Test und Verlässlichkeit Foliensatz 5: Dynamische Tests

Prof. G. Kemnitz

Institut für Informatik, TU Clausthal (TV_F5)

27. Juni 2017

Inhalt TV_F5: Dynamische Tests

Dynamische Tests

- Ausprobieren der Funktion mit einer Stichprobe von Eingaben.
- Testauswertung in der Regel Vergleich mit Sollwerten.
- Testdurchführung idealerweise automatisch.

Voraussetzung für die Durchführbarkeit:

- Entwurf / Fertigung abgeschlossen.
- Statisch nachweisbare Fehler, bei HW und CPS insbesondere die mit potentieller Zerstörungsgefahr, beseitigt[1].
- Prüfgerechter Entwurf.
- Testbeispiele und Testumgebung (Prüftechnik, Testrahmen, ...).

[1] Weitgehend funktionierendes System ohne größere Selbstzerstörungsgefahr.

Systemtypen und ihre Besonderheiten

Auch für den dynamischen Test hat jeder Systemtyp (SW, HW, SW+HW, CPS) seine testspezifischen Besonderheiten:

- Software: Programm als Testrahmen. Prüfgerechtheit bezieht sich vor allem auf die funktionale Dekomposition und die Interaktionsschichten. Agile Entwürfe. Potentiell Zugriff auf alle Zwischenergebnisse. ...
- Hardware-Entwürfe, Simulationsmodelle: Entwurfsablauf, Test, prüfgerechter Entwurf, ... ähnlich wie bei Software.
- Schaltkreise: Eingeschränkter Zugriff auf interne Signale. Fehlerbeseitigung durch Ersatz. Dadurch hohe Anforderungen an Fehlervermeidung und Test. .... Automatische Testberechnung seit ca. 1980.
- Baugruppen, CP-Systeme: Prüftechnik zur Bereitstellung elektrischer und physikalischer Ein- und Ausgaben. ...

Software

Besonderheiten der Testauswahl

Besonderheiten von Software für die Testauswahl

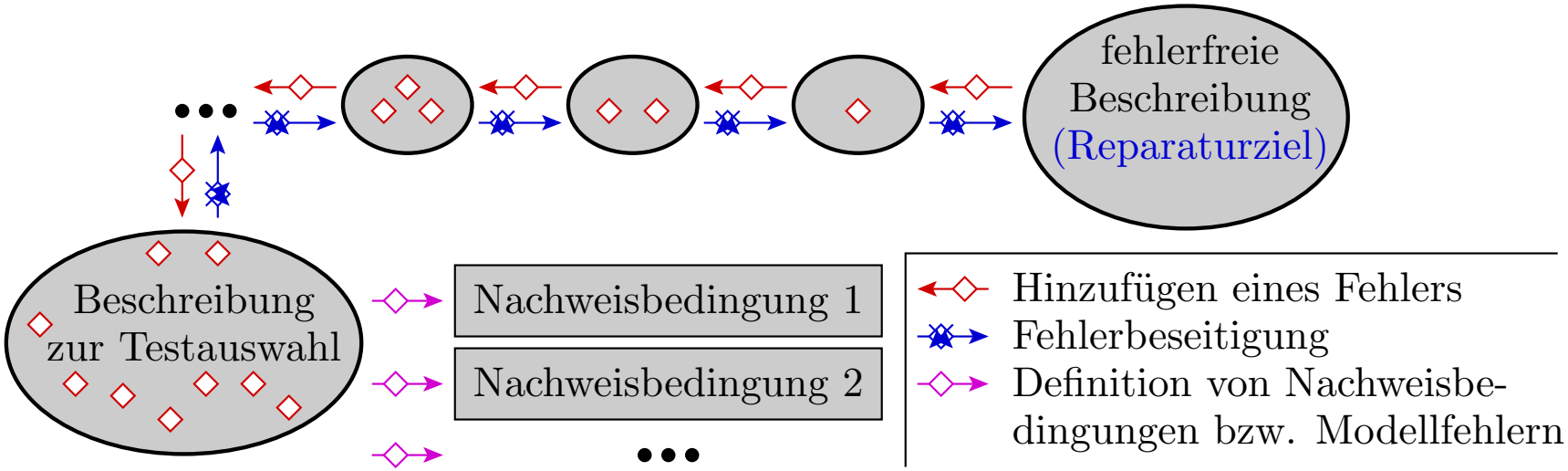
- Fehler entstehen im gesamten Entwurfsprozess (Spezifikation, Architekturentwurf, Programmierung) und auch bei der Fehlerbeseitigung.
- Es existiert in der Regel keine fehlerfreie Sollbeschreibung.
- Eine geziele Kontrolle und Veränderung beliebiger Zwischenergebnisse zur Fehlereingrenzung oder der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

Zu erwartende Fehler

- Anforderungsfehler: vergessene, überflüssige, falsch formulierte, nicht umsetzbare Anforderungen.
- Fehler bei der Architekturfestlegung: fehlende, überflüssige, falsch umgesetzte Anforderungen, Fehler im Algorithmus, unberücksichtigte selbstverständliche Anforderungen, z.B., dass ein Programm installierbar und bedienbar sein muss, nicht prüfgerecht, ...
- Codierung: fehlende, falsche überflüssige Anweisungen, Fallunterscheidungen, Ausnahmebehandlungen, ...

Es gibt keine korrekte Beschreibung zur Zusammstellung einer Menge potentieller bzw. Modellfehler.

Mutationen statt Modellfehler



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

Mutationen für Programme

Mutationen auf der Hochsprachenebene sind geringfügige Verfälschungen im Programmtext:

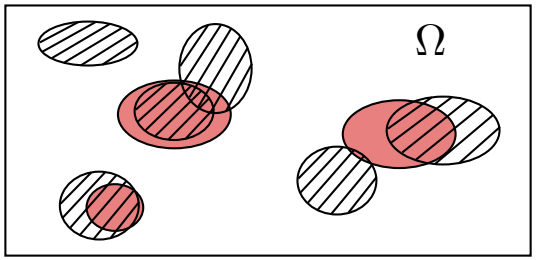
- Verfälschung arithmetischer Ausdrücke (x=a+b ⇒ x=a*b)
- Verfälschung boolescher Ausdrücke (if(a>b){} ⇒ if(a<b){})
- Verfälschung der Wertezuweisung (value=5 ⇒ value=50)
- Verfälschung der Adresszuweisung (ref=obj1 ⇒ ref=obj2)
- Entfernen von Schlüsselworten (static int x=5 ⇒ int x=5)

Bestimmung der Modellfehler- (Mutations-) überdeckung:

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

Kostet viel Rechenzeit, ist aber prinzipiell durchführbar.

Gezielte Testauswahl



Ω Menge der Eingabewerte /Teilfolgen die einen Fehler nachweisen können

Nachweismenge einer Mutation

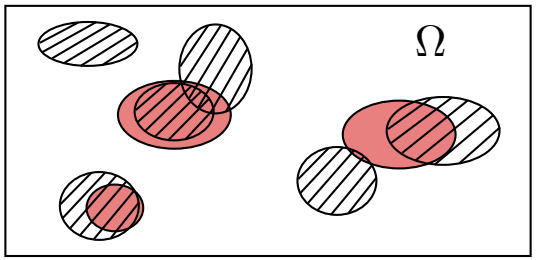
Nachweismenge eines tatsächlichen Fehlers

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

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

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

Zufällige Testauswahl



Ω Menge der Eingabewerte /Teilfolgen die einen Fehler nachweisen können

Nachweismenge einer Mutation

Nachweismenge eines tatsächlichen Fehlers

- Alle potentiellen Fehler und alle Mutationen haben Nachweismengen, die sich mehr oder weniger überschneiden.
- Die FHSF-Funktion[2] der Mutationen unterscheiden sich um einen Skalierungsfaktor c von der der richtigen Fehler:

$$H_{\text{Mut}}(x) \sim H(c \cdot x)$$

Zufallstest ist für alle potentiellen Fehler, auch die vom Typ »fehlt in der fehlerhaften Beschreibung« geeignet.

[2] FHSF-Funktion: **F**ehler**h**äufigkeit in Abhängigkeit von der mittleren Anzahl der **S**ervice-Leistungen zwischen zwei **F**ehlfunktionen.

Aus

$$H_{\mathrm{Mut}}(x) \sim H(c \cdot x)$$

folgt, für einen Zufallstest der Länge n ist die Mutationsüberdeckung etwa die Fehlerüberdeckung der c-fachen Testsatzlänge:

$$FC_{\mathrm{Mut}}(n) \approx FC(c \cdot n)$$

(vergl. TV_F2, Abschn. 3.7 Länge von Zufallstests).

Software und Hardware-Entwürfen sollten immer einem ausreichend langen Zufallstest unterzogen werden.

Fehlende Sollfunktion

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

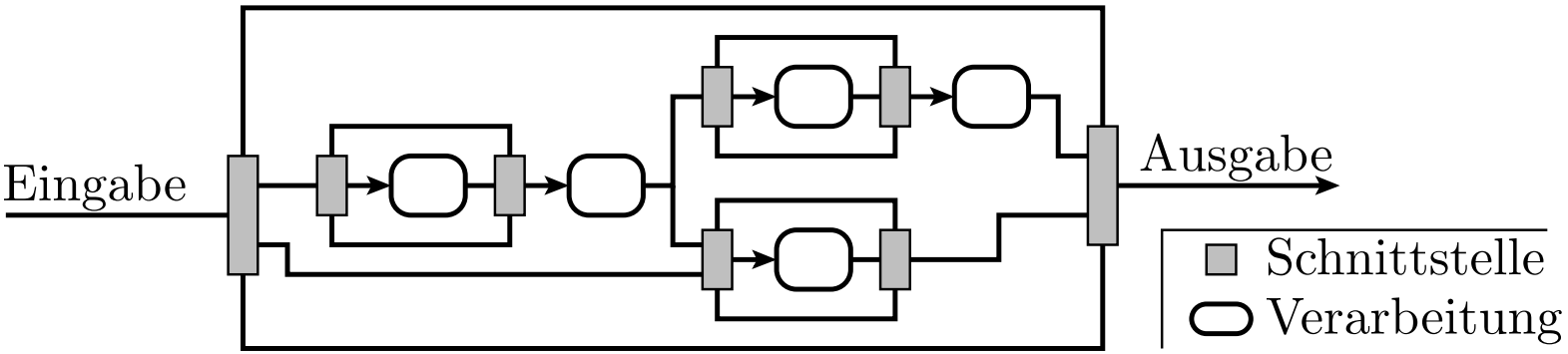
Erfordert eine diversitäre Sollwertberechnung. Maskierung durch übereinstimmende Fehlfunktionen schwer ausschließbar.

Konzeptionelle Ansätze:

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

Module und Schichten

Modularer Test



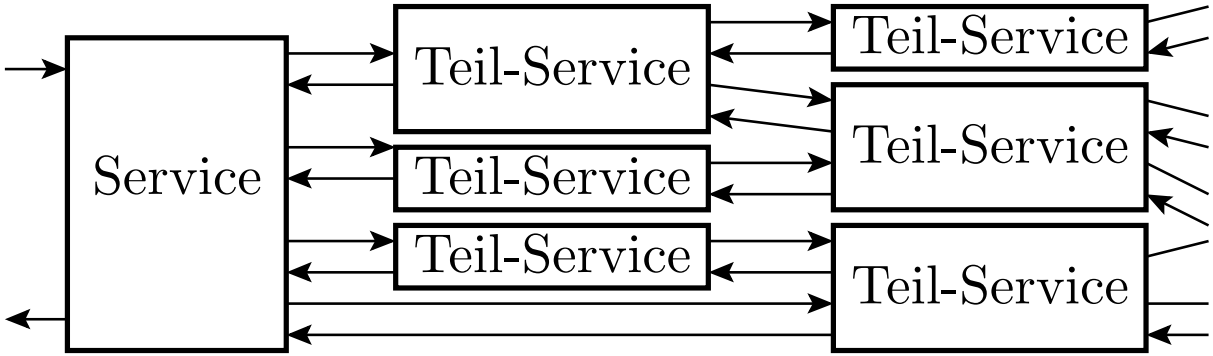
Wiederhole für jedes Modul:

- direkte Bereitstellung von Testeingaben und
- direkte Kontrolle der Ausgaben.

Prüfgerechter Entwurf: Wahl der Modulaufteilung so, dass

- sich die Eingaben gut auswählen und bereitstellen lassen und
- die Ergebnisse gut kontrollierbar sind, ...

Modultest, FHSF-Funktion und Testaufwand



Für den separaten Test von Teil-Service-Leistungen / -Modulen gilt:

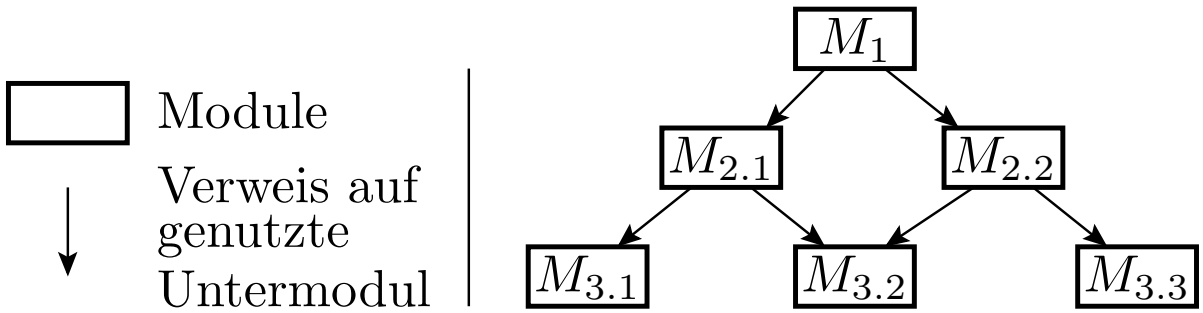
- Erhöhung der $h_i \ll 1$ selten genutzter Module auf 1.
- Erhöhung der $b_i \ll 1$ schlecht beobachtb. Teilergebnisse auf 1.

Einfluss auf die FHSF-Funktion:

$$H_{\text{Mod}}(x) \sim H(h \cdot b \cdot x)$$

Beim separaten Test aller Module genügt tendentiell die $h \cdot b \ll 1$-fache Testsatzlänge eines ganzheitlichen Tests für eine vergleichbare Fehlerüberdeckung und Zuverlässigkeitserhöhung.

Entwurfs- und Testreihenfolge

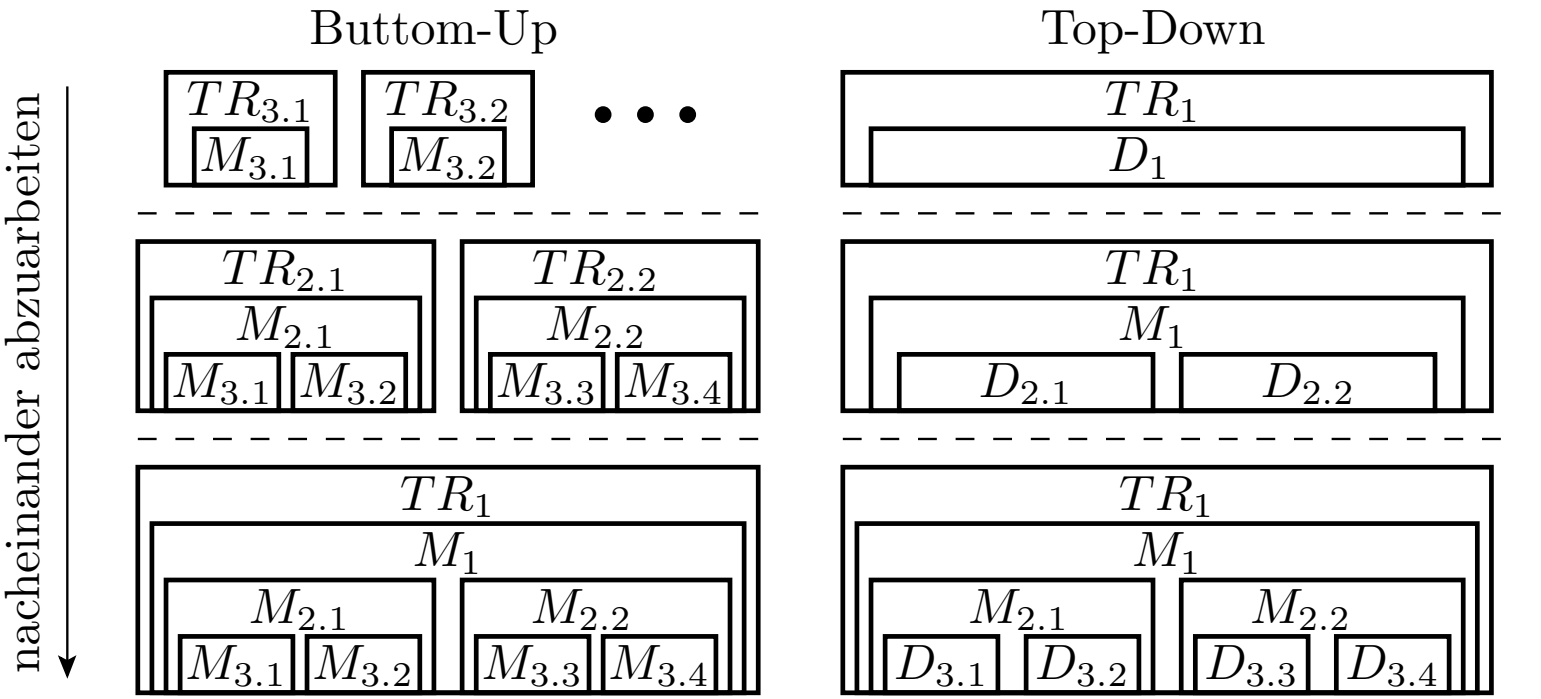


Strategien für die Entwurfs- und Testreihenfolge:

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

Modulstruktur und Entwurfsfluss haben erheblichen Einfluss auf den Aufwand und die Güte der Tests. Prüfgerechter Entwurf!

Testrahmen und Dummies



M_i Modul i D_i Dummie für Modul i TR_i Testrahmen für Modul i

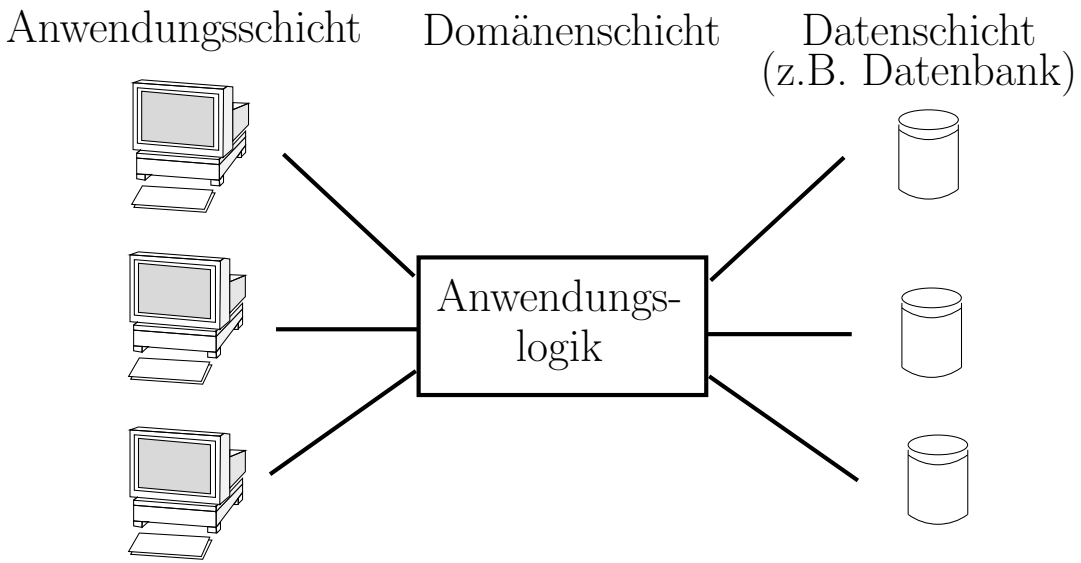
Der Buttom-Up-Entwurf verlangt für jedes Modul einen eigenen Testrahmen. Der Top-Down-Entwurf kommt mit einem Testrahmen für das oberste Modul aus und verlangt aber Dummies für Module unterer Schichten.

Schichtenarchitektur

Übergeordnetes Organisationsprinzip für Module:

- Jedes Modul ist einer Schicht zu geordnet.
- Einschränkung der Abhängigkeiten, dass nur »höhere« Schichten »tiefere« Schichten benutzen dürfen.

Beispiel 3-Schichtenmodell:



Software-Schichten

- Low-Level-Benutzerinteraktion (Tasteneingaben, Mouse-Ereignisse, ...),
- High-Level-Benutzerinteraktion (Buttons, Menüs, ...),
- Funktionsschicht (z.B. Speichern, Compilieren, ...),
- Betriebssystem (Prozessverwaltung, Dateisystem, ...), ...

Betriebssysteme und andere Schichten stellen oft eigene Scriptsprachen für die Steuerung von Applikationen und damit auch für die Durchführung von Tests bereit: Windows-Script, Perl, Python, TCL, Unix-Shell, ...

Testerstellung für einen realen Bug

Bei agilen Entwürfen muss für jedes beobachtbare Fehlverhalten ein programmgesteuerter Test entwickelt werden, mit dem es reproduzierbar nachweisbar ist (siehe später Foliensatz TV_F6)

Folgendes Fehlverhalten sei beobachtbar[3]:

- Start mozilla. Go to bugzilla.mozilla.org. Select search for bug.
- Print to file setting the bottom and right margins to .50.
- When done do the same thing again on the same file.
- This causes the browser to crash with a segfault.

Für die Fehlersuche und -beseitigung wird ein automatisierter Test mit reproduzierbarem Fehlverhalten benötigt (siehe später TV_F6).

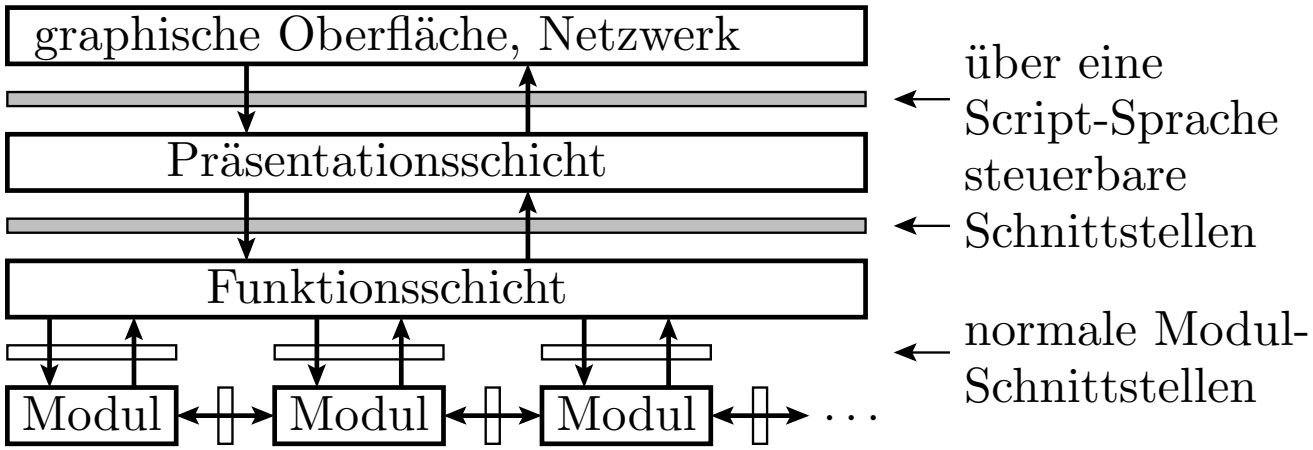
[3] Mozilla Bug #24735 nach [Zeller], Foliensatz MakingProgrammsFail, Folie 6. Mit aktueller Version nicht mehr nachstellbar.

Testautomatisierung

Wie und auf welcher Schicht lässt sich ein solcher Test automatisch durchführen?

- Synchronisation: Wie lässt sich mit einem Testprogramm prüfen, dass sich ein Fenster, das richtige Fenster, ... geöffnet hat?
- Eingabedatenbereitstellung: Wie kann ein Programm Daten in Fenster eintragen, Tastatur- und Mouse-Ereignisse nachbilden, ...
- Wie lassen sich die Ausgaben von Text und Graphik in Fenster auf Zulässigkeit und Richtigkeit kontrollieren?

Schichten für die Testautomatisierung



- Schicht: Zusammenfassung von Service-Leistungen einer bestimmten Abstraktion und Funktionalität.
- Ein Programm mit graphischer Benutzeroberfläche, Netzanbindung, ... hat i. Allg. mehrere Schichten zwischen der Ein- und Ausgabe, ... auf denen Tests programmierbar sind:
  - Betriebssystemschicht, Benutzerinteraktion,
  - Funktionsschicht, ...

Betriebssystemschicht

Komplette Kontrolle des Rechners:

```
# Power on the machine and wait for 5s
power <= true; wait for 5000;
# Click mouse button 1
m_b1 <= true; wait for 300; m_b1 <= false;
# Click the CDROM change button
cdctrl'shortcut_out_add("/cdrom%change/...");
```

- Scripte automatisch mitschreib- und abarbeitbar.
- Beispielfehlverhalten (Absturz mit »sementation fault«) ist beobachtbar.

Problematisch:

- Kontrollen von Text- und Graphikausgaben, ...
- Wenig robust[4] gegenüber Variationen der Hardware, Treibereinstellungen, Fensteranordnung, ...

[4] Jede Variante braucht ihr eigens angepasstes Testprogramm.

Low-Level-Benutzerinteraktion

```
# 1. Launch mozilla and wait for 2 seconds
exec mozilla & send_xevents; wait 2000
# 2. Open URL dialog (Shift+Control+L)
send_xevents keydn Control_L
send_xevents keydn Shift_L
send_xevents keyup Shift_L
send_xevents keyup Control_L
send_xevents wait 500
# 3. Load bugzilla.mozilla.org
send_xevents @400,100
send_xevents type {http://bugzilla.mozilla.org}
send_xevents key Return
```

- Scripte automatisch mitschreib- und abarbeitbar.
- Einprogrammierung von Kontrollen weiterhin schwierig.
- Bei Änderung Fenstergröße, -position Neuaufzeichnung.

High-Level-Benutzerinteraktion

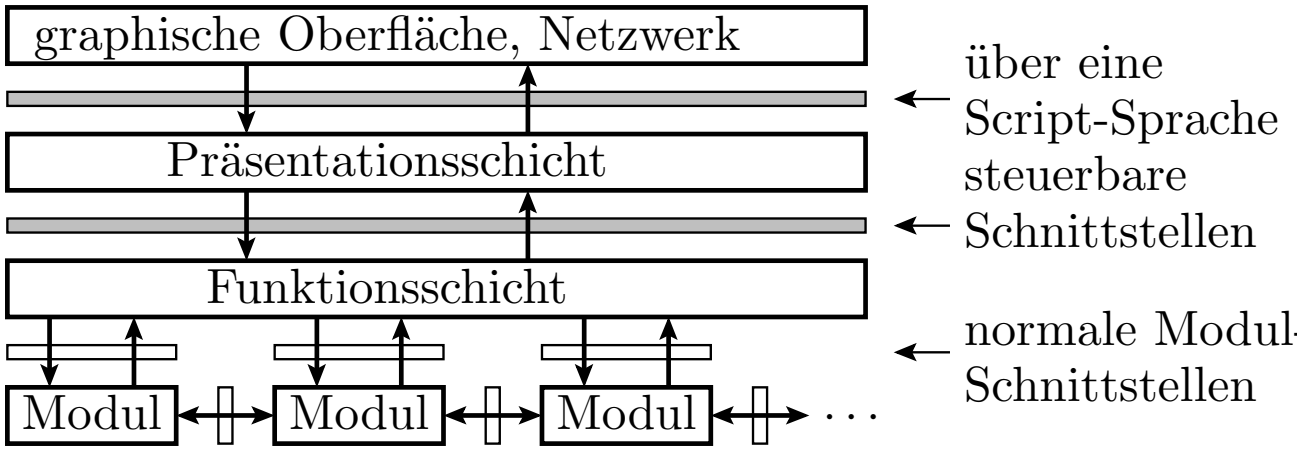
```
-- 1. Activate mozilla
tell application "mozilla" to activate
-- 2. Open URL dialog via menu
tell application "System Events" to ¬
 tell process "mozilla" to tell menu bar 1 to ¬
   tell menu bar item "File" to ¬
    click menu item "Open Web Location"
-- 3. Load bugzilla.mozilla.org
tell window "Open Web Location"
 tell sheet 1 to ¬
  set value of text field 1 to "http://bugzilla.mozilla.org/"
 click button 1
end tell
```

- Namensreferenz der GUI-Elemente, statt durch Koordinaten.
- Robuster gegenüber Änderungen von Größe und Position.
- Neuaufzeichnung bei Änderungen des Oberflächen-Layouts. ...

