenden Folgen lassen sich auch als Polynome des Operators z, der eine Bitverschiebung beschreibt, dargestellt:

- $10011 \Rightarrow (1\cdot z^4) \oplus (0\cdot z^3) \oplus (0\cdot z^2) \oplus (1\cdot z^1) \oplus (1\cdot z^0) = z^4 \oplus z \oplus 1$
- $10010101101 \Rightarrow z^{10} \oplus z^7 \oplus z^5 \oplus z^3 \oplus z^2 \oplus 1$

Die Multiplikation »·« und die Addition »⊕« erfolgen bitweise modulo-2, also durch UND und EXOR. Potenzen z^i beschreiben Verschiebungen um i Bitstellen. Das Produkt beider Polynome

$$\mathbf{x} \odot \mathbf{g} = \mathbf{p} = \left(z^{10} \oplus z^7 \oplus z^5 \oplus z^3 \oplus z^2 \oplus 1\right) \odot \left(z^4 \oplus z \oplus 1\right) = z^{14} \oplus z^{10} \oplus z^9 \oplus z^8 \oplus z^5 \oplus z^2 \oplus z \oplus 1$$

repräsentiert denselben Bitvektor, wie auf der Folie zuvor berechnet. Die Polynomdivision:

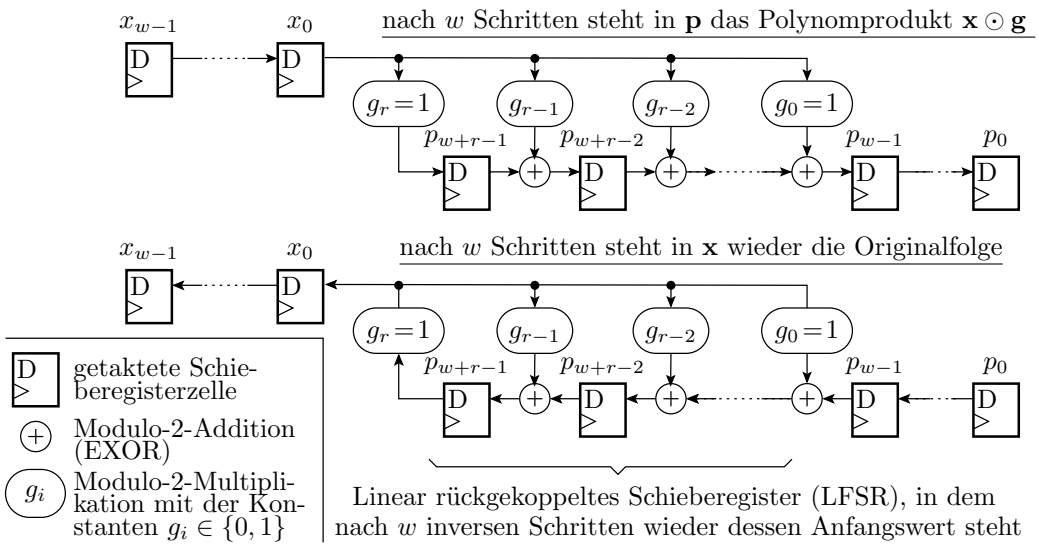
$$\mathbf{p} : \mathbf{g} = \mathbf{x} = \left(z^{14} \oplus z^{10} \oplus z^9 \oplus z^8 \oplus z^5 \oplus z^2 \oplus z \oplus 1\right) : \left(z^4 \oplus z \oplus 1\right) = z^{10} \oplus z^7 \oplus z^5 \oplus z^3 \oplus z^2 \oplus 1$$

liefert ohne Rest das Polynom der Originalfolge.

---

z^i Operator für eine Verschiebung um i Bitstellen.

5.63 Linear rückgekoppelten Schieberegister

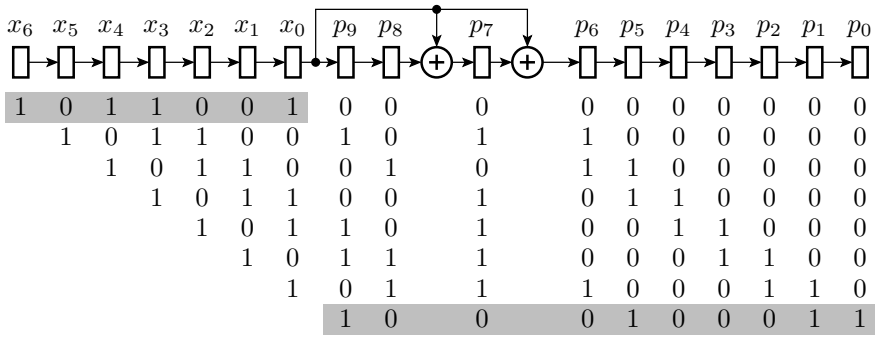


Linear rückgekoppeltes Schieberegister (LFSR), in dem nach w inversen Schritten wieder dessen Anfangswert steht

Schieberegister, vorwärts Multiplikation, rückwärts Division (modulo-2). Fehlererkennende Codes mit LFSR als Coder und Decoder werden als zyklisch redundant (CRC) bezeichnet.

LFSR Linear rückgekoppeltes Schieberegister (Linear feeback shift register).

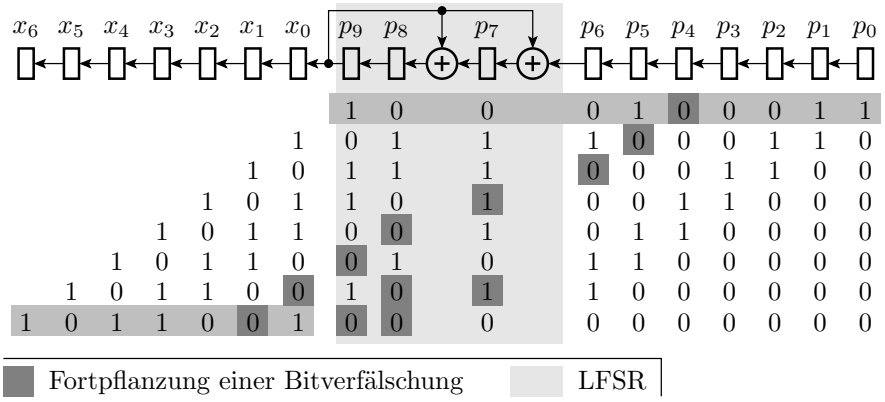
5.64 Zahlenbeispiel



Originalfolge (-polynom):	1011001	$\mathbf{x} =$	$(x^6 \oplus x^4 \oplus x^3 \oplus 1)$
Generatorfolge (-poynom):	1011	$\mathbf{g} =$	$(x^3 \oplus x \oplus 1)$
Produktfolge (-polynom):	1000100011	$\mathbf{p} =$	$(x^9 \oplus x^5 \oplus x \oplus 1)$

Produktfolge um $r = 3$ redundante Bits länger, also 2^3 mal so viele darstellbare wie zulässige Werte. Pseudo-zufällig Umcodierung.

5.65 Rückgewinnung durch Polynomdivision



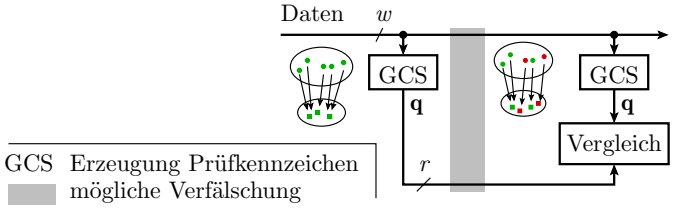
Im LFSR eingegangene Bitverfälschungen zirkulieren dort und werden durch weitere Bitverfälschungen nur mit Wahrscheinlichkeit von 2^{-3} ausgelöscht ($\mu_{\mathrm{MC}} = 1 - 2^{-3}$).

LFSR Linear rückgekoppeltes Schieberegister (Linear feeback shift register).

μ_{MC} Zu erwartende Fehlfunktionsabdeckung.

3.2 Prüfkennzeichen

5.66 Prüfkennzeichen



- Jedem w-Bit-Datenwort wird pseudo-zufällig genau eines der r-Bit-Prüfkennzeichen $\mathbf{q}$ zugeordnet ($w \gg r$).
- Nach der Übertragung oder Speicherung wird das Prüfkennzeichen ein zweites mal gebildet.
- Wenn weder die Daten noch das Prüfkennzeichen verfälscht sind, stimmen beide Prüfkennzeichen überein.

Für pseudo-zufällig gebildete Prüfkennzeichen gilt:

- Anzahl der zulässigen Prüfkennzeichen-Werte-Paare 2^w,
- Anzahl darstellbarer Paare 2^{w+r}:

(1.35) $$\mu_{\mathrm{MC}} \geq 1 - 2^{-r}$$

r	Anzahl der redundanten Bits.
w	Anzahl der Datenbits (Number of data bits).

5.67 Prüfsummen

Prüfkennzeichbildung durch Byteaddition oder EXOR:

einfache Genauigkeit	doppelte Genauigkeit	bitweises EXOR
1011 11	1011 11	1011
0010 2	0010 2	0110
1101 13	1101 13	1101
0100 4	0100 4	1100
(1) 1110 14 (+16)	0001 1110 30	1100

Bei »einfacher Genauigkeit« und »bitweisem EXOR« erscheint die Annahme »pseudo-zufällige Abbildung« gerechtfertigt*:

$$\mu_{\mathrm{MC}} = 1 - 2^{-4}$$

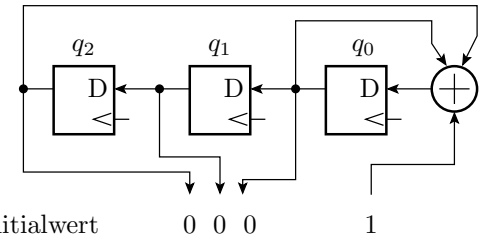
Bei »doppelter Genauigkeit« bilden sich Verfälschungen vorzugsweise auf die niederwertigen Bits ab:

$$1 - 2^{-4} \leq \mu_{\mathrm{MC}} < 1 - 2^{-8}$$

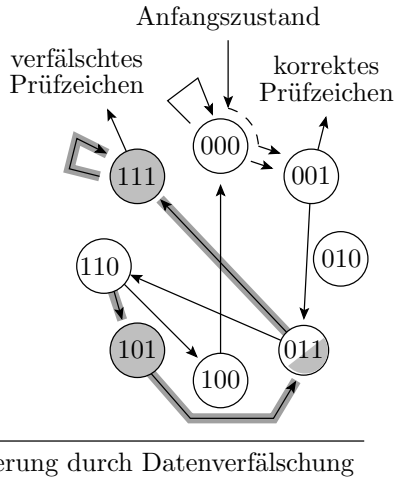
Kein Nachweis für vertauschte Summationsreihenfolge. Falls das eine typ. Verfälschung, zu erwartende Fehlfunktionsabdeckung geringer.

μ_{MC}	Zu erwartende Fehlfunktionsabdeckung.

