Einleitung

Die Mikroelektronik hält zunehmend Einzug in Bereiche unseres täglichen Lebens. Die Abhängigkeit des Menschen von der Technik wächst ständig, und damit kommt der Frage nach deren Zuverlässigkeit eine steigende Bedeutung zu.

Diese Frage nach der Zuverlässigkeit stellt sich insbesondere bei der Fertigung hochintegrierter Schaltkreise. Leider ist die Chipfertigung, sich immer an der Grenze des technisch machbaren bewegend, sehr fehleranfällig: Defektraten von über 40 % sind im VLSI–Bereich keine Seltenheit.
Man benötigt darum unbedingt leistungsfähige Verfahren, die gefertigte Chips auf ihre Korrektheit überprüfen, sie also testen. Welche Bedeutung der Fertigungstest in der Chipfertigung einnimmt, zeigt eine Schätzung von Milne [Mil85], nach der heute mehr als 25 % der Produktkosten im VLSI–Bereich auf den Testvorgang entfallen.

Um den Anwender bei der Erstellung von Fertigungstests zu unterstützen, wurden im Sonderforschungsbereich 124, Teilprojekt B1, verschiedene Testwerkzeuge entwickelt. Im Rahmen von Diplomarbeiten entstanden dabei ein Fehlersimulator [HSS92, Sch91], ein Testmustergenerator [Nik95], sowie eine graphische Oberfläche zur Steuerung dieser Tools [BHHK+94, Kra94]. Alle diese Testwerkzeuge wurden in das Entwurfssystem CADIC [BHKM+87] integriert.

In der Konzeption von Fehlersimulator, Testmustergenerator und der zugehörigen graphischen Oberfläche spiegelt sich eine übergeordnete Zielsetzung wider. Bei allen drei Sytemen lag das Schwergewicht auf folgenden beiden Punkten:

(a) Es sollte eine **flexible, bibliotheksbasierte Fehlermodellierung** unterstützt werden. Erlaubt wird hierbei die Beschreibung beliebiger kombinatorischer Fehlerverhaltens auch bei komplexeren Grundzellen (Volladdierer, Multiplexer, Exor-Gatter, ...) des Schaltkreises.

    Diese Flexibilität in der Fehlermodellierung ist von großer Bedeutung, da die in der Praxis auftretenden Defekte von vielen Faktoren, wie z.B. der Technologie, dem Fertigungsprozeß oder dem Layout der Grundzellen, abhängen. Sie erlaubt es, das Fehlermodell jeweils genau auf eine gegebene Anwendung zuzuschneiden.

(b) Ein weiteres einigendes Konzept bildete die Unterstützung einer **interaktiven Testmustergenerierung**. Die Grundidee besteht hierbei darin, dem Schaltungsdesigner bzw. Testingenieur ein System zur Verfügung zu stellen, das es ihm ermöglicht, sein „High-Level-Wissen“ über die Schaltkreisstruktur zur Verbesserung der Resultate automatischer Testmustergenerierungsalgorithmen zu nutzen. Hierzu müssen diese Algorithmen um einfach handhabbare, effektive Mechanismen zur Steuerung durch Entwerferwissen erweitert werden.

    Mit Hilfe der durch die graphische Benutzeroberfläche [Bur95] des VLSI-Entwurfssystems CADIC [BHKM+87] gegebenen hierarchischen Schaltkreisdarstellung können die Ergebnisse der automatischen Testmustergenerierungsalgorithmen visualisiert und Steuerungsinformationen eingelesen werden.

Der vorliegende Bericht, im wesentlichen identisch mit der Diplomarbeit von Ulrich Nikolaus [Nik95], beschreibt die Realisierung des im Rahmen des oben genannten Projektes entwickelten Testmustergenerators COAT (**C**ell **O**riented **A**kers–logic–based **T**est pattern generator).

Die Automatische Testmustergenerierung (*engl. Automatic Test Pattern Generation,* kurz ATPG) ist ein intensiv untersuchtes Forschungsgebiet, und es gibt heute bereits eine ganze Reihe leistungsfähiger ATPG–Systeme (vgl. [Roth66, Goel81, Fuji83, STS87, GMOD90, KP92, HWA95]). Gemäß der oben angegebenen allgemeinen Zielsetzung lag das Schwergewicht bei der Entwicklung des ATPG–Tools COAT im Vergleich zu den bekannten Ansätzen vor allem auf den folgenden Punkten:

1. *Flexible Fehlermodellierung.* Wie schon oben erwähnt, sollte ein bibliotheksorientiertes Fehlermodell eine möglichst komfortable Anpassung des ATPG–Algorithmus an die jeweilige Fertigungstechnologie erlauben. In dieser Arbeit wird geschildert, wie dieses bereits im Fehlersimulator COFS [HSS92] verwandte Fehlermodell zur automatischen Testmustergenerierung genutzt werden kann.

2. *Unterstützung komplexer Basiszellen.* Im Testmustergenerator COAT können neben den üblichen elementaren Gattern (AND, OR,...) auch komplexere Zellen wie z.B. Volladdierer oder Multiplexer als Basiszellen definiert werden.

   Diese Flexibilisierung bei den Grundbausteinen ist gerade in Verbindung mit dem oben genannten Fehlermodell sinnvoll; hat sich doch gerade bei Komplexgattern gezeigt, daß sich an ihnen häufig auftretende Fertigungsdefekte mit dem üblichen stuck_at–Fehlermodell nur unvollkommen beschreiben lassen.

3. Was die Zielsetzung der *interaktiven Testmustergenerierung* betrifft, untersucht die vorliegende Arbeit, welche Möglichkeiten es bei der ATPG konkret gibt, um Wissen des Schaltkreisentwerfers zur Effizienzsteigerung einzusetzen.

   Dazu ein Beispiel: Für den Schaltkreisentwerfer ist es aus seiner high–level–Sicht häufig leicht zu entscheiden, wie eine Testeingabe zu einem bestimmten Teilmodul der Gesamtschaltung weitergeleitet werden kann. Für den ATPG–Algorithmus, der auf Gatterebene arbeitet und dem aufgrund dessen der Gesamtüberblick über die Schaltung fehlt, kann dieselbe Propagation hingegen ein schwieriges Problem darstellen. Könnte man dieses Wissen dem gate–level–Algorithmus zur Verfügung stellen, so die Überlegung, dann müßte es möglich sein, ihn wesentlich zu beschleunigen.

Besonders wichtig ist bei der Testmustergenerierung ganz allgemein eine *exakte Darstellung des Schaltkreiszustandes:* Schaltkreisbelegungen spielen bei der automatischen Testmustergenerierung eine zentrale Rolle. Sie beschreiben die aktuelle Berechnungssituation und dienen dem ATPG–Algorithmus, bei dem es sich in der Regel um ein *branch & bound*–Verfahren handelt, im branch–Schritt als Entscheidungsgrundlage.
Die 5–wertige Logik nach Roth [Roth66], die den Schaltkreisbelegungen der meisten Testmustergeneratoren zugrundeliegt, ist hier recht ungenau. Aus diesem Grund findet in COAT die exaktere 16–wertige *Akers–Logik* [Ake76] Verwendung.

Ein weiterer Schwerpunkt dieser Arbeit war es daher, zu untersuchen, wie sich die bekannten ATPG–Vorgehensweisen für Roth-Logik auf die Akers-Logik übertragen lassen.

Eine praktische Erprobung des Testmustergenerators COAT erlaubte die Kooperation mit dem Lehrstuhl von Prof. Dr. W. J. Paul im Rahmen des SB–PRAM (**S**aar**B**rücker **P**arallel **R**andom **A**ccess **M**achine) Projektes [ADKP+93]. COAT wurde dort von Robert Knuth bei der Testmustergenerierung und –validierung für den PRAM–Prozessorchip erfolgreich eingesetzt [Knu95].

Danksagungen

An dieser Stelle möchten wir uns zunächst bei Prof. Dr. Hotz für die vielfältige Unterstützung dieser Arbeit bedanken.

Mathias Krallmann danken wir für eine fruchtbare und harmonische Zusammenarbeit. Für ihre Unterstützung und zahlreiche Anregungen bedanken wir uns darüber hinaus bei Lars Köller, Ralf Osbild, Jörg Ritter und den anderen Mitarbeitern der 'Testen'-Gruppe.

Zuletzt möchten wir uns bei Robert Knuth vom Lehrstuhl von Prof. Dr. W. J. Paul für die gelungene Zusammenarbeit bedanken; für seine wertvollen Anregungen und die Weitergabe seiner Erfahrungen aus der Praxis.

Inhaltsverzeichnis

Kapitel 1

Grundlagen

Einleitung

Im ersten Kapitel beschreibe ich einige Begriffe aus den Gebieten der Schaltkreistheorie und des VLSI–Entwurfes, die für das Verständnis meiner Arbeit notwendig sind. Danach werde ich Grundlegendes aus dem Bereich der ATPG *(Automatic Test Pattern Generation)* vorstellen, Aufgaben und Ziele der Automatischen Testmustergenerierung beschreiben ($\rightarrow$ 1.3) und mich mit dem Problem der Fehlermodellierung beschäftigen ($\rightarrow$ 1.2).
Wo sich meine Begriffsbildungen von den aus bekannten ATPG–Systemen wie PODEM [Goel81] oder FAN [Fuji83] gewohnten unterscheiden, werde ich darauf besonders hinweisen (die Unterschiede betreffen insbesondere das *flexible kombinatorische Zellenfehlermodell* [vgl. Abschnitt 1.2.2] sowie das von mir verwendete leistungsfähigere Verfahren zur Darstellung von *Schaltkreisbelegungen* [vgl. Kapitel 2], welches auf einer von Sheldon B. Akers im Jahre 1974 vorgestellten Logik [Ake76] basiert).

1.1 Schaltkreise

Für die Begriffe *gerichteter Graph, kombinatorischer Schaltkreis* und *Zelltyp* werde ich in meiner Arbeit folgende Formalisierungen verwenden:

Definition 1.1.1:
$\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E})$ *heißt* **gerichteter Graph**, *wenn gilt:*

- V, E *sind disjunkte und endliche Mengen*
- $E \subseteq (V \times V)$

V wird auch als Menge der Knoten, *E als Menge der* Kanten *von G bezeichnet. Die* Richtung *der Kanten $e = (u, v) \in E$ ist gegeben durch $\mathcal{Q}, \mathcal{Z} : E \rightarrow V$ mit:*

- $\mathcal{Q}(e) := u$ *heißt* Quelle *von e*
- $\mathcal{Z}(e) := v$ *heißt* Ziel *von e*

Schreibweise: Sei $v \in V$.

indeg(v) $:= \operatorname{card}\{e \in E \mid \mathcal{Z}(e) = v\}$ heißt *Eingangsgrad von v.*
outdeg(v) $:= \operatorname{card}\{e \in E \mid \mathcal{Q}(e) = v\}$ heißt *Ausgangsgrad von v.*

Definition 1.1.2:
Ein **Zelltyp** *ist ein Tripel* $\boldsymbol{T} = (\boldsymbol{ig}, \boldsymbol{og}, \boldsymbol{\tau})$ *mit* $ig, og \in N_0$ *und* $\tau : \{0,1\}^{ig} \rightarrow \{0,1\}^{og}$. *$ig$ heißt auch* Anzahl der Eingänge, *og* Anzahl der Ausgänge *des Zelltyps. τ berechnet die sogenannte* Gatterfunktion.

Beispiel:
Das Tripel $(2, 1, AND_2)$ mit

$$AND_2(i_1, i_2) = \begin{cases} 1 & \text{, falls } i_1 = 1 \wedge i_2 = 1 \\ 0 & \text{, sonst.} \end{cases}$$

ist der *Zelltyp eines AND–Gatters mit 2 Inputs.* ■

Im Gegensatz zu anderen Schaltkreismodellen werde ich auch die primären Ein– und Ausgänge einer Schaltung als Zelltypen beschreiben. Es sei:

$PI := (0, 1, undefiniert)$ und
$PO := (1, 0, undefiniert)$.

Um den Begriff *Schaltkreis* zu definieren, gibt es verschiedene Möglichkeiten. Ich werde das folgende Modell wählen, welches sich auf die Darstellung der für meine Arbeit wesentlichen Schaltkreiseigenschaften beschränkt.

Definition 1.1.3:
Ein 2–Tupel $\boldsymbol{C} = (\boldsymbol{G}, \boldsymbol{F})$ *heißt* **kombinatorischer Schaltkreis**, *falls gilt:*

- $G = (V, E)$ *zykelfreier gerichteter Graph mit*
  - $V = (Z \cup S); \quad Z \cap S = \emptyset$.
    Z ist die Menge der Zellknoten *und korrespondiert zu den im Schaltkreis vorkommenden Gattern.*
    S ist die Menge der Signalknoten*; sie entspricht den Signalnetzen, welche die Gatter des Schaltkreises miteinander verbinden.*
  - $E \subseteq ((Z \times S) \cup (S \times Z))$, *d.h. C bipartit in Z und S*
  - $\forall\, s \in S : indeg(s) = 1$
  - $\forall\, z \in Z$ *mit* $typ(z) = (ig, og, \tau) : \quad indeg(z) = ig \ \wedge \ outdeg(z) = og$
- $F = \{pi, po, in, out\}$ *Menge von Funktionen.*

  Dabei sind pi (po) bijektive Abbildungen, die die primären Eingänge (Ausgänge) der Schaltung durchnumerieren:

  - $pi : \{1, ..., \#Z_{PI}\} \rightarrow Z_{PI}$, *wobei* $Z_{PI} := \{z \in Z \mid typ(z) = PI\}$
  - $po : \{1, ..., \#Z_{PO}\} \rightarrow Z_{PO}$, *wobei* $Z_{PO} := \{z \in Z \mid typ(z) = PO\}$

  Die Funktionen in und out geben an, welche Signalknoten mit den I/O–Pins einer Zelle z verbunden sind.

  - $in(z, i) = s$ *bedeutet, daß der i–te Input von Zelle z mit dem Signalknoten s verbunden ist* $((s, z) \in E)$.
  - $out(z, i) = s$ *bedeutet, daß der i–te Output von Zelle z mit dem Signalknoten s verbunden ist* $((z, s) \in E)$.

1.2 Fehlermodelle

1.2.1 Flexible Fehlermodellierung

Bei der Schaltkreisdefinition im letzten Kapitel handelt es sich um ein Modell auf *Gatterebene:* In ihm wird die *logische Struktur* der Schaltung, nicht aber deren *physikalische Realisierung* beschrieben.
Viele Testmustergeneratoren arbeiten auf dieser Abstraktionsebene; zum Beispiel deshalb, weil zum Entwurfszeitpunkt noch gar nicht feststeht, in welcher Technologie ein Chip später gefertigt wird. Damit untersuchen die Testmustergeneratoren also nicht die eigentlichen *physikalischen Defekte,* sondern die aus diesen resultierenden Veränderungen des *Schaltverhaltens.*
Die Abstraktion von der Transistor– auf die Gatterebene hat Vorteile. Unter anderem gibt es viele Transistorfehler, die — obwohl physikalisch verschieden — zum selben logischen Fehlverhalten führen. Betrachtet man von vorneherein logische Auswirkungen, so reduziert man die Anzahl der Fehler und erhält einen effizienteren Algorithmus.