Funktionsschicht

```
tell application "Safari"
 activate
 if not (exists document 1)
  make new document at the beginning of documents
 end if
 set the URL of the front document ¬
  to "http://bugzilla.mozilla.org/"
 delay 5
end tell
```

- Eingaben und Ergebnisse sind formatierte Daten.
- Damit alle behandelten Möglichkeiten der Ergebnisüberwachung einsetzbar: Format, Wertebereich, Soll-/Ist-Vergleich, ...
- Tests sind robust gegenüber GUI-Änderungen.
- ...



- Schichtenstruktur und Scriptsprachen sind Voraussetzung, um bei komplexen Systemem (mit Bedienoberfläche, Netzanbindung, ...) für beobachtete Fehlverhalten Tests zu programmieren. Teil des prüfgerechten Entwurfs.
- Die am besten geeignete Schicht hängt vom nachzuweisenden Fehlverhalten und vom Fehlerort ab:
  - Funktionsschicht: Fehlverhalten der Module.
  - High-Level-Benutzerinteraktionsschicht: Fehler Oberfläche, Interaktion ...

Kontrollflussorientierte Testauswahl

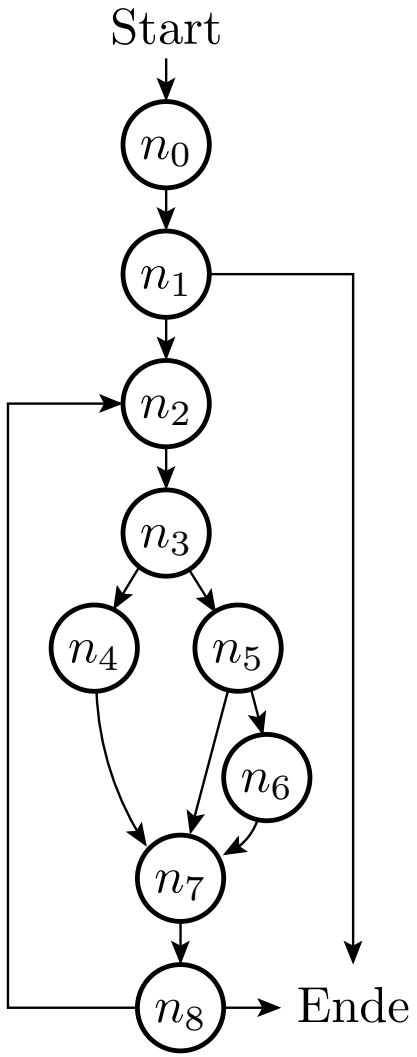
Die nachfolgend vorgestellten Testauswahlverfahren

1. kontrollflussorientierte Testauswahl,
2. Äquivalenzklassenbasierte Testauswahl,
3. Testauswahl für Ursache-Wirkungs-Beziehungen und
4. Testauswahl für Automaten.

eignen sich nur für Module, d.h. für Tests auf der Funktionsschicht. Beim kontrollflussorientierten Test ergeben sich die Testbespiele aus dem fertigen Programm. Bei den Verfahren 2 bis 4 sind die Testbeispiele bereits aus der Zielfunktion der zu testenden Module ableitbar.

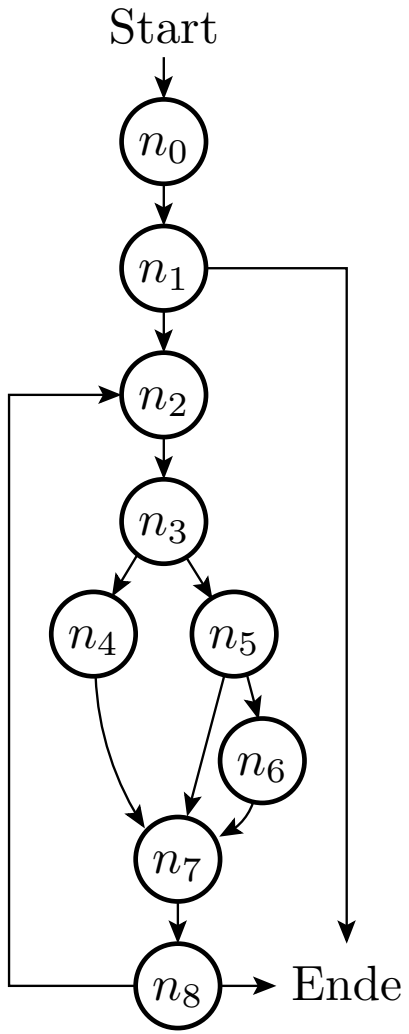
Ein Beispielprogramm und sein Kontrollflussgraph

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



Auswahlkriterien für Tests

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



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

Bestimmung der Überdeckung bei manueller/zufälliger Testauwahl:

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

Kontrolle der Testergebnisse:

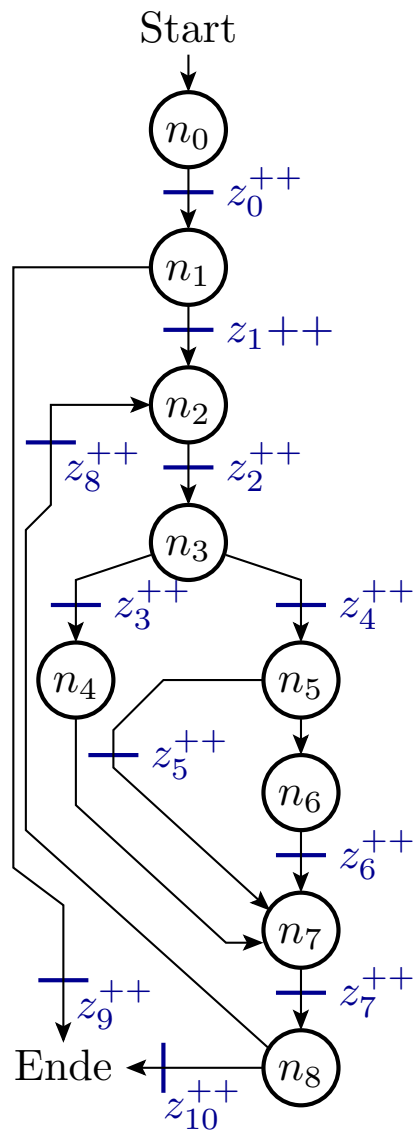
- einprogrammierte Überwachungsfunktionen,
- Trace-Aufzeichnung und Review aufgezeichneter Daten,
- Regressionstest.

Erweiterung der Kontrolle auf Anweisungsergebnisse möglich:

- Schrittbetrieb und manueller Kontrolle der Zwischenergebnisse,
- Trace-Aufzeichnung und Inspektion aller Zwischenergebnisse.
- Kontrolle mit Def-Use-Ketten.

Bestimmung der Kantenüberdeckung

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



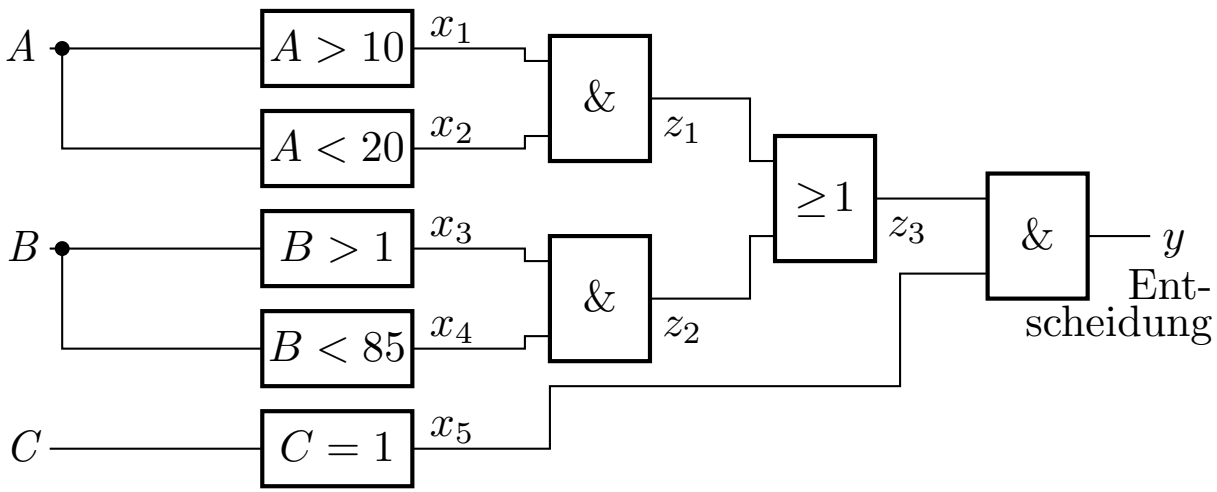
[5] Zur Unterbringung aller Zähler Schleife in Maschinenbefehle auflösen.

Bedingungsüberdeckung

Ein logischer Ausdruck, z.B.

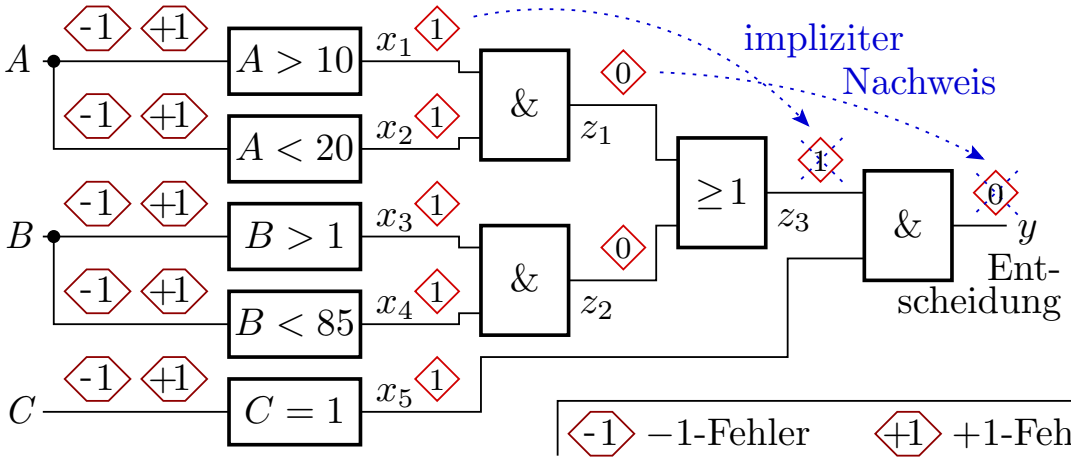
```
n1: if (((A>10) && (A<20)) || ((B>1) && (B<100))
        && (C==1)) {
n2:   ... }
    else {
n3: ...   }
```

ist nachbildbar durch einen Schaltplan aus Gattern und Vergleichern:



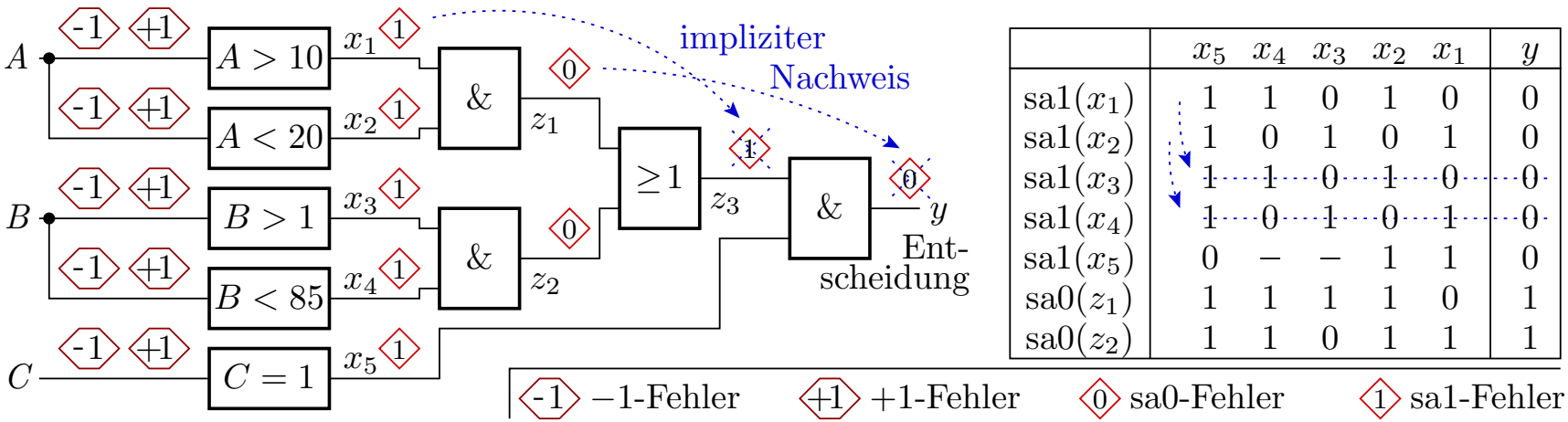
Berechnungsfluss mit eingezeichneten Fehlern

Die Bestimmung der Bedingungsüberdeckung lässt sich auf die Modellierung von Haftfehlern, (+1)-Fehler und (-1)-Fehlern zurückführen.



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

Weiter wie bei Haftfehlern



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

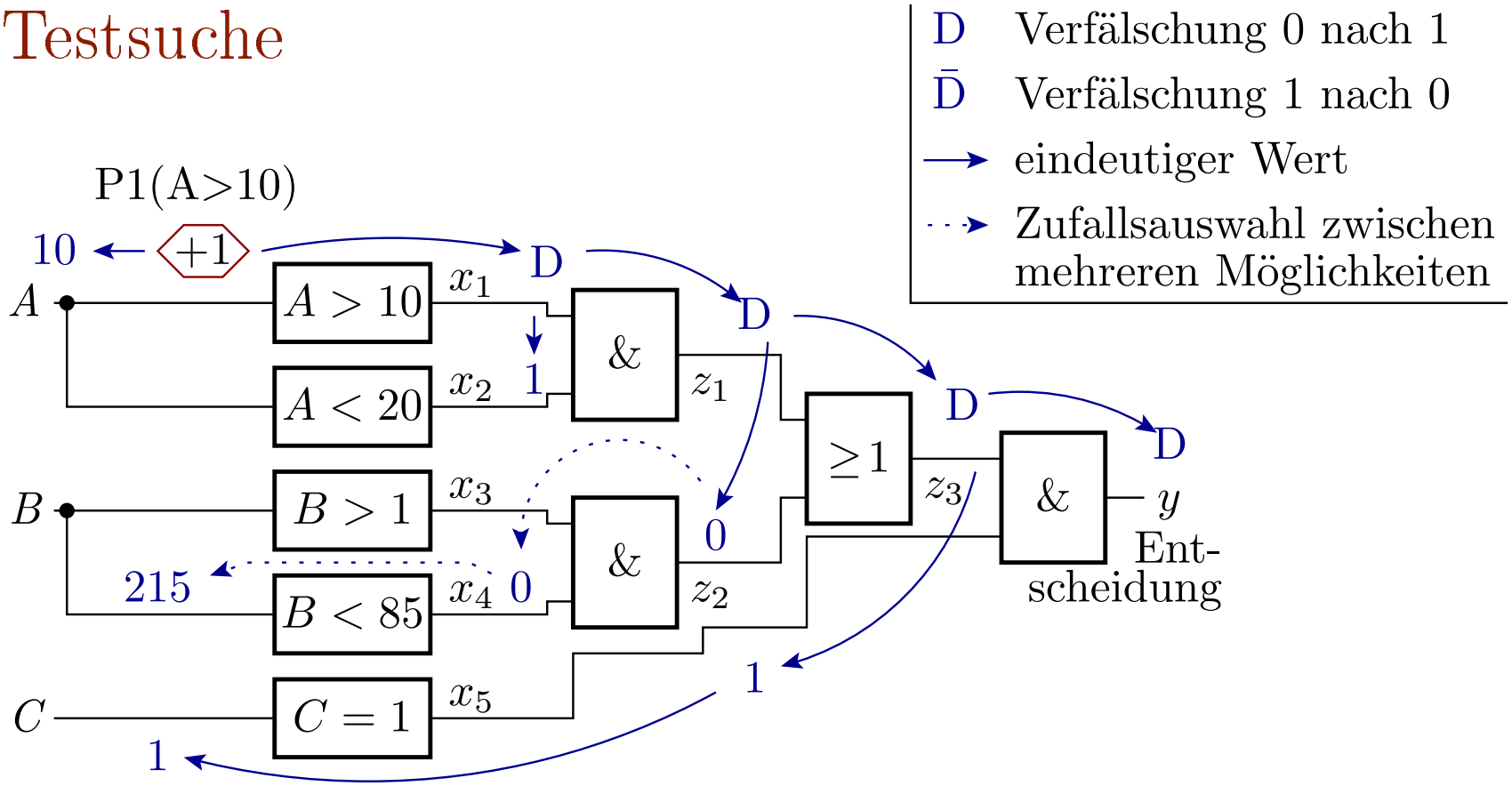
- Zusammenfassen identischer Fehler. Streichen redundanter und implizit nachweisbarer Fehler (vergl. Foliensatz F2, Abschn. 1.3 und dieser Foliensatz, Abschn. 2.2).
- Die ∓ 1-Fehler implizieren den Nachweis aller Haftfehler nach den Vergleichsoperatoren, ...
- Eventuelle redundante Fehler deuten auf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

Fehlersimulation und Testberechnung für die so zusammengestellte Modellfehler- (Mutations-) Menge innerhalb der logischen Ausdrücke:

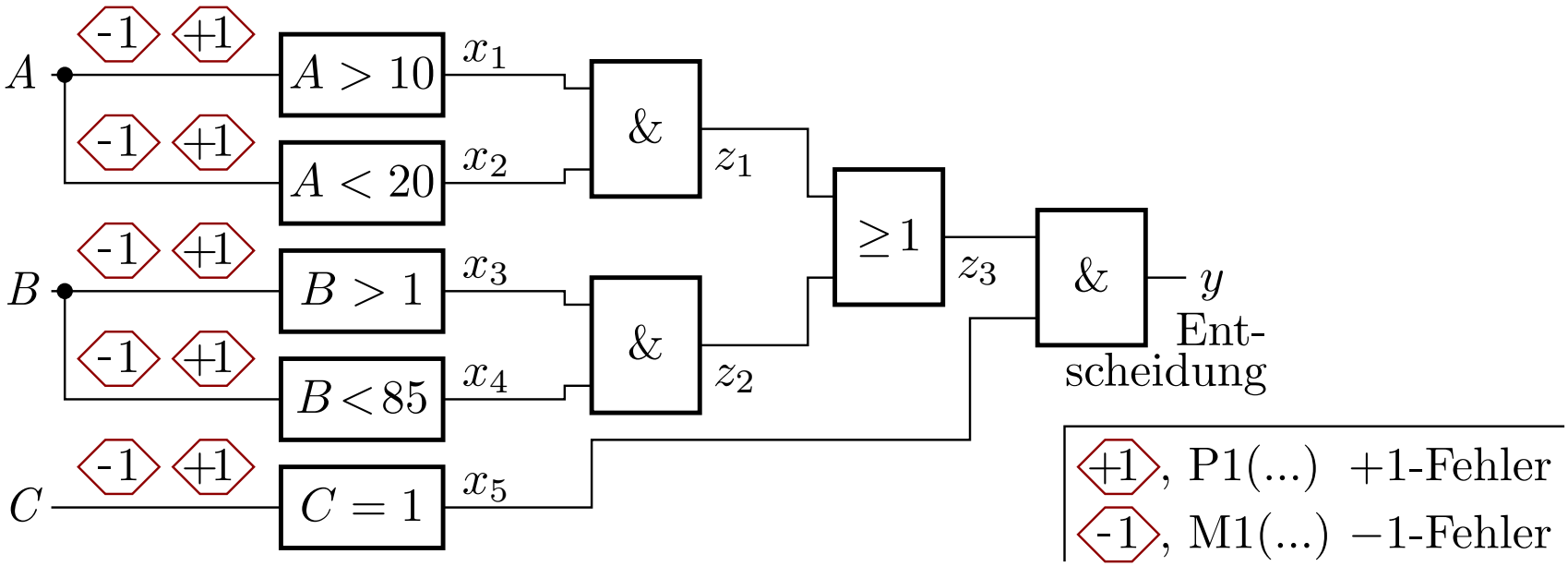
- könnte mit den für digitale Schaltungen etablierten Verfahren erfolgen.
- Das ist aber noch nicht Stand der Technik.

Die nächste Folie zeigt das Prinzip, wie für eine äquivalente Schaltung mit dem D-Algorithmus Tests berechnet werden.
D-Algorithmus siehe später Folie 106.

Testsuche

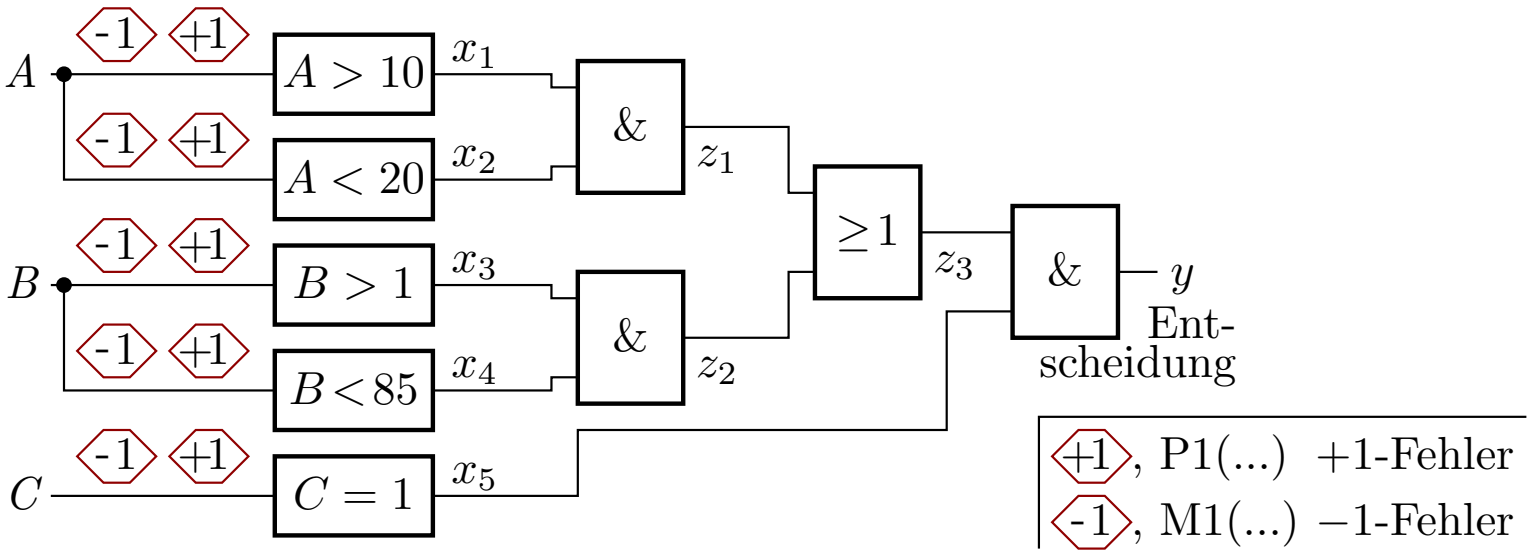


Die Testsuche im digitalen Teil der Beschreibung erfolgt im Beispiel mit dem für digitale Schaltungen etablierten D-Algorithmus (siehe später Folie 104), D – Pseudo-Wert zur Beschreibung einer fehlerbedingten Änderung von 0 nach 1).



	A	B	C	x_1	x_2	x_3	x_4	x_5	z_1	z_2	z_3	y
P1(A>10)	11	215	1	D	1	-	0	1	D	0	D	D
M1(A>10)												
P1(A<20)												
M1(A<20)												
P1(B>1)												
M1(B>1)												
P1(B<85)												
...												

Lösung



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

Testanforderungen für heutige Software

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

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

Nach Stand der Technik noch nicht gefordert:

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

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

Def-Use-Ketten

Def-Use-Ketten

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

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

Var	Def	Use
d	n1	n3
d	n1	n4
d	n1	n5
d	n1	n6
d	n1	n8
d	n8	n4
d	n8	n5
d	n8	n6
d	n8	n8
c	n0	n2
...	...	...

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

Berechnung und Verwendung von Def-Use-Ketten

Berechnung aller Def-Use-Tupel:

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

Verwendung als Testvollständigkeitskriterien:

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

Statische Code-Analyse:

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

Fehlerlokalisierung:

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

Beispiel: Rückverfolgung in »größter gemeinsamer Teiler«:

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

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

Beispielaufgabe

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

1. »n6« sei verfälscht. Was sind die möglichen »Defs«?
2. Wie vereinfacht sich die Rückverfolgung von Verfälschungen bei Aufzeichnung aller Anweisungsergebnisse als Trace?

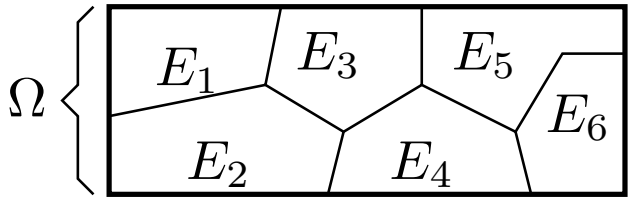
---

1. mögliche »Defs« für die Variable c: »n0« und »n6«. Mögliche »Defs« für die Variable d: »n1« und »n8«.
2. Erspart bei Rückverfolgungsschritten die Testwiederholung bis zu den potentiellen »Defs« davor.

Äquivalenzklassen

Testauswahl mit Äquivalenzklassen

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



Ω Eingaberaum
E_i Äquivalenzklasse

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

Testauswahl / Basis:

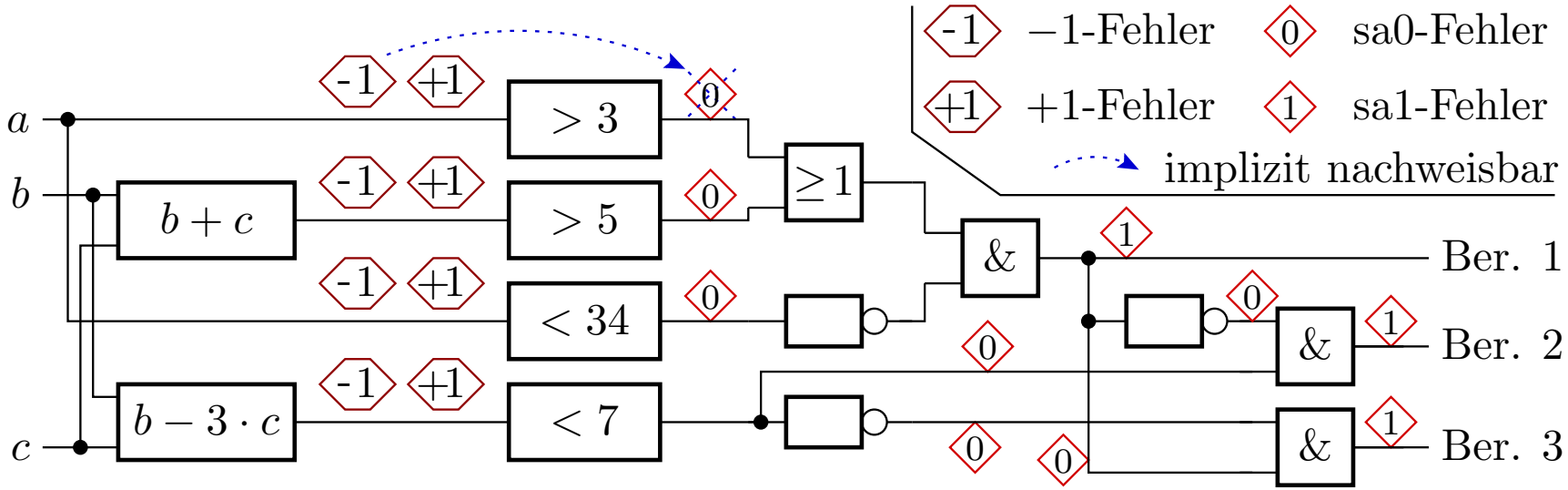
- Fertiges Programm: Testauswahl vergleichbar mit der für »Bedingungsüberdeckung« (siehe Folie 36).
- Spezifikation: Erstellen einer zum Testobjekt diversitären Fallbeschreibung. Weiter wie »Bedingungsüberdeckung«.