5.68 Prüfkennzeichenbildung mit LFSR



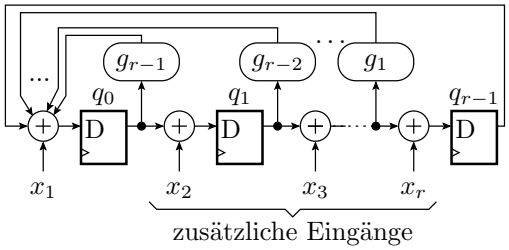
Initialwert	0	0	0	1
Schritt 1:	0	0	1	0
Schritt 2:	0	1	1	1
Schritt 3:	1	1	0	1
Schritt 4:	1	0	0	1
Schritt 5:	0	0	0	0
Schritt 6:	0	0	0	1
Schritt 7:	0	0	1	



Im Beispiel hat das LSFR abweichend von Polynomdivision (Abschn. 5.3.1) eine zentrale Rückführung. Abbildung auch pseudo-zufällig.

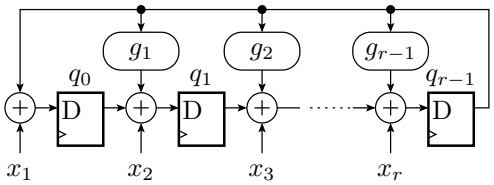
q_i	Registerbit i des linear rückgekoppelten Schieberegisters.
q	Prüfkennzeichen (gesamter Registerzustand).
LFSR	Linear rückgekoppeltes Schieberegister (Linear feeback shift register).

5.69 Parallele Signaturregister



LFSR zur Prüfkennzeichenbildung werden als Signaturregister bezeichnet. Das auf Folie mit nur einem Eingang ist ein serielles Signaturregister. Parallele Signaturregister addieren in jedem Schaltschritt mehrere Eingabebits modulo-2 zu den Registerzuständen. Abbildung auch (in der Regel) pseudo-zufällig hinsichtlich zu erwartender Verfälschungen.

x_i	Mehrere parallel eingespeiste LFSR-Eingabebits.
g_i, q_i	Rückführkoeffizienten und Registerbit i des linear rückgekoppelten Schieberegisters.
r	Bitanzahl des linear rückgekoppelten Schieberegisters.



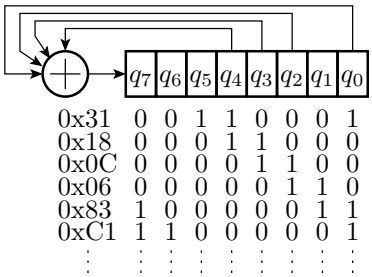
Die Rückführung darf wie bei der Polynom-Division auch dezentral sein. Die Koeffizienten g_i der Rückführung, bei der Polynom-Division das Divisor-Polynom, bestimmen die autonome Zyklusstruktur*. Die autonome Zyklusstruktur ist bei zentraler und dezentraler Rückführung mit denselben Rückführkoeffizienten gleich.

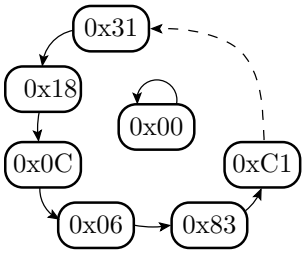
Bevorzugt werden lange Zyklen, insbesondere sog. primitive Polynome, bei denen alle Zustände außer »alles null« einen $2^r - 1$ langen Maximalzyklus bilden.

Ein über die Streuung hinausgehender Einfluss von Rückführung, Eingangsanzahl und anderer struktureller Freiheitsgrade außer Bitanzahl r auf die Fehlfunktionsabdeckung ist kaum nachzuweisen (Folie 6.82).

r	Bitanzahl des linear rückgekoppelten Schieberegisters.
*	Zyklusstruktur ohne Eingaben.

5.71 Autonome Zyklusstruktur von LFSR

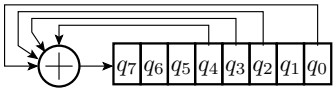




Autonom bedeutet ohne Eingabe oder Eingabefolge »alles 0«. Der Zustand 00...0 geht immer in sich selbst über. Das Beispiel-8-Bit-LFSR hat eine primitive Rückführung, so dass alle anderen Zustände einen 2^r-1 Schritte langen Zyklus bilden. Allgemein gibt es mehr als zwei Zyklen. Schieberegisterringe, bei denen nur die letzte auf die erste Zelle rückgeführt ist, haben nur r-Bit lange Zyklen und gelten als weniger gut geeignet.

q_i Registerbit i des linear rückgekoppelten Schieberegisters.

5.72 Zyklusstruktur durch Simulation

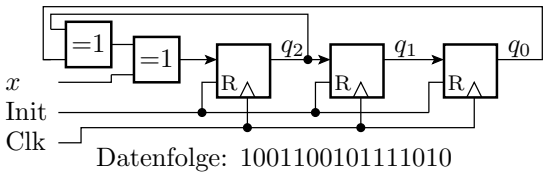



```
...
#define SA 0x31
uint8_t q = SA;      // set initial value
while(1){
  q = (q>>1) ^ (q<<7) ^ ((q<<5)&0x80)
      ^ ((q<<4)&0x80) ^ ((q<<3)&0x80);
  <Ausgabe von s>
  if (q==SA) break; // abort when the initial
  ...               // state is reached again
}
```

- 0x00 geht in sich selbst über.
- Alle anderes 255 Zustände gehen zyklisch ineinander über.
- max. Zykluslänge $2^r-1 \Rightarrow$ primitive Rückkopplung.

Beispiel 5.3 Prüfkennzeichen mit LFSR

Gegeben ist folgendes linear rückgekoppelte Schieberegister:



Datenfolge: 1001100101111010

a) *Welches Prüfkennzeichen* $\mathbf{q} = q_2q_1q_0$ *hat die Datenfolge. Abbildung beginnend mit dem linken Bit. Startwert 000?*

	x	q_2	q_1	q_0
0	1	0	0	0
1	0	1	0	0
2	0	1	1	0
3	1	1	1	1
4	1	1	1	1
5	0	1	1	1
6	0	0	1	1
7	1	1	0	1

	x	q_2	q_1	q_0
8	0	1	1	0
9	1	1	1	1
10	1	1	1	1
11	1	1	1	1
12	1	1	1	1
13	0	1	1	1
14	1	0	1	1
15	0	0	0	1

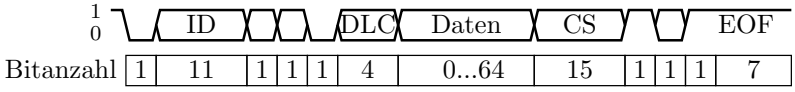
Prüfkennzeichen: **1 0 0**

b) *Wie hoch ist die zu erwartende Fehlfunktionsabdeckung?*

$$\mu_{\mathrm{MC}} = 1 - 2^{-3} = 87{,}5\%$$

q_i	Registerbit i des linear rückgekoppelten Schieberegisters.
x	Eingang zur Abtastung der Bitfolge am Testobjektausgang.
R	Rücksetzeingang.
Init	Initialisierungssignal.
Clk	Taktsignal (Clock signal).
μ_{MC}	Zu erwartende Fehlfunktionsabdeckung.

5.74 Beispielanwendung CAN-Bus



ID	Nachrichtennummer	CS	15-Bit Prüfkennzeichen
DLC	Datenlängencode	EOF	Ende des Datenrahmens

Anwendung: Vernetzung von Fahrzeugsteuergeräten, ...

Prüfkennzeichen (CS): Länge $r = 15$ Bit:

$$\mu_{\mathrm{MC}} = 1 - 2^{-15} \approx 1 - 3 \cdot 10^{-5}$$

Fehlerfunktionsbehandlung:

- Sendeabbruch bei Bitverfälschung (Nachrichtenkollision) und verzögerte Wiederholung.
- Empfangsfehler: Protokollierung in einem Fehlerspeicher. Je nach Problemschwere, Warnung, Notbetrieb oder Notabschaltung.

5.75 5.75 Beispielanwendung Ethernet

Sicherungs-schicht			MAC-Empfänger	MAC-Absender	Protokoll-typ	Nutzlast max. 1500 Bytes	Prüfkennzeichen 4 Byte	
Bitübertra-gungsschicht	Präambel	Start-byte						Lücke zum nächsten Paket
Byteanzahl	7	1	6	6	2	46 bis 1500	4	12

Ein Ethernet-Datenpaket hat ein 4-Byte-Prüfkennzeichen:

$$\mu_{\mathrm{FC}} = 1 - 2^{-32} \approx 1 - 2 \cdot 10^{-10}$$

Fehlerfunktionsbehandlung: Neuanforderung.

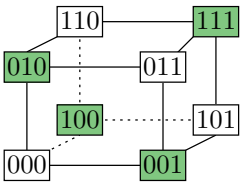
Beispielabschätzung der mittleren Zeit zwischen zwei nicht erkannten verfälschten Empfangsdatenpaketen bei im Mittel einem verfälschten Datenpaket je Sekunde:

$$\frac{1\,\mathrm{s}}{2^{-32}} > 135\,\mathrm{Jahre}$$

Die meisten Nutzer / genutzten Programme werden nie mit einem nicht erkannten verfälschten Empfangsdatenpaket konfrontiert.

3.3 Hamming-Codes

5.76 Hamming-Codes



Die mit LFSR erzeugten Codes gehören zu den zyklisch-redundanten Codes (CRC cyclic redundancy codes) und sind auf zufällige Verfälschungen ausgerichtet.

Hammingcodes werden durch die Hammingdistanz $\#HD$ beschrieben. Das ist die Anzahl der Bitpositionen, in denen sich zwei zulässige Codeworte mindestens unterscheiden. Hammingdistanzen von 2 oder mehr garantiert, dass eine 1-Bit Verfälschung nicht zu einem anderen gültigen Codewort führt. Erforderliche Hamming-Distanz zum Erkennen von Verfälschungen mit bis zu $\#DB$ verfälschten Bits:

$$\#HD \geq \#DB + 1 \tag{5.8}$$

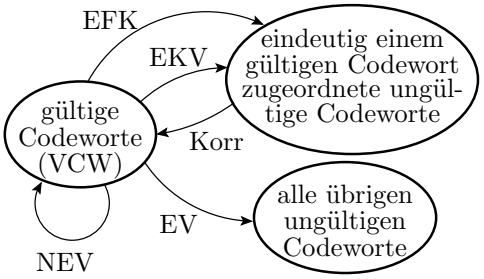
Erforderliche Hamming-Distanz zur Korrektur von Verfälschungen mit bis zu $\#CB$ verfälschten Bits:

$$\#HD \geq 2 \cdot \#CB + 1 \tag{5.9}$$

$\#HD$ Hamming-Distanz.