Das 'klassische' Fehlermodell der ATPG ist das *(Einzel–) stuck_at– oder Haftfehlermodell*. Es ist leitungsorientiert und beschreibt vor allem einfache Defekte wie z.B. Leitungsbrüche. Im *stuck_at*–Modell wird angenommen, daß

1. *genau eine* Leitung des Schaltkreises defekt ist[1].
2. der Fehler sich so auswirkt, daß die Leitung entweder fest auf logisch 0 oder logisch 1 liegt (man bezeichnet den Fehler je nachdem mit *stuck_at_0 (s_a_0)* bzw. *stuck_at_1 (s_a_1)).*

Die bei der Fertigung eines Chips auftretenden Defekte variieren stark mit der verwendeten Fertigungstechnologie. Dies bedeutet, daß man das verwendete Fehlermodell immer wieder neu auf seine Eignung zum Modellieren der auftretenden Defekte untersuchen muß.
Während das stuck–at–Fehlermodell für die Fertigung in TTL–Logik durchaus geeignet war, stellte die Entwicklung neuer Technologien (NMOS und insbesondere CMOS) seine Eignung zur Beschreibung der real auftretenden Fehler zunehmend in Frage.
Auch durch die Verwendung komplexer Gattertypen beim Layout ergaben sich Probleme mit dem stuck–at–Modell. Gatter, die — wie z.B. der Volladdierer — intern eine nichttriviale Schaltlogik, aber nur eine kleine Schnittstelle nach außen besitzen (Volladdierer: 3 Gattereingänge, 2 Ausgänge), zeigen bei Defekt oft Fehlfunktionen, welche Haftfehler nur unvollkommen beschreiben können.

Üblicherweise wird dieses Problem durch Auflösung des Komplexgatters in elementare Komponenten und anschließende stuck–at–Betrachtung auf sogenannten 'internen Leitungen' gelöst. Da es für jede komplexe Schaltung mehrere verschiedene Elementarrealisierungen gibt, ist jedoch fraglich, ob die zur Auflösung gewählte Darstellung die tatsächliche Realisierung auf Transistorebene widerspiegelt.
Generell sind Fertigungsfehler (und damit auch das Fehlermodell) von verschiedenen Faktoren abhängig: Von der verwendeten *Technologie,* vom *Layout,* vom *Fertigungsprozeß.* Bei jeder Veränderung einer Komponente muß das Fehlermodell geeignet angepaßt werden, will man die Qualität der generierten Tests sichern. Dies wiederum führt zu permanenten Modifikation am Testalgorithmus, sofern das Fehlermodell fest in diesen hineincodiert ist[2].

Solche Beobachtungen motivierten die Entwicklung eines flexiblen, mächtigeren Fehlermodelles im Rahmen des *SFB 124 — VLSI–Entwurfsmethoden und Parallelität.* Auf zwei Punkte wurde dabei besonderer Wert gelegt: Erstens ein beliebiges kombinatorisches Fehlverhalten beschreiben zu können, zweitens eine flexible Schnittstelle — eine sogenannte *Fehlerbibliothek* — zur Verfügung zu stellen, die einen möglichst einfachen Wechsel zwischen verschiedenen Fehlermodellen ermöglicht[3].

[1] Man geht in diesem Fall von einer *single fault assumption* aus, d.h. man unterstellt, daß sich stets nur *ein* Fehler auf einmal im Schaltkreis befindet. Diese Einzelfehlerannahme ist allgemein üblich.
Der Grund für den Verzicht auf Mehrfachfehlermodellierung liegt zum einen begründet in der stark wachsenden Komplexität des Testproblems bei *multiple fault assumption,* zum anderen in der praktischen Beobachtung, daß Testsätze für Einfachfehler auch die überwiegende Zahl der Mehrfachfehler testen.

[2] Ein wichtiges Prinzip zur Minimierung solcher Modifikationen am Testalgorithmus besteht in der klaren Trennung von fehlermodellabhängigen und -unabhängigen Aufgaben bei der Testmustergenerierung [HWA95].

[3] Um allgemeinere Fehlermodelle bemüht man sich auch an anderer Stelle — so besitzt z.B. auch der Fehlersimulator COMSIM [MA93] die Möglichkeit zu flexibler Fehlermodellierung.

Jenes Fehlermodell wurde bereits praktisch erprobt und hat sich bei der *Fehlersimulation* [HSS92] bewährt. Aufgabe der vorliegenden Arbeit war es unter anderem, die *bibliotheksorientierte Fehlermodellierung von beliebigem kombinatorischem Fehlverhalten der Grundzellen* in einen *Testmustergenerator* zu integrieren.

1.2.2 Das kombinatorische (Einzel–)Zellenfehlermodell

Das *beliebige kombinatorische Zellenfehlermodell* beschreibt (Zellen–)Fehler — wie schon erwähnt — über sogenannte *Fehlerbibliotheken.* In diesen können für jeden Zelltyp beliebige Veränderungen der Gatterfunktion modelliert werden, solange die Zelle kombinatorisch bleibt. Die Spezifikation dieses logischen Fehlverhaltens geschieht über *ceps:*

Definition 1.2.1:
Ein **C***ondition–***E***ffect* **P***air* **(cep)** *zu einem Zelltyp* $t = (ig, og, \tau)$ *und einem Fehler* F_t *ist ein 2–Tupel* $(\boldsymbol{ib}, \boldsymbol{ob})$ *mit* $ib \in \{0,1\}^{ig}$ *und* $ob \in \{0,1\}^{og}$.
Es beschreibt eine Input–Output–Belegung, bei der sich das Verhalten einer korrekten Zelle vom typ t von einer F_t*–fehlerbehafteten Zelle unterscheidet.*
Man bezeichnet die Eingangsbelegung ib *auch als* condition *und die Ausgangsbelegung* ob *als* effect *des ceps.*

Ein cep beschreibt folglich *genau einen* inkorrekten Schaltzustand des defekten Gatters. (Physikalischen) Fehlern, die sich auf verschiedene Schaltzustände auswirken, sind mehrere ceps zugeordnet.

Beispiel:
Ein stuck_at_1–Fehler am Ausgang eines AND2–Gatters läßt sich im Zellenfehlermodell durch die folgende Menge von ceps beschreiben:

$$\{((0,0),1),((0,1),1),((1,0),1)\}$$

■

Eine Fehlfunktion in einem beliebigen Schaltkreis läßt sich somit durch die Angabe des von dem Fehler betroffenen Gatters sowie eines ceps (der die Fehlfunktion beschreibt) eindeutig beschreiben. Ich werde ein solches Paar im folgenden als *elementaren Zellenfehler* bezeichnen.

Definition 1.2.2:
Ein **elementarer Zellenfehler** *ist ein Paar* $\mathbf{f} := (\mathbf{z_f}, \mathbf{c_f})$; *wobei* z_f *eine Basiszelle und* c_f *ein zugehöriger cep ist.*

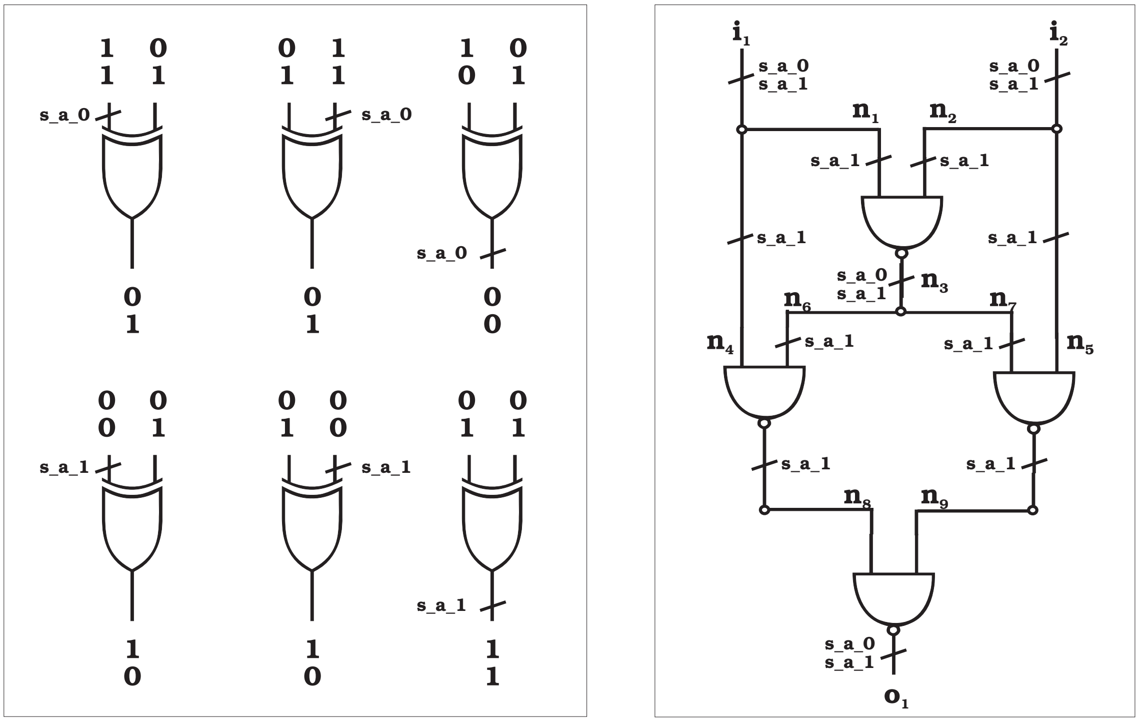
Man überlegt sich leicht, daß sich jedes beliebige kombinatorische Fehlverhalten einer Basiszelle durch eine Menge elementarer Zellenfehler darstellen läßt: Jeder cep entspricht der Veränderung *eines* Eintrags in der Logiktabelle von z_f; durch Angabe mehrerer Zellenfehler läßt sich *jede* Veränderung der Schaltfunktion beschreiben.
Man kann sich bei der Testmustergenerierung daher auf elementare Zellenfehler beschränken. Ich werde deshalb im folgenden den Begriff 'Zellenfehler' als Synonym zum '*elementaren* Zellenfehler' verwenden. Zellenfehler in ihrer allgemeinen Form werde ich explizit als '*allgemeine* Zellenfehler' bezeichnen.

Interessant ist natürlich der Zusammenhang zwischen Zellenfehler– und stuck–at–Fehlermodell. Ein direkter Vergleich wird zunächst dadurch erschwert, daß das stuck–at–Modell *leitungsorientiert,* das Zellenfehlermodell jedoch *gatterorientiert* arbeitet.
Da aber jede Signalleitung der Schaltung (jede Kante e des bipartiten Graphen) laut Definition mit genau einem eindeutig bestimmten Gatter g verbunden ist[4], kann man den stuck–at–Fehler F_{s_a} als Defekt eines Pins von g interpretieren und so durch einen Zellenfehler simulieren. Damit ist das stuck–at–Modell ein *Spezialfall* des Zellenfehlermodelles (siehe auch obiges Beispiel).

Ein Nachteil des Zellenfehlermodelles scheint auf den ersten Blick ein höherer Rechenaufwand zu sein, denn es sind eventuell *mehrere* elementare Zellenfehler nötig, um *einen* stuck–at–Fehler zu beschreiben. Daß die Verwendung des Zellenfehler–Modelles trotzdem zu einer Effizienz*steigerung* führen kann, zeigt das folgende Beispiel:

[4] Wegen der bipartiten Eigenschaft des Schaltkreisgraphen, vgl. Abschnitt 1.1.

Beispiel:



a) Aufzählung aller stuck_at_Fehler des EXOR-Makros samt der fehleraktivierenden Schaltkonfigurationen.

b) Basiszellenauflösung des EXOR mit verbleibenden nichtäquivalenten stuck-at-Fehlern

Abbildung 1.1: Vorteil des Zellen– gegenüber dem stuck–at–Fehlermodell

Das EXOR–Gatter in Abbildung 1.1 sei erschöpfend zu testen. Unter dem stuck–at–Fehlermodell gibt es 6 mögliche Fehler (nämlich jeweils s_a_0 & s_a_1 an den Inputs i_1, i_2 sowie dem Ausgang o). In Abbildung 1.1 a) sind diese Fehler zusammen mit den ceps, welche die zugehörigen fehlerhaften Schaltzustände beschreiben, aufgeführt.
Wie die Abbildung zeigt, reichen schon *drei* Inputbelegungen am EXOR für einen vollständigen stuck–at–Testsatz aus; z.B. die Elemente der Menge $\{(0,0);(0,1);(1,0)\}$. Damit ist zwar eine 100 %–ige Fehlerüberdeckung im *Modell* erreicht, nicht aber in der *Realität:* Wäre das EXOR–Gatter intern beispielsweise so realisiert wie in Abbildung 1.1 b) gezeigt[5], so bliebe ein s_a_1–Fehler an der internen Leitung n_3 ungetestet, da er nur durch die Eingabebelegung (1,1) zu testen ist. Fazit: Selbst eine 100%–ige stuck–at–Fehlerüberdeckung am EXOR wäre in der Praxis nicht ausreichend.

Dieses Dilemma löst man — sofern man auf das stuck–at–Modell festgelegt ist — üblicherweise durch *Auflösung* des EXOR–Makros. Durch die internen Leitungen steigt jedoch die Gesamtanzahl der stuck-at–Fehler stark an[6]; beim betrachteten EXOR beispielsweise von 6 auf 24. Nutzt man lokale Fehleräquivalenzen[7] aus, so verbleiben immer noch 16 zu untersuchende Fehler.
Ein stuck–at–Testmustergenerator würde also 16 Testgenerierungsläufe durchführen. Das EXOR als *Ganzes* besitzt jedoch nur *vier* mögliche Schaltkonfigurationen: Als fehlerhafte Input–Output–Belegungen kommen daher nur die vier ceps $((0,0),1)$, $((0,1),0)$, $((1,0),0)$ und $((1,1),1)$ in Frage.

[5] Diese Realisierung des EXOR wird in den ISCAS–Benchmarkschaltungen [BF85] verwendet.
[6] Als Leitungsabschnitt zählen die Abschnitte zwischen einem Gatterpin und einem Signalknoten, also die Kanten des bipartiten Graphen. Zu jedem Leitungsabschnitt müssen zwei Haftfehler betrachtet werden.
[7] Zwei unterschiedliche Defekte, die nach außen hin dieselben Auswirkungen haben, bezeichnet man als *äquivalent*. Äquivalent ist beispielsweise in Abbildung 1.1 b) der Fehler s_a_0 an n_8 mit dem s_a_1 an o_1.