Spezifikationsbasierte Testauswahl

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

Beispiel einer aus der Spezifikation gewonnen Äquivalenzklassenbeschreibung

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



Testauswahl / Überdeckungskontrolle weiter wie »Bedingungsüberdeckung« ab Folie 36.

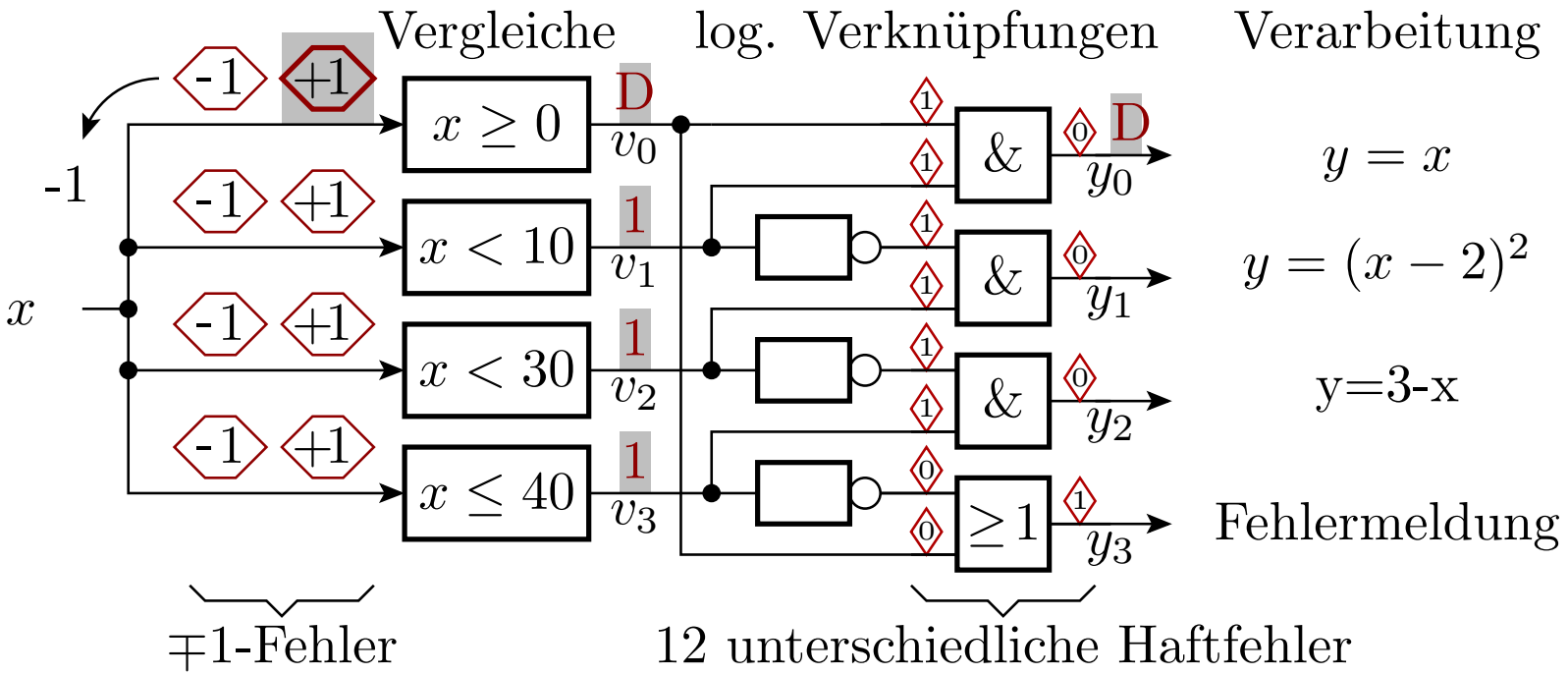
Beispielaufgabe

Gegeben ist die als Tabelle spezifizierte Funktion:

x	y
$0 \leq x < 10$	$y := x$
$10 \leq x < 30$	$y := (x-2)^2$
$30 \leq x \leq 40$	$y := 3 - x$
sonst	Fehlermeldung

1. Skizzieren Sie den Berechnungsfluss für eine äquivalenzklassenbasierte Testauswahl.
2. Zeichnen Sie alle nicht äquivalenten ∓ 1- und sa-Fehler ein.
3. Berechnen Sie einen Test für den $+1$-Fehler der Bedingung $(0 \leq x)$.

Lösung



Der $+1$-Fehler verlangt zur Anregung $x = -1$ und ist an

- $y_1 = \mathrm{D}$ bzw.
- »y=x« wird nicht ausgeführt

beobachtbar.

UW-Analyse

Ursache-Wirkungs-Analyse

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

Der empirische Ansatz ist anders.

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

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

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

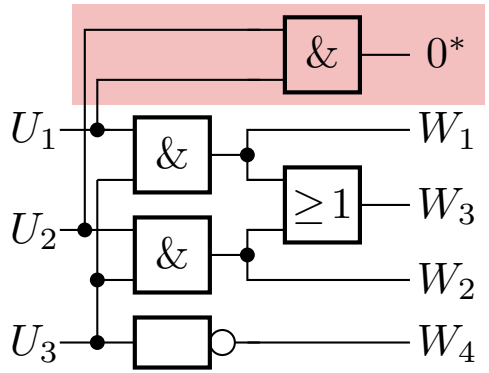
Wirkungen sind einzeln spezifizierte Zielfunktionen, ergänzte selbstverständliche Funktionen und Fehlerbehandlungen.

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

Zwischen den Ursachen und Wirkungen werden logische Verknüpfungen formuliert.

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

Beispiel »Zähle Zeichen«



* Eingabe kann nicht gleichzeitig Typ A und B sein

- Wirkungen:
  W_1: Anzahl_TypA +1[8]
  W_2: Anzahl_TypB +1
  W_3: Gesamtzahl +1
  W_4: Programm beenden
- Ursachen:
  U_1: Zeichen ist vom Typ A
  U_2: Zeichen ist vom Typ B
  U_3: Zeichenanzahl < Maximalwert
- Sich ausschließende Ursachen:
  UND-Verknüpfung muss »0« sein.

Test mit allen einstellbaren Ursachen

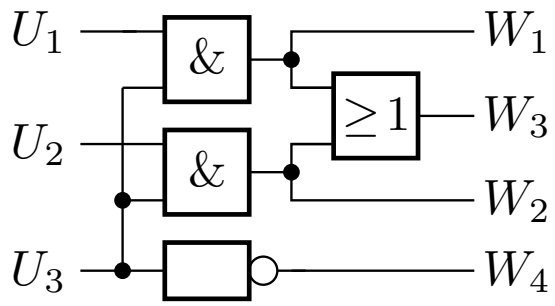
U_1	0	1	0	1	0	1	0	1
U_2	0	0	1	1	0	0	1	1
U_3	0	0	0	0	1	1	1	1
W_1	0	0	0		0	1	0	
W_2	0	0	0		0	0	1	
W_3	0	0	0		0	1	1	
W_4	1	1	1		0	0	0	

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

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

Beispielimplementierung als C-Funktion

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



Test mit allen einstellbaren Ursachen

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

Testbeispiel konkret /symbolisch

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

U_1 Zeichen ist vom Typ A (Ziffer)
U_2 Zeichen ist vom Typ B (Großbuchstabe)
U_3 max. Zählwert nicht erreicht
– es wird kein Zeichen gelesen

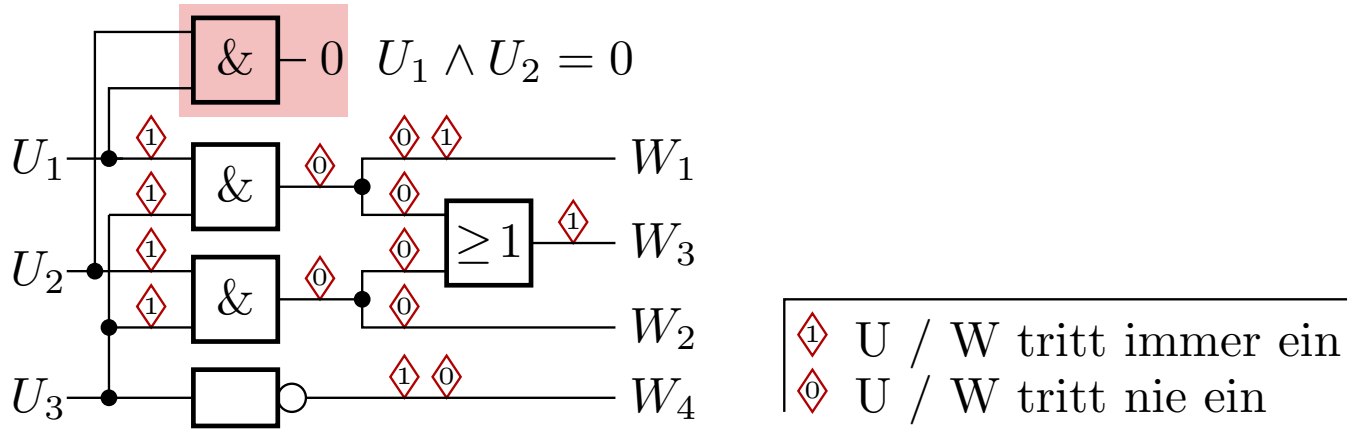
W_1 Ct_A++
W_2 Ct_B++
W_3 Ct_N++
W_4 Ende

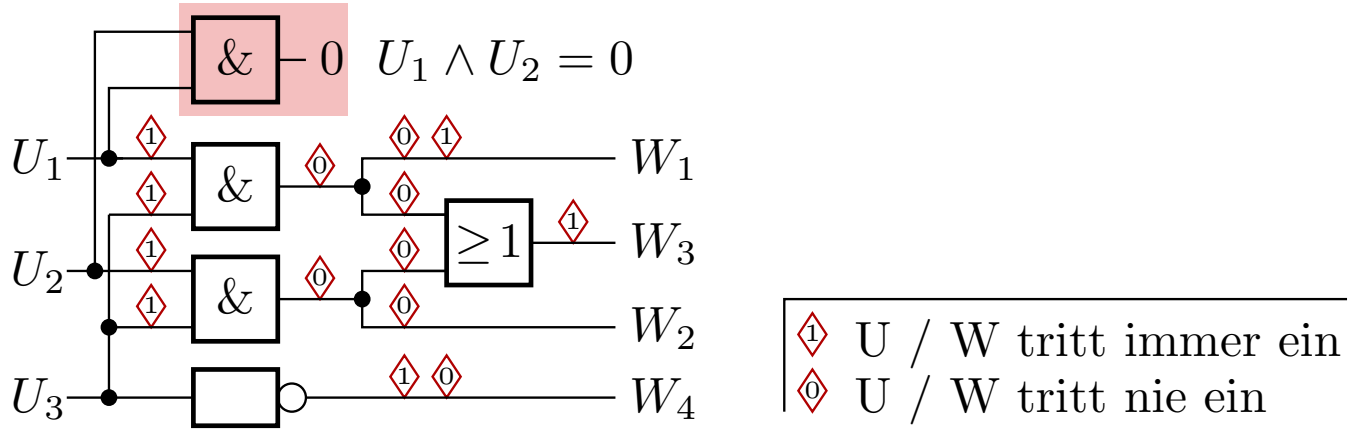
Ungereimtheiten / Haftfehler

Erkennbare Ungereimtheiten:

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

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

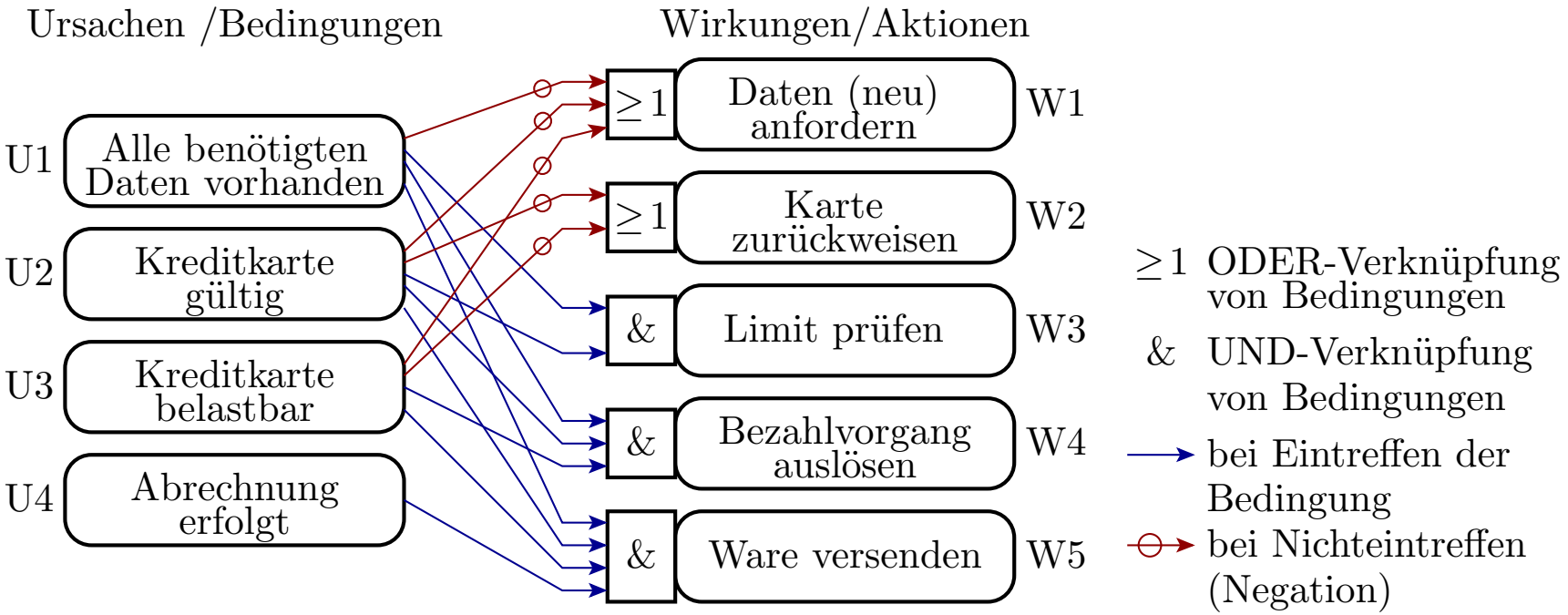




- Im Beispiel würde ein Test mit allen Kombinationen von Ursachen auch alle nachweisbaren Haftfehler erfassen.
- Für eine große Anzahl von Ursachen kann die Anzahl der Haftfehler auch wesentlich kleiner als die Anzahl der Ursachenkombinationen sein.
- Testsuche und Simulation für Haftfehler, siehe später Folie 82 und 104).
- Nach Berechnung der gleichzeitig zu (de-) aktivierenden Ursachen folgt die Suche geeigneter Eingaben und Kontrollen.

Beispielaufgabe

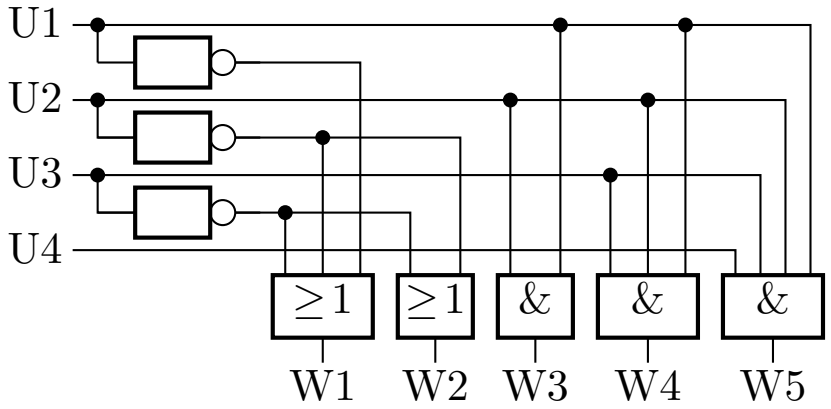
Gegeben ist das Ergebnis einer Ursache-Wirkungs-Analyse in einer anderen Darstellung aus [http://test.silke-wingens.de/].



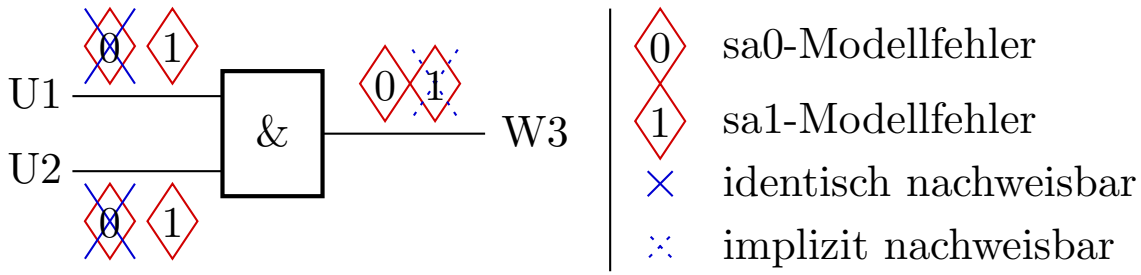
1. Stellen Sie die dargestellte Ursache-Wirkungs-Beziehung als logischen Signalflussplan dar.
2. Bestimmen Sie in dieser Darstellung für Wirkung W3 die Menge der unterschiedlich nachweisbaren Haftfehler ohne redundante und implizit nachweisbare Fehler.
3. Suchen Sie für alle (drei) Haftfehler eine Menge von Ursachenkombinationen, mit denen sie anhand ihrer Wirkung nachweisbar sind.
4. Bestimmen Sie für die (drei) Haftfehler die Nachweiswahrscheinlichkeiten für die Auftrittshäufigkeiten der Ursachen $h\,(\mathrm{U1}) = 30\%$, $h\,(\mathrm{U2}) = 70\%$, $h\,(U3) = 20\%$ und $h\,(U4) = 80\%$.

Lösung Aufgabenteil 1 und 2

1. Ursache-Wirkungs-Beziehung als logischen Signalflussplan:



2. Anfangsfehlermenge 6 Haftfehler. sa0(U1), sa0(U2) und sa0(W3) sind identisch und sa1(W3) implizit von sa1(U1) und sa1(U2) nachweisbar:



Lösung Aufgabenteil 3 und 4

3. Möglicher Testsatz:

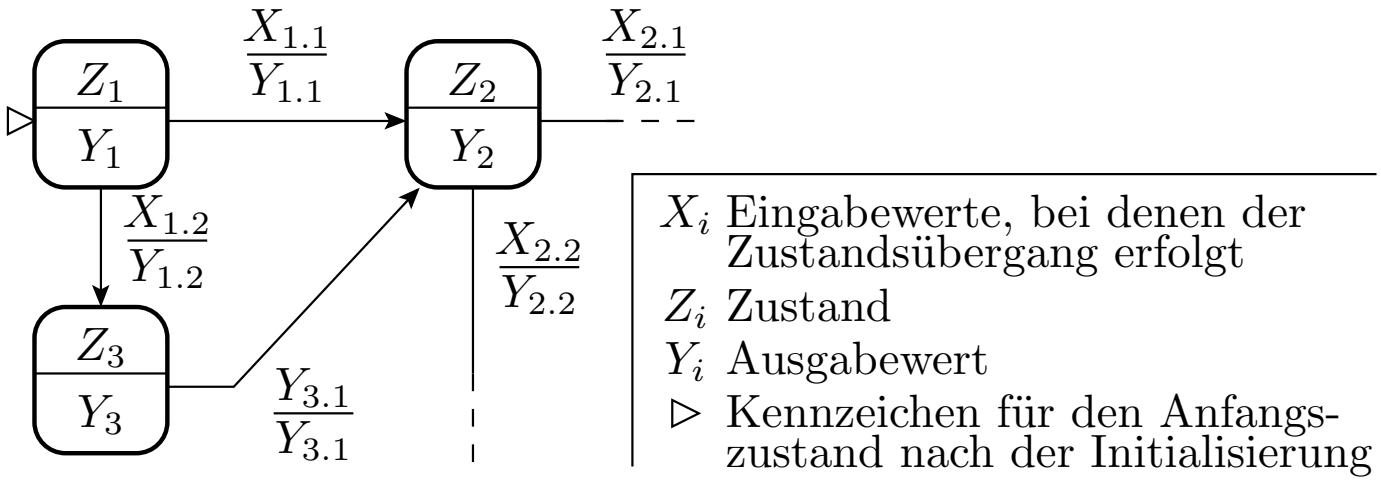
Fehler:	sa1(U1)	sa1(U2)	sa0(W2)
Test:	U1=0, U2=1	U1=1, U2=0	U1=1, U2=1

4. Nachweiswahrscheinlichkeit für $h\,(\mathrm{U1}) = 30\%$ und $h\,(\mathrm{U2}) = 70\%$:

U2	U1	Auftrittshäufigkeit	sa1(U1)	sa1(U2)	sa0(W2)
0	0	$30\% \cdot 70\% = 21\%$			
0	1	$30\% \cdot 30\% = 9\%$		x	
1	0	$70\% \cdot 70\% = 49\%$	x		
1	1	$70\% \cdot 30\% = 21\%$			x
Nachweiswahrscheinlichkeit:			49%	9%	21%

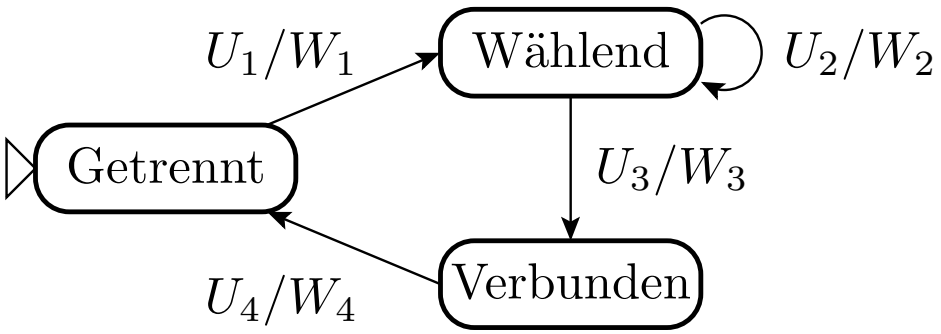
Automaten

Zielfunktion als Automat



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

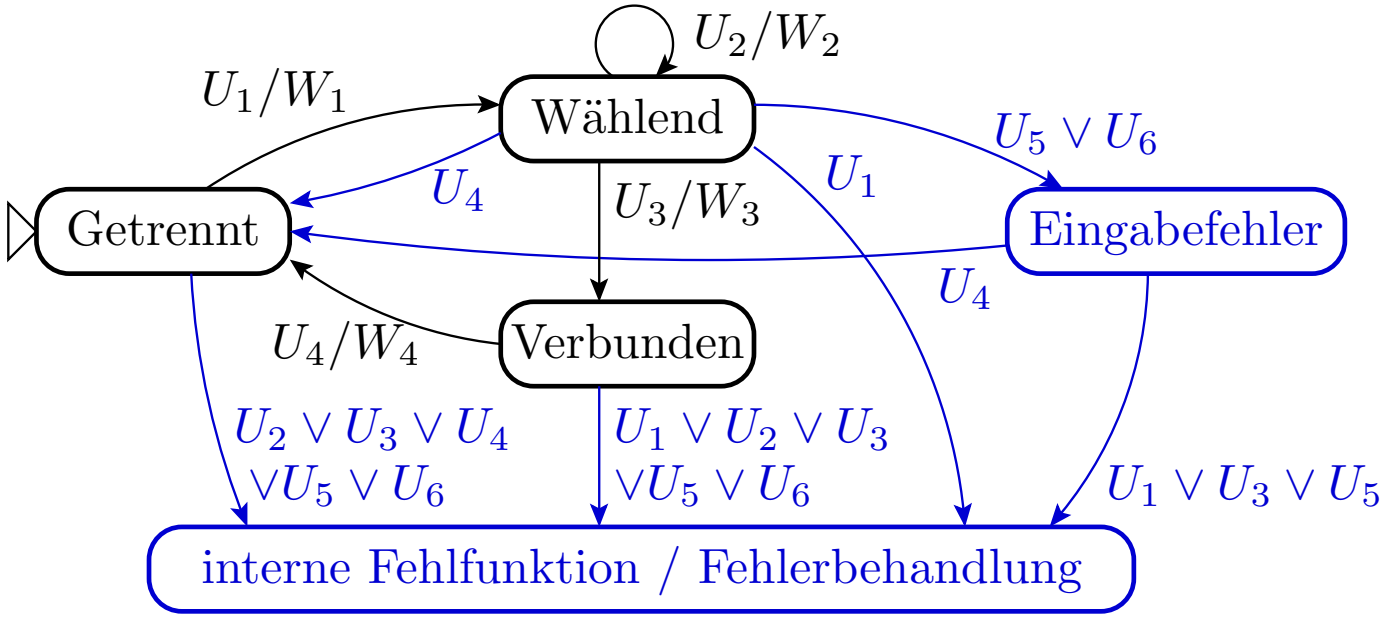
Verbindungsaufbau und -abbau beim Telefonieren



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

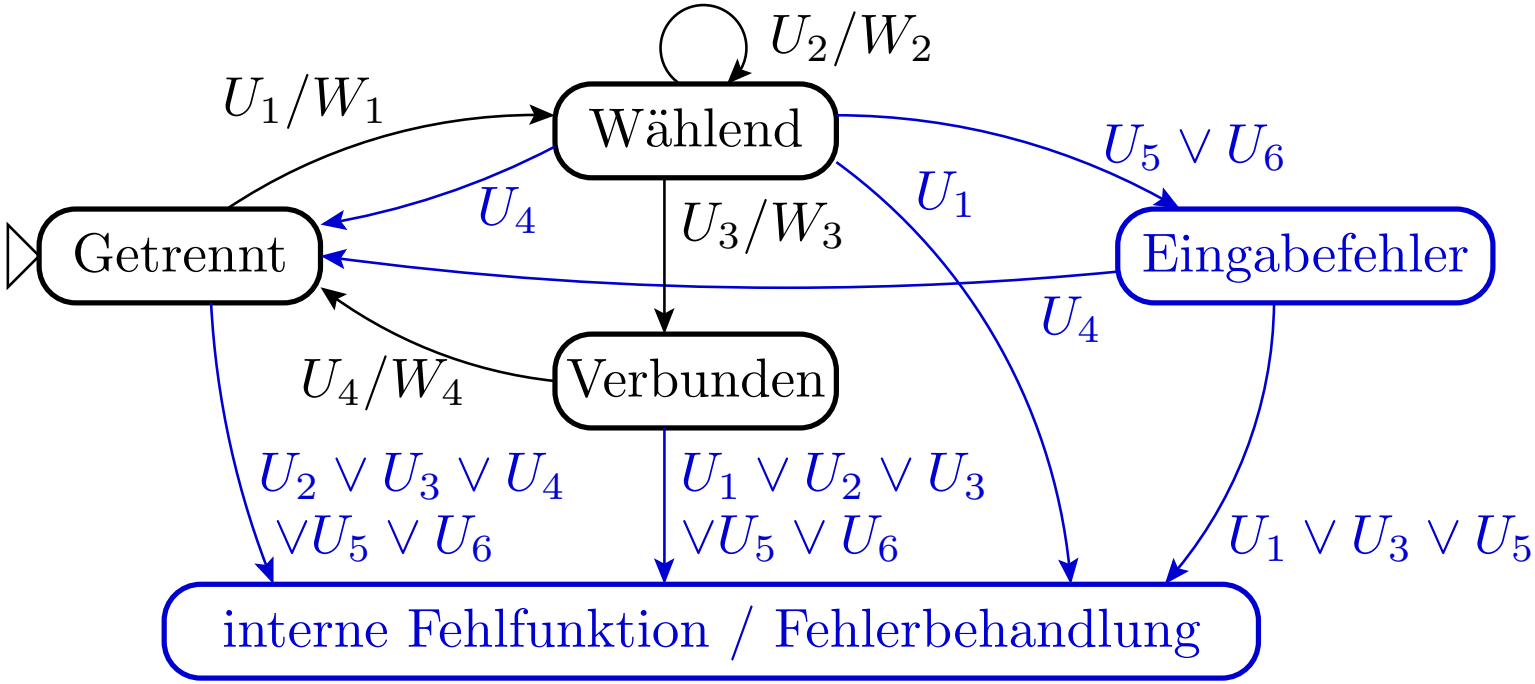
- Test der Sollfunktion: $U_1 \to U_2 \to \ldots \to U_2 \to U_3 \to U_4$
- Verhalten für andere Eingabefolgen?
  - Abnehmen, Wählen, Auflegen ($U_1 \to U_2 \to U_4$)
  - Abnehmen, Wählen, Wählen, falsche Nummer)
  - ...