5.77 Fehlerkorrektur mit Hamming-Code

EFK	erkennbar, aber falsch korrigierte Datenverfälschung
EKV	erkennbare und korrigierbare Datenverfälschung
EV	erkennbare, nicht korrigierbare Datenverfälschung
NEV	nicht erkennbare Datenverfälschung
Korr	Korrektur



Erweiterung der Menge der darstellbaren Codeworte um eine viel größere Menge korrigierbarer Codeworte und optional um unzulässige nicht korrigierbare Codeworte. Mindestbitanzahl:

$$2^{\#\text{Bit}} \geq \#VCW + \#VCW \cdot \#CVC \tag{5.10}$$

$\#Bit$ Bitanzahl des Codeworts.
$\#VCW$ Anzahl der gültigen Codeworte.
$\#CVC$ Anzahl korrigierbare Codeworte je gültiges Codewort.

5.78 Beispiel: Korrektur von Einzelbitfehler

(5.10) $2^{\#\text{Bit}} \geq \#VCW + \#VCW \cdot \#CVC$

Anzahl korrigierbare Codeworte je gültiges Codewort gleich Bitanzahl:

$$\#CVC = \#\text{Bit}$$

Mindestbitanzahl:

$$2^{\#\text{Bit}} \geq \#VCW + \#\text{Bit} \cdot \#VCW = (\#\text{Bit} + 1) \cdot \#VCW$$

Für #VCW = 256 gültige Codeworte:

$$\begin{aligned} 2^{\#\text{Bit}} &\geq 256 \cdot (1 + \#\text{Bit}) \\ \#\text{Bit} &\geq 12 \end{aligned}$$

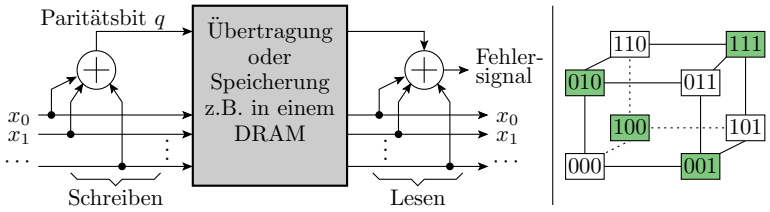
Probe:

$$2^{12} > 2^8 \cdot (1 + 12)$$

$\#Bit$ Bitanzahl des Codeworts.
$\#VCW$ Anzahl der gültigen Codeworte.
$\#CVC$ Anzahl korrigierbare Codeworte je gültiges Codewort.

3.4 Paritätstest

5.79 Paritätsbit als Prüfkennzeichen ($\#HD = 2$)



Paritätsbit q gleich modulo-2 Summe (EXOR) der Datenbits:

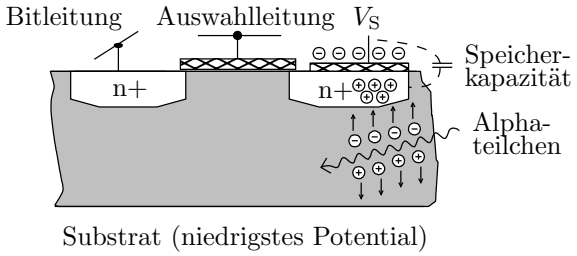
$$q = x_{n-1} \oplus x_{n-2} \oplus \ldots \oplus x_1 \oplus x_0$$

bei gerader Anzahl von Einsen »0« sonst »1«.

Erkannt werden alle ungeradzahligen Bitverfälschungen. Wenn Fehlfunktionen weder geradzahlige noch ungeradzahlige Verfälschungen bevorzugen, beträgt die zu erwartende Fehlfunktionsabdeckung nach (Gl. 1.35) $\mu_{\mathrm{MC}} = 50\%$. In seinen Anwendungsbereichen ist die Fehlfunktionsabdeckung viel größer.

μ_{MC}, r Zu erwartende Fehlfunktionsabdeckung, Anzahl der redundanten Bits.

5.80 Paritätstest für DRAMs und Speicherriegel



- Informationsspeicherung in winzigen Kapazitäten.
- Häufigste Ursache für Datenverfälschungen: Alphastrahlung.
- Deren Quellen radioaktiver Zerfall von Uran und Thorium, enthalten als Spurenelemente im Gehäuse und im Aluminium der Leiterbahnen oder Kernprozesse im Silizium durch Höhenstrahlung.
- Mittlerer Ereignisabstand: Stunden, Tage.

V_S Versorgungsspannung.

- Energie eines Alphateilchen: 5 MeV. Energieverlust bei der Generierung eines Elektronen-Loch-Paares $\approx 3{,}6\,\mathrm{eV} \Rightarrow$ Generierung von $\approx 10^6$ Ladungsträgerpaaren. Reichweite $\approx 89\,\mu$m. gespeicherte Ladung $\approx 10^5$ Ladungsträger. Datenverlust einer oder mehrerer benachbarter Zellen möglich.
- Gleichzeitige Verfälschung durch zwei Alphateilchen unwahrscheinlich.
- Geometrische Trennung der Zellen eines Datenworts (getrennte Schaltkreise oder Speichermatrizen) $\Rightarrow$ Je Zerfall Einzelbitverfälschung je gelesenes Datenwort.

Auch bei der Übertragung sind Verfälschungen selten. Datenobjekte so verschränken, das auch Fehler-Bursts auf Einzelbitfehler je Datenobjekt abgebildet werden.

5.82 Nachweis seltenere Bitverfälschungen

Für seltene unabhängige Bitverfälschungen ist die Anzahl X der verfälschten Bits je Datenobjekt poissonverteilt:

$$\mathbb{P}[X=k] = \mathrm{e}^{-\lambda} \cdot \frac{\lambda^k}{k!} \tag{4.40}$$

Die zu erwartende Fehlfunktionsüberdeckung ist mindests so groß, wie die bedingte Wahrscheinlichkeit, dass genau ein Bit verfälscht ist, wenn nicht null Bits verfälscht sind:

$$\mu_{\mathrm{MC}} \geq \frac{\mathbb{P}[X=1]}{1-\mathbb{P}[X=0]} = \frac{\mathrm{e}^{-\lambda}\cdot\lambda}{1-\mathrm{e}^{-\lambda}} \overset{(\lambda \ll 1)^*}{=} \frac{(1-\lambda)\cdot\lambda}{1-(1-\lambda)} = 1-\lambda$$

Zu erwartende Anzahl der Bitfehler je Datenwort:

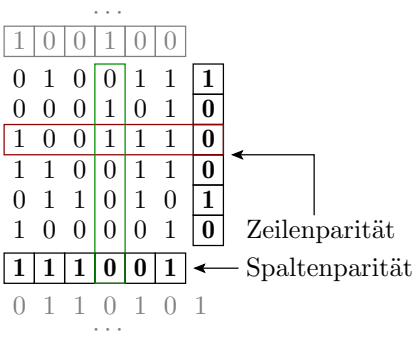
$$\begin{aligned} \lambda &= \zeta_{\mathrm{Bit}} \cdot w \\ \mu_{\mathrm{MC}} &\geq 1 - \zeta_{\mathrm{Bit}} \cdot w \end{aligned} \tag{5.11}$$

Bei geringer Bitfehlerrate ζ_{Bit} und geringer Bitanzahl $w \gg 50\%$.

λ	Zu erwartende Anzahl der Bitfehler je Datenwort.
μ_{MC}	Zu erwartende Fehlfunktionsabdeckung.
ζ_{Bit}, w	Bitfehlerrate, Anzahl der Bit je Datenwort.
*	Taylor-Reihe $\mathrm{e}^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots$. Approximation bis zum linearen Glied.

3.5 Einzelbitkorrektur

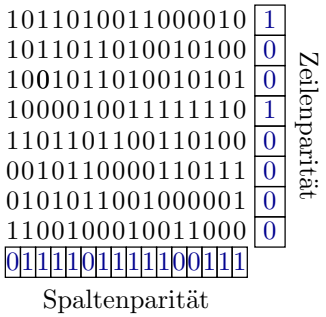
5.83 Kreuzparität



Datenorganisation in einem 2-dimensionalen Array. Paritätsbildung für alle Zeilen und Spalten. Erlaubt Lokalisierung und Korrektur von 1-Bit Fehlern. Einsatz in redundanten Festplatten-Arrays (RAID 3 und 5).

Wenn mindestens ein falsches, aber nicht genau ein falsches Zeilen- und ein falschen Spaltenparitätsbit, Verfälschung nicht korrigierbar.

Beispiel 5.4 Kreuzparität



a) *Kontrolle der Zeilen- und Spaltenparität?*

1011010011000010	1	
1011011010010100	0	
1001011010010101	0	
1000010011111110	1	
1101101100110100	**0**	
0010110000110111	0	
0101011001000001	0	
1100100010011000	0	
0 1 1 1 1 0 **1** 1 1 1 1 0 0 1 1 1		

Querparität

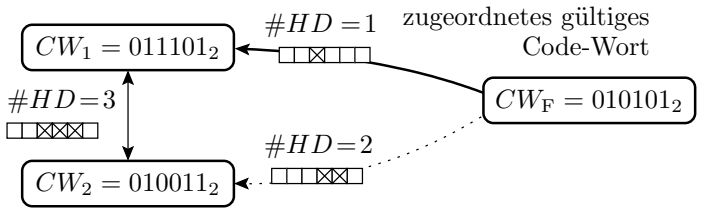
Längsparität

b) *Liegt keine, eine erkennbare nicht korrigierbare oder eine erkenn- und korrigierbare Verfälschung vor?*

Korrigierbar durch Invertierung des Bits in Zeile 5, Spalte 7 (null setzen).

5.85 1-Bit fehlerkorrigierende Hamming-Codes

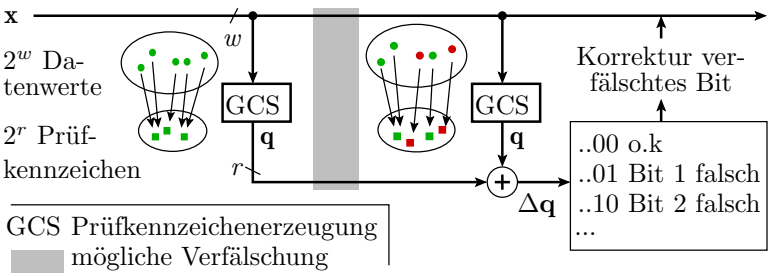
Ab einem Hamming-Abstand $\#HD \geq 3$ ist jede 1-Bit-Verfälschung eindeutig einem gültigen Codewort zugeordnet.



Korrektur durch Ersatz des verfälschten Codeworts durch das mit Hamming-Distanz $\#HD = 1$. Bei Hamming-Distanz $\#HD = 3$ werden Codeworte mit zwei oder mehr verfälschten Bits falschen gültigen Codeworten zugeordnet.

$\#HD$	Hamming-Distanz.
CW_i	Code-Wort i.
CW_F	verfälschtes Code-Wort.