Bei Verwendung des Zellenfehlermodelles reduziert sich so die Anzahl der Programmläufe von 16 auf 4, also auf ein Viertel. Die dadurch erreichte Laufzeitersparnis wiegt den höheren Initialisierungsaufwand bei Verwendung des Zellenfehlermodelles auf. ■

1.3 Aufgaben und Ziele der ATPG

Ist das Fehlermodell bestimmt, kann das *Testen* beginnen. Zunächst auch hier eine formale Definition:

Definition 1.3.1:
Ein Eingabemuster **m** *für eine Schaltung* C *heißt* **Test eines elementaren Zellenfehlers** $\mathbf{f} := (\mathbf{z_f}, \mathbf{c_f})$, *wenn* m, *angelegt an die Primären Eingänge (PI) der Schaltung* C_f *(in der das Gatter* z_f *das durch* c_f *beschriebene Fehlverhalten zeigt)* zu einer fehlerhaften Ausgabe an mindestens einem Primären Ausgang (PO) von C_f *führt.*

Nicht jedes Eingabemuster ist als Test für einen cep c geeignet. Viele Input–Belegungen führen zu einer *Fehlermaskierung,* bei der die Schaltung trotz Defekt eine korrekte Ausgabe liefert. Eine Fehlermaskierung tritt auf, wenn:

1. das gewählte Eingabemuster keine Fehlfunktion am Fehlergatter z_f bewirkt, z.B. weil es eine stuck–at–0–Leitung auf logisch 0 legt → keine *Aktivierung* des Fehlers.
2. das Muster zwar einen logischen Fehler an der Fehlerstelle erzeugt, welcher sich aber auf die primären Ausgänge nicht auswirkt → keine *Propagation* des Fehlers.

Um *Aktivierung* und *Propagation* eines Fehlers zu erreichen — und damit einen *Test* zu generieren — wird in der Praxis ein iteratives Vorgehen gewählt. Ausgangspunkt ist ein unspezifizierter Schaltkreiszustand, d.h. die primären Eingänge sind logisch noch nicht festgelegt. 0 oder 1 — beides ist möglich.
Über einen *branch & bound*–Algorithmus werden nacheinander PIs auf feste Werte gesetzt und das Eingabemuster schrittweise zu einem Test verfeinert[8]. Da das Problem der Testgenerierung für kombinatorische Schaltungen NP–schwer ist, werden in jedem Schritt zur Auswahl von PI und logischem Wert *Heuristiken* verwendet; Fehlentscheidungen sind dadurch möglich.
Um die Gesamtlaufzeit des Algorithmus zu begrenzen, wird für jeden Fehler eine *Laufzeitschranke* angegeben. Kann in dem durch die Laufzeitschranke gegebenen Zeitrahmen kein Test gefunden werden, wird die Berechnung abgebrochen und der nächste Fehler untersucht.

Manche Fehler sind überhaupt nicht testbar; man bezeichnet sie als *Redundanzen.* Redundanzen entstehen bei Schaltkreisdesigns, die die Schaltmöglichkeiten der verwendeten Gatter nur zum Teil ausnutzen. Dadurch bleiben Fehler, die nur mit einem ganz bestimmten, bei diesem Entwurf aber nicht genutzten Eingabemuster am fehlerhaften Gatter getestet werden können, unentdeckt. Ein redundanter Fehler hat keine Auswirkungen auf das logische Verhalten der Schaltung.
Häufig sind Redundanzen Folge von ineffizientem Design. Der Nachweis von Redundanzen stellt das laufzeitintensivste Problem der ATPG dar.

Zusammenfassend kann man sagen, daß jeder allgemeine/elementare Zellenfehler nach Ablauf der Testmustergenerierung einem der drei folgenden Zustände zugeordnet werden kann:

1. Status *unbekannt:* Zu dem entsprechenden allgemeinen/elementaren Zellenfehler wurde noch kein Testmuster gefunden. Der Fehler wurde entweder noch gar nicht bearbeitet, oder die Berechnung wurde beim Erreichen der Laufzeitschranke abgebrochen.
2. Status *getestet:* Ein elementarer Zellenfehler ist getestet, wenn für ihn ein Testmuster angegeben werden kann. Ein allgemeiner Zellenfehler ist getestet, wenn mindestens einer der elementaren Zellenfehler, die seine Fehlfunktion beschreiben, getestet ist.
3. Status *redundant:* Ein elementarer Zellenfehler ist redundant, wenn *kein* Eingabemuster zu einer fehlerhafte Ausgabe führt. Ein allgemeiner Zellenfehler ist redundant, wenn alle seine elementaren Zellenfehler redundant sind.

[8] Es gibt auch Testmustergeneratoren, die neben PIs auch interne Leitungen setzen (vgl. Kapitel 3.1).

Kapitel 2

Beschreibungen des Schaltkreiszustandes

Während der automatischen Testmustergenerierung ist es wichtig, stets Informationen über den aktuellen Zustand des Schaltkreises zu besitzen — darüber, ob der zu testende Fehler schon *aktiviert* ist, wie weit die *Propagation* fortgeschritten ist etc.

Datenstrukturen, in denen Informationen dieser Art gespeichert werden, bezeichnet man als *Schaltkreisbelegungen.* Ich werde in diesem Kapitel erklären, wie Schaltkreisbelegungen aussehen können, in welcher Weise sie die oben genannten Informationen darstellen und welche Probleme dabei auftreten.

2.1 Die 5–wertige Roth–Logik

Es liegt in der Natur der Sache, daß bei der Testmustergenerierung neben dem fehlerhaften Schaltkreis auch der korrekte Schaltkreis betrachtet werden muß: *Aktivierung* eines Fehlers bedeutet, daß sich der logische Wert an mindestens einer Leitung des fehlerhaften Schaltkreises von dem in der korrekt arbeitenden Schaltung unterscheidet. *Vollständige Propagation* ist gleichbedeutend mit der Existenz eines Pfades von der Fehlerstelle zu einem primären Ausgang (PO), auf dem sich die logischen Werte im korrekten und inkorrekten Falle *an jeder Stelle* voneinander unterscheiden. Es ist also eine Art *Parallelbetrachtung* von korrektem und fehlerhaftem Schaltkreis notwendig.

Ein von J. P. Roth entwickeltes Verfahren ([Roth66]), welches eine parallele Darstellung erlaubt und längst zu einer Standardtechnik geworden ist, zeigt die Abbildung 2.1:

Die beiden Schaltkreise werden, gedanklich gesehen, übereinandergelegt und gleichzeitig betrachtet. Zwei einander entsprechende Signalleitungen in korrektem und fehlerhaftem Schaltkreis werden in der parallelen Darstellung zu einer Leitung verschmolzen. Zu jedem Signal der parallelen Darstellung gehört damit statt *einem* logischen Wert ein *Paar:* Der Wert im *korrekten* Schaltkreis und der im *inkorrekten;* üblicherweise getrennt durch einen Schrägstrich.
Beispiel:
Betrachte Abbildung 2.1. Die Signalleitung S liegt im korrekten Schaltkreis auf logisch 0, im inkorrekten hingegen auf logisch 1 — in der parallelen Darstellung modelliert durch 0/1. ■

Da die Ausgangs–Schaltungen auf einer Binärlogik basieren, können in der parallelen Darstellung vier verschiedene Wertepaare auftreten: 0/0, 0/1, 1/0 und 1/1. In der Regel werden diese Grundwerte abkürzend wie folgt bezeichnet:

$$\begin{aligned} \mathbf{0} &:= 0/0 \\ \mathbf{1} &:= 1/1 \\ \mathbf{D} &:= 1/0 \\ \overline{\mathbf{D}} &:= 0/1 \end{aligned}$$

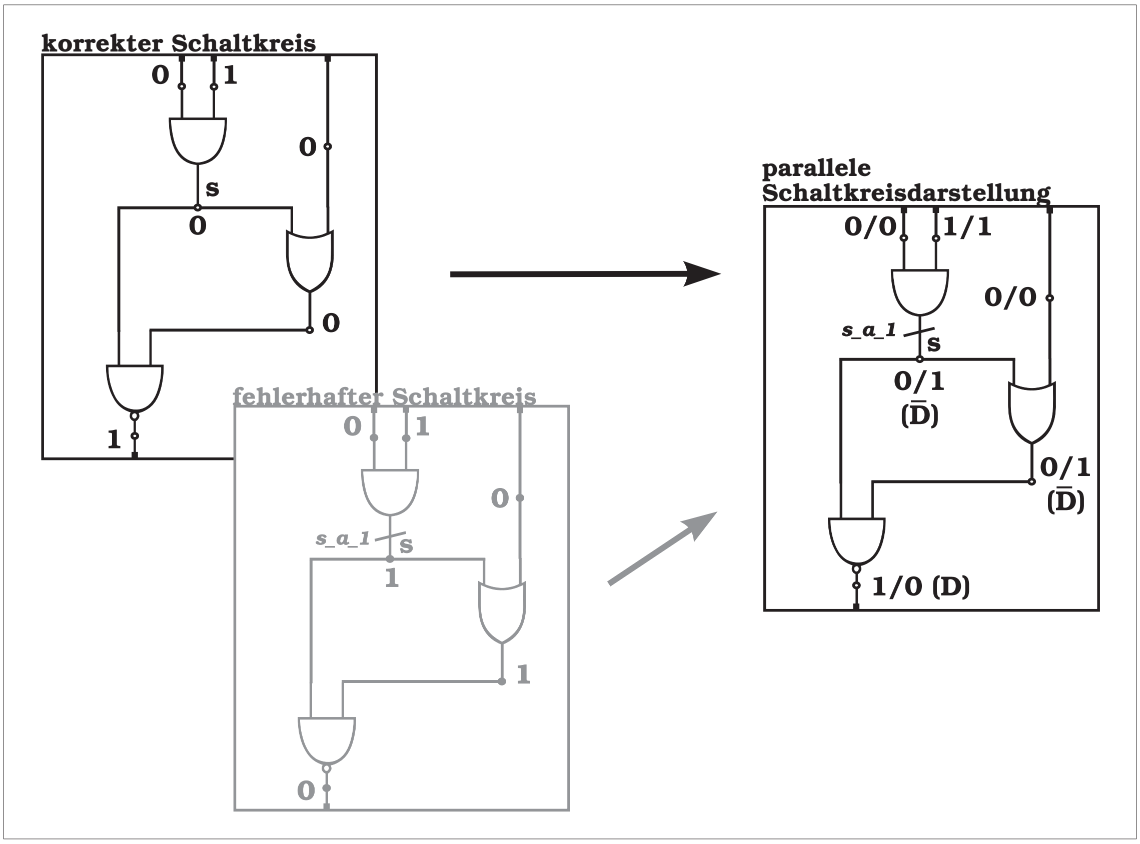


Abbildung 2.1: Darstellung von korrektem und fehlerhaftem Schaltkreis

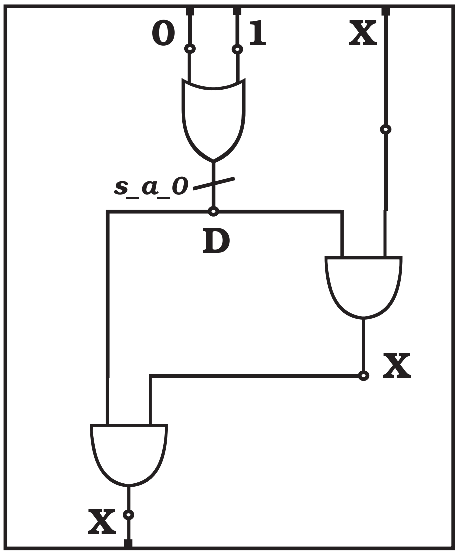


Abbildung 2.2: Beispiel einer partiellen Schaltkreisbeschreibung

Um auch *partielle* Berechnungssituationen modellieren zu können, bei denen die logischen Werte an einigen PIs noch unspezifiziert sind, erweiterte J. P. Roth in seinem sogenannten *D–Algorithmus* die Menge der logischen Grundwerte um den *don't care–Wert X.*

Total spezifizierte Leitungen (also solche, deren logisches Verhalten bereits feststeht) werden mit den Werten 0, 1, D und $\overline{D}$ belegt. Alle anderen Signale werden mit X beschriftet (vgl. dazu Abbildung 2.2).

Damit war es Roth möglich, beliebige Berechnungssituationen mittels der 5–wertigen Logik $\mathbf{L_R} := \{\mathbf{0}, \mathbf{1}, \mathbf{D}, \overline{\mathbf{D}}, \mathbf{X}\}$ darzustellen.

Zur Unterscheidung von *partiellen* Schaltkreisbelegungen bezeichnet man Darstellungen eindeutig spezifizierter Berechnungssituationen, die ohne den Wert X auskommen, als *totale* Schaltkreisbelegungen.

2.2 Die 16–wertige Akers–Logik

2.2.1 Schwächen der herkömmlichen Beschreibungen

Die Roth–Logik ist *die* Standardlogik der ATPG schlechthin. Viele wichtige Arbeiten ([Roth66], [Goel81], [Fuji83], [STS87] etc.) basieren auf ihr. Doch die Roth–Logik hat ihre Grenzen.
Ein Beispiel soll dies verdeutlichen. Gegeben sei der EXOR–Baum in Abbildung 2.3. Durch einen Defekt sei die Leitung s stuck_at_0. Die einzige Möglichkeit, diesen Fehler zu aktivieren, besteht darin, Leitung s (wie im Bild gezeigt) mit D zu belegen. — Alle anderen primären Eingänge des EXOR–Baumes seien zunächst unspezifiziert.
Wertet man die Gatterfunktion des EXOR E_1 für die beiden möglichen Eingangsbelegungen $(D, 0)$ bzw. $(D, 1)$ aus, so stellt man fest, daß am Ausgang o_{E1} nur noch die Signalwerte D oder $\overline{D}$ möglich sind. Damit ist der Signalwert an o_{E_1} *in jedem Fall* fehlerabhängig und die Propagation über E_1 damit sichergestellt.
Analog kann man für die anderen Gatter entlang des Pfades zum Ausgang o schließen, daß der Fehlereffekt im EXOR–Baum *stets* propagiert wird; unabhängig davon, welcher von den mit X markierten Schaltkreiseingängen auf 0 bzw. 1 gesetzt wird.
Unter Roth–Logik ist jedoch weder die Information 'nur noch 0 oder 1 möglich' noch 'entweder D oder $\overline{D}$' darstellbar. Zur Beschreibung nicht vollständig spezifizierter Zustände steht bei Roth–Logik einzig der *don't-care–Wert X* zur Verfügung. Folge: Außer Leitung s müssen *alle* anderen Leitungen der Schaltung mit X markiert werden.