  $\Rightarrow$ Ablaufgraph ist noch unvollständig



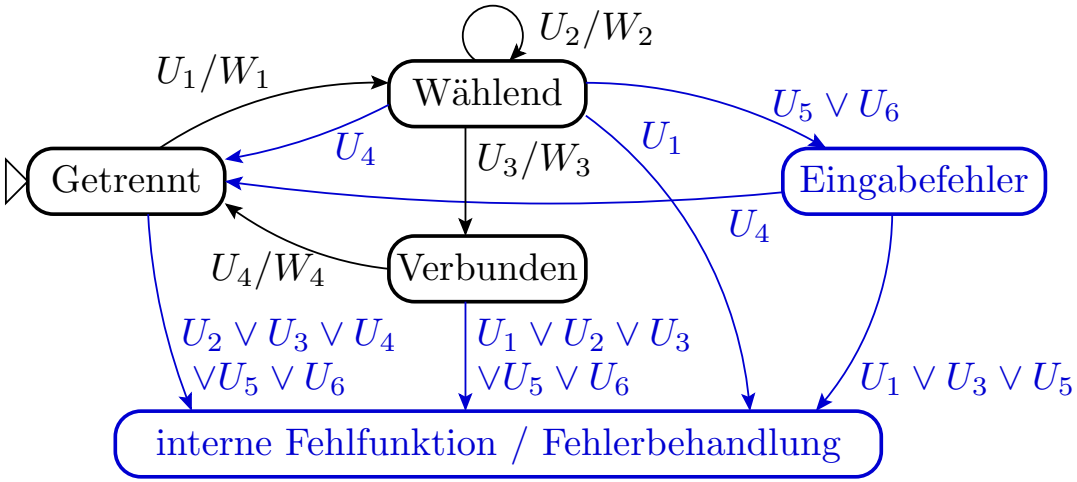
U_1	Abnehmen	W_1	Rufnummer zurücksetzen
U_2	Ziffer wählen	W_2	Ziffer zur Rufnummer hinzufügen
U_3	Rufnummer gültig	W_3	Verbindung aufbauen
U_4	Auflegen	W_4	Verbindung trennen
U_5	Rufnummer ungültig		
U_6	Timeout		

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



Test aller Zustandsübergänge, Wirkungen, ...

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



Test der Reaktion auf interne Fehlfunktionen

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

Auswahlregeln sind wie bei der kontrollflussorientierten Auswahl:

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

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

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

Schaltkreistest

Test digitaler Schaltkreise

Besonderheiten:

- Tausende bis Millionen von Logikfunktionen je Testobjekt.
- Sehr eingeschränkte Beobachtbarkeit interner Signale.
- Hohe Anforderungen an Ausbeute und Test. Überschlag:
  - Ausbeute $Y \approx 50\%$,
  - angestrebter Fehleranteil nach dem Test $DL \approx 100\,\mathrm{dpm}$,
  - zu fordernde Fehlerüberdeckung $FC \approx 99{,}9\%$.

Für digitale Schaltungen sind

- fehlerorientierte Testauswahl, Haftfehler,
- Zufallstests mit sehr vielen Testbeispielen und
- automatische Testsatzberechnung

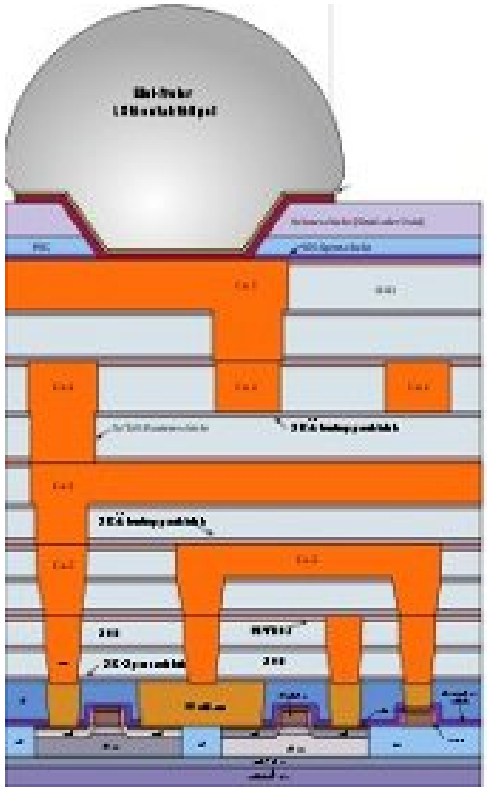
Seit über drei Jahrzehnten Stand der Technik.

Fertigungsfehler

Fertigungsfehler integrierter Schaltkreise

Schaltkreise werden schichtweise hergestellt:

- Auftragen von Schichten (z.B. Fotolack oder Metall).
- Belichten des Fotolacks durch eine Maske, die die Geometrie der zu erzeugenden Schichtelemente festlegt.
- Entfernen der belichteten (unbelichteten) Bereiche des Fotolacks.
- Fortätzen der freiliegenden Schichten neben dem Fotolack und entfernen des Fotolacks.

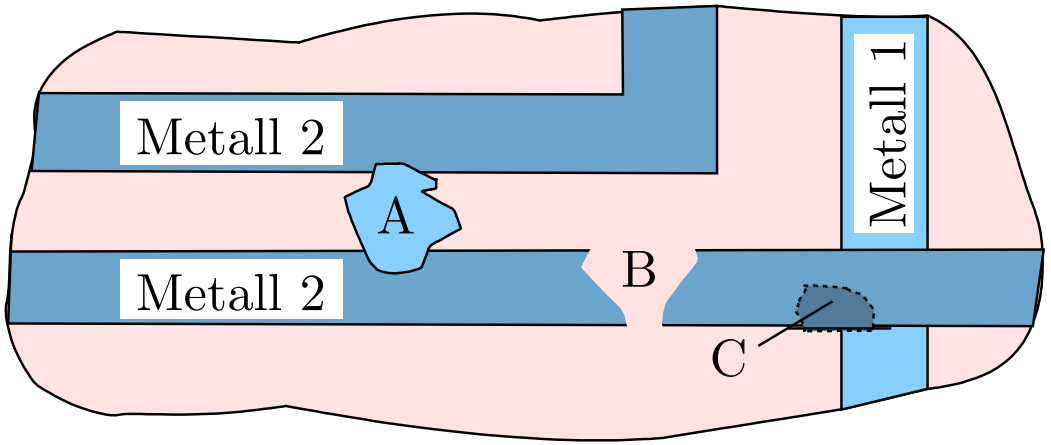


Typische Herstellungsfehler:

- fehlendes (zu wenig aufgetragenes zu viel weggeätztes) und
- überflüssiges (zu viel aufgetragenes, zu wenige weggeätztes)

Leitungs- oder Isolationsmaterial in einer Schicht.

Wirkung: Kurzschlüsse, Unterbrechungen, nicht richtig ein- oder ausschaltende oder zu langsam schaltende Transistoren.



A zuätzliches Metall
B fehlendes Metall
C fehlende Isolation

- Mehrfachfehler durch einzelne Fehlerfläche möglich.
- Einzelfehlerannahme genügt, weil ein Test für Einzelfehler auch die meisten Mehrfachfehler nachweist.

Notwendige Nachweisbedingungen:

- Anregung: Einstellung 0 bzw. 1 am Fehlerort.
- Beobachtung: Sensibilisierung eines Beobachtungspfads vom Fehlerort zu einem Ausgang.

Alternativen/Ergänzungen zur Kontrolle des logischen Verhaltens:

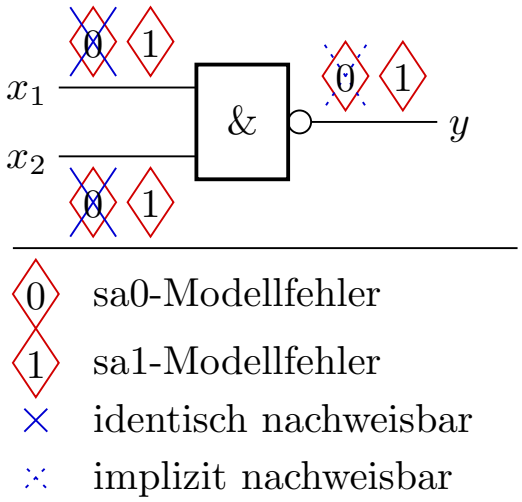
- Kontrolle der Signalverzögerungen und
- Kontrolle der Stromaufnahme, insbesondere der Ruhestromaufnahme, die bei CMOS-Schaltungen sehr geringe Sollwerte hat,

während der Tests.

Haftfehler

Haftfehlermodell

Das gebräuchlichste Fehlermodell ist das bereits auf Foliensatz F1 eingeführte Haftfehlermodell. Es unterstellt für jedes binäre Datensignal, dass es durch einen Fehler ständig auf null (stuck-at 0) oder ständig auf eins (stuck-at-1) gehalten wird.



x_2	x_1	$\overline{x_2 \wedge x_1}$	sa0(x_1)	sa1(x_1)	sa0(x_2)	sa1(x_2)	sa0(y)	sa1(y)
0	0	1	1	1	1	1	0	1
0	1	1	1	1	1	0	0	1
1	0	1	1	0	1	1	0	1
1	1	0	1	0	1	0	0	1
				Nachweisidentität (gleiche Nachweismenge)				

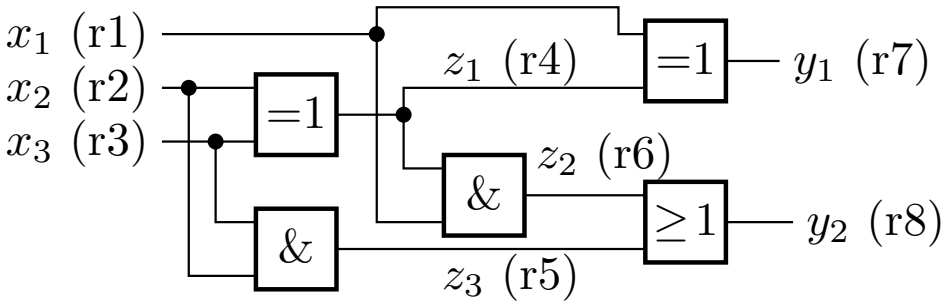
Nachweisimplikation

zugehörige Eingabe ist Element der Nachweismenge

Aus der so berechneten Anfangsmenge werden redundante Fehler, identisch nachweisbare Fehler und (optional) implizit nachweisbare Fehler gestrichen.

Haftfehler sind einfach zu simulieren

Schaltung eines Volladdierers



r1 bis r8 Prozessorregister

Programm für die Gutsimulation

lade x_1 in Register r1
lade x_2 in Register r2
lade x_3 in Register r3
r4 = r2 xor r3
speichere Inhalt r4 in z_1
r5 = r2 and r3
speichere Inhalt r5 in z_3
r6 = r1 and r4
speichere Inhalt r6 in z_2
r7 = r1 xor r4
speichere Inhalt r7 in y_1
r8 = r5 or r6
speichere Inhalt r8 in y_2

- Jede zweistellige Logikoperation ist ein Maschinenbefehl.
- In jeder der 8, 16, 32 oder 64 Bits der Operanden kann ein anderer Testfall oder ein anderer Fehler simuliert werden.

Aufwandsabschätzung am Beispiel

- Schaltungsgröße: 10^4 Gatter
- Anzahl der Testschritte / Testeingaben: 10^4
- Anzahl der Modellfehler: 10^4
- Simulationsaufwand je Gatter: 10 ns

Rechenaufwand, wenn jeder Fehler mit allen Testeingaben simuliert wird und ohne bitparallele Simulation: 10^4 s, ca. 3 h.

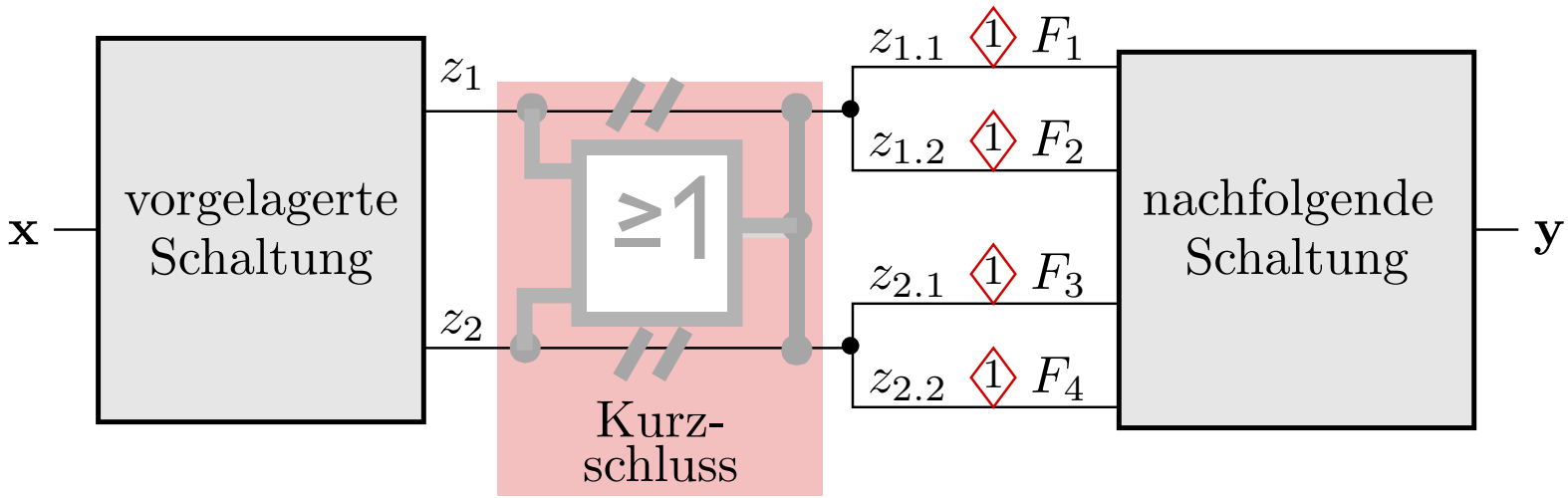
Wenn mit jedem der 32 bzw. 64 Bits ein anderer Fehler simuliert wird, nur 6 bzw. 3 Minuten.

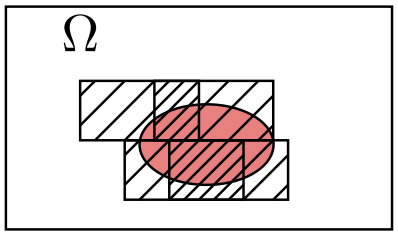
Kurzschlussnachweis mit einem Haftfehlertestsatz

- Zum Nachweis eines Kurzschlusses müssen auf den beteiligten Leitungen unterschiedliche Werte eingestellt und das dabei verfälschte Signal beobachtet werden.
- Die Anzahl der möglichen Kurzschlüsse nimmt im ungünstigsten Fall[9] mit dem Quadrat der Leitungsanzahl zu. Viel größer als die Anzahl der Haftfehler.
- Fehlersimulation und Testsatzberechnung aufwändiger als für Haftfehler, z.B. durch Berücksichtigung zusätzlicher Speicherzustände (siehe später Folie 87).
- Für Haftfehler ausgewählte Testsätze erkennen die meisten Kurzschlussmöglichkeiten, so dass auf eine explizite Modellierung in der Regel verzichtet wird.

[9] Verdrahtung und damit potentielle Kurzschlusskandidaten unbekannt.

Kurzschlussnachweis mit einem Haftfehlertestsatz



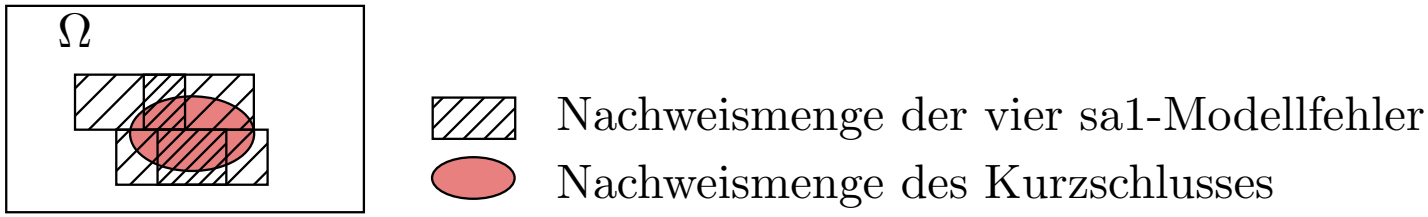


1 Modellfehler (ständig 1)

Nachweismenge der vier Modellfehler

Nachweismenge des Kurzschlusses

- Für jeden Haftfehler wird mindestens ein Test gesucht.
- Wie wahrscheinlich ist der Kurzschlussnachweis?



Der Kurzschluss ist nachweisbar

- $z_2 = 0$ und F_1 oder F_2 nachweisbar oder
- $z_1 = 0$ und F_3 oder F_4 nachweisbar ist.

Überschläge:

- Gezielte Testsatzsuche: $FC_{\mathrm{sa}} = 1$, $N \geq 4$ Versuche mit Kurzschlussnachweiswahrscheinlichkeit 50%:

$$p_{\mathrm{E}} \geq 1 - 0{,}5^4 = 93{,}7\%$$

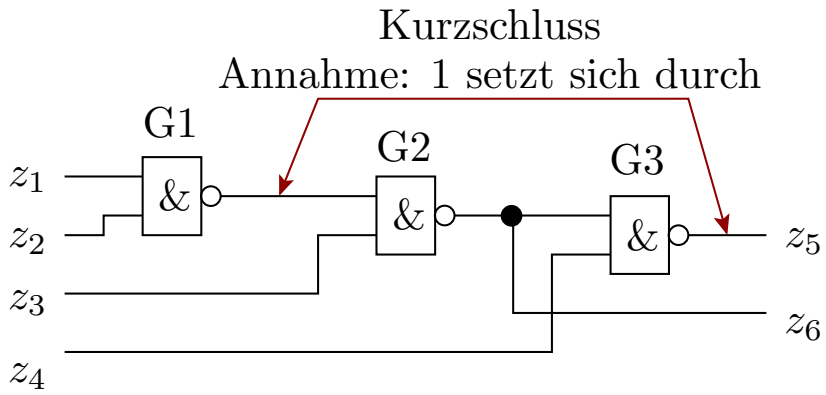
- Zufällige Testauswahl: Tendenziell doppelt so große Nachweismenge wie die der vier ähnlich nachweisbaren Haftfehler:

$$H_{\mathrm{Kurzschl}}(x) \sim H_{\mathrm{sa}}(2 \cdot x)$$

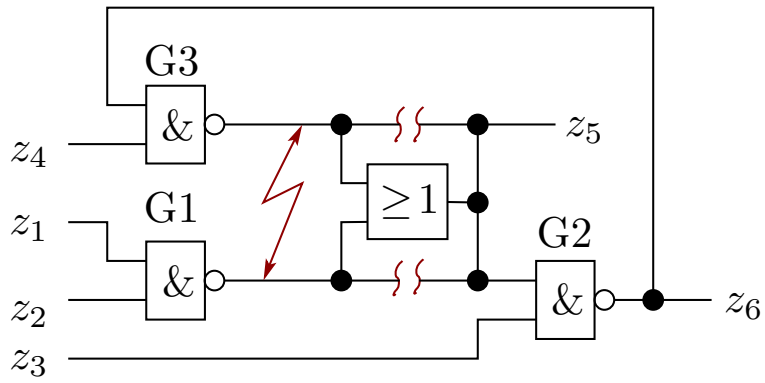
- Grobabschätzung: $FC_{\mathrm{Kurzschl}}(n) \approx FC_{\mathrm{sa}}\left(\frac{n}{2}\right)$.

Kurzschlüsse können Gatterschaltungen zu Speicherzellen umbilden.

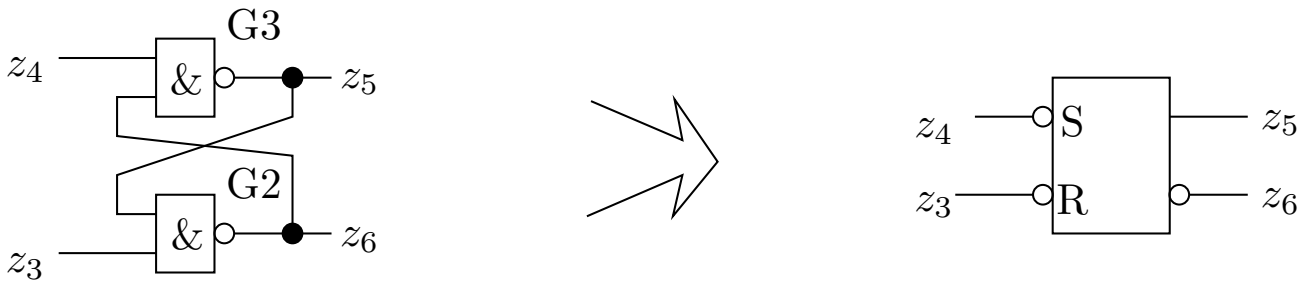
Schaltung mit Kurzschluss



Kurzschlussnachbildung durch ein ODER



Ersatzschaltung für $z_1 = z_2 = 1$

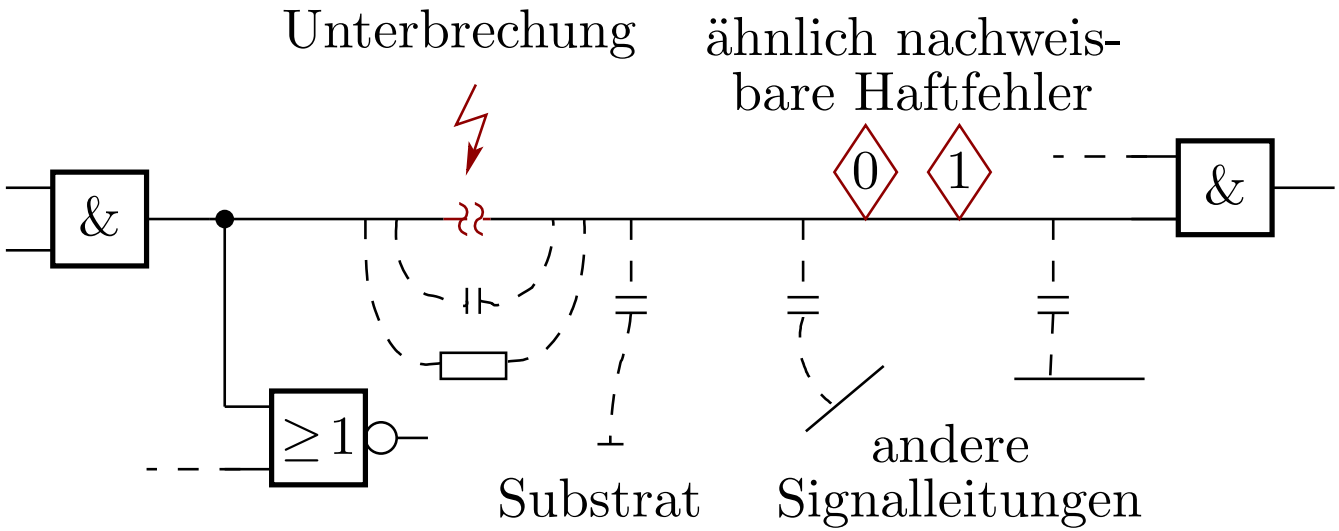


Für Fehler mit Speicherverhalten Fehlersimulation und Testberechung nach pseudo-kombinatorischem Iterationsmodell (siehe später Abschn. 2.5).

Zusammenfassung

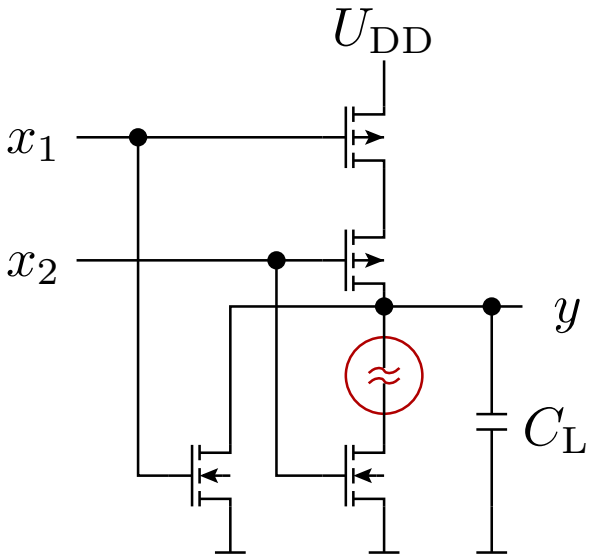
- Kurzschlüsse können sehr vielfältige Fehlerwirkungen haben.
- Berücksichtigung aller Möglichkeiten in der Regel weder notwendig, noch vom Aufwand her zu rechtfertigen.
- Gezielt berechnete Haftfehlertestsätze mit hoher Fehlerüberdeckung erkennen die meisten Kurzschlüsse.
- Die Kurzschlussüberdeckung hängt dabei jedoch weniger von der Haftfehlerüberdeckung, sondern mehr von der Anzahl der Tests, die je Haftfehler gesucht werden, ab.
- Bei zufälliger Testauswahl ist die zu erwartende Kurzschlussüberdeckung etwa gleich der zu erwartenden Haftfehlerüberdeckung für einen kürzeren Testsatz.
- Haftfehlerüberdeckung hat für Zufallstests mehr Aussagewert als für gezielt berechnete Testsätze.

Fehlerwirkung von Unterbrechungen



- Die abgetrennten Gattereingänge können dauerhaft auf null oder eins liegen, driften oder den korrekten Wert erst nach erheblicher Verzögerung übernehmen.
- Überwiegender Nachweis mit Haftfehlertests für die nachfolgenden Gattereingänge.
- Sicherer Nachweis durch Kontrolle der Signalverzögerung.

Stuck-open-Fehler



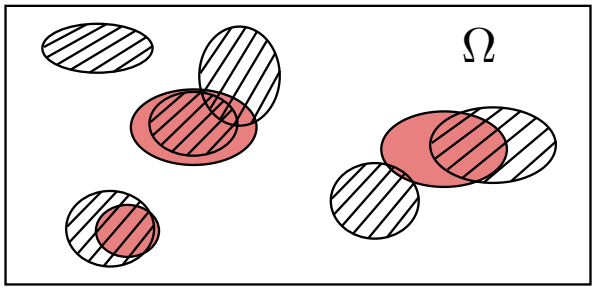
x_2	x_1	y
0	0	1 (setzen)
0	1	0 (rücksetzen)
1	0	speichern
1	1	0 (rücksetzen)

x_2	x_1	y
0	0	1
0	1	0
0	0	1
1	0	0

Unterbrechung innerhalb eines Gatters, so dass der Gatterausgang für bestimmte Eingaben nicht auf- bzw. entladen wird.

- Überwiegender Nachweis mit Haftfehlertests.
- Sicherer Nachweis durch Kontrolle der Signalverzögerung.

Reale Schaltkreisfehler und Haftfehler



Ω Menge der Eingabewerte /Teilfolgen die einen Fehler nachweisen können

Nachweismenge einer Mutation

Nachweismenge eines tatsächlichen Fehlers

- Die möglichen Wirkungen von Schaltkreisfehlern sind wesentlich vielfältiger als gezeigt (z.B. null setzt sich bei Kurzschluss durch, ...)
- Jeder logisch nachweisbare Schaltkreisfehler teilt sich Anregungsbedingungen und Beobachtungspfade mit Haftfehlern. Tendenz $H(p) \sim H_{\mathrm{sa}}(c \cdot p)$ mit $c > 1$.
- Zufallstest: $FC(n) \approx FC_{\mathrm{sa}}\left(\frac{n}{c}\right)$.
- Gezielte Testauswahl: $FC_{\mathrm{sa}} \Uparrow \Rightarrow FC \Uparrow$, wobei FC erheblich davon abhängt, wie viele Tests je Haftfehler gesucht werden.

Validierung mit Zahlen aus der Literatur

Der typische Schaltkreistest hat eine Überdeckung für Haft- oder Verzögerungsfehler von 95% bis 100% und erkennt etwa 99,9% der Herstellungsfehler. Der Wert 99,9% ist keine publizierte Zahl, sondern über folgenden Überschlag mit typ. Werten abgeschätzt:

Von 10^6 gefertigten Schaltkreisen

- sind etwa 10% bis 50% fehlerhaft,
- werden etwa 99,9% der defekten Schaltkreise von den Fertigungstests erkannt und aussortiert.
- Jeder 1000ste bis 10.000ste eingesetzte Schaltkreis hat einen kaum nachweisbaren Fehler.
- Jeder 10te Arbeitsplatzrechner enthält einen defekten Schaltkreis.