5.86 Beispiel für die Codekonstruktion



Erzeugung des r-Bit Prüfkennzeichens **q** so aus dem Datenwort **x**, dass die mod-2 Summen **Δq** des: übertragenen und des danach gebildeten Prüfkennzeichens die Nummer des verfälschten Bits ergibt:

Δq	..000	..001	..010	..011	..100	..101	...
verfälschtes Bit	–	1	2	3	4	5	...

q, **x**	r-Bit Prüfkennzeichen, w-Bit Datenwort.
Δq	EXOR-Differenz der erhaltenen und der neu berechneten Prüfbits.

5.87 Konstruktion der Prüfkennzeichen

Anzahl der Prüfbits r nach Gl. 5.10 für $w = 8$:

$$2^{w+r} \geq \left(1 + \underbrace{w+r}_{\text{Anz. Einzelbitfehler}}\right) \cdot \underbrace{2^w}_{\#VCW} = (1+12)\cdot 2^8;\ r = 4$$

$\Delta\mathbf{q}$	..000	..001	..010	..011	..100	..101	...
verfälschtes Bit	–	1	2	3	4	5	...

$$\begin{aligned}
\Delta q_0 &= \underline{b_1} \oplus b_3 \oplus b_5 \oplus b_7 \oplus b_9 \oplus b_{11} \\
\Delta q_1 &= \underline{b_2} \oplus b_3 \oplus b_6 \oplus b_7 \oplus b_{10} \oplus b_{11} \\
\Delta q_2 &= \underline{b_4} \oplus b_5 \oplus b_6 \oplus b_7 \oplus b_{12} \\
\Delta q_3 &= \underline{b_8} \oplus b_9 \oplus b_{10} \oplus b_{11} \oplus b_{12}
\end{aligned}$$

Die niedrigsten Bits jeder Prüfsumme werden Prüfbits q_i, die anderen als Datenbits x_i. Zuordnung der b_i zu x_i und q_i ...

---

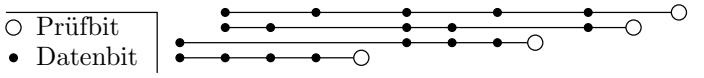
w, r Anzahl der Datenbits, Anzahl der Prüfbits.

b_i Bit i des Gesamtcodeworts.

Δq_i Prüfkennzeichendifferenz Bit i.

5.88 Zuordnung der Bitnummern

Gesamtwort	b_{12}	b_{11}	b_{10}	b_9	b_8	b_7	b_6	b_5	b_4	b_3	b_2	b_1
Zuordnung	x_7	x_6	x_5	x_4	q_3	x_3	x_2	x_1	q_2	x_0	q_1	q_0



$$\begin{aligned}
\Delta q_0 &= b_1 \oplus b_3 \oplus b_5 \oplus b_7 \oplus b_9 \oplus b_{11} &&= q_0 \oplus x_0 \oplus x_1 \oplus x_3 \oplus x_4 \oplus x_6 \\
\Delta q_1 &= b_2 \oplus b_3 \oplus b_6 \oplus b_7 \oplus b_{10} \oplus b_{11} &&= q_1 \oplus x_0 \oplus x_2 \oplus x_3 \oplus x_5 \oplus x_6 \\
\Delta q_2 &= b_4 \oplus b_5 \oplus b_6 \oplus b_7 \oplus b_{12} &&= q_2 \oplus x_1 \oplus x_2 \oplus x_3 \oplus x_7 \\
\Delta q_3 &= b_8 \oplus b_9 \oplus b_{10} \oplus b_{11} \oplus b_{12} &&= q_3 \oplus x_4 \oplus x_5 \oplus x_6 \oplus x_7
\end{aligned}$$

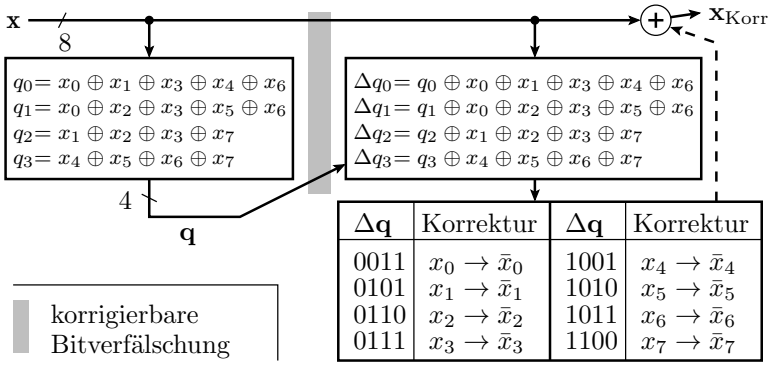
Ohne Verfälschung $\Delta q_i = 0$. Umstellung nach q_i:

$$\begin{aligned}
q_0 &= x_0 \oplus x_1 \oplus x_3 \oplus x_4 \oplus x_6 \\
q_1 &= x_0 \oplus x_2 \oplus x_3 \oplus x_5 \oplus x_6 \\
q_2 &= x_1 \oplus x_2 \oplus x_3 \oplus x_7 \\
q_3 &= x_4 \oplus x_5 \oplus x_6 \oplus x_7
\end{aligned}$$

---

b_i, x_i, q_i Gesamtbit i, Datenbit i, Prüfbit i.

5.89 Codier-, Erkennungs- und Korrekturschaltung



$\Delta\mathbf{q}$	Korrektur	$\Delta\mathbf{q}$	Korrektur
0011	$x_0 \to \bar{x}_0$	1001	$x_4 \to \bar{x}_4$
0101	$x_1 \to \bar{x}_1$	1010	$x_5 \to \bar{x}_5$
0110	$x_2 \to \bar{x}_2$	1011	$x_6 \to \bar{x}_6$
0111	$x_3 \to \bar{x}_3$	1100	$x_7 \to \bar{x}_7$

- Gleiches Prüfbit-Bildung vor und nach Speichern/ Übertragung.
- Differenzbildung durch bitweises EXOR.
- Wenn $\Delta\mathbf{q} \neq 0$ und Datenbit verfälscht, Bit »$\Delta\mathbf{q}$« invertieren.

Beispiel 5.5 Fehlerkorrigierender Code

b_{12}	b_{11}	b_{10}	b_9	b_8	b_7	b_6	b_5	b_4	b_3	b_2	b_1
x_7	x_6	x_5	x_4	q_3	x_3	x_2	x_1	q_2	x_0	q_1	q_0

$$q_0 = x_0 \oplus x_1 \oplus x_3 \oplus x_4 \oplus x_6 \qquad q_2 = x_1 \oplus x_2 \oplus x_3 \oplus x_7$$
$$q_1 = x_0 \oplus x_2 \oplus x_3 \oplus x_5 \oplus x_6 \qquad q_3 = x_4 \oplus x_5 \oplus x_6 \oplus x_7$$

a) *Bestimmung des Codeworts* $\mathbf{b}$ *für den Datenwert* $\mathbf{x} = 0x8B$.

Bitnummer	12	11	10	9	8	7	6	5	4	3	2	1	
Zuordnung	x_7	x_6	x_5	x_4	q_3	x_3	x_2	x_1	q_2	x_0	q_1	q_0	
Kontrollbits	=	=	=	=		=	=	=	=	=	=	=	
$\mathbf{x}$ = 0x8B													$\mathbf{b}$ =
$\mathbf{b}$ = 0xA9B													$\Delta\mathbf{q}$ =
korrigiert													$\mathbf{x}$ =

Bitnummer	12	11	10	9	8	7	6	5	4	3	2	1	
Zuordnung	x_7	x_6	x_5	x_4	q_3	x_3	x_2	x_1	q_2	x_0	q_1	q_0	
Kontrollbits	=	=	=	=		=	=	=	=	=	=	=	
$\mathbf{x}$ = 0x8B	1	0	0	0	1	1	0	1	1	1	0	1	$\mathbf{b}$ = 0x8DD

Der Wert 0x8B hat das Codewort 0x8DD.

b) *Extrahieren des Werts* $\mathbf{x}$ *aus dem Codewort* $\mathbf{b} = 0xA9B$*?*

Bitnummer	12	11	10	9	8	7	6	5	4	3	2	1	
Zuordnung	x_7	x_6	x_5	x_4	q_3	x_3	x_2	x_1	q_2	x_0	q_1	q_0	
Kontrollbits	=	=	=	=		=	=	=	=	=	=	=	
$\mathbf{b}$ = 0xA9B	1	0	1	0	1	0	0	1	1	0	1	1	$\Delta\mathbf{q} = 12_{10}$
korrigiert	0	0	1	0		0	0	1		0			$\mathbf{x}$ = 0x22

Im Codewort 0x49B steckt der Wert 0xA2 mit $\Delta\mathbf{q}$ = 0b1100 = 12. Das verfälschte Bit 12 im Codewort ist Datenbit x_7. Invertierung von x_7 ergibt den Datenwert 0x22.

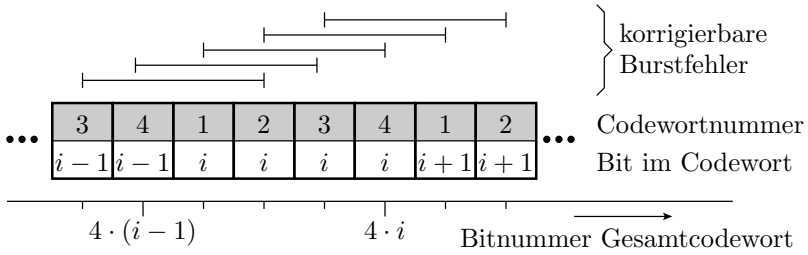
3.6 Burst-Verfälschung

5.91 Burst-Verfälschungen, Verschränkung

Burst-Verfälschungen

Verfälschung einer Folge von bis zu m aufeinanderfolgenden Bits. Typisch für Lesefehler von CDs und Übertragungsfehler.

Zusammensetzen eines fehlerkorrigierenden Codes für m-Bit Burst-Verfälschungen für eine $m \cdot n$ Bit lange Folgen aus m fehlerkorrigierenden Codeworten für 1-Bit-Fehler für n Bit lange Folgen durch Verschränkung.