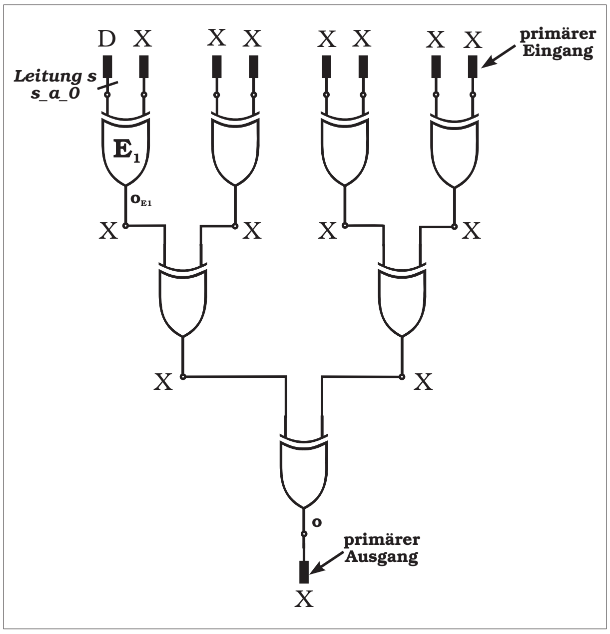


Abbildung 2.3: Keine Fehlerpropagation im EXOR–Baum bei Standarddarstellung

Diese Modellierungsungenauigkeit hat zur Folge, daß die Information über die bereits sichergestellte Fehlerpropagation verloren geht. Ein auf der Roth–Logik basierender Testmustergenerator ist dadurch gezwungen, *sämtliche* Eingänge der Baum–Schaltung auf feste Werte zu setzen, um die (in Wirklichkeit längst gesicherte) Propagation auch im Modell garantieren zu können — ein ineffizientes Vorgehen.
Abgesehen davon, daß die Laufzeit des Algorithmus unnötig erhöht wird, verschlechtert sich auch die Qualität der erzeugten Testmuster. Im Anschluß an die ATPG erfolgt nämlich häufig eine *Testmengenreduzierung,* bei der versucht wird, die berechneten Testmuster soweit wie möglich miteinander zu verschmelzen. Dies gelingt umso besser, je geringer die Anzahl der spezifizierten PIs je Testmuster ist.

Die hier genannten Nachteile legen die Verwendung einer präziseren Logik zur Modellierung des Berechnungszustands nahe.

2.2.2 Partielle Darstellung unter Akers–Logik

Ein vielversprechender Versuch in diese Richtung wurde 1976 von Sheldon B. Akers, Jr. ([Ake76]) unternommen. Seine 16–wertige Logik ist der 5–wertigen Logik von Roth in der Darstellungsgenauigkeit deutlich überlegen. Sie ist insbesondere in der Lage, die Fehlereffekt–Propagation im Beispiel aus Abbildung 2.3 richtig darzustellen.

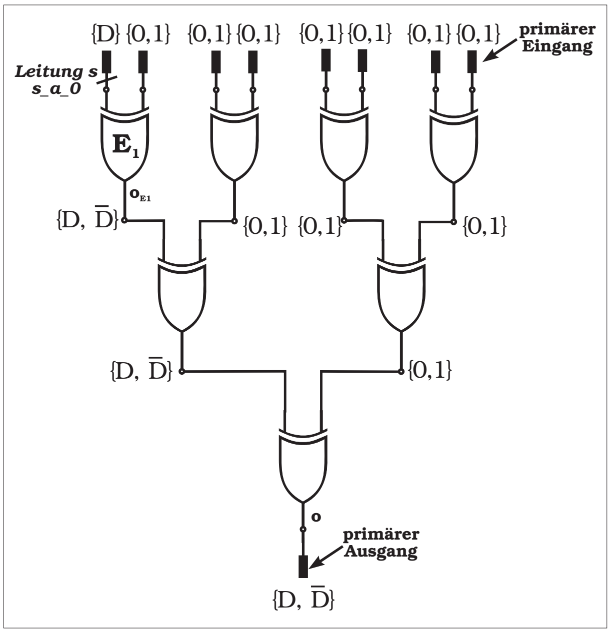


Abbildung 2.4: Propagation im EXOR–Baum in Akers–Darstellung

Der grundlegende Unterschied zwischen Roth– und Akers–Logik zeigt sich in der Behandlung von nicht vollständig spezifizierten Signalbelegungen. Während in Roth–Logik alle diese Signale mit X markiert werden, differenziert die Akers–Logik: Die entsprechende Leitung wird mit einer Werte*menge* beschriftet, die all jene Grundwerte aus $\{0, 1, D, \overline{D}\}$ enthält, die zum gegenwärtigen Zeitpunkt noch *nicht eindeutig ausgeschlossen* werden können.
Abbildung 2.4 zeigt dieselbe Berechnungssituation, wie sie in Abb. 2.3 in Roth–Logik dargestellt war, unter Akers–Logik. Die noch unspezifizierten aber fehlerunabhängigen Signale sind hier mit der Menge $\{0, 1\}$ markiert, sicher fehlerpropagierende Signale wie z.B. das an o_{E_1} hingegen mit $\{D, \overline{D}\}$. $\{0\}$ beschreibt unter Akers–Logik ein auf den logischen Wert *false* festgelegtes, $\{0, 1, D, \overline{D}\}$ ein völlig unspezifiziertes Signal. Faßt man alle auf diese Art bildbaren Teilmengen der logischen Grundmenge $\{0, 1, D, \overline{D}\}$ zusammen, so erhält man als Grundwertemenge zur Beschreibung eines Schaltkreiszustandes unter Akerslogik:

$$\begin{aligned} \boldsymbol{L_A} := \wp\{0, 1, D, \overline{D}\} = \{&\emptyset, \{0\}, \{1\}, \{D\}, \{\overline{D}\}, \{0, 1\}, \{0, D\}, \{0, \overline{D}\}, \{1, D\}, \\ &\{1, \overline{D}\}, \{D, \overline{D}\}, \{0, 1, D\}, \{0, 1, \overline{D}\}, \{0, D, \overline{D}\}, \{1, D, \overline{D}\}, \{0, 1, D, \overline{D}\}\}. \end{aligned}$$

Wie man in Abbildung 2.4 erkennen kann, ist die Akers–Logik im Gegensatz zum Ansatz von Roth in der Lage, die im EXOR–Baum stattfindende Fehlerpropagation korrekt darzustellen.

2.2.3 Vergleich von Roth– und Akers–Logik

Es ist sicherlich eine Tatsache, daß das Rechnen mit Akers–Logik einen höheren Berechnungsaufwand pro Gatter erfordert als unter Roth–Logik. Die Anzahl der Input–Belegungen an einem AND2–Gatter (d.h. der Einträge in seiner Logiktabelle) erhöht sich beispielsweise von $5^2 = 25$ unter Roth–Logik auf $16^2 = 256$ bei Akers. Auf den ersten Blick scheint es also, als würde man die größere Genauigkeit der 16–wertigen Logik mit einen merklichen Laufzeit–Overhead erkaufen.
Diesem scheinbaren Nachteil stehen aber die großen Vorteile gegenüber, die aus der exakteren Belegungsdarstellung resultieren. Die Schaltkreisbelegung ist nämlich von großer Bedeutung für den Testmustergenerierungsprozeß, denn sie dient dem *branch & bound*–Algorithmus als Entscheidungsbasis dafür, welches (PI/Wert)–Paar als nächstes zur Untersuchung ausgewählt wird.

Eine präzisere Belegungsdarstellung erlaubt daher eine bessere Auswahl im branch–Schritt und damit ein gezielteres Vorgehen des Algorithmus. Sie kann die Testsuche bzw. die Identifizierung von Redundanzen beschleunigen und obendrein zu qualitativ besseren Testmustern führen (wenn eine Input–Belegung eher als Test identifiziert werden kann; vgl. Abb. 2.4).

Der Testingenieur muß sich somit zwischen höherer Rechengeschwindigkeit je Gatter und einem ‘intelligenten’ Algorithmus entscheiden. Aufgrund der immer komplexer werdenden Problemstellungen im Bereich der ATPG scheint auf lange Sicht die zweite Alternative erfolgversprechender.

<table>
<tr><th>Akers–Logik</th><th>Roth–Logik</th><th>Akers–Logik</th><th>Roth–Logik</th></tr>
<tr><td>$\emptyset$</td><td>$\emptyset$</td><td>$\{1, D\}$</td><td rowspan="8">X</td></tr>
<tr><td>$\{0\}$</td><td>0</td><td>$\{1, \overline{D}\}$</td></tr>
<tr><td>$\{1\}$</td><td>1</td><td>$\{D, \overline{D}\}$</td></tr>
<tr><td>$\{D\}$</td><td>D</td><td>$\{0, 1, D\}$</td></tr>
<tr><td>$\{\overline{D}\}$</td><td>$\overline{D}$</td><td>$\{0, 1, \overline{D}\}$</td></tr>
<tr><td>$\{0, 1\}$</td><td rowspan="3">X</td><td>$\{0, D, \overline{D}\}$</td></tr>
<tr><td>$\{0, D\}$</td><td>$\{1, D, \overline{D}\}$</td></tr>
<tr><td>$\{0, \overline{D}\}$</td><td>$\{0, 1, D, \overline{D}\}$</td></tr>
</table>

Tabelle 2.1: Zusammenhang zwischen Signalmarkierungen in Roth– und Akers–Logik

Diese Überlegung motivierte unsere Entscheidung, die im Zusammenhang mit der Testmustergenerierung einige Zeit weniger beachtete Akers–Logik neu zu untersuchen. Dabei stießen wir auf eine Fülle neuer Heuristiken und Algorithmen, die nach Akers' ursprünglicher Veröffentlichung von 1976 entstanden waren und bisher noch nicht unter der 16–wertigen Logik eingesetzt worden waren. Es war ein wesentliches Ziel meiner Arbeit, die Testmustergenerierung unter Akers–Logik dem aktuellen Stand der Forschung anzupassen.

2.3 Belegung und Konsistenz

Die Verwendung der Akers–Logik macht bei einigen Begriffsbildungen der ATPG Modifizierungen nötig. Dies betrifft insbesondere die Definition der *Belegung* und der *Konsistenz*. Ich werde beide Begriffe daher für den Akers–Zusammenhang neu definieren. Der nun folgende Abschnitt mag vielleicht etwas technisch wirken; er wird mir aber die Ausführungen in den folgenden Kapiteln erleichtern.
Als erstes werde ich definieren, was ich von nun an unter einer Belegung eines Schaltkreises, eines Gatters bzw. eines Signals verstehe.

Definition 2.3.1:
Sei $C = (G, F)$ Schaltkreis mit $G = (V, E)$; $V = (Z \cup S)$; L Logik.

Eine Funktion $b : S \to L$ bezeichnet man als **Belegung des Schaltkreises C**.

Zu $s \in S$ heißt $b(s)$ die **aktuelle Belegung des Signals s.**

Die **aktuelle Belegung eines Gatters z** *($z \in Z$, $typ(z) = (ig, og, \tau)$) ist definiert als das ($ig + og$)–Tupel*

$$(b(in(z, 1)), ..., b(in(z, ig)), b(out(z, 1)), ...b(out(z, og))).$$

Abbildung 2.5 erläutert den Belegungsbegriff an mehreren Beispielen. Abb. 2.5 a) zeigt die Schaltkreisbelegung $b : S \to L_A$, die ich am Schaltungsgraphen $G = ((Z \cup S), E)$ dadurch visualisiert habe, daß jedes Signal $s \in S$ mit seinem Bild aus L_A markiert wird. Abb. 2.5 b) und c) zeigen in ähnlicher Weise eine Signal– bzw. eine Gatterbelegung.
In neuem Licht erscheinen unter 16–wertiger Logik die totalen und partiellen Schaltkreisbelegungen. Unter Roth–Logik waren partielle Beschreibungen daran zu erkennen, daß sie mindestens ein X enthielten (s.o.). Den Akers–Begriff für totale und partielle Belegungen erhält man, indem man dieses Roth–Kriterium anhand der Tabelle 2.1 auf die 16–wertige Logik überträgt:

Definition 2.3.2:
Eine Schaltkreisbelegung heißt **total**, *falls sämtliche Signalbelegungen* einelementig *sind, also*

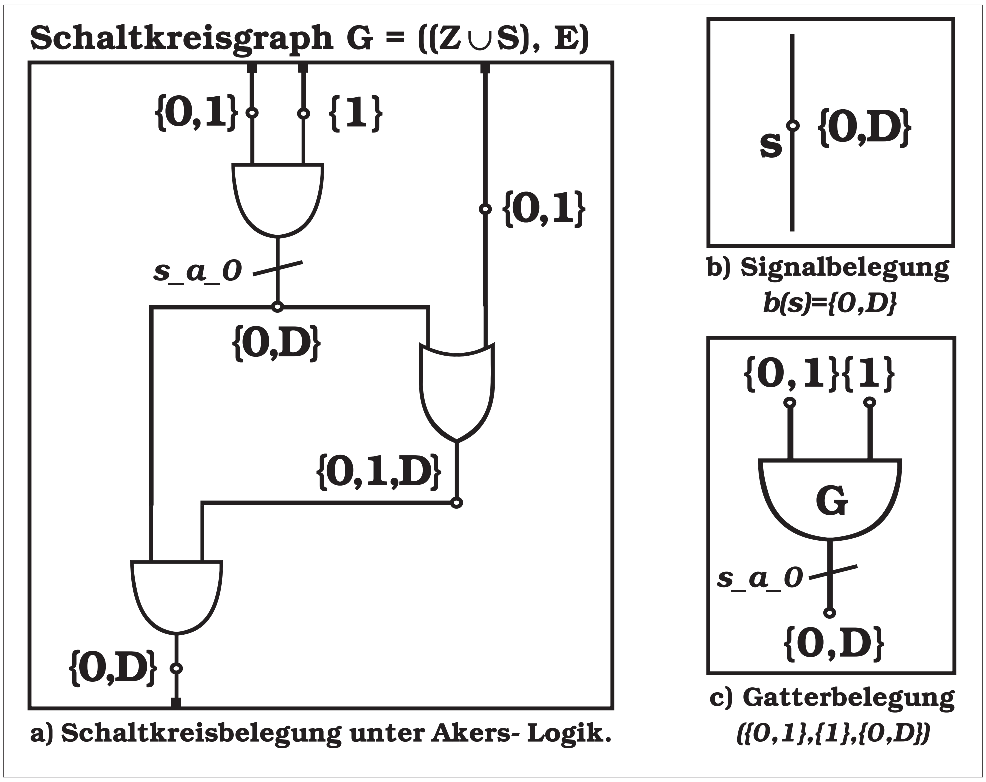


Abbildung 2.5: Beispiel für Schaltkreis–, Signal– und Gatterbelegung

$$\#(b(s)) = 1 \quad \forall\, s \in S.$$

Eine Schaltkreisbelegung heißt **partiell**, *falls* mindestens eine mehrelementige *Signalbelegung existiert, also*

$$\exists\, s \in S \;\; mit \;\; \#(b(s)) > 1.$$

Schaltkreisbelegungen ändern sich im Verlauf der automatischen Testmustergenerierung. Wird beispielsweise vom ATPG–Algorithmus ein PI auf einen festen Wert gesetzt, so werden dadurch Wertemengen eingeschränkt. Es entsteht eine neue Belegung, die der Ausgangsbelegung ähnelt, in der einige Signalmengen jetzt aber weniger Werte enthalten. Man könnte sagen, daß die erste Belegung die zweite 'umfaßt' (da die Zahl der *Freiheitsgrade,* d.h. die Auswahlmöglichkeiten in den Belegungswertemengen, bei ihr größer ist).