Daraus folgt, dass die tatsächlichen Schaltkreisfehler im Mittel deutlich besser als Haftfehler nachweisbar zu sein scheinen.

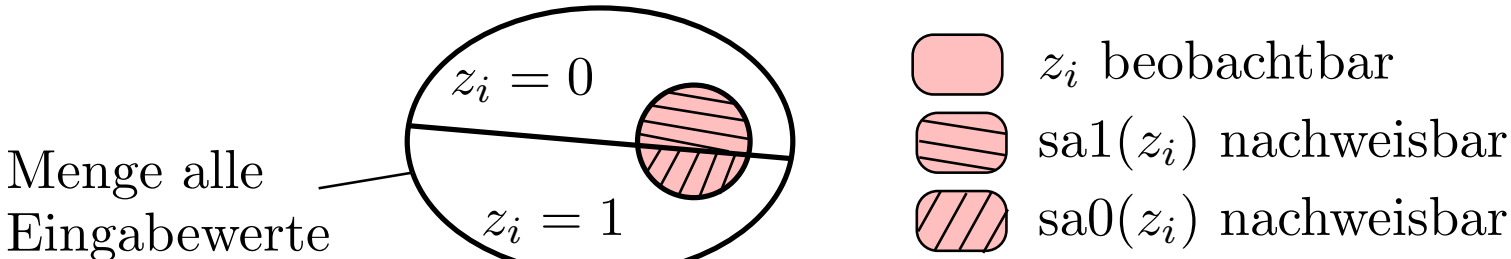
Andere Fehlermodelle

Weitere für Schaltkreise diskutierte Fehlermodelle

- Toggle Test: Der Testsatz muss jede Leitung mindestens einmal auf null und einmal auf eins steuern.
- Zellenfehlermodell: Teilschaltung mit 1-Bit-Ausgabe funktioniert genau mit einer Eingabe nicht.
- Gatterverzögerungsmodell: Für jedes Schaltelement werden die beiden Modellfehler verzögerter Anstieg (slow to rise) und verzögerter Abfall (slow to fall) unterstellt.
- Pfadverzögerungsfehler: Für jeden Schaltungspfad werden die beiden Modellfehler verzögerter Anstieg (slow to rise) und verzögerter Abfall (slow to fall) unterstellt.

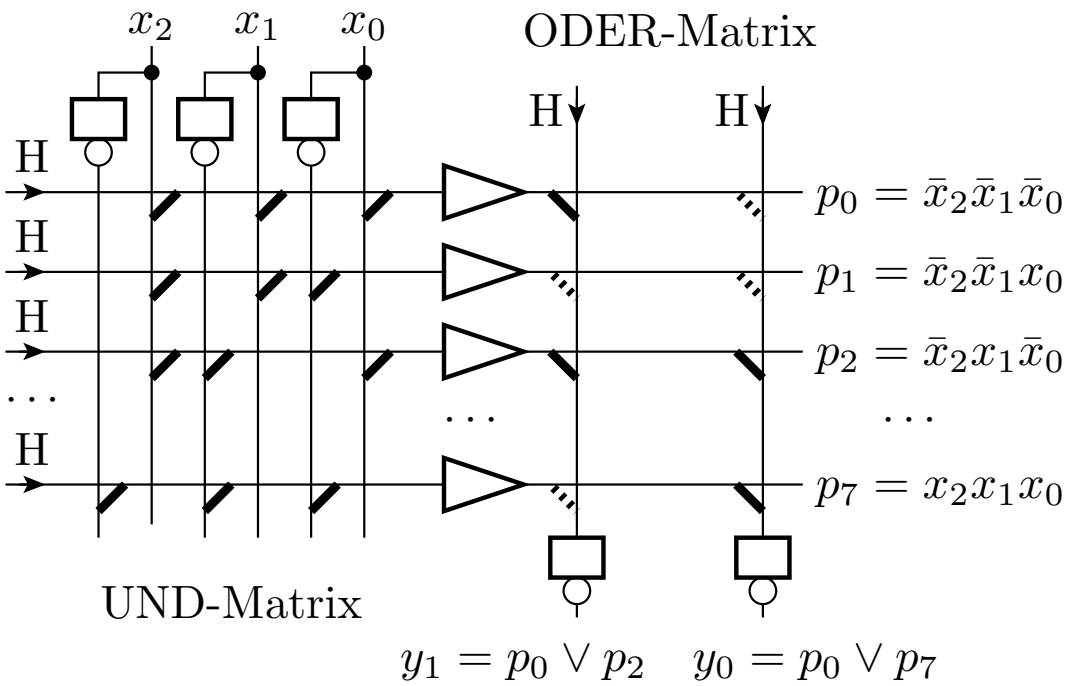
Toggle-Test

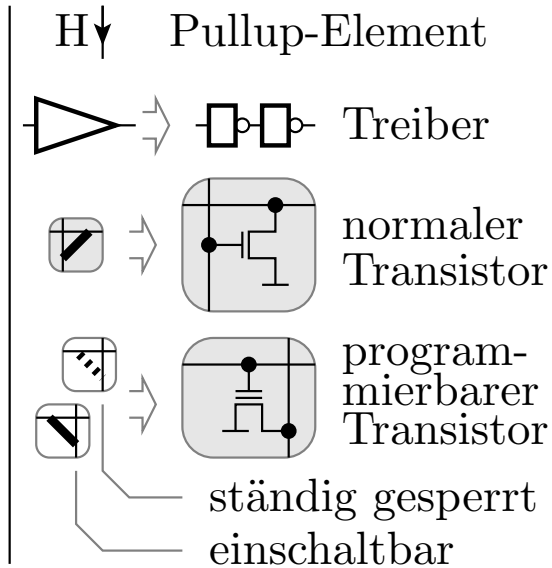
Auswahlkriterium Toggle Test: Jedes logische Signal bei Testabarbeitung mindestens einmal »0« und einmal »1«.



- Garantiert Steuerbarkeit für alle Haftfehler. Gleichzeitige Beobachtbarkeit ist Zufall ($H(p) \sim H_{\text{Toggle}}(c \cdot p)$ mit $c \ll 1$).
- Vergleichbar mit »Anweisungsüberdeckung« für SW.
- Zufallstest: Fehlerüberdeckung ist etwa die Toggle-Überdeckung der $\left(\frac{1}{c} \gg 1\right)$-fachen Testsatzlänge.
- Gezielter Testauswahl: Toggle-Überdeckung erlaubt kaum Aussagen über die Fehlerüberdeckung.
- Für HW veraltet.

Zellenfehlermodell (ROM, LUTs, ...)





Für jede Programmierstelle, Annahme falsch gesetzt oder für jedes Ausgabebit in der Wertetabelle Ausgabe invertiert. Ein Testsatz mit 100% Zellenfehlerüberdeckung weist jede kombinatorische Funktionsabweichung und auch jeden Haftfehler nach.

Sollfunktion

$x_2\ x_1\ x_0$	$y_1\ y_0$
0 0 0	1 1
0 0 1	0 0
0 1 0	0 1
0 1 1	1 1

Fehler	verfälschtes Bit
1	y_0 für $\mathbf{x} = 000$
2	y_1 für $\mathbf{x} = 000$
3	y_0 für $\mathbf{x} = 001$
4	y_1 für $\mathbf{x} = 001$

Anzahl der Modellfehler:

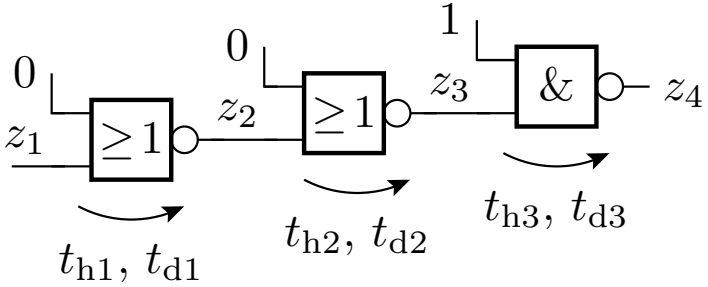
$$\varphi_{\mathrm{M}} = N_{\mathrm{A}} \cdot 2^{N_{\mathrm{E}}}$$

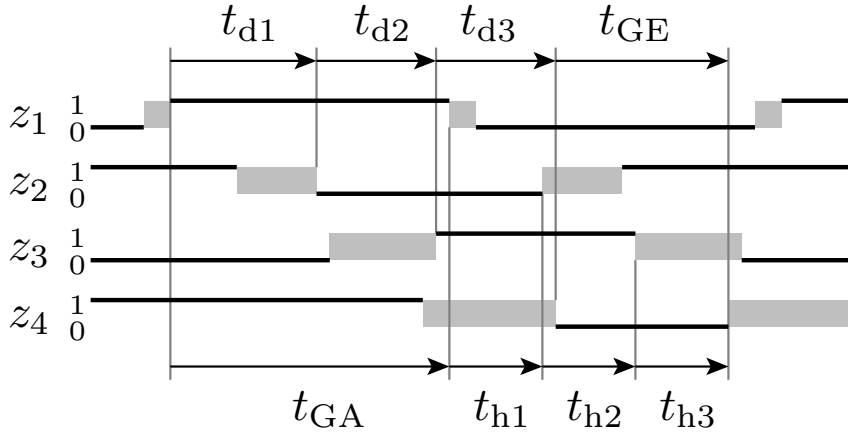
(N_{E} – Anzahl der Eingänge; N_{A} – Anzahl der Ausgänge). Vergleich Zellenfehler und Haftfehler für freistrukturierte Schaltungen (Realisierung aus Gattern statt durch programmierte Speicher):

- u.U. wesentlich mehr Modellfehler,
- viele davon im Systemverbund redundant,
- Nachweis der Redundanz schwer zu erbringen.

Geeignet für LUTs, Volladdierer, ... Für Gatterschaltungen aus UND, ODER, ... ist das Haftfehlermodell besser geeignet.

Gatterverzögerungsfehler





t_{GA}	Gültigkeitsdauer am Pfadanfang
t_{GE}	Gültigkeitsdauer am Pfadende

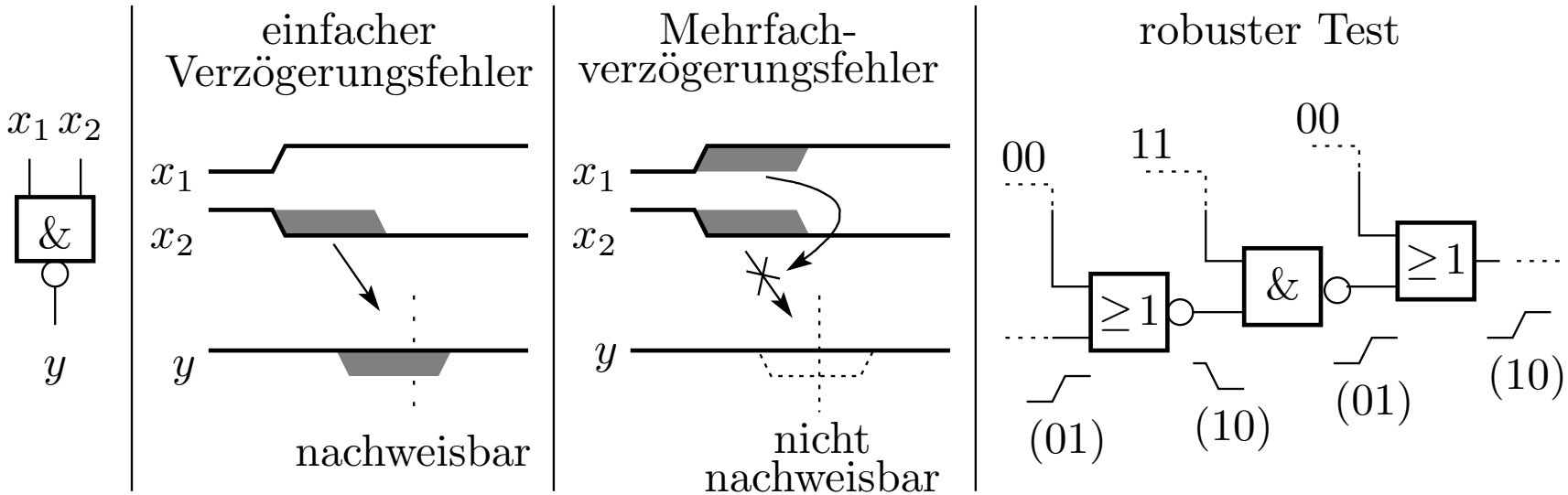
- Modellfehler: slow-to-rise-/ slow-to-fall-Fehler.
- 2-Pattern-Test: Initialisierungseingabe + Haftfehlertest.
- FHSF-Funktion unter Annahme $g = 50\%$:

$$H_{\mathrm{GVerz}}(x) \sim H_{\mathrm{sa}}\left(\frac{x}{2}\right)$$

- Zufallstest: Gleiche Überdeckung wie für Haftfehler erfordert etwa doppelte Testsatzlänge ($FC_{\mathrm{GVerz}}(n) \approx FC_{\mathrm{sa}}\left(\frac{n}{2}\right)$).
- Gezielte Auswahl: mindestens 2 Tests je Haftfehler suchen.

Pfadverzögerungsfehler

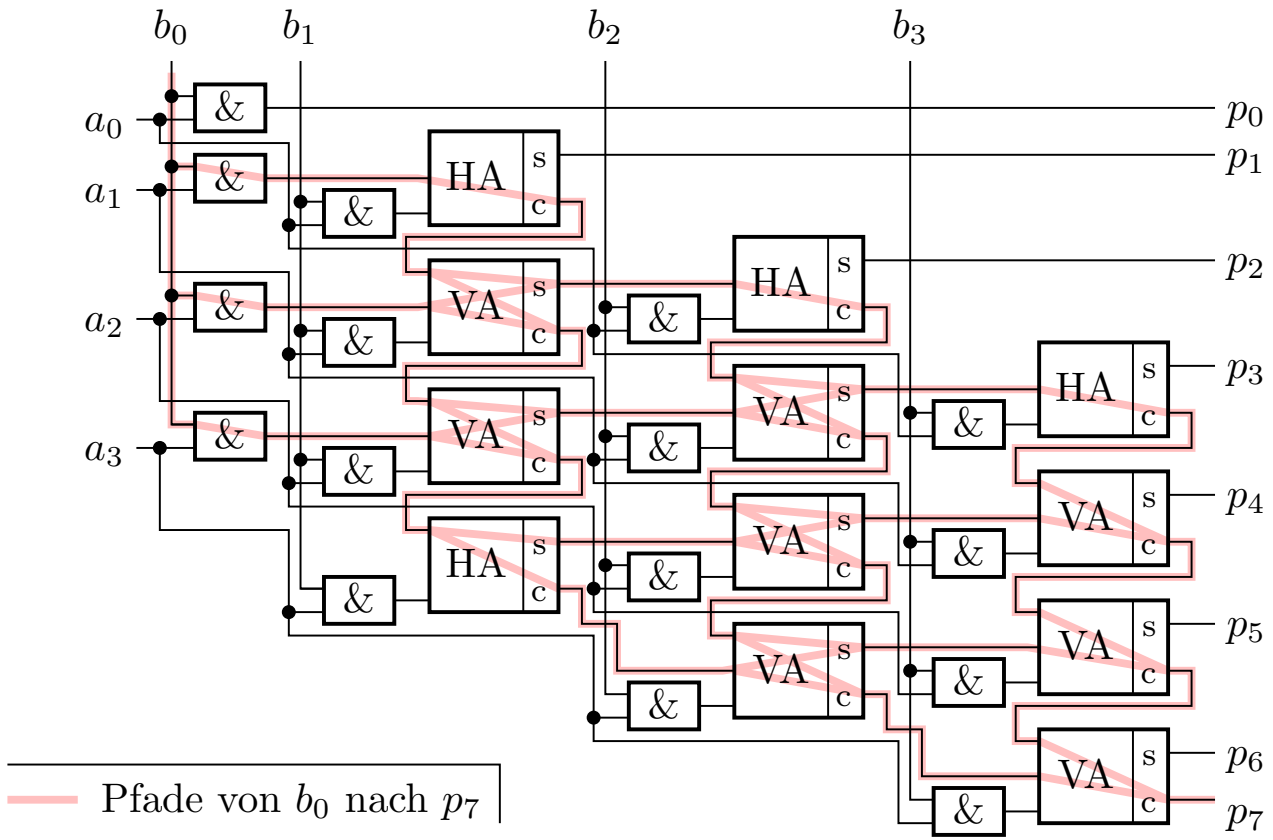
- Modellfehler: Für alle Pfade durch die Schaltung
  - slow-to-rise-Fehler (erhöhte 01-Verzögerung),
  - slow-to-fall-Fehler (erhöhte 10-Verzögerung).
- Robuster Test: Je Testschritt max. eine Signaländerung an den Eingängen jedes Gatters.

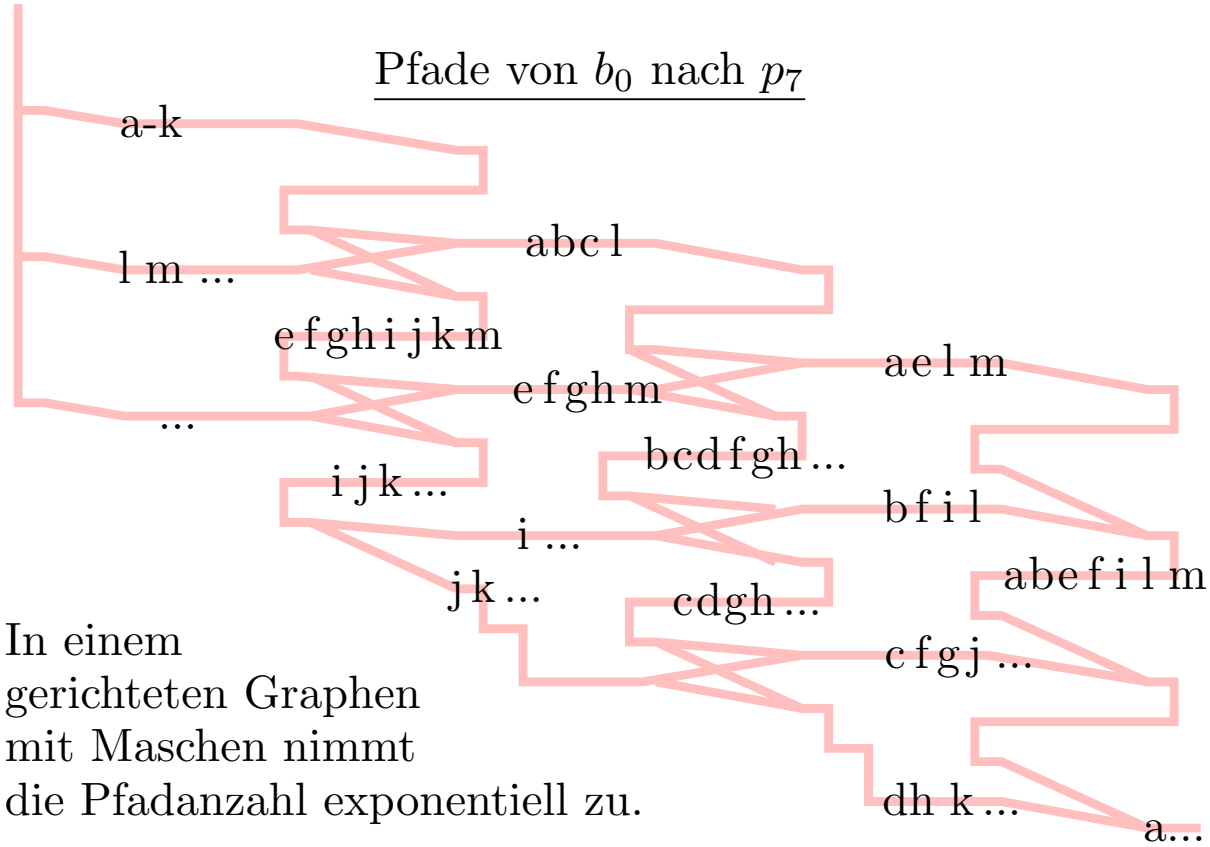


Exponent. Wachstum Modellfehleranzahl (Bsp. Multiplizierer)

$$(a_3 \cdot 2^3 + a_2 \cdot 2^2 + a_1 \cdot 2^1 + a_0 \cdot 2^0) \cdot (b_3 \cdot 2^3 + b_2 \cdot 2^2 + b_1 \cdot 2^1 + b_0 \cdot 2^0) =$$

$$\begin{array}{r} a_3b_0 \cdot 2^3 + a_2b_0 \cdot 2^2 + a_1b_0 \cdot 2^1 + a_0b_0 \cdot 2^0 \\ a_3b_1 \cdot 2^4 + a_2b_1 \cdot 2^3 + a_1b_1 \cdot 2^2 + a_0b_1 \cdot 2^1 \\ a_3b_2 \cdot 2^5 + a_2b_2 \cdot 2^4 + a_1b_2 \cdot 2^3 + a_0b_2 \cdot 2^2 \\ a_3b_3 \cdot 2^6 + a_2b_3 \cdot 2^5 + a_1b_3 \cdot 2^4 + a_0b_3 \cdot 2^3 \\ \hline p_7 \quad p_6 \quad p_5 \quad p_4 \quad p_3 \quad p_2 \quad p_1 \quad p_0 \end{array}$$

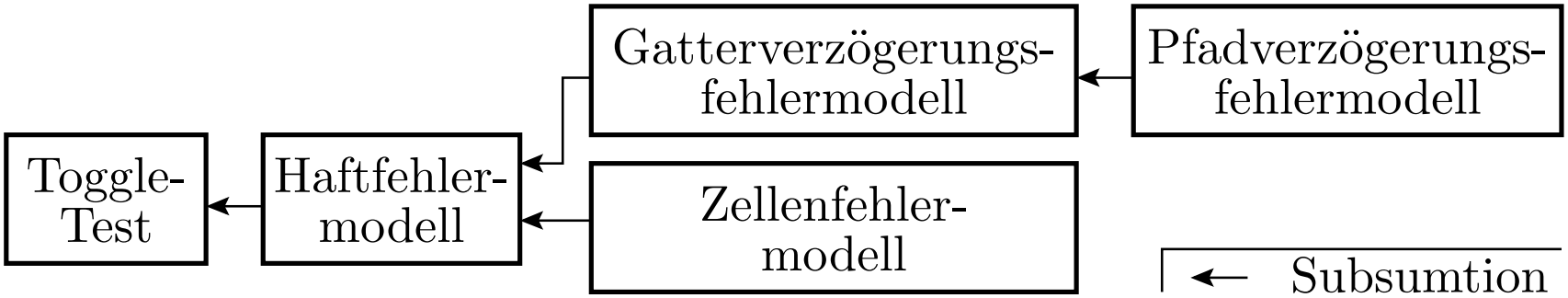




- Die Anzahl der Pfadverzögerungsfehler wächst exponentiell mit der Schaltungsgröße. Nicht praxistauglich.
- Bei mehreren Pfaden je Gatter reicht Kontrolle des Pfades mit der geringsten Laufzeitreserve (Gatterverzögerungsmodell mit Zusatzbedingung).

Subsumtionshierarchie von Fehlermodellen

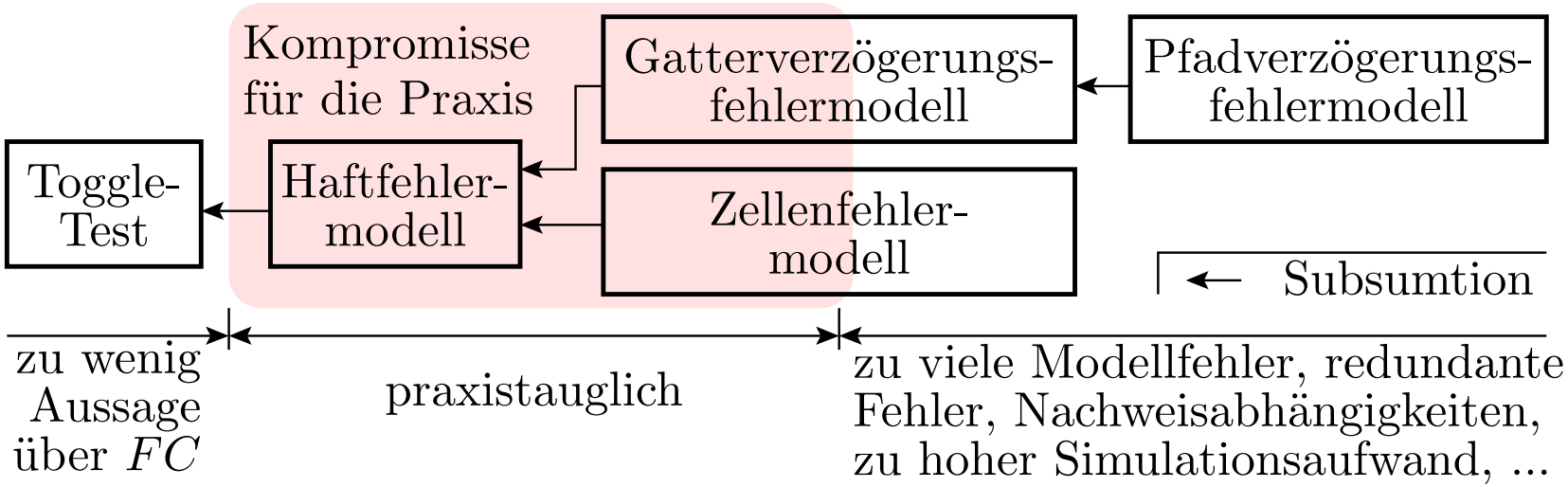
- Subsumption bedeutet in den Beschreibungslogiken, dass ein Konzept (eine eindeutig beschriebene Menge von Objekten) eine Teilmenge eines anderen Konzepts ist.
- Fehlermodell A subsumiert Fehlermodell B, wenn ein Testsatz, der alle Modellfehler von A nachweist, garantiert auch alle Modellfehler von B nachweist.



Bei 100% Modellfehlerüberdeckung haben auch die Modellfehlermengen aller subsumierten Fehlermodelle 100% Überdeckung.

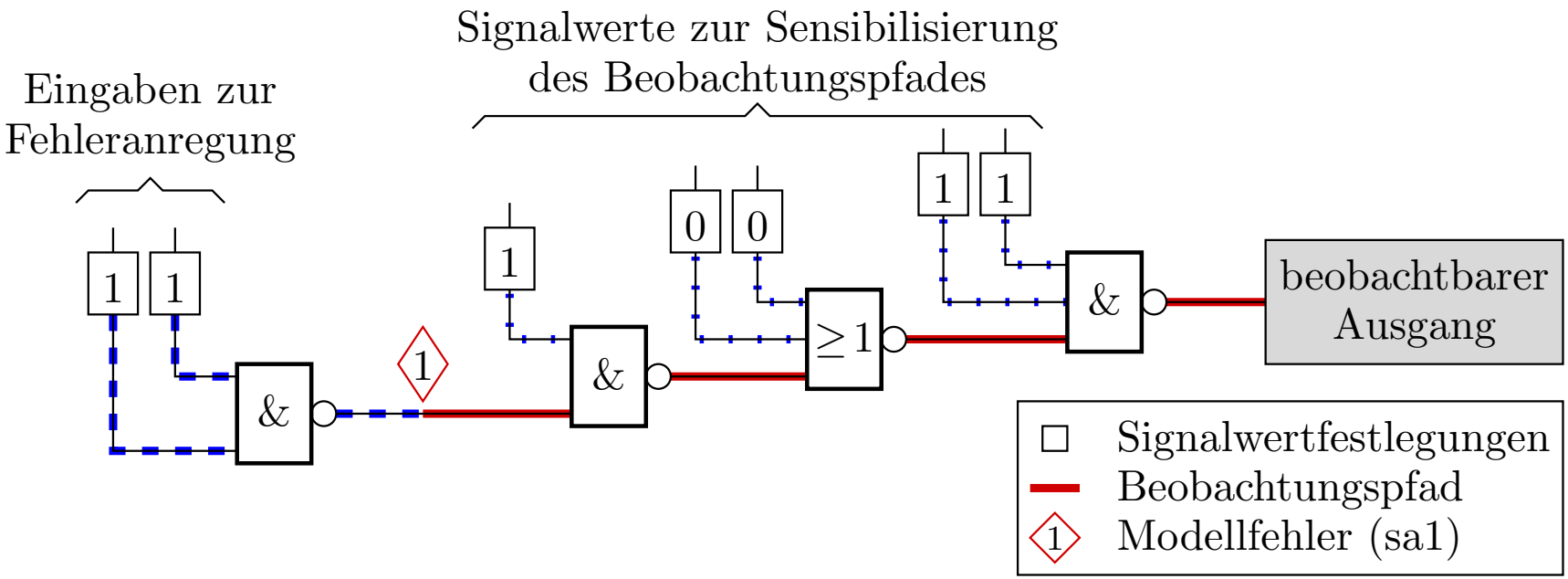
Je höher ein Fehlermodell in der Subsumtionshierarchie steht:

- desto geringer ist tendenziell die Modellfehlerüberdeckung für denselben Testsatz,
- desto größer ist tendenziell die Fehlerüberdeckung bei gleicher Modellfehlerüberdeckung,
- desto größer ist der Aufwand für die Testsuche,
- desto länger muss ein Zufallstestsatz für eine angestrebte Fehlerüberdeckung sein.