Beispiel 5.6 Burst-Verfälschungen

a) *Codierung Datenfolge 0x8B, 0x22, 0x9C so, dass bis zu 3-Bit lange Burst-Verfälschungen korrigierbar sind, durch Verschränkung von je drei aufeinanderfolgende Codeworten für Einzelbitfehlerkorrektur?*

Bitnummer	12	11	10	9	8	7	6	5	4	3	2	1
Zuordnung	x_7	x_6	x_5	x_4	q_3	x_3	x_2	x_1	q_2	x_0	q_1	q_0
Kontrollbits	=	=	=	=	=	=	=	=	=	=	=	=
x_1 = 0x8B												
x_2 = 0x22												
x_3 =0x9C												

Bits 30 bis 32

Bitnummer	12	11	10	9	8	7	6	5	4	3	2	1
Zuordnung	x_7	x_6	x_5	x_4	q_3	x_3	x_2	x_1	q_2	x_0	q_1	q_0
Kontrollbits	=	=	=	=	=	=	=	=	=	=	=	=
x_1 = 0x8B	1	0	0	0	1	1	0	1	1	1	0	1
x_2 = 0x22	0	0	1	0	1	0	0	1	1	0	1	1
x_3 =0x9C	1	0	0	1	0	1	1	0	1	0	0	0

b) *Zeigen Sie, dass eine Invertierung der Bits 30 bis 32 korrigiert wird?*

Bitnummer	12	11	10	9	8	7	6	5	4	3	2	1	
Zuordnung	x_7	x_6	x_5	x_4	q_3	x_3	x_2	x_1	q_2	x_0	q_1	q_0	
Kontrollbits	=	=	=	=	=	=	=	=	=	=	=	=	
x_1 = 0x8B	1	0	0	0	1	1	0	1	1	1	0	1	$\Delta\mathbf{q} = 1011_2$
x_2 = 0x22	0	0	1	0	1	0	0	1	1	0	1	1	$\Delta\mathbf{q} = 1011_2$
x_3 =0x9C	1	0	0	1	0	1	1	0	1	0	0	0	$\Delta\mathbf{q} = 1010_2$

Durch Burstfehler invertierte Bits 30 bis 32,

3.7 Zusammenfassung

5.93 Codebasierte Kontrollverfahren

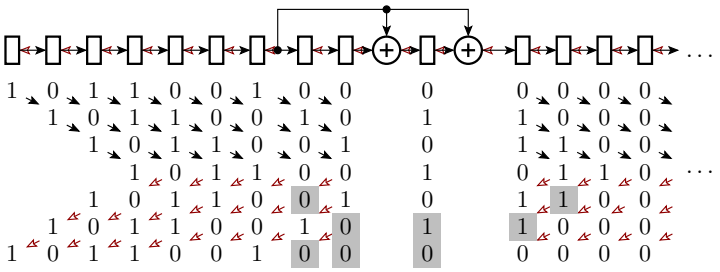
Formatkontrolle mit r redundanten Bits ist perfekt, wenn garantiert werden kann, dass Verfälschungen durch Fehlfunktionen gleichmäßig auf zulässige und erkennbar unzulässige Werte abgebildet und alle unzulässigen Werte erkannt werden:

(1.35) $$\mu_{\mathrm{MC}} \geq 1 - 2^{-r}$$

Erfordert in der Regel geeignete Codierung oder Prüfkennzeichen und eignet sich deshalb nur für die Datenaufbewahrung und Weitergabe, nicht aber für berechnete Ergebnisse:

- Fehlererkennende Codes: Pseudozufällige Codierung unter Erhöhung der Bitanzahl. Decodierung und Kontrolle mit Umkehralgorithmus der Codierung.
- Prüfkennzeichen: Pseudozufällige Bildung eines r-Bit Kennzeichens aus dem viel größeren Datenobjekt. Anhängen an das Datenobjekt. Kontrolle durch wiederholte Kennzeichbildung und Vergleich mit dem angehängten Kennzeichen.

5.94 Schieberegister als Coder und Decoder



Für umgekehrbare pseudo-zufällige Abbildung mit Bitanzahlerhöhung gibt viele Möglichkeiten, z.B. die Multiplikation mit einer großen Konstanten. Aufwandsgünstiger und genauso wirkungsvoll ist die zyklisch redundante Codierung und Decodierung mit Schieberegistern. Auch mit den Maschinenbefehlern von Rechnern programmierbar.

Zur Prüfkennzeichenbildung eignet sich das linear rückgekoppelte Schieberegister (LFSR) im Rückwärtsbetrieb im Bild, ein andere rückgekoppelte Schieberegister oder Prüfsummen.

5.95 Hamming-Codes

Gültigen Codeworte eines Hammingcodes unterscheiden sich in $\#HD > 1$ Bits ($\#HD$ – Hammingabstand). Dazu wird das uncodierte Codeworte um ein Prüfkennzeichen aus lineare Summen ergänzt.

Der Nachweis von bis zu $\#DB$ verfälschten Bits bzw. für die Korrektur von bis zu $\#CB$ Bits verlangt Hamming-Distanzen von:

(5.8) $$\#HD \geq \#DB + 1$$

(5.9) $$\#HD \geq 2 \cdot \#CB + 1$$

Genutzt werden hauptsächlich

- Paritätstest für den Nachweis von Einzelbitverfälschungen und
- fehlerkorrigierende Codes für Einzelbitverfälschungen.

Beides setzt geringe Bitverfälschungsraten und Unabhängigkeit der Verfälschungen voraus, sicherzustellen durch geringe Bitverfälschungsraten, kleine Codeworte und geeignete Zusammenfassung physikalischer Bits zu Code-Worten.

5.96 Paritätstest

Ergänzung aller Datenworte um eine Paritätsbit gleich EXOR aller Datenbits. Hamming-Distanz $\#HD = 2$. Kontrolle durch EXOR aller Datenbits und Vergleich mit dem Paritätsbit.

Erkannte werden ungeradzahligen incl. Einzelbitverfälschungen. Fehlfunktionsüberdeckung für unabhängige Bitverfälschungen und eine zu erwartende Anzahl verfälschter Bits je Datenwort $\zeta_{\mathrm{Bit}} \cdot w \ll 1$:

(5.11) $$\mu_{\mathrm{MC}} \geq 1 - \zeta_{\mathrm{Bit}} \cdot w$$

Einsatzbeispiel:

- Erkennen umgekippte Bits in DRAMs durch Radioaktivität. Verlangt das die Codeworte aus Bits räumlich getrennter Speicherblöcke zusammengesetzt werden.
- Übertragung kleiner Datenobjekte mit geringer Bitfehlerrate.

Kreuzparität: Mit einer Zeilen- und Spaltenparität für zweidimensionale Bitfelder lässt sich sogar eine einzelne Bitverfälschung lokalisieren und korrigieren. Anschaulichster fehlerkorrigierender Code.

5.97 Fehlerkorrigierende Hamming-Codes

Bitnummer	12	11	10	9	8	7	6	5	4	3	2	1	
Zuordnung	x_7	x_6	x_5	x_4	q_3	x_3	x_2	x_1	q_2	x_0	q_1	q_0	
Kontrollbits	=	=	=	=	=	=	=	=	=	=	=	=	
$\mathbf{b}$ = 0xA9B	1	0	1	0	1	0	0	1	1	0	1	1	$\Delta\mathbf{q} = 12_{10}$
korrigiert	0	0	1	0		0	0	1		0			$\mathbf{x}$ = 0x22

Erweiterung der Menge der darstellbaren Codeworte um eine viel größere Menge korrigierbarer Codeworte und optional um unzulässige nicht korrigierbare Codeworte. Mindestbitanzahl:

(5.10) $$2^{\#\text{Bit}} \geq \#VCW + \#VCW \cdot \#CVC$$

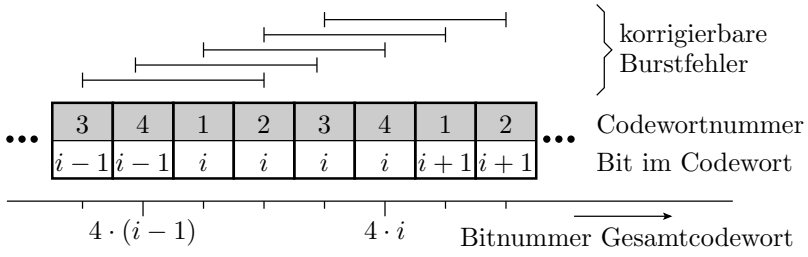
Beispielcode:

- 8 Datenbits, $\#VCW = 256$, 4 Kontrollbits, $\#\text{Bit} = \#CVW = 12$

$$12 + 256 \cdot 12 = 3328 < 2^{12} = 4096\surd$$

- Zahl der Kontrobitabweichung gleich Nummer verfälschtes Bit.

5.98 Burst-Verfälschungen



Bei Betrachtung typischer Verfälschungsursachen

- Kratzer auf einer CD,
- elektromagnetische Übertragung über Leitungen oder Funk und
- radioaktiven Zerfall

sind nicht vereinzelte unabhängige verfälschte Einzelbits, sondern vereinzelte Burst-Verfälschungen begrenzter Länge typisch.

Verschränkung. Fehlerkorrigierender Code für Einzelbitverfälschungen.

4 Ergebniskontrolle

5.99 Kontrollen für Berechnungsergebnisse

Fehlererkennende Codes und Prüfkennzeichen eignen sich nur zur Kontrolle auf Unverändertheit. Kontrollverfahren für berechnete Ergebnisse (Abschn. 1.2.3):

- Ausgabevergleich mit diversitären Ergebnissen (Master-Checker),
- Eingabevergleich mit rückgewonnenen Eingaben (Loop-Test),
- Formatkontrollen (Typ, Wertebereich, ...), ...

Für Merkmale, die durch statische Tests oder Compilierung zugesichert werden (Typen, Wertebereiche, Gleichheit, ...) sind Kontrolle zur Laufzeit überflüssig.

Kontrollen auf Gleichheit großer Datenobjekte lassen sich durch Prüfkennzeichenbildung und -vergleich (Hash-Werte oder Prüfsummen) vereinfachen. Bei ausreichend großen Prüfkennzeichen ist die Verschlechterung der Fehlfunktionsüberdeckung vernachlässigbar.

5.100 Die Güte regelbasierter Kontrollen

Die typischen Formatkontrollen für Ergebnisse (Typ, Wertebereich, ...) und die gleichwertigen statischen Tests erkennen bzw. schließen in der Regel alle Regelverletzungen aus, auf die sie abzielen. Die zu erwartende Fehlfunktionsabdeckung ist der Anteil der Fehlfunktionen, die die Regel verletzen:

$$\mu_{\mathrm{MC}} = \frac{\#RV}{\#MF}$$

Oft nicht besser als $\mu_{\mathrm{MC}} \leq 20\% \ldots 90\%$ je Kontrolle.

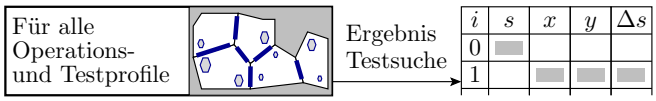
Hohe Fehlfunktionsabdeckung verlangt Kontrollen für viele verschiedene Regeln.

Der Einbau vieler Kontrollen verlangt Automatisierung und diese rechnerlesbare Beschreibungen für die zu kontrollierenden Merkmale.

μ_{MC} Zu erwartende Fehlfunktionsabdeckung.

$\#RV$, $\#MF$ Experimentell bestimmte Zählwerte für Regeleverletzungen und Fehlfunktionen.

5.101 Kontrolle von Testausgaben



Software sollte mit vielen Zufallstests einsprechend Nutzungs- und unterschiedlichen Testprofile getestet werden (Abschn. 2.3.8). Die zu kontrollierenden Ergebnisse je Testschritt sind »kein Absturz«, Fehlfunktionsbehandlung oder Ausgaben y_i und Objektraumänderungen Δs_i.

Es gibt keine fehlerfreie Beschreibung für die Ableitung von Sollwerten. Ohne automatisierte Kontrollen sind die Ist-Ergebnisse manuell durch Inspektion zu kontrollieren. Aufwand groß, MF-Abdeckung gering.