Es gibt also gewisse Ähnlichkeiten und Zusammenhänge zwischen den Schaltkreisbelegungen. Zwei besonders wichtige beschreibe ich formal so (vgl. auch Abbildung 2.6):

Definition 2.3.3:
Eine Schaltkreisbelegung b_1 heißt **Einschränkung** *einer (partiellen) Belegung b_2 (in Zeichen:* $\mathbf{b_1 \leq b_2}$*), falls gilt:*

$$\forall\, s \in S: \quad b_1(s) \subseteq b_2(s)$$

Eine Schaltkreisbelegung b_1 heißt **totale Einschränkung** *einer (partiellen) Belegung b_2 (in Zeichen:* $\mathbf{b_1 \dot{\leq} b_2}$*), falls $b_1 \leq b_2$ und b_1 total.*

Eine (partielle) Belegung b_p heißt **Vereinigung** *einer Menge von Belegungen $\{b_1, b_2, ..., b_n\}$ (in Zeichen: $\mathbf{b_p} := \mathcal{V}(\mathbf{b_1}, \mathbf{b_2}, ..., \mathbf{b_n})$), falls gilt:*

$$\forall\ s \in S: \quad b_p(s) = b_1(s) \cup b_2 \cup ... \cup b_n(s).$$

Die Begriffe der Einschränkung und Vereinigung einer Signal– bzw. Gatterbelegung ergeben sich analog[1].

Der Begriff der *Einschränkung* beschreibt den Übergang zu stärker spezifizierten Berechnungssituationen, wie es während der Testmustergenerierung z.B. durch Setzen neuer PIs auf feste Werte geschieht. Man kann gewisse Phasen der Testsuche daher auch als Folge von Belegungseinschränkungen interpretieren (→ Abschnitt 3.2). Der Begriff definiert implizit eine *partielle Ordnung* auf der Menge der Belegungen. Es gibt sowohl partielle als auch totale Einschränkungen (vgl. Abbildung 2.6).

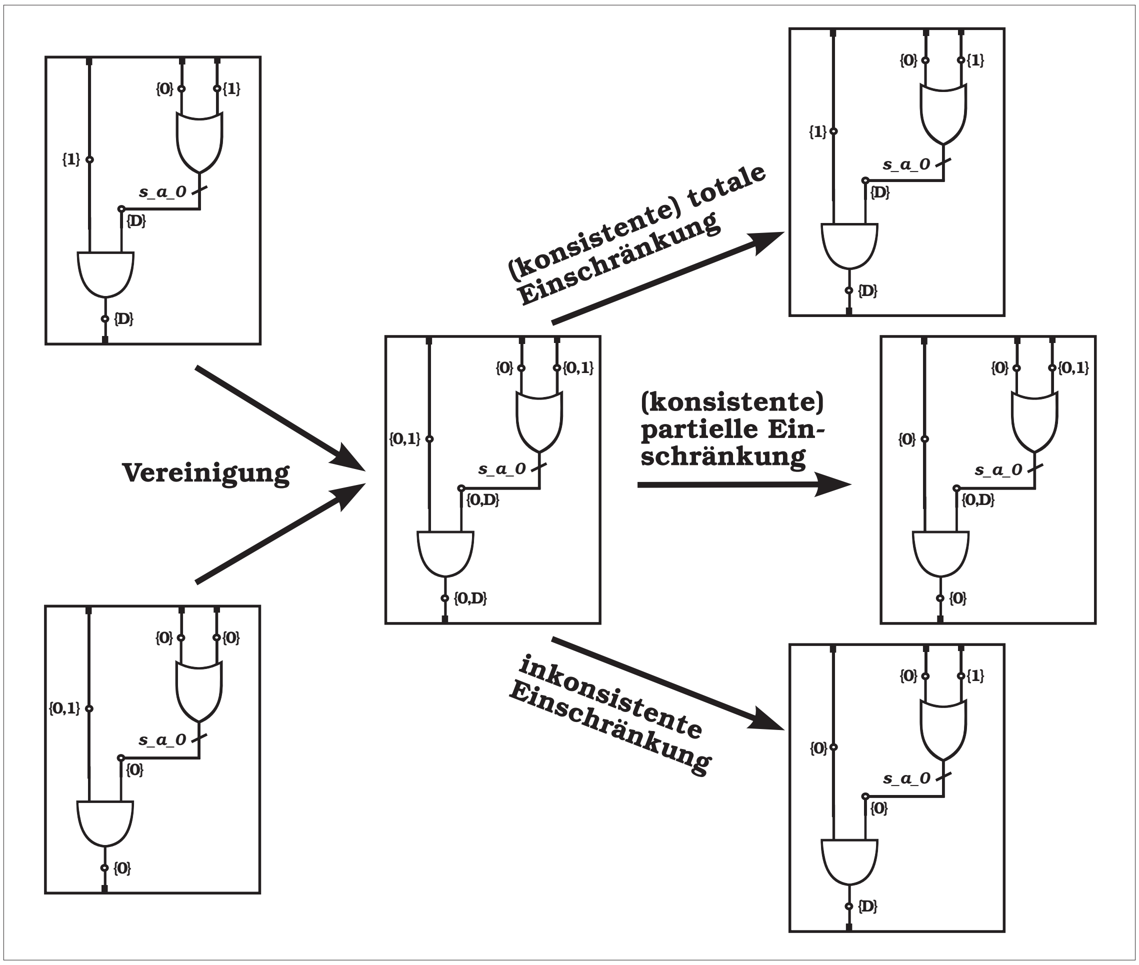


Abbildung 2.6: Einschränkung und Vereinigung

Der Begriff der *Vereinigung* beschreibt eine Zusammenfassung verschiedener Berechnungszustände zu einer neuen partiellen Beschreibung. Die resultierende Belegung kann man als Repräsentation aller ihrer Ausgangsbeschreibungen begreifen. Insbesondere kann man so jede partielle Belegung b_p als 'abkürzende Darstellung' aller ihrer totalen Unterbelegungen (aller b_t mit $b_t \dot{\leq} b_p$ gemäß Definition) interpretieren.
Umkehrbar ist der Vereinigungsprozeß übrigens nicht, zumindest nicht eindeutig: Wie Abbildung 2.6 zeigt, lassen sich aus der resultierenden Schaltkreisbeschreibung die ursprünglichen Belegungen nicht mehr ohne weiteres rekonstruieren.

[1] Bei *Signalbelegungen* gelten die beiden obigen Bedingungen für das betreffende Signal. Zur Definition für Gatterbelegungen ersetze man in der angegebenen Definition '$\forall\ s \in S$' durch '$\forall\ s \in \{s \in S \,|\, \exists e \in E$ mit $(Z(e) = z \wedge Q(e) = s) \vee (Q(e) = z \wedge Z(e) = s)\}$', wobei $z \in Z$ das betrachtete Gatter ist.

Abb. 2.6 zeigt auch, daß beim Einschränken Schaltkreisbelegungen entstehen können, denen keine realen Berechnungssituationen entsprechen. Die in der Abbildung als 'inkonsistente Einschränkung' bezeichnete Belegung ist zwar *formal* korrekt[2], sie beschreibt jedoch einen unzulässigen Schaltkreiszustand, da die durch sie gegebene Signalbelegung den Schaltfunktionen der angeschlossenen Gatter widerspricht[3].

Belegungen, die *sinnvolle* Schaltkreiszustände modellieren, bezeichnet man als *konsistent.* Dabei ist mit 'sinnvoll' gemeint, daß alle Basiszellen des Schaltkreises sich ihrer Schaltfunktion gemäß verhalten.

In diesem Zusammenhang eine kurze Bemerkung zur Anpassung des Begriffs der Schaltfunktion: τ war bisher zwar für binäre Logik definiert, nicht aber für Paralleldarstellungen von korrektem und fehlerhaftem Schaltkreis.
Zur Anpassung an den Belegungsbegriff genügt es aber, die übliche Schaltfunktion τ^{bin} auf beiden Schaltkreisen (dem korrekten und dem fehlerhaften) einzeln auswerten und die resultierenden Teilbelegungen wieder in die Paralleldarstellung überführen. Am Beispiel eines AND_2–Gatters sei illustriert, wie sich die Schaltfunktion τ der parallelen Darstellung aus der zugehörigen binären Funktion τ^{bin} ergibt:

$$\tau(1, D) = \tau(\mathbf{1}/1, \mathbf{1}/0) =$$
$$(\tau^{bin}(\mathbf{1}, \mathbf{1})/\tau^{bin}(1, 0)) = 1/0 = D.$$

Da ich mich im folgenden fast ausschließlich mit parallelen Schaltkreisbelegungen beschäftigen werde, wird mit der 'Schaltfunktion τ eines Gatters' von nun an stets die Funktion in Paralleldarstellung gemeint sein.

Nach diesem Exkurs über Schaltfunktionen werde ich nun den Begriff der *Konsistenz,* den ich oben als 'der Schaltfunktion gemäßes Verhalten' charakterisiert hatte, für die 16–wertige Logik formal definieren. Zur Erläuterung dient auch hier die Abbildung 2.6.

Definition 2.3.4:
Sei $C = (G, F)$ Schaltkreis mit $G = (V, E)$; $V = (Z \cup S)$; b Belegung von C.
Sei ferner $z \in Z$ mit $typ(z) = (ig, og, \tau)$.

- *Eine* totale Gatterbelegung $(b(in(z, 1)), ..., b(in(z, ig)), b(out(z, 1)), ...b(out(z, og)))$ *heißt* **konsistent**, *wenn gilt:*

$$\tau(b(in(z, 1)), ..., b(in(z, ig))) = b(out(z, 1)), ..., b(out(z, og)).$$

- *Eine* totale Schaltkreisbelegung b_t *heißt* **konsistent**, *falls $b_t(z)$ konsistent $\forall\ z \in Z$.*

- *Eine* partielle Schaltkreisbelegung b_p *heißt* **konsistent**, *falls mindestens eine ihrer totalen Einschränkungen $b_t \dot{\leq} b_p$ konsistent ist (für partielle Gatterbelegungen analog).*

Ich will den Begriff der Konsistenz noch ein wenig erweitern, indem ich eine Verbindung zwischen Belegungen und elementaren Zellenfehlern herstelle. Der Begriff der *f–Konsistenz* wird es mir erlauben, eine Charakterisierung der *testenden Belegung,* also einer Zielbelegung der ATPG, anzugeben.

Definition 2.3.5:
Sei C Schaltkreis, b Belegung von C und $f := (z_f, c_f)$ mit $c_f := (ib, ob)$ ein elementarer Zellenfehler. Die Belegung b heißt ***f*–konsistent,** *wenn gilt:*

- *die Gatterbelegung am fehlerhaften Gatter z_f ist gegeben durch den cep c_f : ib beschreibt die Inputbelegung von z_f und ob die Outputbelegung des fehlerhaften Schaltkreises. Die Outputbelegung des fehlerfreien Schaltkreis ist korrekt.*

- *b_t konsistent für mindestens ein $b_t \dot{\leq} b$.*

[2]Laut Definition wird nur gefordert, daß sich die Belegung durch eine Funktion $b : S \rightarrow L_A$ darstellen läßt.
[3]$AND_2(0,0) = 0 \neq D$, und die Gatterbelegung des OR_2–Gatters mit s_a_0–Fehler am Output müßte statt $(0,1,0)$ richtig lauten: $(0,1,D)$.

Eine Belegung b heißt **testende Belegung** *zu f, falls gilt:*

- *b ist f–konsistent und*
- *$\exists$ mindestens ein propagierender Pfad*[4] *von z_f zu einem PO.*

Der Begriff der Konsistenz hat für die automatische Testmustergenerierung zentrale Bedeutung, denn über einen Konsistenzcheck kann man entscheiden, ob ein Test gefunden ist (also eine *testende Belegung* vorliegt, s.o.), ob die Testsuche fortgesetzt werden muß oder ob es sich bei dem zu testenden Fehler um eine *Redundanz* handelt.

Umso bedauerlicher ist es daher, daß der obigen definierte Konsistenzbegriff praktisch nur von begrenztem Nutzen ist: Es ist nämlich ein NP–hartes Problem, für eine partielle Belegung zu entscheiden, ob sie konsistent ist oder nicht[5].
Um wenigstens *grundlegende* Konsistenzprüfungen durchführen zu können, behilft man sich mit einem abgeschwächten, aber in Linearzeit berechenbaren Konsistenzkriterium:

Definition 2.3.6:
Eine partielle Schaltkreisbelegung b heißt **lokal konsistent,** *falls $b(z)$ konsistent $\forall\ z \in Z$.*

Satz 2.3.1 *Jede konsistente Belegung ist auch lokal konsistent. Die Umkehrung gilt nicht.*

Beweis:
$\Rightarrow$: **klar nach Definition.**

$\not\Leftarrow$: **durch Gegenbeispiel.**

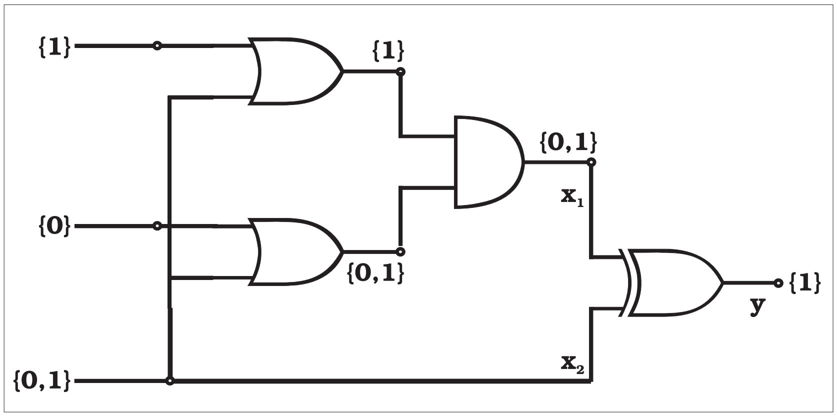


Abbildung 2.7: lokal, aber nicht global konsistente Belegung

Die Abbildung 2.7 zeigt eine lokal, aber nicht global konsistente Belegung. Obwohl an jedem einzelnen Gatter noch konsistente Einschränkungen der Gatterbelegung möglich sind, gibt es doch keine totale Schaltkreisbelegung mehr, die an *allen Gattern gleichzeitig* konsistent ist[6].

■

[4] Ein Pfad p im Schaltungsgraphen heißt *propagierend,* wenn an seinen Signalknoten nur noch fehlerabhängige Wertebelegungen möglich sind, also wenn $\forall s \in S,\ s$ auf $p : b(s) \subset \{D, \overline{D}\}$.

[5] Zum Beweis betrachte man einen beliebigen kombinatorischen Schaltkreis mit einem PO und eine partielle Belegung mit $b(pi) = \{0,1\}$ (unspezifiziert) $\forall$ primären Eingänge pi sowie $b(po) = 1$ für den primären Ausgang. Ein Nachweis der Konsistenz dieser Belegung entspräche der Lösung des Erfüllbarkeitsproblems.