Testberechnung (D-Alg.)

Pfadalgorithmen



Vom Fehlerort werden durch Festlegung von Signalwerten

- in Signalflussrichtung Beobachtungspfade sensibilisiert und
- entgegen der Signalflussrichtung Steuerbedingungen eingestellt.

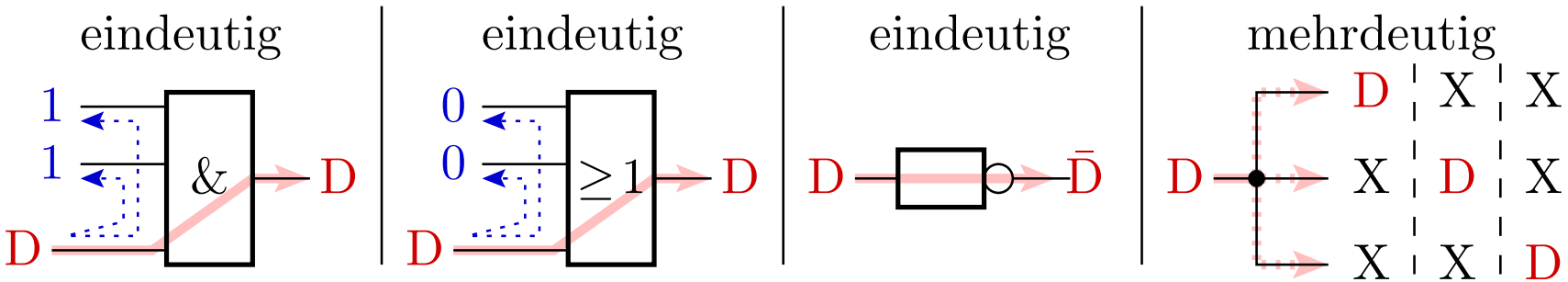
Geeignet für Haftfehler (Pfadverzögerungsfehler, ...)

D-Algorithmus

Erweiterung der Logikwerte um 3 Pseudo-Werte[10]:

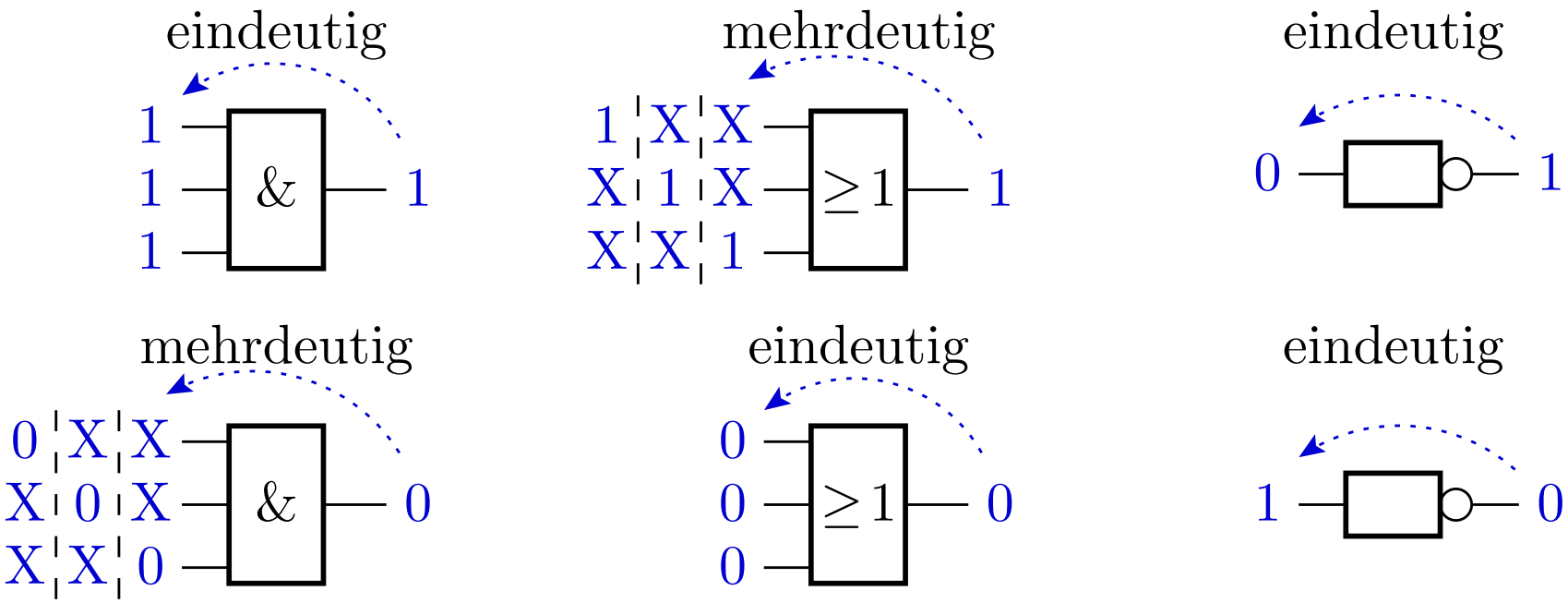
- D 0 wenn unverfälscht, 1 wenn verfälscht.
- $\bar{D}$ 1 wenn unverfälscht, 0 wenn verfälscht.
- X Signalwert ist ungültig oder für den Fehlernachweis ohne Bedeutung.

Regeln für die Sensibilisierung eines Beobachtungspfades:

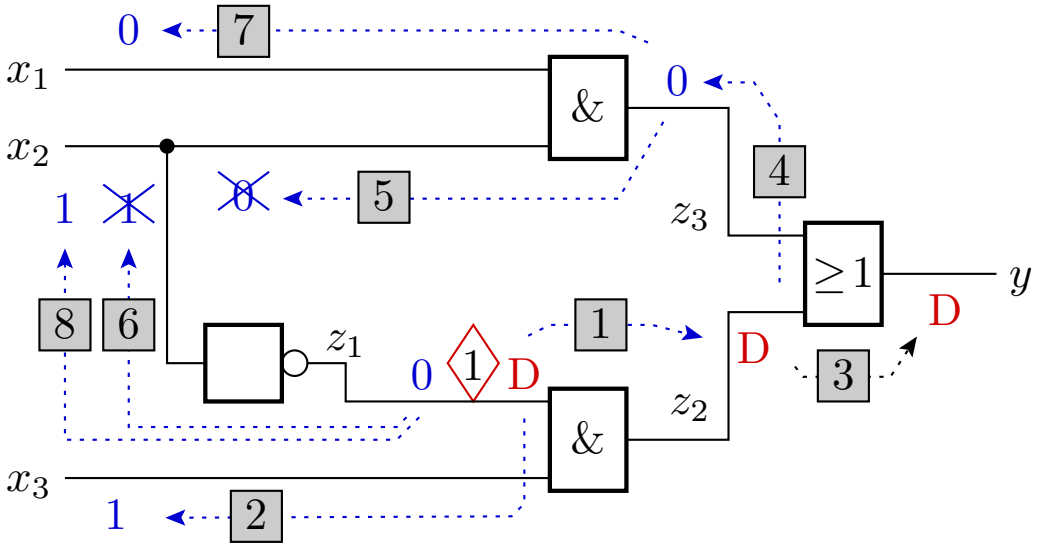


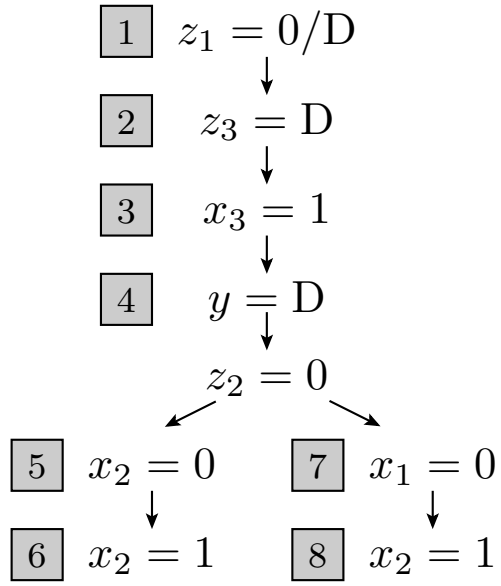
[10] W. Daehn: Testverfahren in der Mikroelektronik: Methoden und Werkzeuge. Springer 1997.

Einstellung von Steuerwerten



- Jede kombinatorische Schaltung kann durch eine Schaltung aus AND, OR, NOT nachgebildet werden.
- Später Verallgemeinerung auf LUT (**L**ook-**U**p **T**able, Tabellenfunktionen).





(Widerspruch)

	x_3	x_2	x_1	z_3	z_2	z_1	y
0	X	X	X	X	X	0D	X
1	1	X	X	X	X	0D	X
2	1	X	X	X	D	0D	X
3	1	X	X	X	D	0D	D
4	1	X	X	0	D	0D	D
5	1	0	X	0	D	0D	D
6	1	~~01~~	X	0	D	0D	D
7	1	X	0	0	D	0D	D
8	1	1	0	0	D	0D	D

Baumsuche:

- Bei der Wertefestlegung können Widerspüche auftreten.
- Zurück zur letzten mehrdeutigen Entscheidung.
- Keine Lösung nach Durchmusterung des gesamten Baums. $\Rightarrow$ Fehler nicht nachweisbar

Erfolgsrate der Testberechnung:

- Anteil der Fehler, für die ein Test gefunden oder für die der Beweis »nicht nachweisbar« erbracht wird.

- Die Testsuche für einen Fehler kann hunderte von Wertefestlegungen beinhalten.
- Der Suchraum wächst exponentiell mit der Anzahl der mehrdeutigen Festlegungen. Suchräume der Größen $> 2^{30...40}$ nicht mehr vollständig durchsuchbar.
- Abbruch der Suche nach einer bestimmten Rechenzeit.

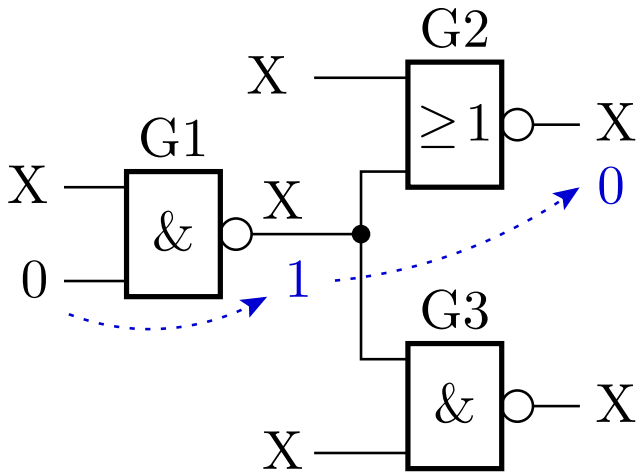
Heuristiken:

- Frühe Erkennung von Widersprüchen,
- Suchraumbegrenzung und
- gute Suchraumstrukturierung.

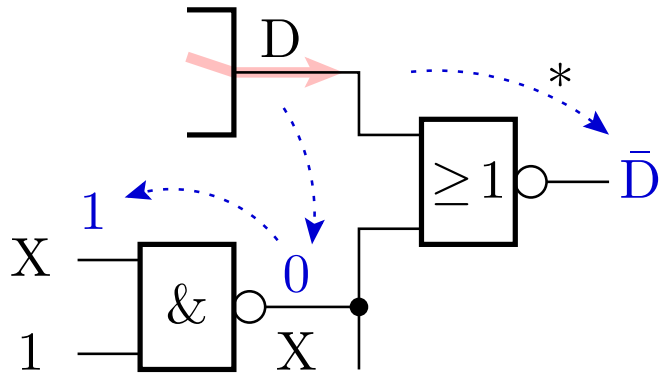
Implikationstest (Widerspruchsfrüherkennung)

- Aus den berechneten Wertefestlegungen alle eindeutig folgenden Werte berechnen.

Implikation in Signalflussrichtung



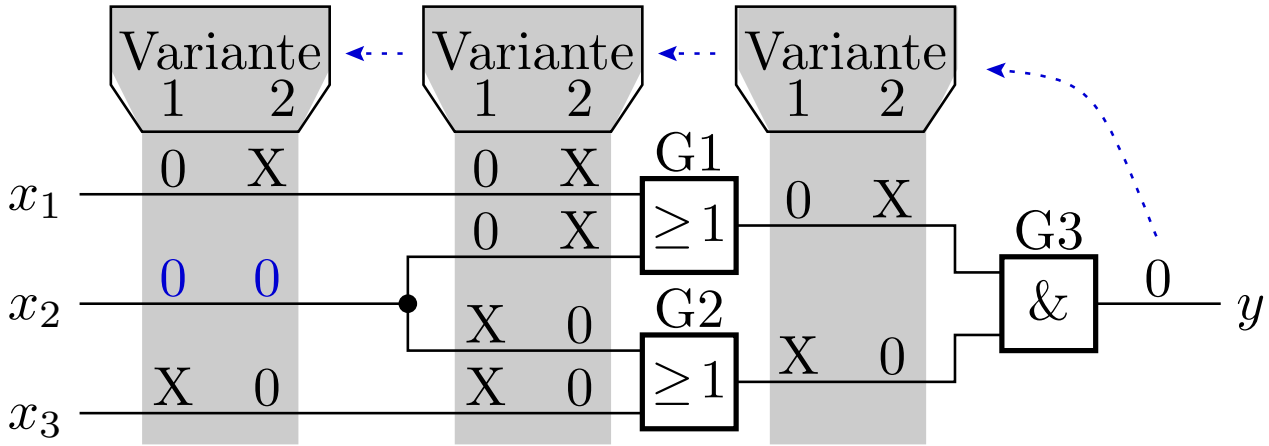
D-Pfad- und Rückwärtsimplikation



* eindeutige Weiterführung

- Mindert die Entscheidungsbaumtiefe.

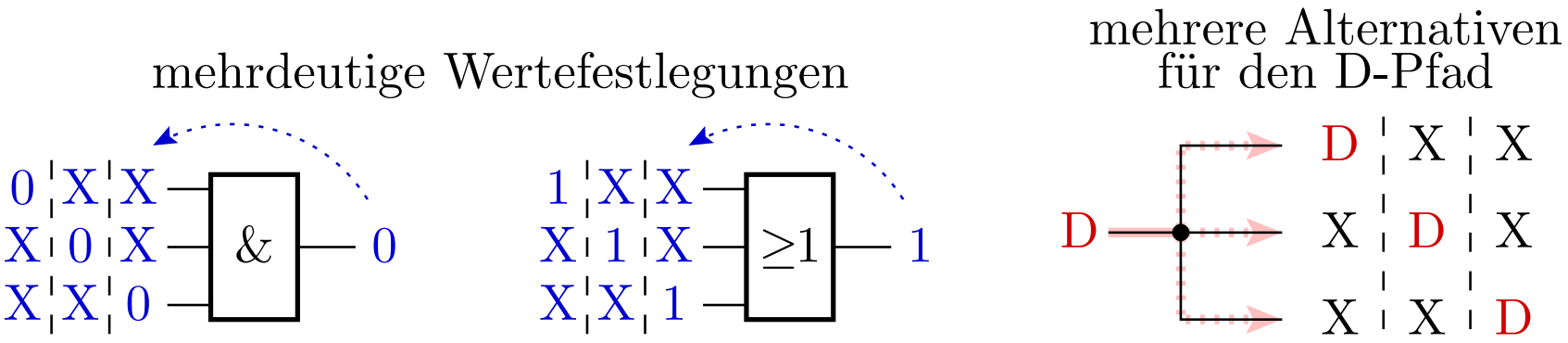
- Rückwärtsimplikation über mehrere Gatterebenen:



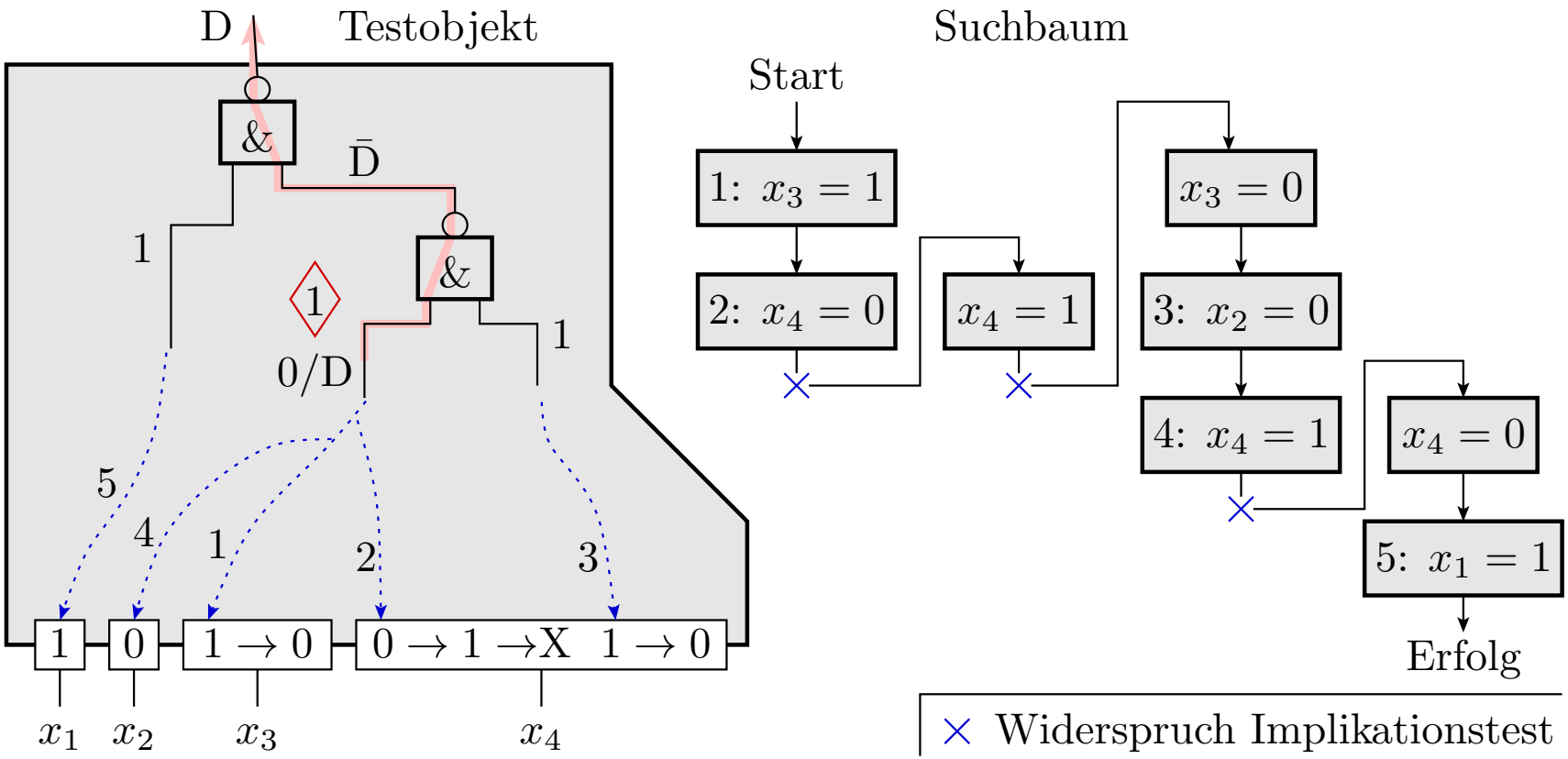
- Für $y = 0$ gibt es zwei Einstellmöglichkeiten.
- Für beide Möglichkeiten muss $x_2 = 0$ sein.
- Das Erkennen von Implikationen dieser Art mindert die Backtracking-Häufigkeit um bis zu 80 %.

Suchraumbegrenzung

- Der D-Algorithmus baut den Suchbaum über alle mehrdeutigen Wertefestlegungen auf:

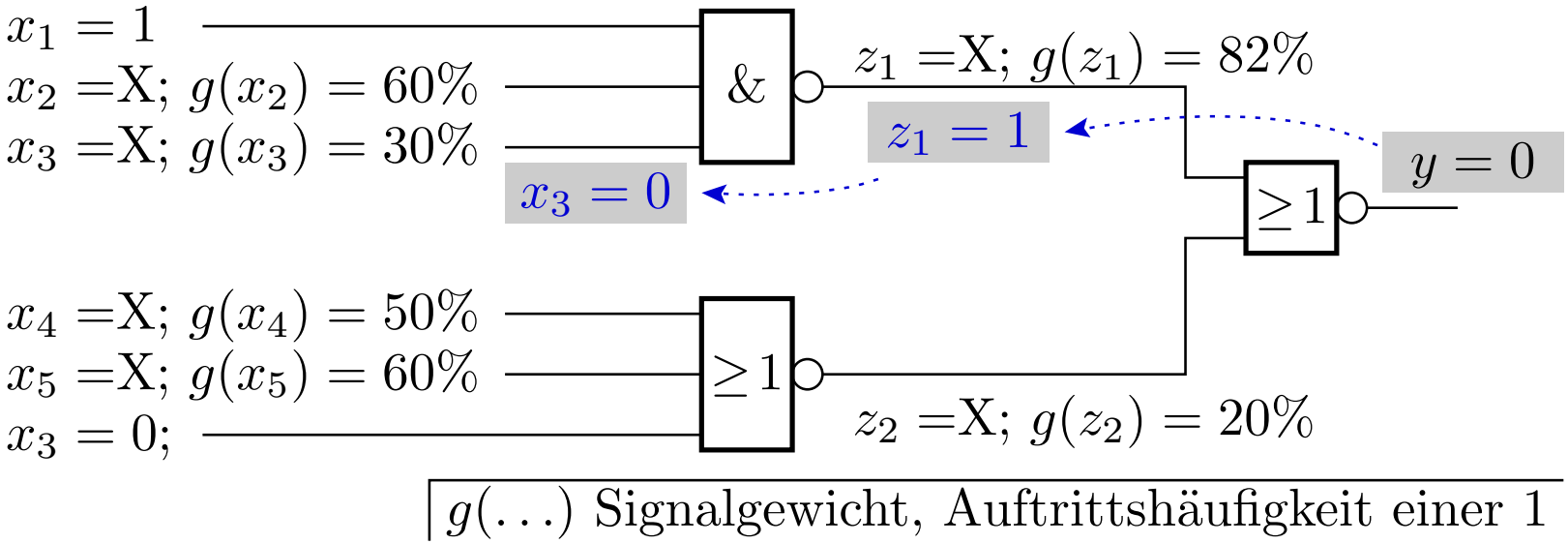


- Nur die Schaltungseingänge können unabhängig voneinander alle Wertevariationen annehmen.
- Es genügt, den Suchbaum mit den Eingabewertefestlegungen aufzubauen.
- Begrenzt Suchraum auf 2^{N_E} (2^{N_E} – Eingangsanzahl). Verringert Rechenaufwand um Zehnerpotenzen.



- Lange Steuerpfade vom Fehlerort und vom D-Pfad zu Eingängen.
- Aufbau des Suchbaums über Eingangssignale.
- Wenn Implikationstest-Widerspruch, letzte Eingabefestlegung invertieren.

Geschätzte Erfolgswahrscheinlichkeiten



- Schätzen der Signalwichtungen[11] über eine kurze Simulation mit Zufallswerten oder analytisch.
- Wahl der Steuerwerte / Beobachtungspfade, die mit größerer Wahrscheinlichkeit aktivierbar / sensibilisierbar sind.

[11] Die Wichtung eines Signals ist die Auftrittshäufigkeit einer »1«.

Komplexe Funktionsbausteine

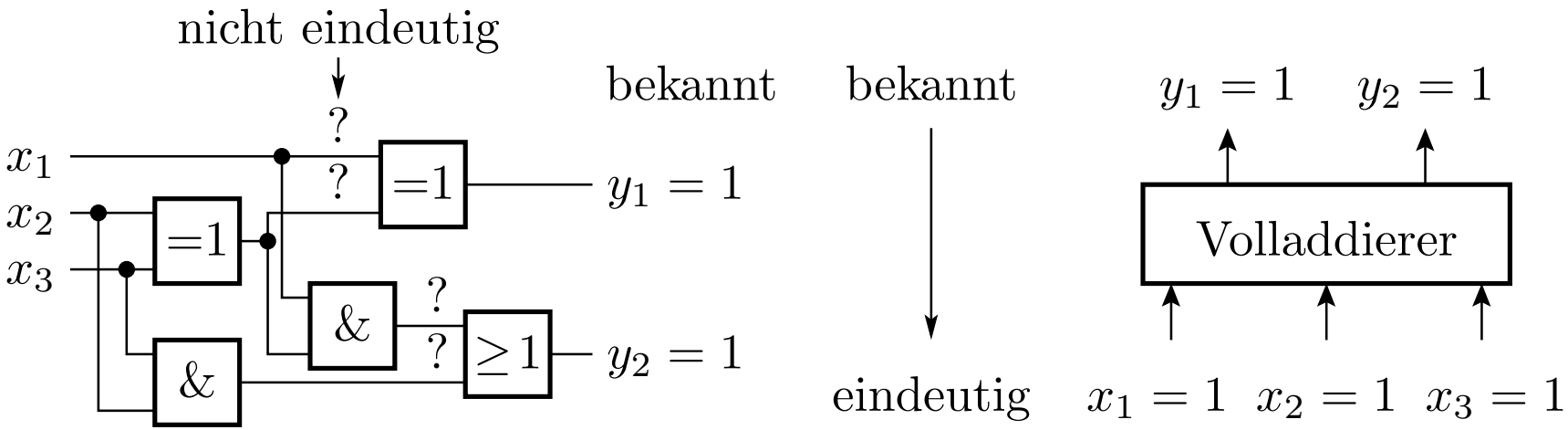
- Beschreibung durch Tabellenfunktion (Bsp. Volladdierer):

x_2	x_1	x_0	s	c
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

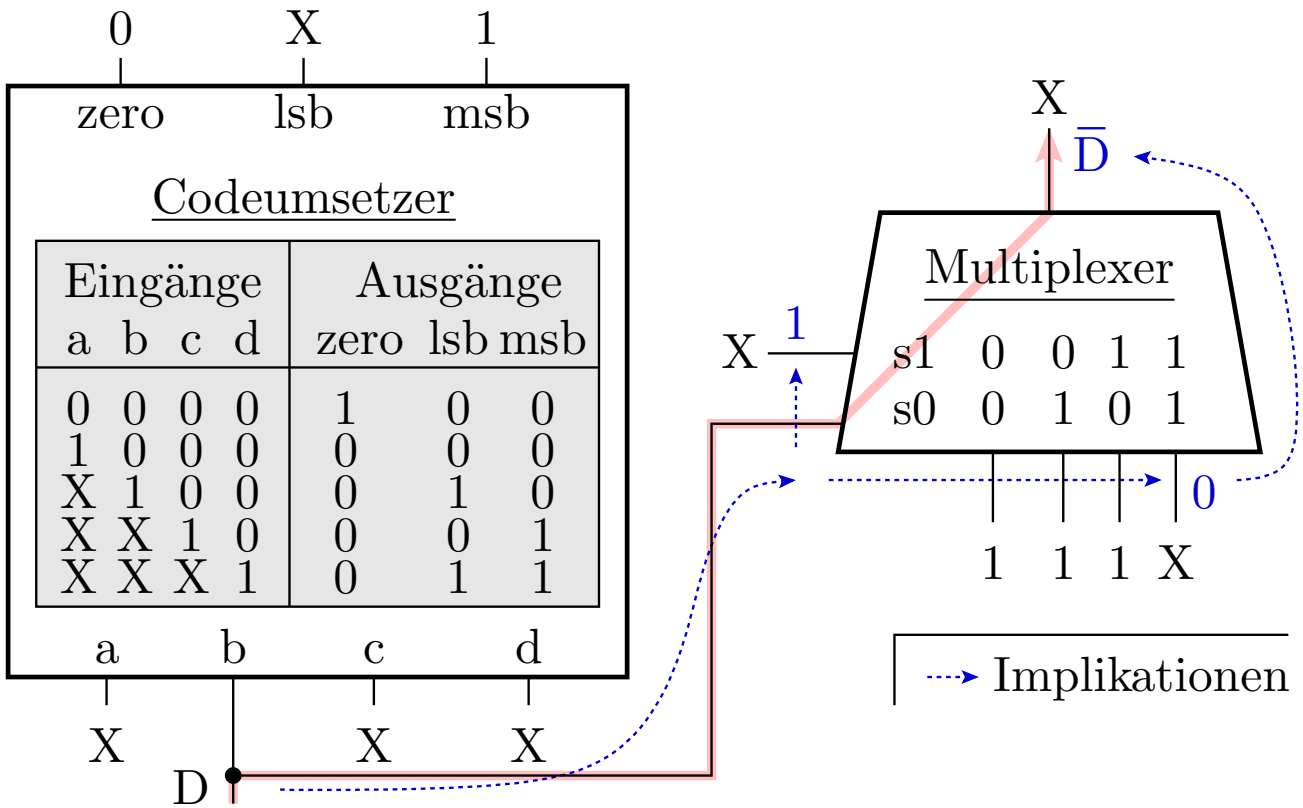
gegeben		Lösungsmenge
XXX00	$\Rightarrow$	00000
01DXX	$\Rightarrow$	01D$\bar{\text{D}}$D
1XXXD	$\Rightarrow$	10D$\bar{\text{D}}$D, 1D0$\bar{\text{D}}$D
11XX1	$\Rightarrow$	11111, 111001

- Vervollständigung des Vektors der gegebenen Anschlusswerte durch Vergleich mit allen Tabellenzeilen:
  - »1« und »0« passen nur auf »1« und »0«.
  - »X« passt immer.
  - »D« muss für »D=0« und für »D=1« passen.