Umfangreiche Tests setzen Kontrollautomatisierung voraus.

4.1 Zusicherungen, OCL

5.102 Zusicherungen

Eine Sprache für einzuhaltende Merkmale für Berechnungsergebnisse ist OCL, eine logische Programmiersprache und Teil von UML:

```
context <classe>::<methode>  -- (1)
  pre: <expr>  -- Vorbedingungen
  inv: <expr>  -- Invarianten
  post: <expr> -- Nachbedingung
```

(1) UML-Beschreibungselement, z.B. eine Methode einer Klasse, für die etwas zugesichert wird; <expr> logische Ausdrücke; Vorbedingung müssen beim Funktionseintritt gelten, Nachbedingungen nach Funktionsabschluss und Invarianten nach jedem Ausführungsschritt:

```
context func   -- func(int* age, int* count)
inv: age>0 and age<100    - (2)
post: count = count@pre+1 - (3)
```

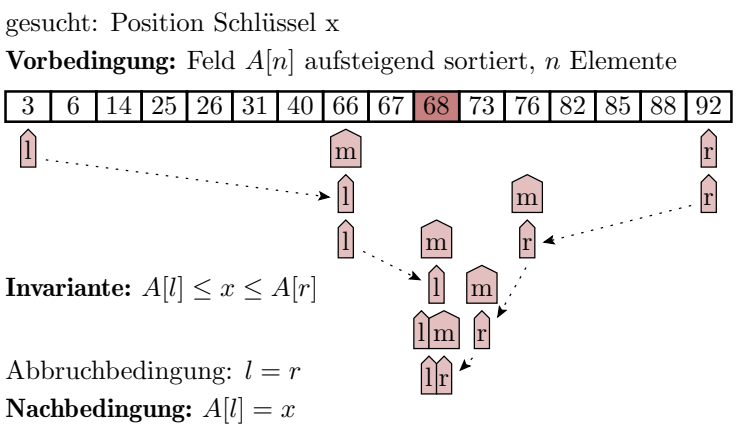
(2) Invariante, nach jedem Schritt einzuhaltender Wertebereich;

(3) konstruktive Ergebniszusicherung, Ausgabe = Eingabe plus eins.

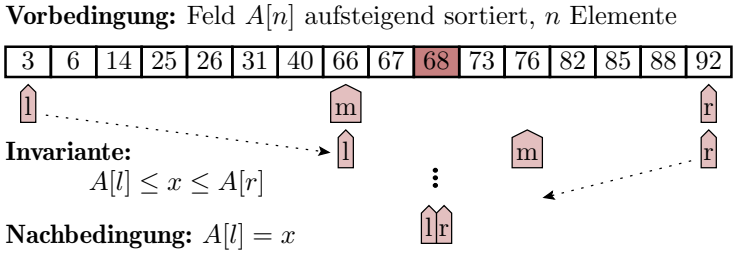
OCL Object Constraint Language. Modellierungssprache für zuzusichernde Eingenschaften.

UML Grafische Modellierungssprache zur Spezifikation von Software-Teilen.

5.103 Vorbedingungen, Invarianten, ...



Innerhalb der binären Suche nach dem Element mit Schlüssel x darf $A[l]$ nie größer und $A[r]$ nie kleiner als der gesuchte Schlüssel sein. Invarianten müssen nach jedem Abarbeitungsschritt gelten.



Das Konzept der Invarianten dient vor allem für Korrektheitsbeweise (statische Tests). Wenn eine Aussage

- zu Begin wahr ist,
- die Gültigkeit in jedem Schritt erhalten bleibt und
- die Berechnung zum Abschluss kommt (terminiert)

gilt sie auch nach Abschluss (Beweis durch vollständige Induktion).

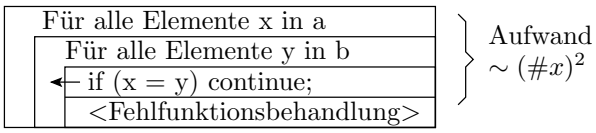
Für den Fehlfunktionsnachweis zur Laufzeit genügt die Kontrolle der Nachbedingung (Terminierung und $A\,(l) = x$). Zur Fehlerlokalisierung ist auch eine Invariantenüberwachung nach Zwischenschritten nützlich.

5.105 Aufwändige Kontrollen

OCL kann in einfacher Weise sehr aufwändig zu kontrollierende Merkmale beschreiben. Die Variablen a und b seien typengleiche Mengen von Objekten, a eine unsortierte Eingabe, b eine sortierte Ausgabe. Für jede Sortierung gilt, dass alle Elemente aus a in b und alle Elemente aus b in a enthalten sein müssen:

```
post: a->forAll(x|(b->includes(x))
post: b->forAll(x|(a->includes(x))
```

Kontrolle der ersten Zusicherung (zweite analog):



Kontrollaufwand größer als Sortieraufwand mit Quicksort.

$\#x$ Anzahl der zu sortierenden Elemente.

5.106 Effizienzsteigerung durch Prüfsummen

Komplexe Datenobjekte (Texte, Listen, Dateien, ...) können auch anhand ihrer Prüfkennzeichen (Hash-Werte) verglichen werden:

```
post: (a->hash()) = (b->hash())
```

Eine Kontrolle, dass alle Elemente aus a in in b und umgekehrt anders sortiert enthalten sind, verlangt ein kommutatives Prüfkennzeichen, z.B. die arithmetische Summe oder bitweises EXOR der Element-Hash-Werte (vergl. Folie 5.67):

```
post: a->sum() = b->sum()
```

Die Vereinfachung auf einen Prüfsummenvergleich reduziert den Kontrollaufwand erheblich und Fehlfunktionsabdeckung um den Faktor $1-2^{-r}$, also für größere Bitanzahl r praktisch überhaupt nicht.

---

hash() Hash-Funktion, Prüfkennzeichenbildung für Programmobjekte.

sum() Prüfsumme über alle Werte des Datenobjekts.

μ_{MC}, r Zu erwartende Fehlfunktionsabdeckung, Anzahl der Prüfkennzeichenbits.

4.2 Wertebereichskontrollen

5.107 Klassischer Rechtschreibtest

Wort im Wörterbuch enthalten?

- Maskierung, wenn Verfälschung zulässig und im Wörterbuch, z.B. »Maus« statt »Haus«.
- Phantom-MF, zulässiges Wort nicht im Wörterbuch.

Fehlfunktionsüberdeckung: Anteil der zulässigen Worte an den darstellbaren Zeichenfolgen fast null, aber beim Schreiben entstehen überdurchschnittlich oft zulässige Worte:

$$\mu_{\mathrm{MC}} = 60\% \ldots 90\% \ll 1 - \frac{\#VP}{\#PP}$$

Phantomfehlfunktionsrate: Viele zulässigen Zusammensetzungen und Abänderungen von Worten fehlen im Wörterbuch, typisch:

$$\zeta_{\mathrm{PM}} = \frac{1\ldots10\,[\mathrm{PM}]}{1000\,[\mathrm{DS}]} \gg 0$$

μ_{MC} Zu erwartende Fehlfunktionsabdeckung.

ζ_{PM} Phantom-Fehlfunktionsrate.

[PM] Zählwert in Phantom-Fehlfunktionen.
[DS] Zählwert in erbrachten Service-Leistungen.

5.108 Zahlenbereiche

Zahlentypen haben in der Regel viel kleinere zulässige als darstellbare Wertebereiche ($\#VP \ll \#PP$), aber bei einer typischen Wertebereichskontrolle, z.B.:

```
post: age>0 and age<100
```

erzeugen Fehlfunktionen bevorzugt falsche zulässige Werte:

- In Programmen werden kleine Werte, allen voran null und eins, viel öfter genutzt als große Werte, z.B. für uint32_t der Wert 0x92A4.5F1B=2.460.245.787.
- Das Alter eines anderen Angestellten ist immer zulässig.
- Eine Hausnummer ist oft auch ein zulässiges Alter, ...

Maskierungswahrscheinlichkeit schwer abschätzbar und viel größer als bei gleichmäßiger Abbildung:

$$p_{\mathrm{M}} = 1 - \mu_{\mathrm{MC}} \overset{(mag.)}{\approx} 10^{-1} \gg \frac{\#VP}{\#PP} = \frac{99}{2^{32}} \approx 2{,}3 \cdot 10^{-8}$$

$\#VP, \#PP$ Anzahl der gültigen Bitmuster, Anzahl der darstellbaren Bitmuster.
$(mag.)$ Größenordnung.

5.109 Bessere Verteilung der zulässigen Werte

Die Annäherung an den Idealfall, im Beispiel $p_{\mathrm{M}} \approx 2{,}3 \cdot 10^{-8}$, verlangt eine pseudo-zufällige Verteilung der zulässigen Werte innerhalb der möglichen Werte. Für Aufzählungstypen, deren Werte nur gelesen, geschrieben und getestet werden, können z.B. statt 0 bis 10 auch 0xF762, 0x38A5, ... genutzt werden.

Eine erhebliche Verbesserung ohne Perfektion versprechen bereits umkehrbar eindeutige lineare Transformationen des Wertebereichs, z.B.

- Bereichsverschiebung durch Addition und
- und Lückenerzeugung durch Multiplikation

großer zufällig gewählter Konstanten.

Konsistente Wertebereichstransformationen in Berechnungen zur Erhöhung der Fehlfunktionsabdeckung von Wertebereichkontrollen

- verlangen Werzeugunterstützung und
- funktionieren nur für eine begrenzte Menge von Verarbeitungsfunktionen.

5.110 Unterschiedliche Übersetzungen

Der Einbau von Kontrollen kann den Rechenaufwand und Speicherbedarf vervielfachen. Ein vernünftiges Verhältnis zwischen Aufwand und Nutzen verlangt für unterschiedliche Übersetzungen unterschiedlich umfangreiche Kontrollen und Fehlfunktionsbehandlungen:

- Test: Max. Kontrollen um möglichst viele Fehler zu finden.
- Fehlersuche: Zusatzkontrollen nach Zwischenschritten (z.B. für Invarianten), Trace-Aufzeichung, ... zur Fehlereingrenzung.
- Einsatz: Verzicht auf Überwachung gründlich getesteter Teile. Möglichst Weiterarbeit nach MF, z.B. durch Ersatz unplausibler Werte durch Standardwerte (fail slow).
- Reifeprozess: Zusätzliche Kontrollen und Werteaufzeichungen für Systemteile mit vermuteten Fehlern zur Lokalisierung.

Kontrollen sollten als Angebote an den Compiler zur optionale Verwendung beschrieben werden. Gleiches für Fehlfunktionsbehandlung, Trace-Aufzeichungen, ...

4.3 Standardkontrollfunktionen

5.111 Kontrolle von Klasseneigenschaften

Objektorientierte Programmierung fasst gemeinsam zu bearbeitende Daten und Beabeitungsmethoden zu Klassen von Objekten zusammen. Objekte stehen in Assoziationen mit anderen Objekten.