[6] Die globale Inkonsistenz ergibt sich aus folgendem Zusammenhang zwischen den Signalleitungen x_1 und x_2: $x_1 = 0 \Rightarrow x_2 = 0$ und $x_1 = 1 \Rightarrow x_2 = 1$. Folglich gilt stets $x_1 = x_2$, während die '1' am Ausgang des EXOR $x_1 \neq x_2$ fordert.

Kapitel 3

Grundlagen der Automatischen Testmustergenerierung

3.1 Historischer Überblick

Die (durch immer größere Schaltungen ständig wachsende) Komplexität des Testproblems beschäftigt die Forschung seit den 60er Jahren. Inzwischen gibt es so viele Algorithmen zur automatischen Testmustergenerierung (vgl. [Roth66, Goel81, Fuji83, STS87, GMOD90, KP93, HWA95]), daß es sehr aufwendig wäre, hier eine vollständige Übersicht zu geben. Ich werde daher Schwerpunkte setzen und mich auf einige zentrale Punkte beschränken, die für meine Arbeit von besonderer Bedeutung sind. (Für ausführlichere Informationen zum Thema vgl. [Wun91, ABF90, Schu88]).

Als 'Urvater' aller modernen ATPG–Algorithmen gilt der *D–Algorithmus* von J. P. Roth ([Roth66]). Nicht nur die schon mehrfach erwähnte Logik, sondern auch die grundlegende Struktur des *branch & bound*–Algorithmus (s.u.) finden sich in Roths Arbeit. Anders als seine Nachfolger macht der D–Algorithmus während des Aufzählungsprozesses nicht nur feste Wertzuweisungen an PIs, sondern auch an beliebigen internen Leitungen. Die Laufzeit des Algorithmus war damit *worst case* exponentiell in der Anzahl der Schaltungssignale [Wun91].
Im Gegensatz zum D–Algorithmus werden im System *PODEM* von P. Goel nur PIs auf feste logische Werte gesetzt. Die Laufzeit ist deshalb nur noch exponentiell in der Anzahl der primären Eingänge. Wird an einer internen Leitung l ein bestimmter logischer Wert w gefordert, sind allerdings sogenannte *Backtrace*–Routinen (→ 3.3) nötig, um aus der Forderung an l eine logische Veränderung an einem PI zu folgern, die hilft, w möglichst schnell an l zu erzeugen. PODEM steuert den Backtrace heuristisch durch *Testbarkeitsmaße.*
Bei der Weiterentwicklung von PODEM war ein zentraler Aspekt die Verbesserung der *Implikation* (→ Abschnitt 4). Die Aufgabe der Implikation besteht darin, zu einer gegebenen Berechnungssituation eine Schaltkreisbelegung zu bestimmen, die diese Berechnungssituation möglichst genau beschreibt. Man unterscheidet dabei zwischen *direkten Implikationen,* die sich lokal aus der Funktion einzelner Gatter ergeben, und *indirekten Implikationen,* die weitergehende Schlüsse aus der globalen Schaltkreisstruktur ziehen.
Im System *FAN* von H. Fujiwara und T. Shimono ([Fuji83]) stellte die Ausnutzung von *Dominatoren* (→ Abschnitt 5.2) einen ersten Schritt zur Berechnung indirekter Implikationen dar. Dominatoren sind Kreuzungspunkte aller Pfade von der Fehlerstelle zu primären Ausgängen. An diesen *muß* ein Fehlereffekt beobachtbar sein, wenn es einen Test für den gerade betrachteten Fehler gibt.
Ein entscheidender Fortschritt bei den Implikationsalgorithmen war auch die Einführung des *Lernens;* entwickelt und im Programmsystem *SOKRATES* implementiert von M. Schulz et al. ([STS87, SA88]). Beim Lernen wird versucht, nichtlokale Zusammenhänge im Schaltkreis durch eine Vorberechnung zu erkennen und später bei der eigentlichen Berechnung auszunutzen (→ Kapitel 5.3).

Der in dieser Arbeit beschriebene Testmustergenerator COAT (*C*ell–*O*riented *A*kers–logic–based *T*est pattern generator) ist ein auf PODEM basierendes System mit Erweiterungen aus FAN und SOKRATES.

Die oben explizit genannten Verfahren wurden in COAT integriert und für eine Verwendung unter Akers–Logik angepaßt. Der genaueren Beschreibung des Systems und der Unterschiede zu seinen Vorgängern dienen die folgenden Kapitel.

3.2 Testmustergenerierung als Suchproblem

Bevor ich den Testmustergenerator COAT selbst vorstelle, möchte ich kurz auf eine Interpretation des Testproblems eingehen, die auf einen Vorschlag von Goel [Goel81] zurückgeht und das im folgenden beschriebene Vorgehen motivieren soll.

Nach Goel ist Testmustergenerierung im Grunde eine *binäre Suche in einem endlichen Suchraum.* Der *Suchraum* wird gebildet von der Menge aller Eingabemuster der Schaltung; *Ziel* der Suche ist es, möglichst schnell einen Test zu finden bzw. nachzuweisen, daß der Suchraum keinen Test enthält (→ *Redundanz;* vgl. Abschnitt 2.3). Das Durchsuchen übernimmt ein *branch & bound*–Verfahren, durch welches alle Eingabekombinationen implizit überprüft werden. Durch Belegen primärer Eingänge mit den logischen Werten 0 oder 1 *(branch)* und durch Verwerfen falscher Entscheidungen *(bound)* entsteht ein *Entscheidungsbaum (decision tree),* wie er in Abbildung 3.1 graphisch dargestellt ist.

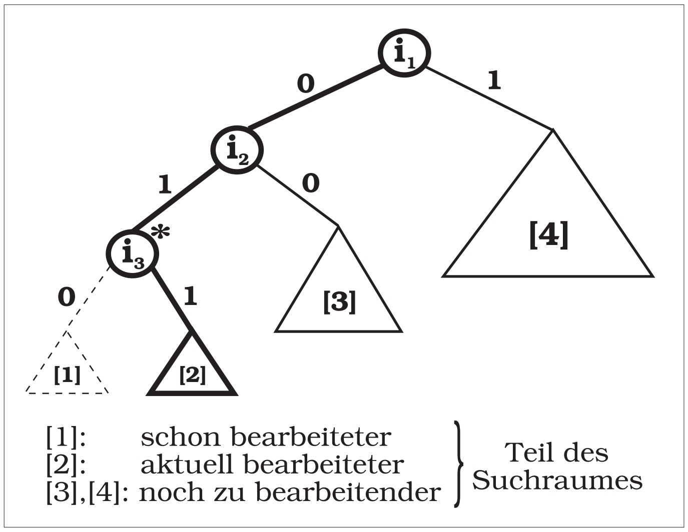


Abbildung 3.1: Beispiel eines Entscheidungsbaumes (decision tree)

Welche Eingabemuster zu den verschiedenen Teilbäumen korrespondieren, läßt sich an den Knoten– und Kantenmarkierungen entlang des Pfades von der Wurzel des Gesamtbaumes zu der Wurzel des betreffenden Unterbaumes ablesen. Der Unterbaum [2] in Abbildung 3.1 repräsentiert z.B. alle Eingabemuster mit der Eigenschaft:

$$\boxed{\begin{array}{c}\mathbf{i_1 = 0, i_2 = 1, i_3 = 1} \\ \textbf{(alle anderen PIs unspezifiziert)}\end{array}}$$

Durch den Entscheidungsbaum wird dargestellt, welche Teile des Suchraumes schon untersucht wurden (in Abbildung 3.1: Teilbaum [1]), wo aktuell gesucht wird (Teilbaum [2]) und welche Bereiche noch zu untersuchen sind (Bäume [3] und [4]). Die Suchreihenfolge ist durch den Baum vorgegeben: Wird z.B. beim Bearbeiten von [2] festgestellt, daß in diesem Teil des Suchraumes kein Test existiert, so wird die Suche in [3] fortgesetzt.

Die Struktur des Baumes (also die Reihenfolge, in der die PIs beim Aufbau desselben ausgewählt werden) ist natürlich entscheidend für die Effizienz des Verfahrens. Die Auswahl der primären Eingänge und der zugehörigen Wertebelegungen sollte daher möglichst *zielgerichtet* erfolgen (→ *Backtrace;* Abschnitt 3.3). Der aktuelle Suchraum selbst und die Information darüber, welche Teile des Gesamtraumes schon untersucht wurden bzw. noch zu untersuchen sind, wird auf einem *Stack* verwaltet, der immer den aktuellen Suchpfad im Binärbaum beschreibt. Ein Stackeintrag besteht jeweils aus gewähltem PI, aktuellem Wert an diesem PI und der Information, ob der komplementäre Wert am PI schon betrachtet wurde.

Beispiel:
Der Stack zum Entscheidungsbaum in Abbildung 3.1 würde wie folgt aussehen (der Pfad ist im Bild fett gedruckt):

$[\mathbf{i_3, 1, k\_b}]$	$[\mathbf{i_2, 1, k\_n\_b}]$	$[\mathbf{i_1, 0, k\_n\_b}]$	**(Stackende)**

Die oben verwendeten Abkürzungen 'k_b' bzw. 'k_n_b' entsprechen dabei Knotenmarkierungen im Suchbaum; graphisch werden diese üblicherweise so dargestellt, daß 'k_n_b'–Knoten mit einem '*' markiert werden (vgl. Abbildung 3.1). Die Kürzel(Markierungen) haben folgende Bedeutung:

k_n_b Komplement nicht bearbeitet (der andere Teilbaum dieses Knotens wurde noch nicht durchsucht)
k_b Komplement bearbeitet (der andere Teilbaum dieses Knotens wurde bereits durchsucht).

■

Die Knoten des Entscheidungsbaums und die Eingabemuster der untersuchten Schaltung entsprechen sich natürlich wechselseitig: Die Wurzel des Suchbaumes korrespondiert zur Menge *aller* möglichen Belegungen (Eingabemuster beliebig), die Blätter des Baumes entsprechen totalen Belegungen. Das Absteigen im Baum ist gleichbedeutend mit einer fortschreitenden Spezifikation der PI–Belegungen.

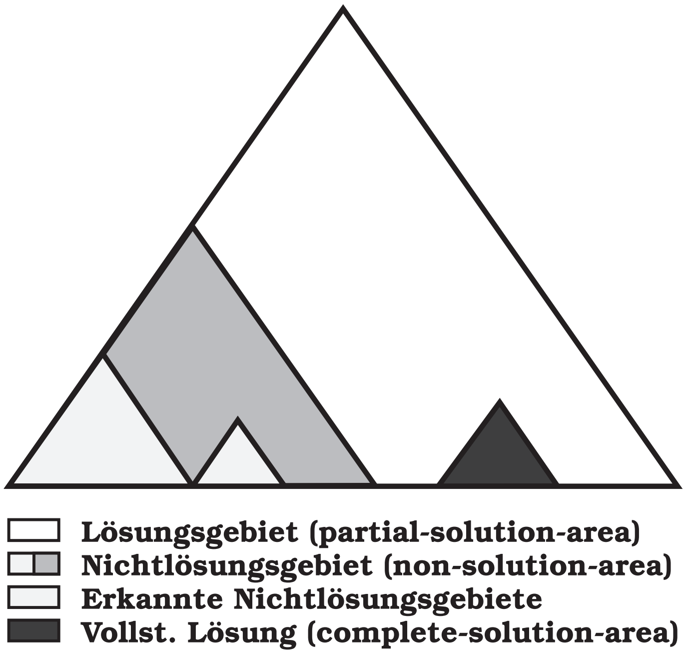


Abbildung 3.2: Aufteilung des Suchraumes in solution– & non–solution–areas

Genau so, wie man die Menge aller Schaltkreiseingaben in disjunkte Teilmengen *testender* und *fehlermaskierender* Eingaben aufteilen kann, lassen sich auch Teilbäume des decision tree klassifizieren [Wun91]. Teilbäume, in denen sich kein Test befindet, bezeichnet man als *Nichtlösungsgebiete* bzw. *non–solution–areas.* Alle anderen Teilbäume gehören zum *Lösungsgebiet (solution–area),* welches man in eine *complete–solution–area* und eine *partial–solution–area* unterteilen kann.
Ein Baum ist eine complete–solution–area, wenn alle seine Blätter Tests sind. Enthält er sowohl Tests als auch fehlermaskierende Muster, so liegt eine partial–solution–area vor.

Abbildung 3.2 veranschaulicht diese Begriffsbildungen. Dabei werden als *erkannte* non–solution–areas jene Unterbäume bezeichnet, bei denen der verwendete Testgenerierungsalgorithmus schon an der *Wurzel* feststellen kann, daß sie keinen Test enthalten.

Zwei Dinge sind für die Effizienz eines Testmustergenerators entscheidend:

1. Existiert ein Test im aktuellen Unterbaum, so sollte sich die Suche schnell auf diesen zubewegen (durch geeignete Wahl von PI und logischem Wert ($\rightarrow$ *branching*)
2. Ist der aktuelle Teilbaum eine non–solution–area, dann sollte dieser möglichst schnell erkannt und verlassen werden ($\rightarrow$ *bounding*).

Die praktische Bedeutung von Punkt 2 möchte ich kurz erläutern. — Zunächst ist natürlich alle Rechenzeit, die zur Untersuchung einer non–solution–area verbraucht wird, unproduktiv und damit Ressourcen–Verschwendung. Wenn sich der Eintritt in ein Nichtlösungsgebiet schon nicht vermeiden läßt, sollte wenigstens die zum Verlassen desselben *(bound–Schritt)* benötigte Laufzeit möglichst gering sein.
Steigt man in eine non–solution area hinab und spezifiziert n PIs, obwohl schon die PI–Entscheidung an der *Wurzel* des Unterbaumes einen Test unmöglich machte (denn gerade das zeichnet eine non–solution–area aus), so wird eine Aufzählung aller Wertekombinationen an den n PIs nötig. Muß diese Untersuchung *explizit* erfolgen — was in der Praxis keine Seltenheit ist — so ergibt sich (bei 2^n Kombinationsmöglichkeiten von n binären Variablen) ein exponentielles Verhalten.
Gerade beim Auffinden von Redundanzen kann dieser Laufzeit–Overhead dafür sorgen, daß die Gesamtdauer der Testgenerierung das praktisch vertretbare Maß überschreitet und die Testsuche abgebrochen werden muß.

3.3 Das System COAT

Der Saarbrücker Testmustergenerator COAT ist ein PODEM–basiertes System mit Erweiterungen aus FAN und SOKRATES. Er ist nicht auf elementare Grundzellen und das stuck_at–Fehlermodell festgelegt; stattdessen können auch komplexe Basiszellen verwendet und beliebiges (kombinatorisches) Fehlverhalten an diesen Zellen vom Benutzer selbst in einer Fehlerbibliothek beschrieben werden (→ Abschnitt 1.2.2).
Ferner benutzt das Programm statt der Roth– die exaktere Akers–Logik.
Ein Programmablaufplan des Testmustergenerators COAT befindet sich in Abbildung 3.3. Den darin beschriebenen Algorithmus möchte ich hier näher erläutern. Ich stelle die wesentlichen prozeduralen Komponenten vor und beschreibe ihr Zusammenwirken. Eine detaillierte Betrachtung der Unterprozeduren folgt dann ab Kapitel 4.