Implikationstest an einem Volladdierer



- An der Gatterbeschreibung eines Volladdierers ist die Implikation $y_1 = y_2 = 1 \Rightarrow x_1 = x_2 = x_3 = 1$ nicht zu erkennen. Lösungsfindung über Baumsuche.
- Bei Zusammenfassung zu einer Tabellenfunktion wird die Lösung bereits bei der Anschlusswertevervollständigung erkannt.



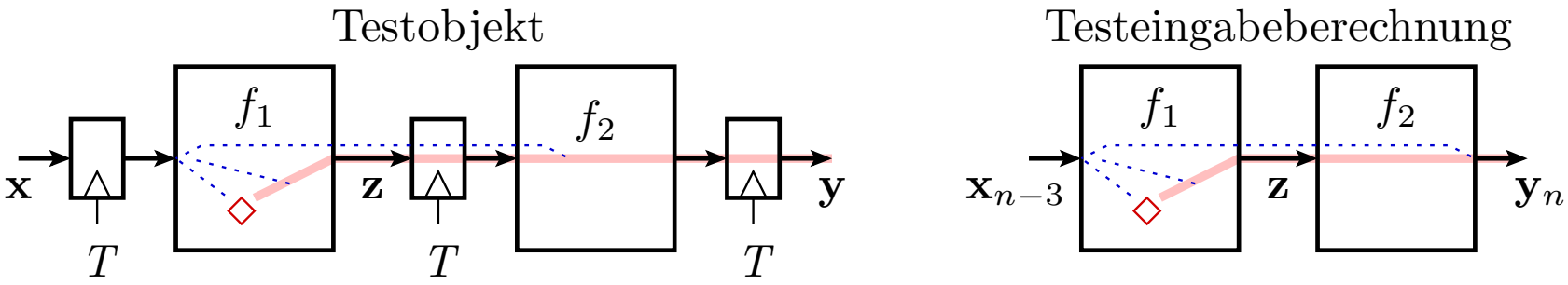
- »lsb« hängt bei »zero=0« und »msb=1« nicht von »b« ab. Eindeutiger D-Pfad über Multiplexer.
- Tabelleneingabewerte »X« (Eingang beeinflusst nicht die Ausgabe) führt zu Tabellen mit $\ll 2^{N_\mathrm{E}}$ Tabellenzeilen (N_E – Anzahl der Eingänge).

Sequentielle Schaltungen

Sequenzielle Schaltungen

Zurückführung auf kombinatorische Schaltungen:

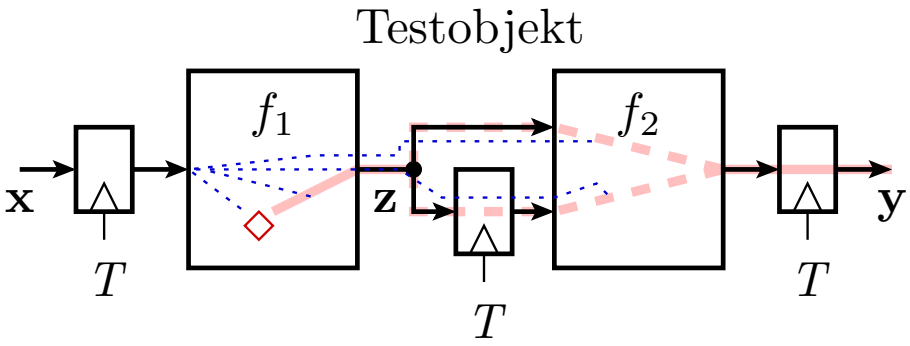
- nur Abtastung

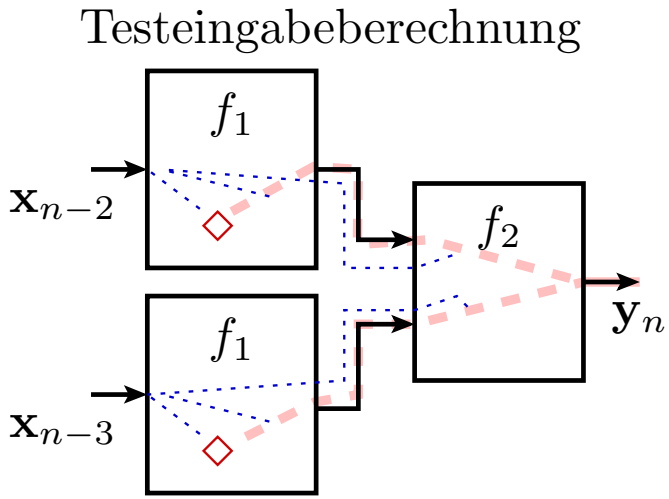


- Testberechnung wie für eine kombinatorische Schaltung.
- Zeitversatz zwischen Ein- und Ausgabe berücksichtigen.

```
x <= Eingabe_1; wait on RISING_EDGE(T);
x <= Eingabe_2; wait on RISING_EDGE(T);
x <= Eingabe_3; wait on RISING_EDGE(T);
x <= Eingabe_4; wait for tP; assert y=Ausgabe_1 ...;
x <= Eingabe_5; wait for tP; assert y=Ausgabe_2 ...;
```

Verarbeitung in mehreren Zeitebenen



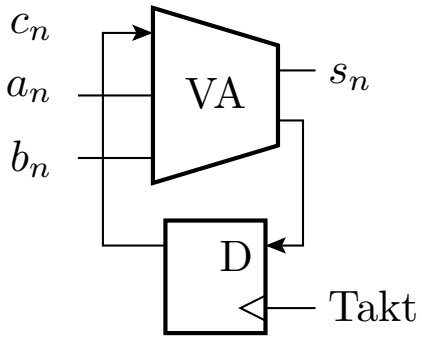


- Die kombinatorische Ersatzschaltung enthält mehrere Kopien derselben Schaltung.
- Die Haftfehler sind in jeder Kopie.
- Eingaben mehrerer Zeitebenen / Mehr-Pattern-Test:

```
x <= Eingabe_1A; wait on RISING_EDGE(T);
x <= Eingabe_1B; wait on RISING_EDGE(T);
... wait on RISING_EDGE(T); assert y = Ausgabe_1 ...;
```

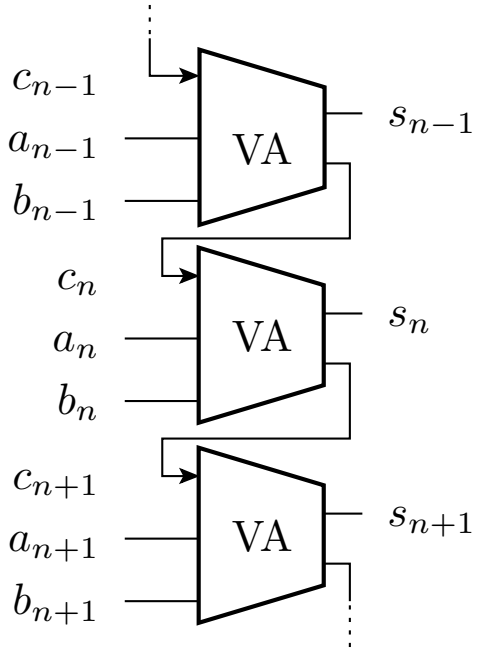
Schaltungen mit Rückführung

serieller Addierer



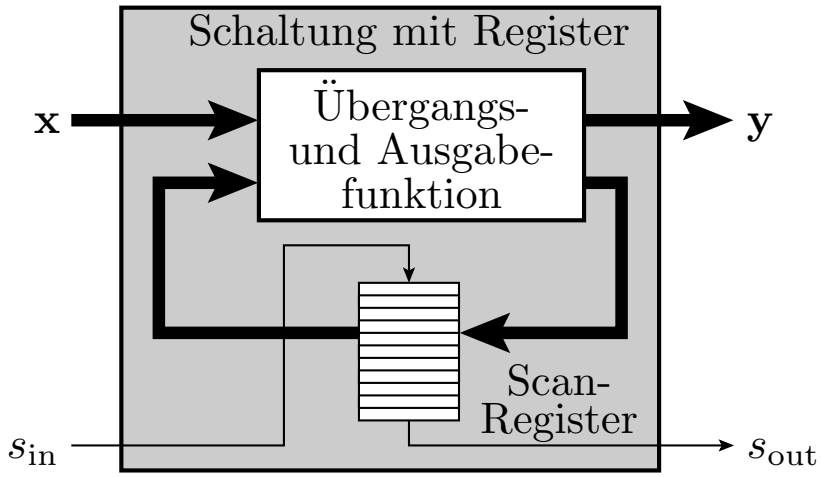
VA - Volladdierer

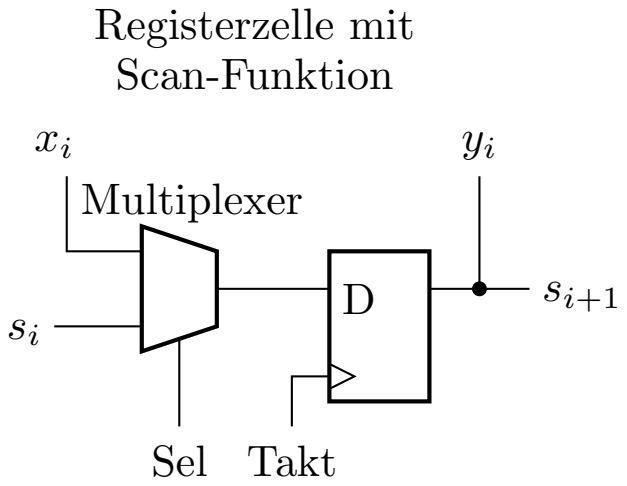
aufgerollter Addierer



- Ersatzschaltung aus unbegrenzt vielen Kopien.
- Regeln zur Begrenzung der Länge der Steuer- und Beobachtungspfade erforderlich.

Auftrennung der Rückführungen für den Test





- Erweiterung der Zustandsregister um eine Schiebefunktion. (Mindestaufwand ein Multiplexer je Speicherzelle.)
- Serielles Auslesen und Neuladen der Zustandsregister zwischen aufeinanderfolgenden Testschritten.

Speichertest

Speichertest

Schreib-Lese-Speicher (RAM) bestehen aus Speichermatrix, Adressdecoder, Eingabelogik und Ausgabelogik. Fehler im Decoder, der Eingabe- und der Ausgabelogik werden als Fehler in der Speichermatrix modelliert. Modellfehler für Speicher:

- Haftfehler (Lesewert ist ständig null oder ständig eins).
- Übergangsfehler (Wert nur in einer Richtung änderbar).
- Stuck-open-Fehler (Ausgabe des zuletzt gelesenen Wertes).
- Zerstörendes Lesen (Löschen des Inhalts beim Lesen).
- Gegenseitige Beeinflussung unterschiedlicher Zellen, ...

Aufsetzend auf den Fehlerannahmen gibt es algorithmisch beschriebene Testabläufe mit der Speicherorganisation als Parameter, die alle unterstellten Modellfehler nachweisen.

beteiligte Zellen	Name	Definition	Fälle	Testfolge für den Nachweis
1	Haftfehler	Wert der Speicherzelle ist nicht setzbar	stuck-at-0 stuck-at-1	$W(i)1$, $R(i)1$ $W(i)0$, $R(i)1$
	Übergangsfehler	Wert der Speicherzelle i ist nur in einer Richtung änderbar	kein Übergang $1 \rightarrow 0$ $0 \rightarrow 1$	 $W(i)1$, $R(i)1$, $W(i)0$, $R(i)0$ $W(i)0$, $R(i)0$, $W(i)1$, $R(i)1$
	Stuck-open-Fehler	kein Zugriff auf Speicherzelle i (Ausgabe des Wertes der vorherigen Leseoperation)		$W(i)0$, $R_1(j)$, $R(i)0$, $W(i)1$, $R_0(j)$, $R(i)1$
	zerstörendes Lesen	Inhalt von Speicherzelle i wird beim Lesen verändert	$R(i) \Rightarrow C(i) = \overline{C(i)}$	$W(i)0$, $R(i)0$, $R(i)0$ $W(i)1$, $R(i)1$, $R(i)1$
2	Kopplung Typ 1	Veränderung des Inhalts von Zelle i bestimmt Zustand in Zelle j	$W(i)0 \Rightarrow C(j) = 0$ $W(i)0 \Rightarrow C(j) = 1$ $W(i)1 \Rightarrow C(j) = 0$ $W(i)1 \Rightarrow C(j) = 1$	$W(j)0$, $W(i)0$, $R(j)0$, $W(i)1$, $R(j)0$ $W(j)1$, $W(i)0$, $R(j)1$, $W(i)1$, $R(j)1$
	Kopplung Typ 2	Veränderung des Inhalts von Zelle i bewirkt eine Änderung in Zelle j	$C(i) = \overline{C(i)} \Rightarrow C(j) = \overline{C(j)}$	$W(j)0$, $W(i)0$, $R(j)0$, $W(i)1$, $R(j)0$, $W(i)0$, $R(j)0$ $W(j)1$, $W(i)0$, $R(j)1$, $W(i)1$, $R(j)1$, $W(i)0$, $R(j)1$

$W(i)0$ Schreibe in Zelle i eine 0
$W(i)1$ Schreibe in Zelle i eine 1
$R(j)$ Lese eine beliebige andere Zelle
$C(\ldots)$ Inhalt Zelle ...

$R(i)0$ Lese Inhalt Zelle i und vergleiche mit Sollwert 0
$R(i)1$ Lese Inhalt Zelle i und vergleiche mit Sollwert 1
$R_0(j)$ Lese eine andere Zelle, in der 0 steht
$R_1(j)$ Lese eine andere Zelle, in der 1 steht

Beispielalgorithmus Marching Test

Adresse i	Initialisierung	March 1	March 2	March 3
0	$W(i)0$	$R(i)0, W(i)1$	$R(i)1, W(i)0$	$R(i)0, W(i)1$
1	$W(i)0$	$R(i)0, W(i)1$	$R(i)1, W(i)0$	$R(i)0, W(i)1$
2	$W(i)0$	$R(i)0, W(i)1$	$R(i)1, W(i)0$	$R(i)0, W(i)1$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$N-1$	$W(i)0$	$R(i)0, W(i)1$	$R(i)1, W(i)0$	$R(i)0, W(i)1$

	March 4		March 1a		March 2a
0	$R(i)1, W(i)0$	Wartezeit	$R(i)0, W(i)1$	Wartezeit	$R(i)1$
1	$R(i)1, W(i)0$		$R(i)0, W(i)1$		$R(i)1$
2	$R(i)1, W(i)0$		$R(i)0, W(i)1$		$R(i)1$
$\vdots$	$\vdots$		$\vdots$		$\vdots$
$N-1$	$R(i)1, W(i)0$		$R(i)0, W(i)1$		$R(i)1$

Mehrfaches Durchwandern des Speichers in unterschiedlicher Reihenfolge mit der Operationsfolge Zelle Lesen, Wert kontrollieren und inversen Wert zurückschreiben.

Adresse i	Initialisierung	March 1	March 2	March 3
0	$W(i)0$	$R(i)0, W(i)1$	$R(i)1, W(i)0$	$R(i)0, W(i)1$
1	$W(i)0$	$R(i)0, W(i)1$	$R(i)1, W(i)0$	$R(i)0, W(i)1$
2	$W(i)0$	$R(i)0, W(i)1$	$R(i)1, W(i)0$	$R(i)0, W(i)1$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$N-1$	$W(i)0$	$R(i)0, W(i)1$	$R(i)1, W(i)0$	$R(i)0, W(i)1$

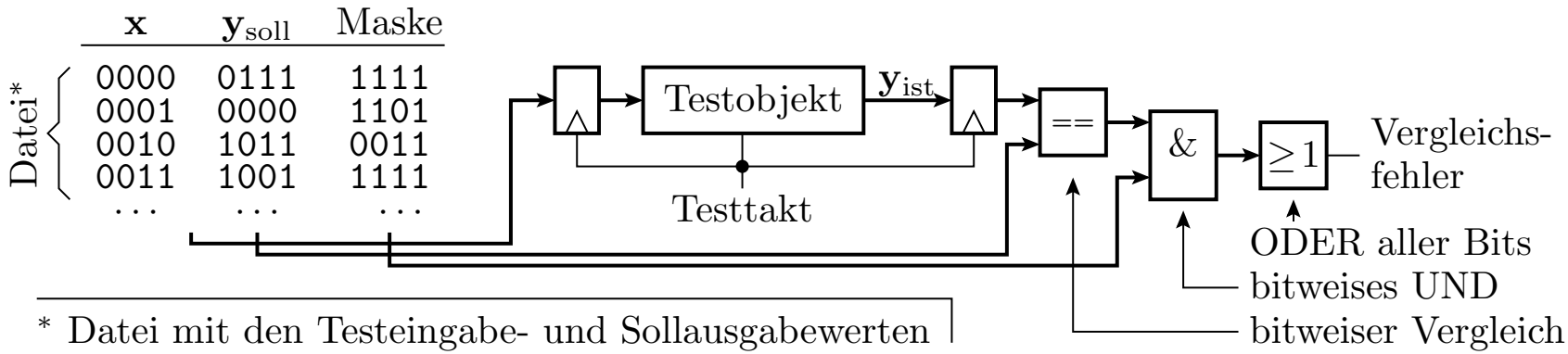
	March 4		March 1a		March 2a
0	$R(i)1, W(i)0$	Wartezeit	$R(i)0, W(i)1$	Wartezeit	$R(i)1$
1	$R(i)1, W(i)0$		$R(i)0, W(i)1$		$R(i)1$
2	$R(i)1, W(i)0$		$R(i)0, W(i)1$		$R(i)1$
$\vdots$	$\vdots$		$\vdots$		$\vdots$
$N-1$	$R(i)1, W(i)0$		$R(i)0, W(i)1$		$R(i)1$

$W(i)0$ Schreibe in Zelle i eine 0 $R(i)0$ Lese Zelle i und vergleiche mit 0
$W(i)1$ Schreibe in Zelle i eine 1 $R(i)1$ Lese Zelle i und vergleiche mit 1

Lässt sich auch ohne großen Zusatzaufwand als Selbsttest implementieren.

Selbsttest

Prinzip eines Digitaltesters



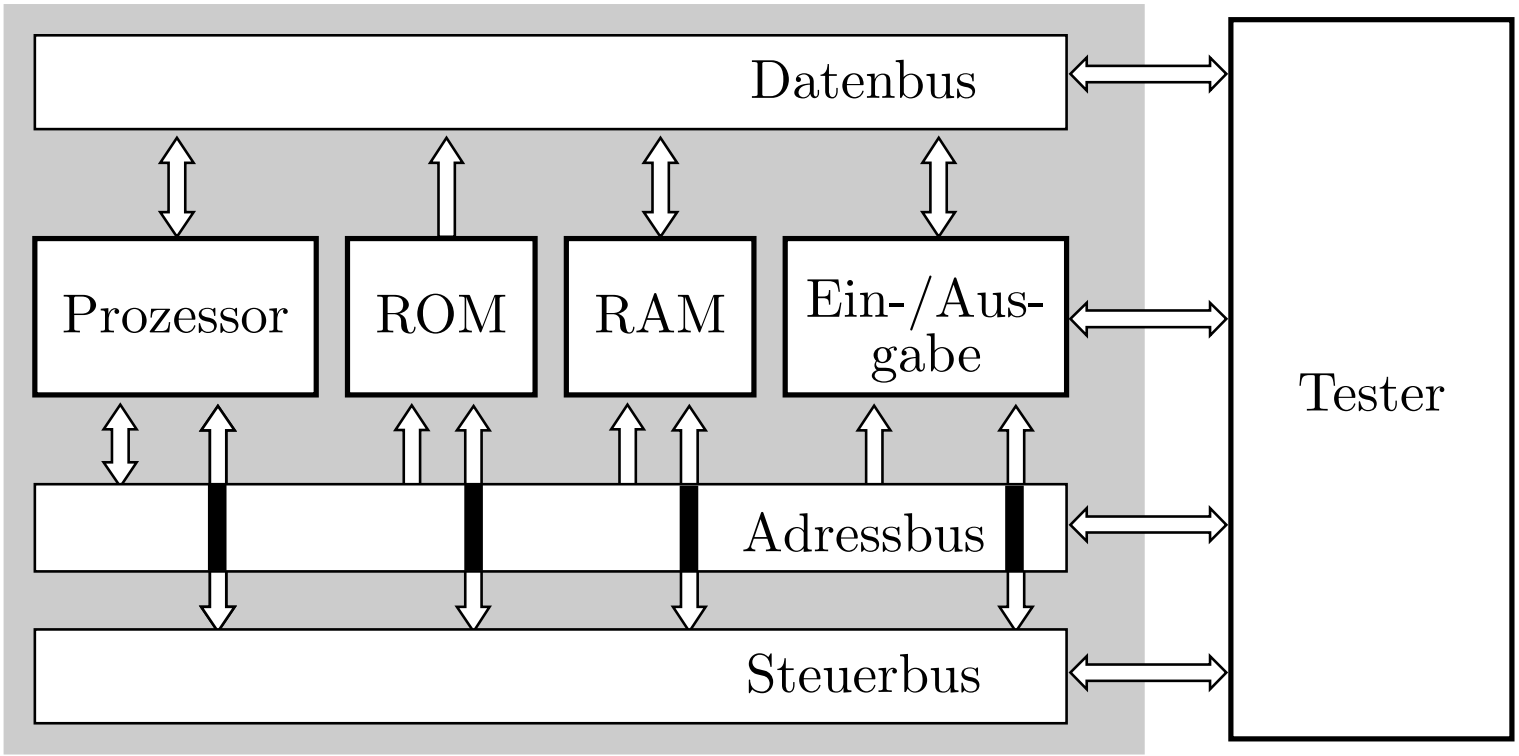
Das Testprogramm beschreibt:

- Eingabe-, Sollausgabe- und Maskenbitvektoren[12] sowie
- den Testtakt zur Festlegung der Zeitpunkte der Eingabesignalwechsel und der Ausgabeabtastung.

Testerfunktionen werden zum Teil in die Schaltkreise integriert.

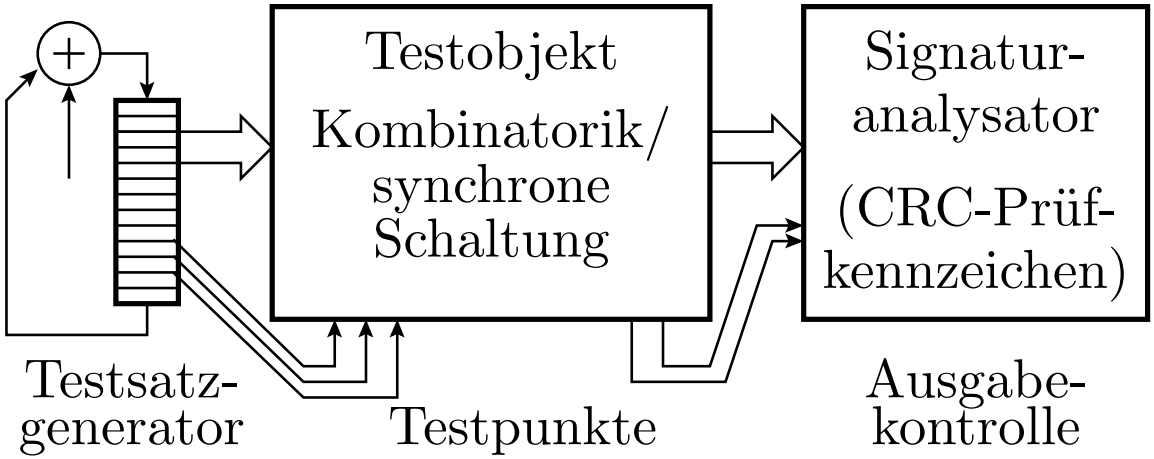
[12] Die Maskenwerte legen die zu kontrollierenden (gültigen) Bitwerte je Testschritt fest.

Auch Schaltkreise werden modular getestet



- Verlangt vom Entwurf Vorkehrungen für den Testerzugriff auf Schnittstellen zwischen Teilsystemen.
- Geeignet sind Busstrukturen, Boundary-Scan, ..., Selbsttest.

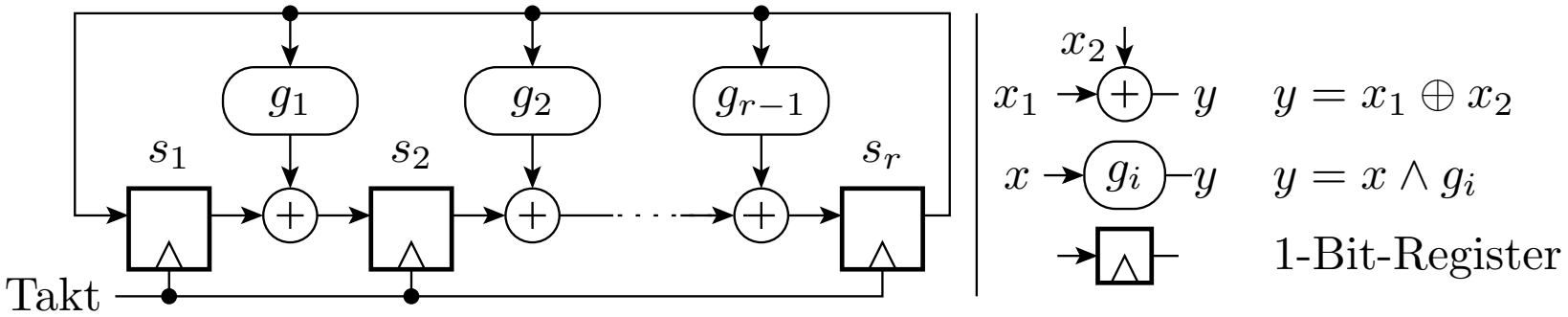
Selbsttests



- Einrahmen der Testobjekte mit Pseudo-Zufallsgeneratoren und Signaturanalysatoren zur Prüfkennzeichenbildung.
- Zugangsmöglichkeit zu internen Schaltungspunkten. Modularisierung.
- Integrierte Test-Hardware schafft die Taktfrequenz des Testobjekts (kurze Verbindungen, gleiche Schaltungstechnik).
- Zeitgleicher Test aller Chips auf einem Waver, ...