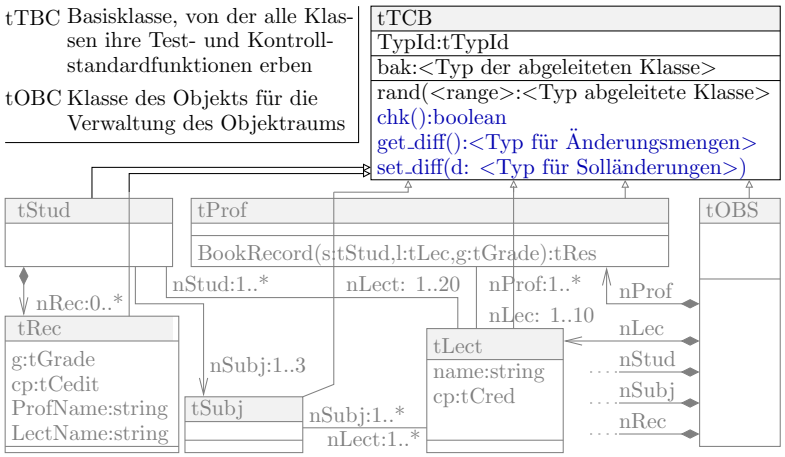
Einfachste Beschreibung für Kontrollmöglichkeiten sind aus der Klassenstruktur abgeleitete Standardfunktionen, die bei Bedarf überladen werden können. Nachfolgend ein Konzept mit drei Standardfunktionen:

```
<T>.chk():boolean;
<T>.get_diff(): tCL;
<T>.set_diff(tCL);
```

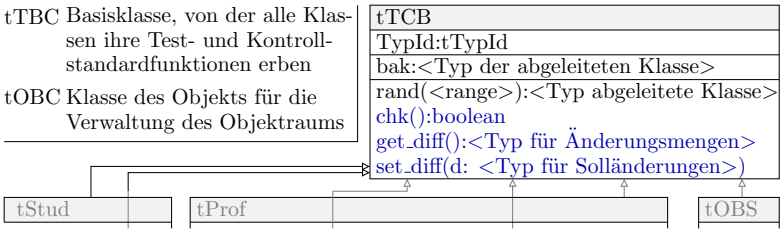
1. Kontrolle der Merkmale einer Klasse <T>, die immer gelten müssen.
2. Bestimmung aller Änderungen seit dem letzten Aufruf.
3. Setzen von Solländerungen für (2).

(2) und (3) dienen zur Programmierung diversitärer Checker und als Filter für versagende Tests, die manuell untersucht werden müssen.

5.112 Unser UML-Klassendiagrammbeispiel



Bezugnehmend auf Folie 5.53 Vererbung weiterer Attribute und Standardfunktionen für Test und Überwachung an alle Objekte.



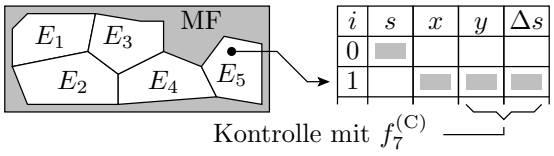
TypId: zufällige gewählte Konstante für den Datentyp. Erlaubt für alle Zeiger auf das Objekt die Kontrolle, das das adressierte Datenobjekt existiert und den richtigen Typ hat.

bak: Kopie aller Attribute und Zeiger. Normale Methoden bearbeiten die Originaldaten und **set_diff(..)** die Kopie. **get_diff()** gibt bei jedem Aufruf alle Abweichungen zurück und kopiert die Originawerte nach bak.

Für tStud, tProf, ... soll **<T>.chk()** nur die eigenen Daten und die Typen der Objekte an den abgehenden Zeigern kontrollieren und **<T>.get_diff()** Änderungen der eigenen Daten zurückgeben.

Für das zentrale OBS-Objekt soll **OBS.chk()** alle Objekte kontrollieren und **OBS.get_diff()** alle Änderungen zurückgeben.

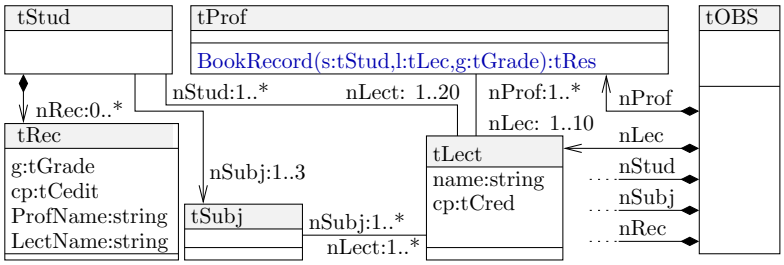
5.114 Kontrollen mit Standardfunktion



Die Eingabebereiche teilen sich in zulässige und unzulässige Werte auf. Für unzulässige Werte ist auf »Fehlfunktionsbehandlung« zu kontrollieren. Die Äqivalenzklassen E_i seien hier Bereiche mit gleicher Ergebniskontrolle, beschrieben durch ihre

- Eingabebedingung c_i und
- eine Kontrollfunktion $f_i^{(C)}$.

Bei der Testgenerieung wird auf jeden gefundenen Test für jede erfüllte Kontrollbedingung c_i die Kontrollfunktion $f_i^{(C)}$ angewendet. Bei Signalisierung eines Fehlers wird für den Reparaturprozess ein seperater Testrahmen mit Objektrauminitialisierung und Kontrolle generiert. OBS.chk() ist nach jedem Testschritt nutzbar.



Bedingungen und Kontrollen für unser Beispiel:

1. mindestens eine ungültige Eingabe: Fehlfunktionsabbruch,
2. sonst kein Student der Lehrveranstaltung, Lehrveranstaltung nicht im Studienplan, ...: keine Veränderung des Objektraums.

3. sonst kein Verbuchsdatensatz für das Fach: OBD.get_diff() → ein neues tRec-Objekt mit den Testeingaben,
4. sonst Notenverbesserung: stud.lect.grade = OBD.get_diff() → verändertes Objekt,
5. sonst: keine Veränderung.

Bei wenig Änderungen im Objektraum verspricht bereits eine Kontrolle, was sich nicht ändern soll, hohe Abdeckung für unerwartete Probleme.

5 Syntax

5.116 Syntaxtest

Syntaxkontrollen sind nicht nur für Programme vor der Übersetzung, sondern für jede Form der manuellen Texteingabe zweckmäßig. Voraussetzung, Textbeschreibung in einer formalen Sprache.

Formale Sprache: Definition zulässiger Zeichenfolgen durch Syntaxregeln, deren Einhaltung sich durch spracherkennende Automaten kontrollieren lassen.

Ein spracherkennender Automat ist zwar selbst kein einfaches, schnell zu schreibendes Programm, aber das Programm für einen spracherkennenden Automaten lässt sich nach bekannten Algorithmen aus den Syntaxregeln der Sprache automatisch generieren.

Syntaxtests erkennen viele menschengemachte Fehler. Für maschinell erzeugte Daten, die nicht manuell bearbeitet werden, haben Prüfkennzeichen, fehlererkennende oder korrigierenden Codes ein viel besseres Verhältnis von Aufwand zu Nutzen.

5.1 Formale Sprachen

5.117 EBNF zur Beschreibung formaler Sprachen

Beschreibungsmittel der EBNF zur Definition von Sprachregeln:

Beschreibungsmittel	EBNF-Darstellung
Definition	NTS = Ersetzungsregel
Aufzählung	..., ...
Endezeichen	...;
Alternative	...|...
Option	[...]
Wiederholung	{...}
Gruppierung	(...)
Zeichenkette (Terminalsymbolfolge)	"..." oder '...'

EBNF Erweiterete Backus-Naur-Form.
NTS Nicht-Terminalsymbol, zu ersetzendes Symbol.

5.118 Beispiele für EBNF-Syntaxregeln

```
Zahl = ['-'],((ZiffernAusserNull,{Ziffer})|'0');
ZifferAusserNull = '1'|'2'|'3'| ... |'9';
Ziffer = ZifferAusserNull | '0';
```

Beispielzeichenfolgen:

```
'-1001': zulässig
'3,23' : nur bis Komma zulässig
'010'  : nur vor 1 zulässig
```

```
Bezeichner = Buchstabe,{Buchstabe|Ziffer},TZ;
TZ = ','|';'|Leerzeichen|...
```

Beispielzeichenfolgen:

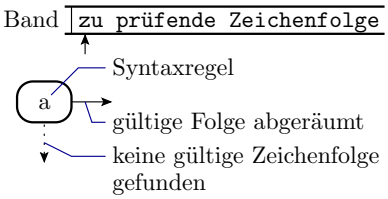
```
'a100;': zulässig,
'aa_3,': unzulässig wegen '_'
'1A'   : unzulässig wegen führender Ziffer
```

5.2 Spracherkennende Automaten

5.119 Spracherkennende Automaten

Die zu prüfende Zeichenfolge liegt auf einem Band mit einem Zeiger auf den Anfang.

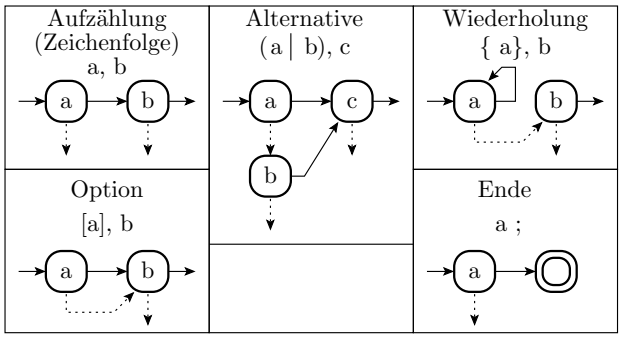
In jedem Automatenzustand wird versucht, eine Zeichenfolge nach der Syntaxregel a des Zustands abzuräumen:



- Wenn möglich, wandert der Zeiger zum ersten Zeichen nach der abgeräumten Folge und der Knoten wird über $\longrightarrow$ verlassen.
- Sonst bleibt der Zeiger und der Knoten wird über $\downarrow$ verlassen.

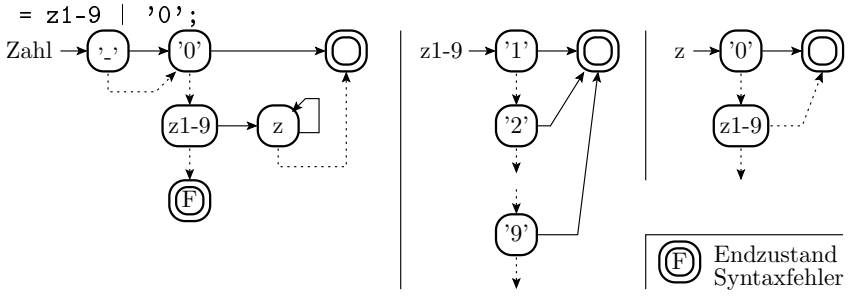
$\downarrow$-Übergänge ohne dargestellten Zielknoten enden im Fehlerzustand.