Grobablauf der Berechnung

Die Testmustergenerierung beginnt im System COAT mit dem *Einlesen eines Schaltkreises C* sowie einer zugehörigen *Fehlerliste.* In dieser sind alle sich aus dem verwendeten *Fehlermodell* (→ Abschnitt 1.2) ergebenden Defekte verzeichnet.
Aus dieser Fehlerliste wird ein elementarer Zellenfehler $f = (z_f, c_f)$ ausgewählt. Der cep $c_f := (ib, ob)$ beschreibt dabei ein Fehlverhalten am 'fehlerhaften Gatter' z_f (→ 1.2).

Es folgt die *Initialisierung des Schaltkreises.* Die partielle Ausgangsbelegung ist dabei abhängig von f: Als Startkonfiguration wird eine (partielle) Belegung gewählt, die alle Signalknoten unterhalb der Fehlerstelle mit $\{0, 1, D, \overline{D}\}$ initialisiert und alle anderen mit $\{0, 1\}$[1]. Die Belegung am fehlerhaften Gatter z_f ist durch den cep c_f gegeben: ib beschreibt die Inputbelegung an z_f und ob die Ausgangsbelegung im fehlerhaften Schaltkreis (die Outputbelegung im fehlerfreien Schaltkreis ergibt sich durch z_f's Gatterfunktion).

Als nächstes werden *Testbarkeitsmaße* berechnet (→ 3.3), die im branch–Schritt zu bestimmen helfen, welcher PI auf welchen Wert gesetzt werden soll. Sie beeinflussen so die Durchmusterung des Suchraumes.

Nach dieser Initialisierungsphase beginnt die eigentliche Berechnung. Die Suche nach einem Test geschieht durch systematisches Durchmustern des des Suchraumes mittels eine *branch & bound*–Algorithmus. Vor jedem branch–Schritt sind folgende vier Arbeitsschritte nötig:

1. *Implikation.* Jede Veränderung logischer Werte an einem Signalknoten kann Auswirkungen auf andere Signale des Schaltkreises haben. Die Aufgabe der Implikation ist es, aus der aktuellen Belegung der primären Pins und aus der Tatsache, daß das aktuelle Eingabemuster zu einem Test für f fortgesetzt werden soll, möglichst viele Folgerungen über die Belegung interner Leitungen zu ziehen. Eine Implikation wird nach jedem Entscheidungsschritt im Suchbaum durchgeführt (→ *Implikation,* Abschnitt 4).

[1] Damit sind insbesondere alle primären Eingänge logisch unspezifiziert (von Sonderfällen, z.B. Defekten an den PIs selbst, einmal abgesehen).

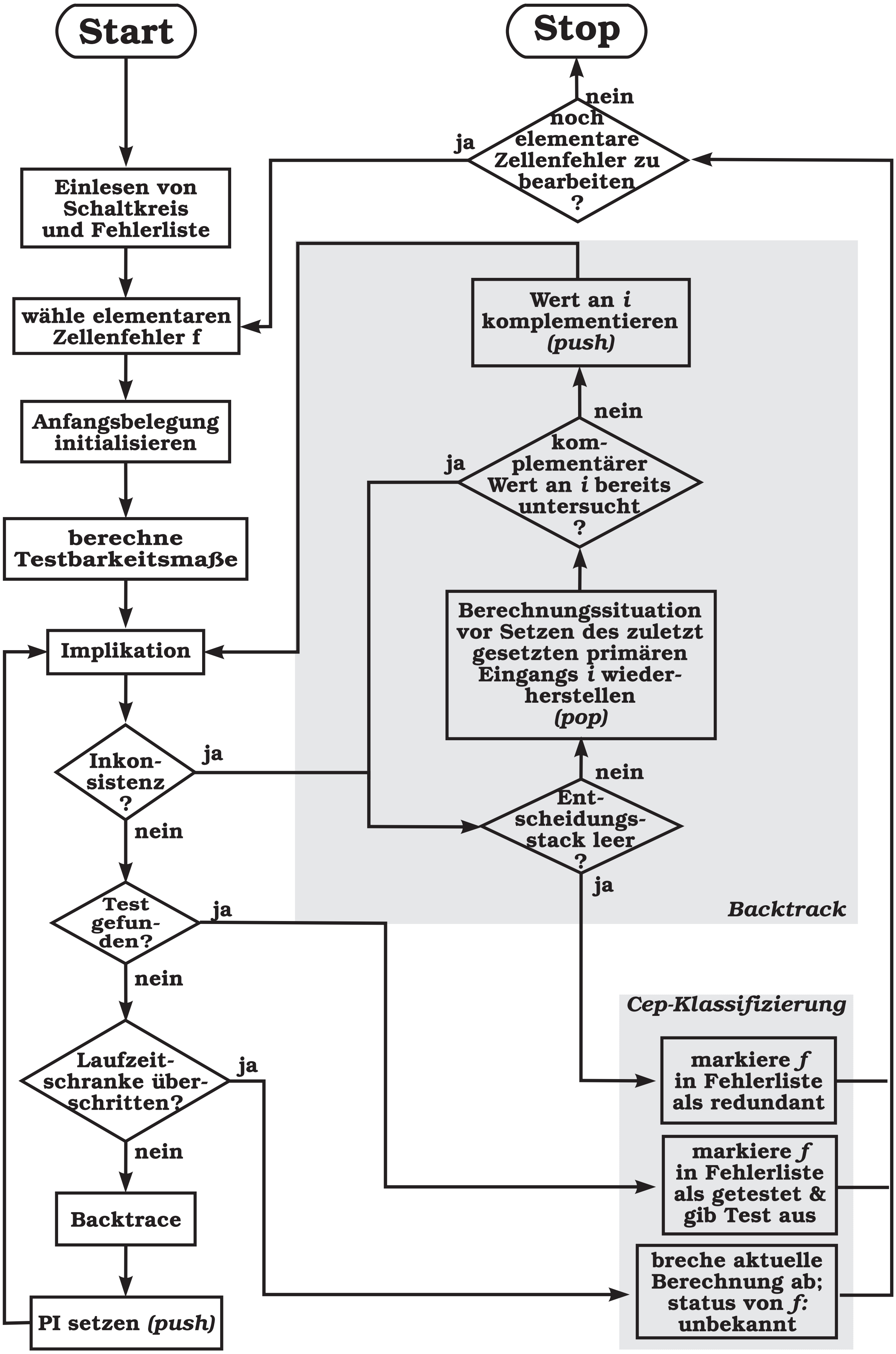


Abbildung 3.3: Flußdiagramm des Testmustergenerators COAT

2. *Inkonsistenz.* Die sich nach der Implikation ergebende Belegung ist darauf zu überprüfen, ob sie noch eine f–konsistente totale Einschränkung besitzt, ob sie also noch zu einem Test fortgesetzt werden kann. Ist dies nicht der Fall, muß ein *Backtrack* durchgeführt werden, um die non–solution–area, in die man geraten ist, schnellstmöglich zu verlassen (wie dies genau geschieht, wird weiter unten beschrieben).

3. *Test.* Eine Analyse zeigt, ob Aktivierung und Propagation des Fehlers f schon sichergestellt sind (d.h. ob alle dem aktuellen Knoten des Suchbaumes zugeordneten Eingabemuster die Kriterien für einen *Test* erfüllen). In diesem Fall ist das Testmuster (bzw. die Testmenge, falls es sich um einen inneren Baumknoten handelt) auszugeben, der cep entsprechend zu klassifizieren und die Testgenerierung, falls nötig, an einem anderen cep fortzusetzen.

4. *Laufzeitschranke.* Bei Überschreiten der schon erwähnten Laufzeitschranke, die den Berechnungsaufwand pro Fehler begrenzt (→ 1.3), ist die Testsuche ebenfalls abzubrechen. Der untersuchte cep bleibt in diesem Fall unklassifiziert, d.h. sein Status ist *unbekannt* (→ *Klassifizierung von elementaren Zellenfehlern;* Abschnitt 1.3). Die Berechnung wird am nächsten elementaren Zellenfehler der Fehlerliste fortgesetzt (sofern vorhanden).

Der Backtrace

Falls keiner der letzten drei Punkte erfüllt ist, so findet an dieser Stelle ein neuer *branch*–Schritt statt. Dazu muß zunächst ein geeigneter PI ($\hat{=}$ *Knoten* im Entscheidungsbaum) ausgewählt werden, der bis jetzt unspezifiziert ist und nun auf einen festen Wert gesetzt werden soll. Die Prozedur, die ein für die Testsuche möglichst sinnvolles PI/Wert–Paar auswählt, wird als *Backtrace* bezeichnet.

Nach welchen Kriterien geht der Backtrace dabei vor? — Ausgangspunkt ist stets eine *initial objective* genannte, sich aus dem Testalgorithmus ergebende Forderung an einer internen Leitung. Initial objectives können entweder der *Aktivierung* dienen *('um den Fehler zu aktivieren, muß Leitung l auf Wert w gesetzt werden')* oder aber eine *Propagation* des Fehlereffektes zum Ziel haben *('um den Fehlereffekt über Gatter z zu propagieren, muß dessen Pin* p_z *auf* w' *gesetzt werden'.*
Aus diesem initial objective entwickelt der Backtrace nun ein *final objective* der Form *'setze PI* i_x *auf Wert y'.* Zur Entwicklung des final aus dem initial objective bedient sich der Backtrace des auf der nächsten Seite beschriebenen Verfahrens:

/∗ initial objective := 'Signal s auf Wert w' ∗/
$z := source(s)$;
    /∗ sei z das s treibende Gatter ∗/
while ($typ(z) \neq PI$)
do
  suche 'wahrscheinlichste' Eingabebelegung $b_i(z)$ aus, welche Wert w an s erzeugt;
    /∗ Intuition: den einfachsten Weg zum Erreichen des Zieles wählen ∗/
  suche den Pin p von z, dessen Belegung $w_p := b_i(p)$ sich am *schwersten* erzeugen läßt;
    /∗ Intuition: das schwierigste Teilproblem als erstes lösen ∗/
  $s := p$;
  $z := source(p)$;
  $w := w_p$;
    /∗ new objective: 'Signal p auf Wert w_p' ∗/
od

Das Verfahren terminiert, wenn der Backtrace einen primären Eingang erreicht hat. Das Gatter z_p und der zugehörige Wert w_p bilden dann das *final objective.*

Testbarkeitsmaße

Zur Bewertung, welche Signal– bzw. Gatterbelegungen 'einfacher' zu erzeugen sind als andere, benötigt man sogenannte *Testbarkeitsmaße.* Die Testbarkeitsmaße in COAT basieren auf der *Schätzung von Signalwahrscheinlichkeiten.*

Solche Strategien sind zwar bekannt (vgl. dazu [BPH84]); allerdings waren auch hier einige Anpassungen an das erweiterte ATPG–Konzept von COAT nötig. Da die Testbarkeitsmaße für meine Ausführungen nur untergeordnete Bedeutung haben, werde ich dieses Thema hier nicht weiter vertiefen. Einen Überblick über verschiedene Testbarkeitsmaße und ihre Vor– und Nachteile findet man z.B. in [BPH84].

Bisher habe ich lediglich den Ablauf der ATPG beschrieben, solange keine Inkonsistenz auftritt d.h. die Berechnung nicht in eine (erkannte) non–solution–area gerät. Dieser noch fehlende Programmteil ist in Abbildung 3.3 im grau markierten und mit *Backtrack* gekennzeichneten Bereich beschrieben. Der nächste Abschnitt wird diese Skizze genauer erläutern.

Verhalten bei Inkonsistenz (der Backtrack)

Die folgende Situation macht einen Backtrack nötig: Eine non–solution–area wird erkannt; aus der aktuelle Belegung ist kein Test mehr ableitbar (d.h. unter seinen konsistenten totalen Einschränkungen befindet sich kein Test). Der Algorithmus muß den aktuellen Suchbereich verlassen und in einen noch nicht untersuchten Teilabschnitt wechseln[2]. Das weitere Vorgehen hängt von der Baumstruktur ab:

1. *Entscheidungsstack leer?* Der in Abschnitt 3.2 erwähnte *Entscheidungsstack* verwaltet den aktuellen Pfad im Suchbaum. Ist er leer, so ist der gesamte Suchraum erfolglos durchmustert worden und der Fehler ist *redundant.* Andernfalls ist das weitere Vorgehen wie folgt:

2. *Berechnungssituation wiederherstellen (pop).* Der oberste Eintrag $SE := [i_x, w, flag]$ des Entscheidungsstacks[3] wird gepopt. Desweiteren muß die Berechnungssituation wiederhergestellt werden, die vor dem Setzen von i_x auf w bestand. Alle aus der Wertzuweisung an i_x abgeleiteten Implikationen und alle Veränderungen an internen Datenstrukturen werden rückgängig gemacht. Dann wird der Entscheidungsstack verändert:

3. *Komplement untersucht?* Ist $flag = k\_b$ (Komplement bearbeitet), so bedeutet dies, daß *beide* Teilbäume von K erfolglos durchsucht wurden. K selbst wird damit zur Wurzel einer erkannten non–solution–area; eines vollständig durchsuchten Unterbaumes. Das Ausgangsproblem stellt sich damit — auf einer höheren Baumebene — erneut: Der Backtrack geht in die Rekursion.

4. *Wert komplementieren (push).* Ist der vom Backtrack betroffene Knoten K unmarkiert (*flag* auf *k_n_b:* Knoten nicht bearbeitet), so wurde der zweite Teilbaum von K noch nicht durchsucht. Um in diesen hinabzusteigen, muß der am zu K korrespondierenden PI anliegende Wert w durch sein Komplement ersetzt werden. Dazu wird der Stackeintrag SE wie folgt verändert: Der logische Wert w wird durch sein Komplement $\overline{w}$ ersetzt, und *flag* wechselt von *k_n_b* auf *k_b*. SE wird wieder gepusht und die Belegung des PI dementsprechend angepaßt.

Die vorgenommenen Änderungen am Schaltkreiszustand machen eine neue Implikation nötig. Die Berechnung kehrt in den in Abschnitt 3.3 beschriebenen Grobablauf zurück.

Erläuterndes Beispiel

Ein Fallbeispiel soll den Ablauf des Backtrack verdeutlichen. Ich lege meinen Ausführungen den Entscheidungsbaum in Abbildung 3.4 zugrunde und beschreibe, wie sich bei einem Backtrack der Entscheidungsstack verändert.

[2] Eigentlich handelt es sich bei dem nun folgenden Aufzählungsprozeß um nichts anderes als eine ausprogrammierte Rekursion. Die Verwaltung der aktuellen Baumposition kann entweder durch rekursive Programmierung oder explizit durch einen Stack verwaltet werden. Beschrieben wird hier die zweite Alternative.

[3] Dabei sei i_x der zuletzt gesetzte PI, w der zugehörige logische Wert und $flag$ die Markierung am korrespondierenden Knoten K im Entscheidungsbaum.