Linear rückgekoppelte Schieberegister

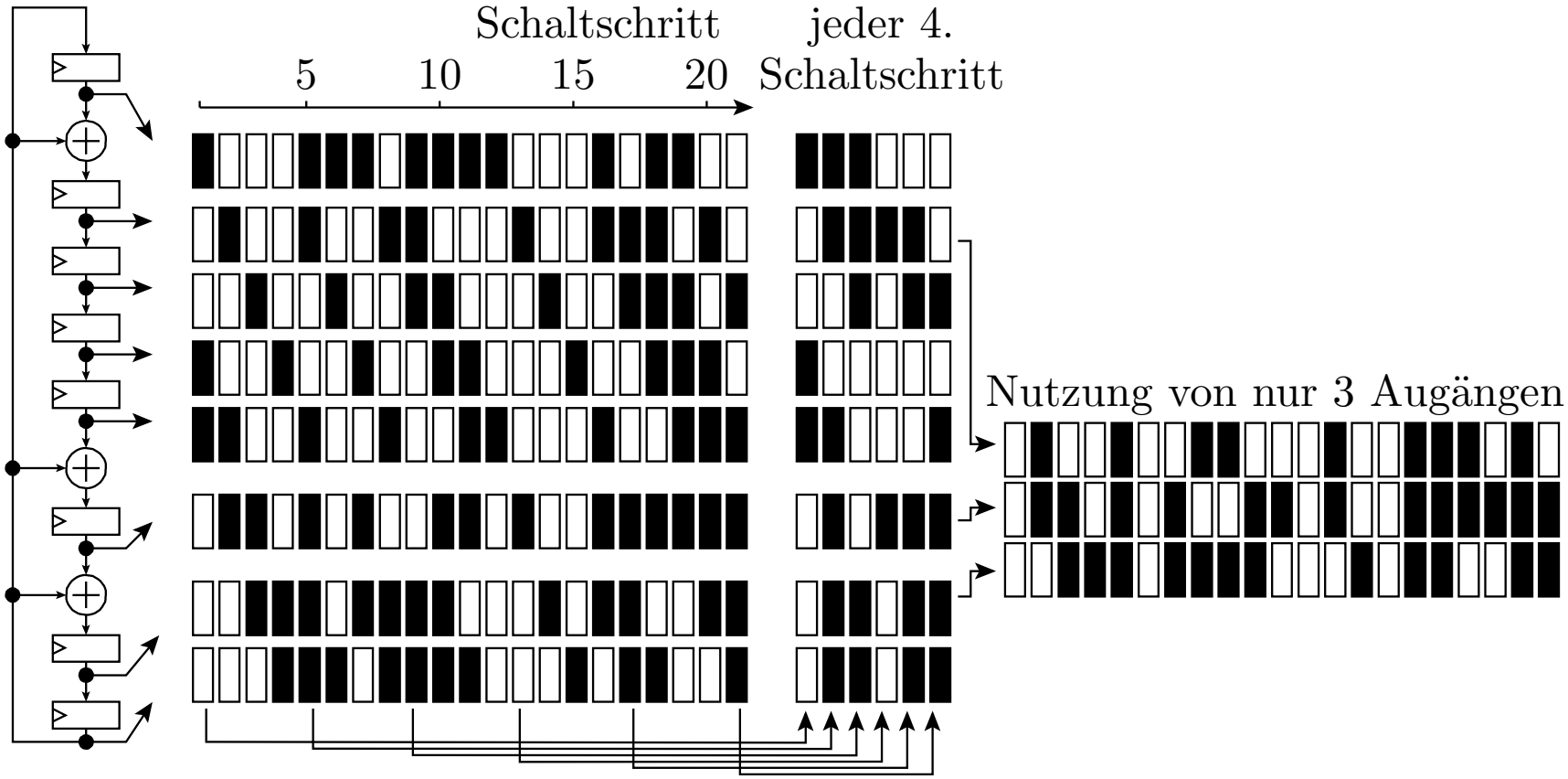
Einfachste und zur Testdatenbereitstellung meist ausreichende Lösung sind linear rückgekoppelte Schieberegister (LFSR **L**inear **F**eeback **S**hift **R**egister) bzw. Algorithmen aus logischen und Verschiebeoperationen ähnlich CRC-Bildung:



Für die Rückführung g_i gehen nur bestimmte Werte, bei denen große Zyklen entstehen. Internet Suchbegriff »Primitive Polynome«. Beispiel für ein 16-Bit LFSR:

$$x^{16} \oplus x^5 \oplus x^3 \oplus x \oplus 1$$

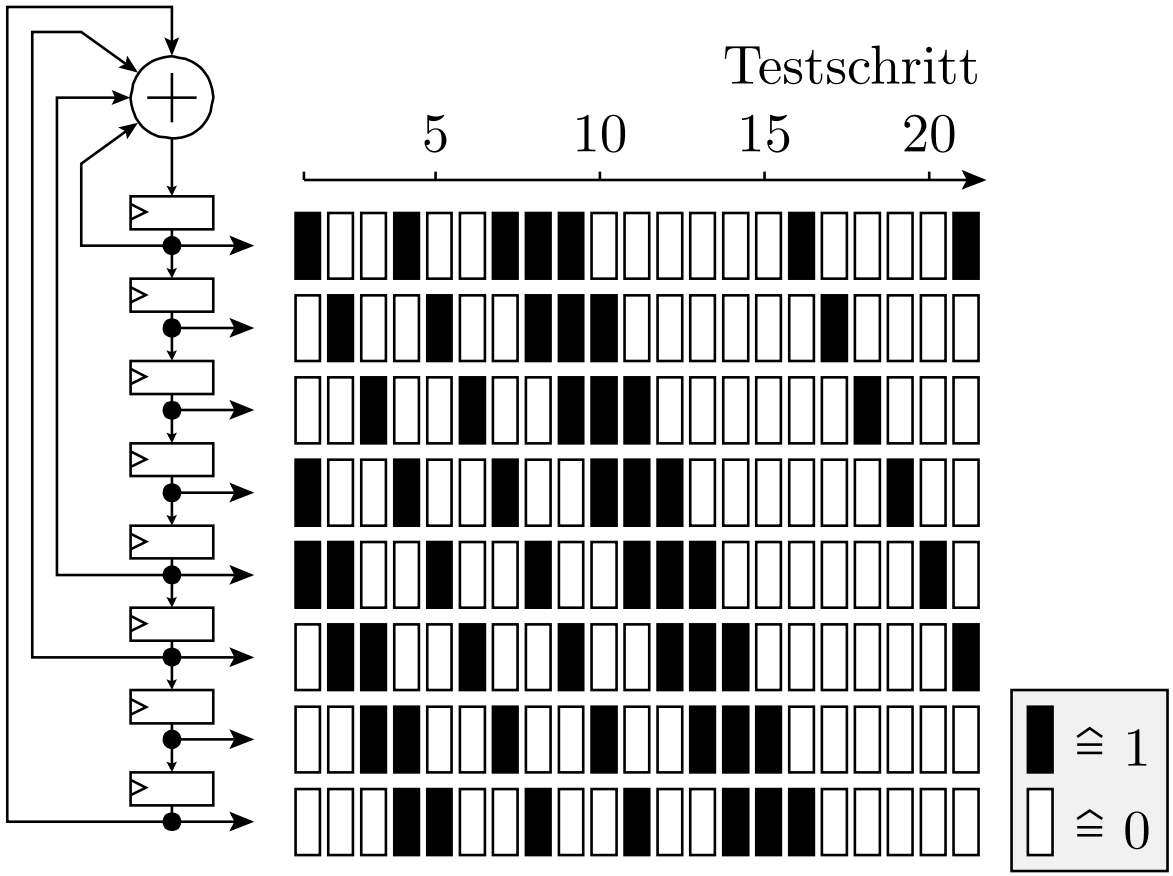
Es bedeutet $g_1 = g_3 = g_5 = 1$, alle anderen null / weglassen.



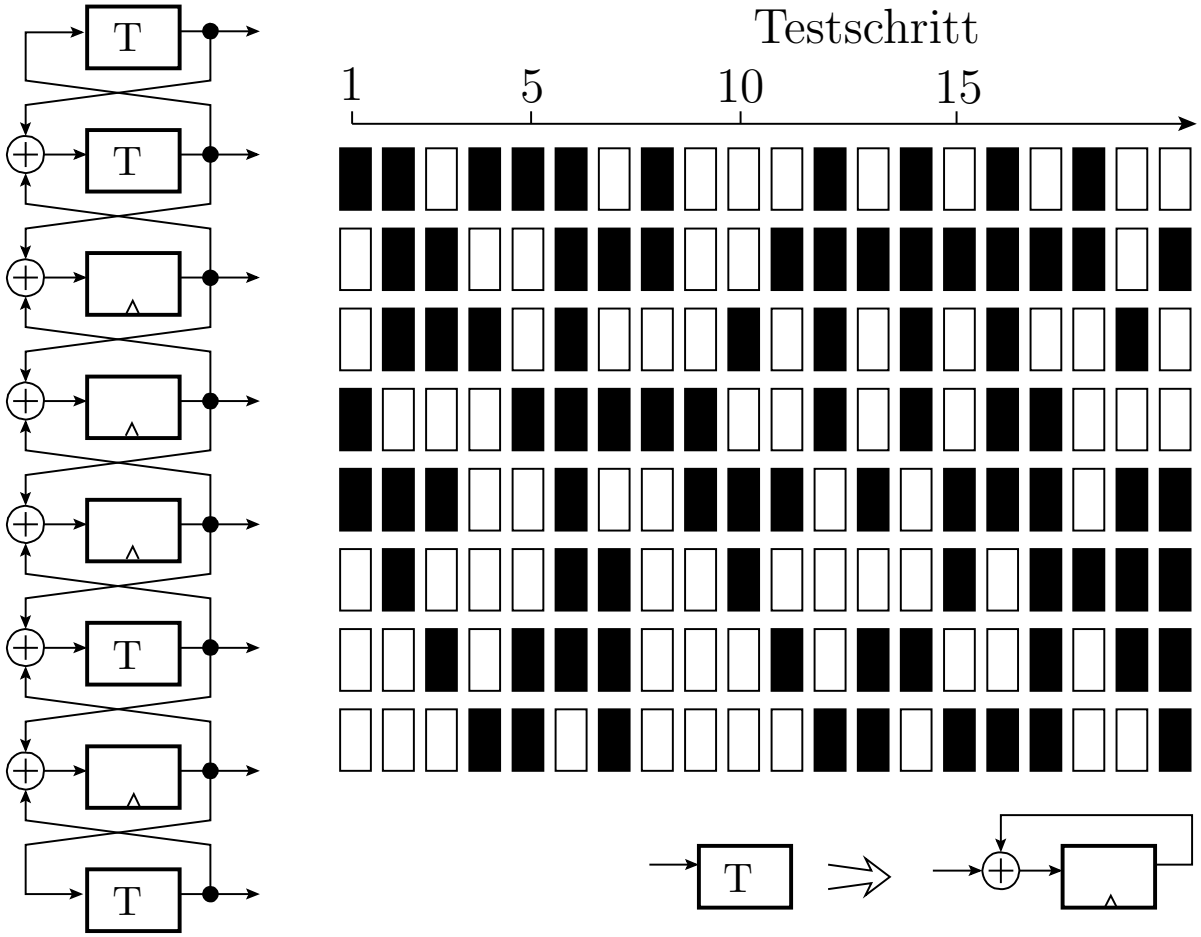
Falls die »Streifenmuster« durch die Schiebeoperationen stören, Generator für jede neue Testeingabe mehrere Schritte weiterschalten oder nur einen Teil der Ausgänge nutzen[13].

[13] G Kemnitz: Test und Verlässlichkeit von Rechnern. Springer 2007.

Statt einer Rückführung des Ausgangs auf mehrere Bitstellen können auch mehrere Bitstellen auf den Eingang rückgeführt werden:



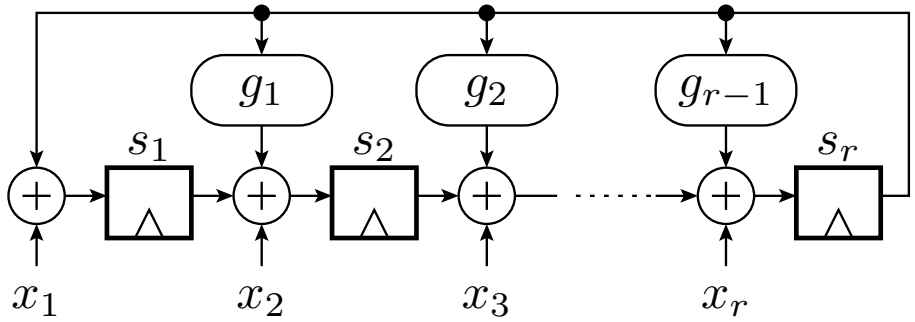
Es gibt viele weitere lineare Automaten, die auch zyklisch Bitfolgen in zufälliger Reihenfolge erzeugen. Beispiel Zellenautomaten, bei denen jedes Folgebit aus dem eigenen und den Zuständen der Nachbarbits gebildet wird:



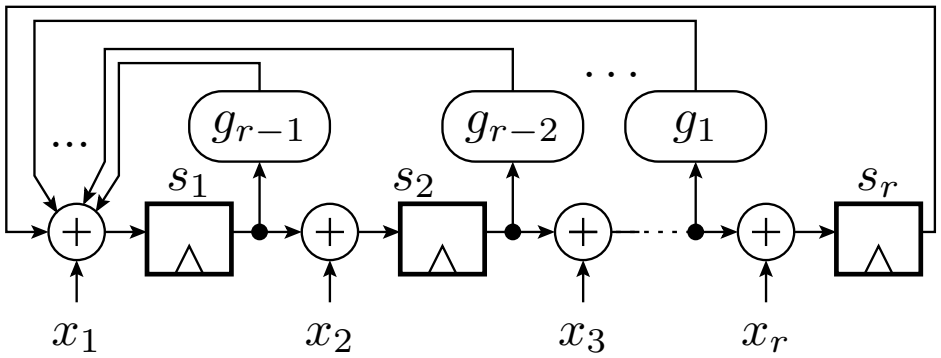
LFSR als Signaturregister

Zur Prüfkennzeichenbildung werden die Ausgaben des Testobjekts modulo-2 (EXOR) zu den LFSR-Zuständen addiert.

Paralleles Signaturregister mit dezentraler Rückführung



Paralleles Signaturregister mit zentraler Rückführung



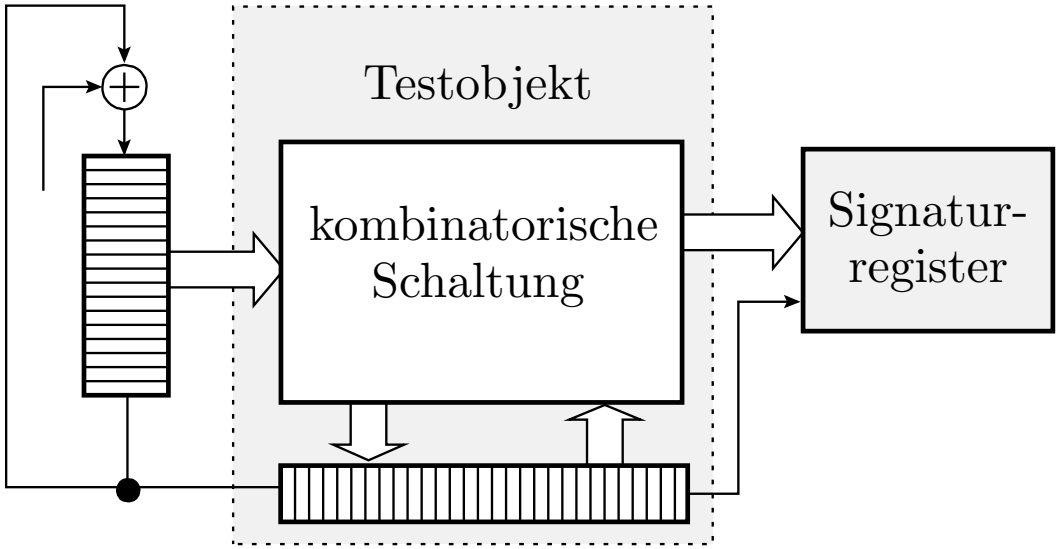
Das LSFR bildet so pseudo-zufällig ein Prüfkennzeichen.

Erkennungssicherheit für abweichende Testausgaben:

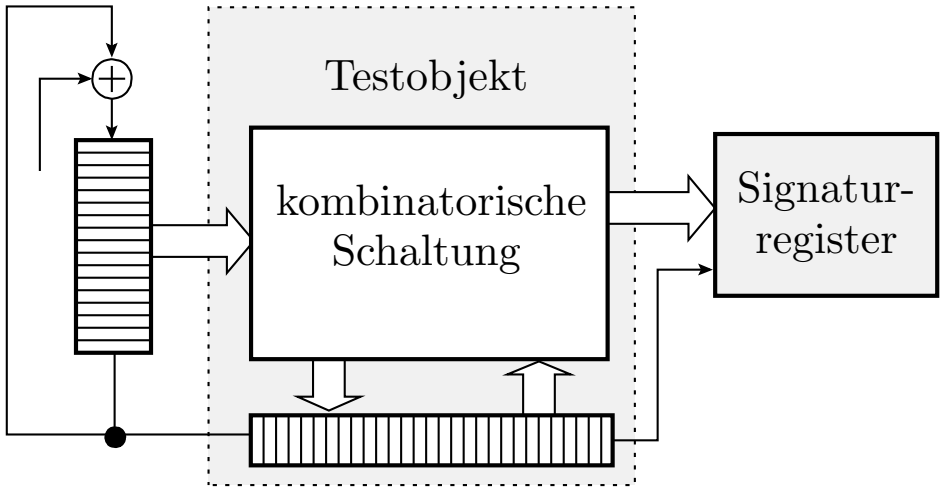
$$p_{\mathrm{E}} \approx 1 - 2^{-r}$$

(r – Registerlänge, siehe Foliensatz TV_F3, Abschn. 1.2 Prüfkennzeichen).

Pseudo-Zufallstest für die Übergangsfunktion



- Die Fehlerüberdeckung für den pseudo-zufälligen Test von Automaten ist oft deutlich höher, wenn die Übergangs- und Ausgabefunktionen isoliert getestet werden.
- Erweiterung der Eingabe-, Zustands- und Ausgaberegister um Schiebefunktionen für den Test.



- Wiederhole für jeden Testschritte (z.B. 1 Million mal)
  - Schiebe den Registerzustand in das Signaturregister und beschreibe das Zustandsregister gleichzeitig mit Pseudo-Zufallswerten aus dem Eingaberegister.
  - Ausführung eines Testschritts mit Pseudo-Zufallswerten am Eingang und in den internen Registern.
  - Ergebnisabbildung in das Zustands- und Signaturregister.

Gewichteter Zufallstest

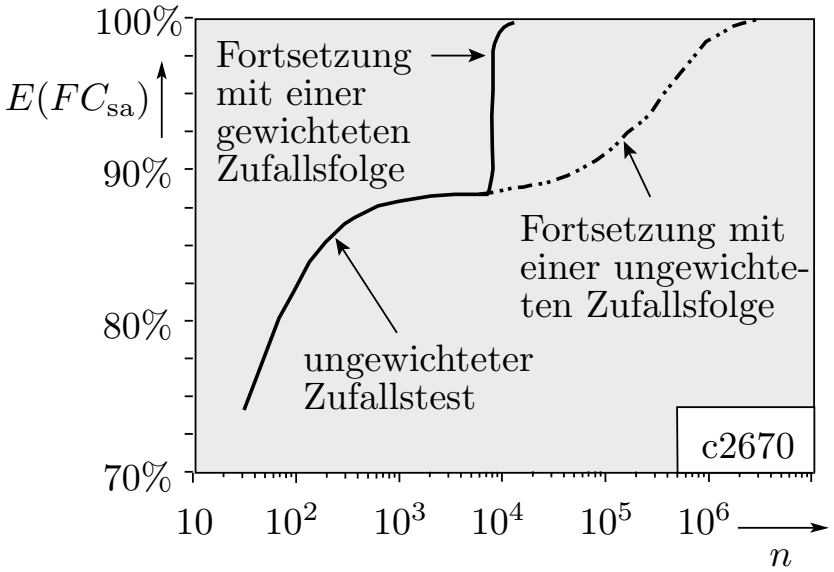
Außer durch feinere Modularisierung lässt sich die Fehlerüberdeckung von Zufallstests auch durch Änderung des Operationsprofils während des Tests erhöhen. Pragmatischer Ansatz[14]:

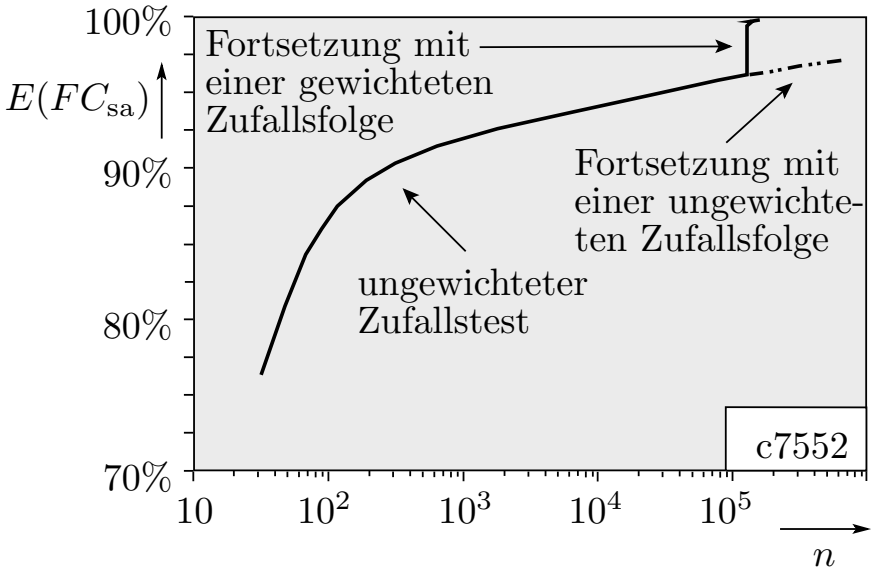
1. Festlegung einer größeren Menge von Modellfehlern.
2. Längerer Test mit ungewichteten Zufallswerten und Abhaken aller damit nachweisbaren Modellfehler.
3. Suche für die restlichen Modellfehler eine Eingabewichtung, die deren Nachweiswahrscheinlichkeiten erheblich erhöht.
4. Längerer Test mit den so gewichteten Zufallswerten und Abhaken aller damit nachweisbaren Modellfehler.
5. Wenn erforderlich, Wiederholung von Schritt 3 und 4.

[14] J. Hartmann, G. Kemnitz: How to do weighted random testing for BIST?. ICCAD 1993.

Experiment mit den Schaltungen c2670 und c7552[15]

- Test mit 10^4 bzw. 10^5 ungewichteten Zufallsmustern, die 90% bzw. 95% der Haftfehler nachweisen.
- Gezielte Testberechnung für die restlichen Haftfehler.
- Individuelle Wichtung aller Eingabebits zur Maximierung der mittleren Auftrittshäufigkeit der berechneten Testeingaben.





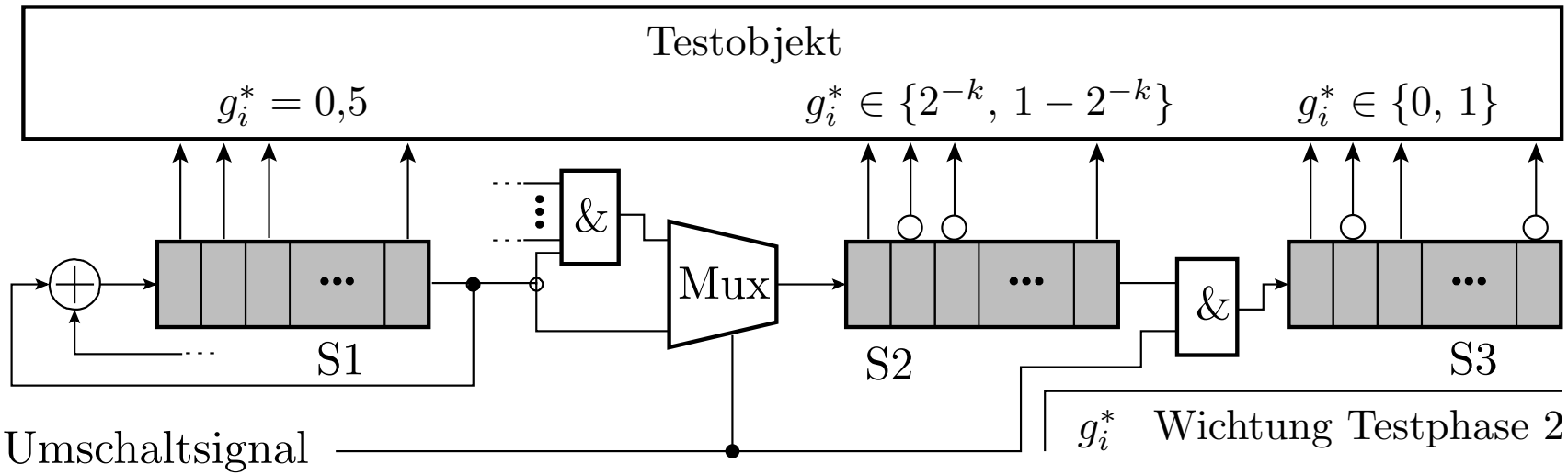
[15] Kombinatorische Benchmarkschaltungen zum Vergleich von Testlösungen. Die Zahl hinter dem »c« ist die Anzahl der Signalleitungen.

Implementierung als Selbsttest

Im Experiment wurde der Wertebereich für die Wichtung auf die schaltungstechnisch einfach einstellbaren Werte begrenzt:

$$g_i \in \left\{0,\ 2^{-k},\ 0{,}5,\ 1-2^{-k},\ 1\right\}$$

Diese werden mit wenigen UND-Gattern erzeugt und mit Scan-Registern an die Eingänge weitergeleitet.



Nicht nennenswert aufwändiger als ohne Wichtung.

Baugruppen und CP-Systeme

Baugruppentest

Inbetriebnahme von Prototypen:

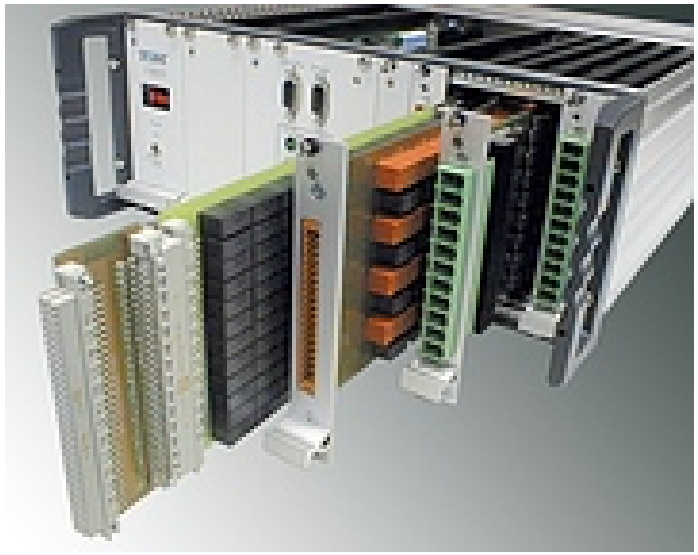
- Sichtprüfung auf Bestückungs- und Lötfehler.
- Kontrolle der Verbindungen über Widerstandsmessungen.
- Anschluss der Spannung. Stromüberwachung. Kontrolle der Bauteile auf unnormale Erwärmung.
- Anschluss von Signalgeneratoren, Oszillographen, ...
- Manuelle Einstellung der Eingaben und Ausgabekontrolle.

Serienfertigung:

- Automatisierte statische Tests auf Verbindungs- und Bestückungsfehler (vergl. Foliensatz TV_F4, Abschn. 3).
- Auf das Ausprobieren von Beispielfunktionen wird zum Teil verzichtet[16].

[16] Annahme: Schaltkreis- und andere Bauteilfehler werden fast alle von den Bauteiltests und die verbleibenden kaum im Verbund erkannt. Statische Tests erkennen (fast) alle Bestückungs- und Verbindungsfehler.

Modularer Funktionstester



Der komplette Test von CP-Systemen[17] verlangt die Bereitstellung von Testverläufen für die Sensorsignale und Kontrollen der Ausgabesignale. Typische Lösung ist ein Rechner mit einem modular zusammensetzbaren System aus

- Logikgenerator- und Logikanalysatorbaugruppen,
- DAU- und ADU-Baugruppen,
- programmierbaren Spannungsversorgungen,
- Baugruppen für Busschnittstellen (RS232, SPI, CAN, ...),
- Lastschaltungen, Adapter, ...

[17] Cypher-Physikalische Systeme

Nachbildung der Systemumgebung

physikalisch, als Simultionsmodell oder gemischt. Maschinen und Anlagenbau:

HIL- (Hardware in the Loop) Tester

- Physikalische Simulation der gesteuerten Maschine oder Anlage,
- 3D-Visualisierung des physikalischen Verhaltens,
- Untersuchung von Grenzwert- und Gefahrensituationen.

Fahrzeugbau, Luft- und Raumfahrt

- physikalische Simulationen von Motoren, Lenksystemen bis hin zu kompletten Flugzeugen,
- Nachstellung komplizierter Testsituationen im Labor (fahrendes Auto, Flugzeug in der Luft, ...)

Jedes Simulationsmodell hat Genauigkeitsgrenzen. Kein vollständiger Ersatz für den Test in der Anwendungsumgebung.