5.120 Von der EBNF zum Automaten



Beispiel:

```
Zahl = ['-'],((z1-9, {z}) | '0');
    z1-9 = '1' | '2' | ... | '9';
    z    = z1-9 | '0';
```



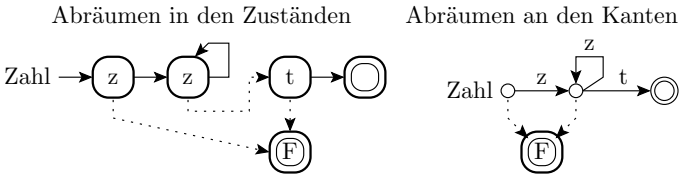
Welche Zustände durchläuft der Automat. Was erkennt er?

- "125_": '-'↓, '0'↓, z1-9→, z→, z→, z↓, erkannt: 125, Zeiger: $125\overset{\downarrow}{\_}$
- "k89": '-'↓, '0'↓, z1-9↓, Syntaxfehler, Zeiger: $\overset{\downarrow}{k}89$
- "-0701" : '-'→, '0' →, erkannt: $0\surd$, Zeiger: $-\overset{\downarrow}{0}701$

5.122 Abräumen an den Kanten

Beschreibung einer Zahl:

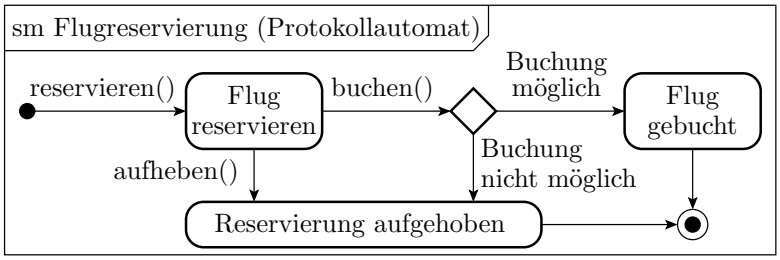
Zahl = z, {z}, t; z = '0' | '1' | ... | '9';
t = ' ' | ',' | ...; (Trennzeichen)



- Das Abräumen kann auch an den Kanten erfolgen.
- Dann kann jeder Knoten auch mehr als zwei abgehende Kanten haben.
- Beschreibung derselben Syntaxregeln mit weniger Zuständen.
- Die »Sonst-Kanten« zum Fehlerzustand müssen im Syntaxgraphen nicht mit gezeichnet werden.

5.3 Ablaufkontrolle

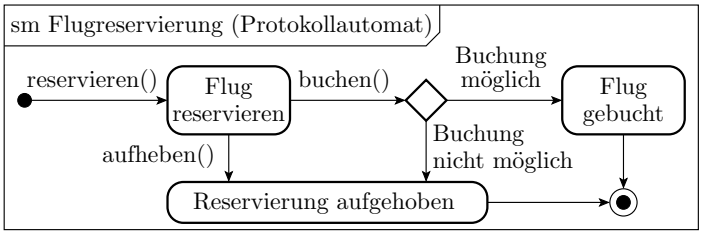
5.123 Protokollautomat



Protokollautomaten sind ein UML-Beschreibungsmittel zur Spezifikation 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.

UML Grafische Modellierungssprache zur Spezifikation von Software-Teilen.

5.124 Vom Automat zur Sprache



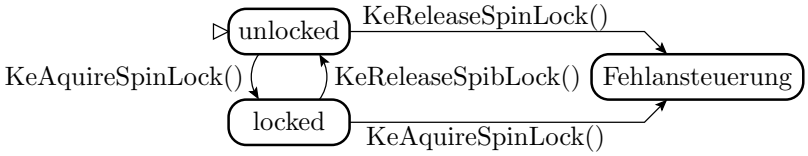
Wenn wir an die Eingaben Terminalsymbole vergeben, z.B. r (reservieren), a (aufheben), b (buchen), m (Buchung möglich) und n n (Buchung nicht möglich) sind folgende Eingaben zulässig:

```
r,a|(b,(m|n))
```

Das kann man für

- die Beschreibung von Tests
- das Auswürfeln von Tests
- und als als Ablaufkontrolle nutzen.

5.125 API-Benutzungsregeln



Eine API definiert u.a. Benutzerregeln für die bereitgestellten Funktionen. Beispiel »spinlock« aus der Windows-API [1]:

- Spinlocks müssen alternierend reserviert und freigegeben werden
- durch Aufruf der Funktionen »KeAquire..« und »KeRelease«

beschreibbar durch einen Protokollautomaten. Protokollautomaten können als Überwachungsfunktionen zusammen mit einer Fehlfunktionsbehandlung für Protokollverletzungen in das System instrumentiert werden. Alternativ Ausschluss durch statische Tests.

API — Eine API (Application Programming Interface) ist ein Satz von Befehlen, Funktionen, Protokollen und Objekten, die Programmierer verwenden können, um eine Software zu erstellen oder um mit einem externen System zu interagieren.

5.126 5.126 Eine zu testende Treiberfunktion

Eine Treiberfunktion ruft »KeAquire..« und »KeRelease...« u.U. mehrfach auf, in Fallunterscheidungen, Schleifen, ...

Für jeden Kontrollpfad muss der Spinlock alternierend bedient werden.

Fehlerausschluss erfordert Kontrolle für alle Pfade.

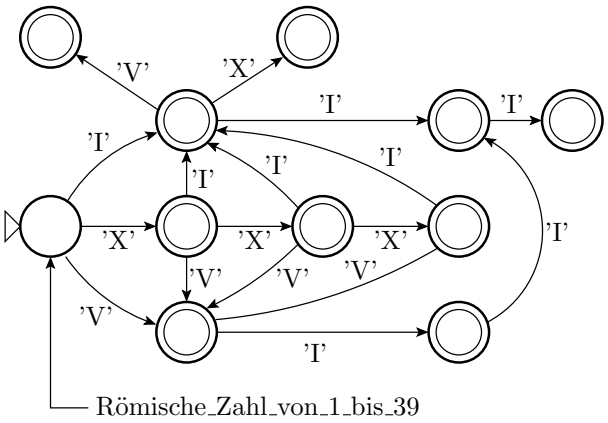
Reale Treiberfunktionen haben hunderte von Code-Zeilen. Kontrolle selbst so einfacher Regeln nicht trivial.

[Fehler in uralten Treibern gefunden]

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

5.4 Würfel für zulässige Eingaben

5.127 Checker für römische Zahlen



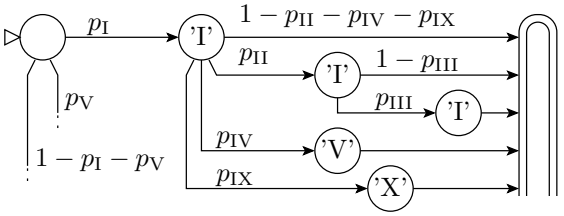
I	XI	XXI	XXXI
II	XII	XXII	XXXII
III	XIII	XXII	XXXII
IV	XIV	XXIV	XXXIV
V	XV	XXV	XXXV
VI	XVI	XXVI	XXXVI
VII	XVII	XXVII	XXXVII
VIII	XVIII	XXVIII	XXXVIII
IX	XIX	XXIX	XXXIX
X	XX	XX	

(potentieller) Endzustand

Der Automat im Bild kontrolliert eine Eingabezeichenfolge auf gültige römische Zahldarstellung.

Man stelle sich an den Kanten Auswahlhäufigkeiten und die Zeichen als Ausgaben vor. Dann wird auf jedem zufällig durchlaufenen Weg durch den Graphen eine gültige römische Zahl erzeugt.

5.128 Würfel für römische Zahlen



$\mathbf{P}(\text{'I'}) = p_I \cdot (1 - p_{II} - p_{IV} - p_{IX})$
$\mathbf{P}(\text{'II'}) = p_I \cdot p_{II} \cdot (1 - p_{III})$
$\mathbf{P}(\text{'III'}) = p_I \cdot p_{II} \cdot p_{III}$
$\mathbf{P}(\text{'IV'}) = p_I \cdot p_{IV}$
$\mathbf{P}(\text{'IX'}) = p_I \cdot p_{IX}$

Die Häufigkeiten, mit denen die einzelen Werte gewürfelt werden, hängt von die Übergangswahrscheinlichkeiten der Kantenübergänge ab. Wenn jede Kante mit gleicher Wahrscheinlichkeit gewählt wird, nimmt die Wahrscheinlichkeit tendentiell mit der Zeichenlänge ab.

Für Häufigkeitvorgaben $\mathbb{P}[\langle Wert \rangle]$ ergeben sich die Übergangshäufigkeiten $p_{\langle Zeichen \rangle}$ als Lösung eines Gleichungssystems.

In Abschn. 7.3.3 werden die zu erkennenden und zu würfelnden Zeichen auch Kontrollausgaben, Zustände oder Eingabebedingungen sein. Dann sind auch die ausgewürfelten Beispiele nur symbolische Tests, für die dann noch symbolweise Werte gesucht werden müssen.

Auf diese Weise lassen sich auch zulässige, wenn meist auch sinnlose komplette gültige Programme als Testbeispiele auswürfeln.

In [Holler##] Java-Scripte für Webbrowser-Test.

Zusammenfassung

5.130 Regelbasierte Kontrollen

Die codebasierten Verfahren (fehlererkennende und fehlerkorrigierende Codes sowie Prüfkennzeichen) eignen sich nur sehr begrenzt für die Überwachung von Berechnungen. Statt dessen lassen sich für Berechnungen (Eingaben, Ablauf und Ergebnis) zahlreiche kontrollierbare Regeln festlegen und zur Laufzeit überwachen:

- Datentypen, Wertebereiche, ...
- Syntax, Invarianten,
- API-Benutzerregeln, Protokollautomaten,
- aufgabenspezifische Kontrollen, ...

Alternativ wird immer ein Teil der Regelverletzungen bei der Programmübersetzung durch statische Tests ausgeschlossen.

Fehlfunktionsüberdeckung und die Phantom-MF-Rate solcher regelbasierter Kontrollen sind sind i. Allg. schlechter als bei den codebasierten Verfahren und auch schlecht vorhersagbar. Die Leistungsfähigkeit liegt in der Vielzahl der Kontrollmöglichkeiten.

5.131 Einordnung in den Vorlesungskontext

Beim Test und in Reifeprozessen werden hauptsächlich Fehler beseitigt, die Abstürze, automatisch erkennbare Fehlfunktionen und auffällige Probleme in der Anwendungsumgebung verursachen. Denn die manuelle Inspektions-MF-Abdeckung ist i.Allg. nicht sehr hoch.

Die Nutzung der Kontrollmöglichkeiten hängt maßgeblich von der Programmiersprache, der Softwarearchitektur etc. ab und wie einfach sich die Kontrollen incl. Problembehandlung beschreiben lassen.

Literatur

Literatur

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

[4] 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.

[5] 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.