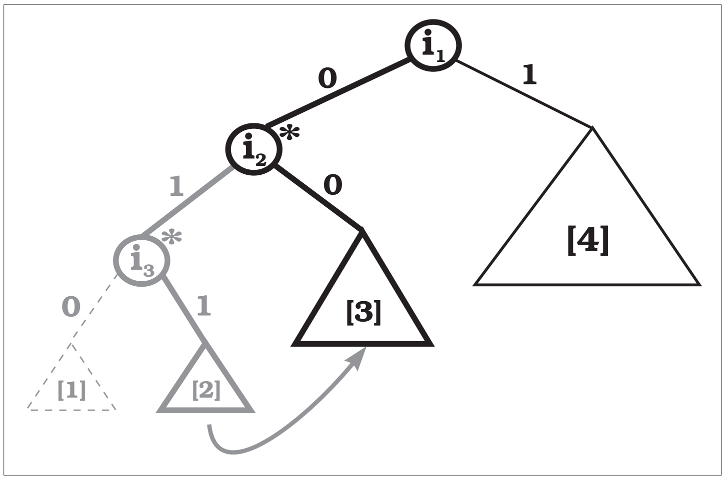


Abbildung 3.4: **Darstellung eines Backtrack auf dem Entscheidungsstack**

Übergang von Unterbaum [2] zu [3] Man betrachte Abbildung 3.4. Soeben sei Teilbaum [2] als non–solution–area identifiziert worden. Der nächste zu bearbeitende Suchraumabschnitt ist Unterbaum [3].
Die letzte Wertzuweisung vor Bearbeitung von [2] war $i_3 = 1$. Diese Zuweisung muß rückgängig gemacht werden. Da der zugehörige Baumknoten mit k_b $(*)$ markiert ist, wird rekursiv i_2 auf 0 gesetzt und der zugehörige Knoten markiert.
Der Stack ändert sich dabei wie folgt:

- Startzustand:

$[\mathbf{i_3}, \mathbf{1}, \mathbf{k\_b}]$	$[\mathbf{i_2}, \mathbf{1}, \mathbf{k\_n\_b}]$	$[\mathbf{i_1}, \mathbf{0}, \mathbf{k\_n\_b}]$	(**Stackende**)

- *pop* $[\mathbf{i_3}, \mathbf{1}, \mathbf{k\_b}]$ (Knoten markiert $\Rightarrow$ Rekursion)
- *pop* $[\mathbf{i_2}, \mathbf{1}, \mathbf{k\_n\_b}]$ (Knoten nicht markiert $\Rightarrow$ Modifikation)
- *push* $[\mathbf{i_2}, \mathbf{0}, \mathbf{k\_b}]$
- Endzustand:

$[\mathbf{i_2}, \mathbf{0}, \mathbf{k\_b}]$	$[\mathbf{i_1}, \mathbf{0}, \mathbf{k\_n\_b}]$	(**Stackende**)

Kapitel 4

Direkte Implikation

Im letzten Kapitel habe ich die Grundzüge des ATPG–Algorithmus im Überblick beschrieben. Nun will ich mir aus der Vielzahl der Unterprozeduren eine herausgreifen, um an dieser exemplarisch zu zeigen, wie sich die ATPG–Algorithmen und –Heuristiken unter dem verallgemeinerten Konzept von COAT verändert haben, welche Modifikationen ich vorgenommen habe und wie sich diese auf den Testmustergenerierungsalgorithmus auswirken. Ich habe zu diesem Zweck das Herzstück des COAT –Systems ausgewählt: die *Implikation.*

Aufgabe der Implikation ist es, aus lokalen Veränderungen der Schaltkreisbelegung (z.B. Wertemengeneinschränkungen durch das Setzen von PIs) Folgerungen über die Belegung an den anderen Signalknoten zu ziehen. Sie sorgt also dafür, daß die aktuelle Berechnungssituation möglichst exakt in einer partiellen Schaltkreisbelegung dargestellt wird.

Diese partiellen Belegungen spielen bei der ATPG eine zentrale Rolle. Sie bilden nicht nur die Entscheidungsgrundlage für den branch & bound–Algorithmus (also die Basis, auf der entschieden wird, welcher PI als nächstes auf welchen Wert gesetzt wird), sondern von ihnen (genauer: von ihrer *Konsistenz*) hängt es auch ab, wie schnell non–solution areas verlassen werden können. Die Frage, ob ein Test möglich ist oder evtl. eine Redundanz vorliegt, wird ebenfalls anhand der implizierten Belegungen entschieden.

Durch das verallgemeinerte ATPG–Konzept von COAT haben sich die Implikationsalgorithmen in zahlreichen Punkten verändert. In einigen Fällen wurde der Algorithmus durch die einfache Übertragung auf die 16–wertige Logik leistungsfähiger (z.B. die direkte Implikation $\rightarrow$ 4.4), in anderen erweiterte sich zusätzlich das Einsatzgebiet der Heuristiken (dies gilt insbesondere für die *Dominatorberechnung* $\rightarrow$ 5.2 und das *Lernen unter der Fehlerstelle* $\rightarrow$ 5.3.4).

Es gibt eine Unzahl verschiedener Implikationsverfahren. Die Verfahren, die ich bei der Betrachtung des Systems COAT vorstellen möchte, habe ich in zwei Gruppen unterteilt: in *direkte* und *indirekte* Implikationsverfahren.

Nach einigen theoretischen Vorbemerkungen werde ich mich zunächst mit den direkten Implikationsverfahren beschäftigen. Den indirekten Verfahren, die den direkten überlegen, dadurch aber auch komplexer sind, widme ich mich in Kapitel 5.

4.1 Theoretische Vorbemerkungen

Ich möchte meine Ausführungen beginnen mit der Betrachtung des folgenden, sich nach jeder Spezifizierung eines PI neu stellenden Problems:

Gegeben sei die aktuelle Inputbelegung der Schaltung sowie die Information, daß ein Test für den cep c_f gesucht wird. *Gesucht* wird eine möglichst genaue Aussage darüber, welche Belegungen an internen Schaltkreissignalen noch möglich sind.

Definition 4.1.1:
Sei $C = ((Z \cup S), E), F)$ Schaltkreis; $M_1, M_2, ..., M_n$ seien Mengen totaler Belegungen von C. Zu M_i sei $\chi_i : \{b \mid b : S \rightarrow L_A\} \rightarrow \{0,1\}$ die charakteristische Funktion, *d.h. $\chi_i = 1 \Leftrightarrow b \in M_i$; $\chi_i = 0$ sonst. Dann ist*

$$\boldsymbol{Imp}_C(\boldsymbol{\chi_1, \chi_2, ...\chi_n}) := \mathcal{V}(\bigcap_{i=1}^{n} M_i)) \text{ die } \textbf{(vollständige) Implikation} \text{ aus } \chi_1, \chi_2, ...\chi_n \text{ [1]}.$$

Für die weiteren Betrachtungen über die Implikation werden zwei charakteristische Funktionen (bzw. die von ihnen beschriebenen Mengen) besonders wichtig sein. Sei dazu b_a die aktuelle Schaltkreisbelegung von C und $f := (z_f, c_f)$ der momentan untersuchte Zellenfehler. Dann ist:

$$\chi_{\dot{\leq} b_a}(b) := \begin{cases} 1, \text{ falls } b \leq b_a;\ b \text{ total} \\ 0, \textit{ sonst} \end{cases} \qquad \chi_f(b) := \begin{cases} 1 \quad , & \text{falls } b \text{ total, f–konsistent und} \\ & \text{Test für Fehler } f = (z_f, c_f); \\ 0 \quad , & \text{sonst} \end{cases}$$

In $\chi_{\dot{\leq} b_a}$ und χ_f sind zwei wichtige Belegungseigenschaften codiert: $\chi_{\dot{\leq} b_a}$ beschreibt b_a und damit implizit die aktuelle PI–Belegung, während χ_f die Konsistenz zum betrachteten Fehler f formalisiert. Die Implikation $Imp_C(\chi_{\dot{\leq} b_a}, \chi_f)$ bildet gemäß obiger Definition also eine partielle Belegung, welche durch die Vereinigung aller totaler Belegungen, die sowohl eine Einschränkung von b_a als auch ein Test für f sind (also sowohl die aktuelle PI–Belegung als auch den zu testenden Fehler respektieren), entsteht.

Satz 4.1.1 *Das Problem der Berechnung von $Imp_C(\chi_{\dot{\leq} b_a}, \chi_f))$ ist NP–hart.*

Beweis:
Sei f_C die Schaltfunktion eines kombinatorischen Schaltkreises $C = ((Z \cup S), E), F)$ mit n primären Inputs, einem primären Ausgang o und der Schaltkreisfunktion τ. Sei f der stuck–at–0–Fehler an o. f ist genau dann testbar, wenn es ein Eingabemuster t gibt mit $\tau(t) = 1$. Oder, anders gesagt:

$$f \textit{ testbar} \Leftrightarrow \tau \textit{ ist erfüllbar} \qquad \textbf{(1)}$$

Man betrachte nun die Belegung b_a mit $\forall s \in S : \ b_a(s) := \{0,1\}$. Offensichtlich ist jede testende Belegung für f auch eine Einschränkung von b_a. Es gilt also:

$$Imp_C(\chi_{\dot{\leq} b_a}, \chi_f) := \mathcal{V}(M_{\dot{\leq} b_a} \cap M_f) = \mathcal{V}(M_f) =: Imp_C(\chi_f).$$

Die vollständige Implikation berechnet folglich die Vereinigung aller Testbelegungen. Für jede testende Belegung b_t von f muß aber gelten, daß $b_t(o) = \{1\}$. Damit gilt:

$$f \text{ testbar} \quad \Leftrightarrow \quad Imp_C(\chi_{\dot{\leq} b_a}, \chi_f)(o) = 1 \qquad \textbf{(2)}$$

Wegen (1) wäre dies aber zugleich eine Lösung des Erfüllbarkeitsproblems.

$\Rightarrow$ das Problem der vollständigen Implikation ist NP–hart. ■

4.2 Implikationsverfahren

Es ist damit gezeigt, daß vollständige Implikation sehr schwer ist; in der Praxis wird man sich mit Näherungslösungen behelfen müssen. Man kann die verschiedenen Implikationsverfahren in zwei Gruppen einteilen:

1) Direkte Implikation Die direkte Implikation betrachtet die Gatter der Schaltung isoliert, um durch diese Beschränkung Berechnungsaufwand je Rechenschritt gering zu halten. Konkret wird dabei an jedem Gatter versucht, möglichst viele Folgerungen aus seiner Gatterfunktion zu ziehen.

Eine Implikation heißt **direkt,** *wenn sie sich als eine Folge (lokaler) Gatterimplikationen darstellen läßt.*

[1] Zur Definition der Vereinigung von Belegungen '$\mathcal{V}$' vgl. Abschnitt 2.3.

2) Indirekte Implikation Indirekte Implikationsverfahren werden dort verwendet, wo das direkte Verfahren versagt. Sie ziehen Schlüsse aus globaleren Zusammenhängen, die sich nicht direkt aus den Logiktabellen der Gatter ergeben, sondern z.B. aus der Schaltkreisstruktur oder aus dem Wissen, daß die Propagation des Fehlereffektes sichergestellt werden muß.

Alle nicht direkten Implikationen heißen **indirekt**.

Zwei wichtige indirekte Verfahren, die in das System COAT integriert wurden, sind die sogenannte *Dominatorenberechnung* und das *Lernen.*

Ich werde nun einige Implikationsverfahren näher beschreiben und dabei schrittweise vom kleineren zum größeren Kontext übergehen: Ich beginne mit der direkten Implikation an einem einzelnen Gatter und beschreibe danach das direkte Vorgehen schaltkreisweit. Den Abschluß werden die indirekten Verfahren bilden.

4.3 Gatterimplikation

4.3.1 Historisches

Das 'natürliche' Implikationsverfahren in kombinatorischen Schaltkreisen scheint auf den ersten Blick die sogenannte *Vorwärtsimplikation* zu sein: Aus der Belegung der Gatterinputs ermittelt man die zulässigen Werte an den Gatteroutputs; Veränderungen des Schaltkreiszustands pflanzen sich so kontinuierlich von den primären Eingängen zu den primären Ausgängen fort. Reine Vorwärtsimplikation findet man beispielsweise im PODEM–Algorithmus von 1981.
Weniger offensichtlich ist, daß auch Schlußfolgerungen in umgekehrter Richtung möglich sind. Solche sogenannten *Rückwärtsimplikationen* können sich z.B. aus dem Einpflanzen des ceps an der Fehlerstelle ergeben.

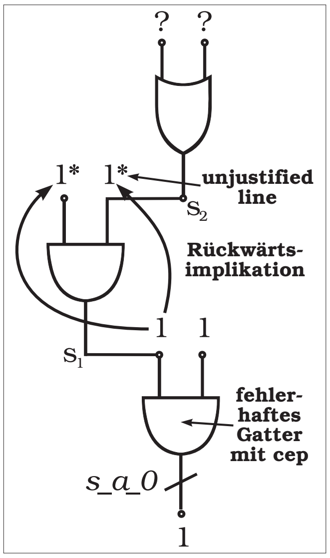


Abbildung 4.1: Rückwärtsimplikation

Solch ein Fall ist in Abbildung 4.1[2] skizziert: Der Fehler am unteren AND_2 ist nur durch einen einzigen cep testbar. Dieser cep fordert eine 1 an beiden Gattereingängen. Das Signal s_1 am linken Gatterinput wird selbst durch ein AND_2 getrieben. Da $b(s_1) = \{1\}$ eine Voraussetzung zum Generieren eines Tests mit dem gegebenen cep ist, müssen auch die Inputs des zweiten AND_2 auf 1 gesetzt werden. Diese Folgerung ist eine *Rückwärtsimplikation.*
Daß Rückwärtsimplikationen nicht in jedem Fall möglich sind, zeigt die Situation an Signal s_2 : Da es mehrere Möglichkeiten gibt, eine 1 am Ausgang des OR_2–Gatters zu erzeugen, sind keine eindeutigen Rückschlüsse auf die Inputbelegung des OR–Gatters möglich. Dies bedeutet aber, daß die Belegung an s_2 nicht unbedingt mit der Inputbelegung des OR–Gatters korresponiert: Wären seine beiden Inputpins mit $\{0,1\}$ belegt, so würde eine reine Vorwärtsimplikation an s_2 ebenfalls $\{0,1\}$, also eine Obermenge der tatsächlichen Belegung bestimmen.

Ein Signal, dessen Wertebelegung nicht durch die Inputbelegung seines treibenden Gatters gerechtfertigt ist, bezeichnet man als *unjustified line.* Unjustified lines kennzeichnet man üblicherweise mit einem '$*$'; ein Beispiel dafür ist die Leitung s_2 in Abbildung 4.1.
Während der Berechnung werden diese Signale in einer sogenannten *unjustified–liste* verwaltet. Ein Test ist erst dann gefunden, wenn keine unjustified lines mehr existieren.

Auch Gatterimplikationsroutinen, die sowohl Vorwärts– als auch Rückwärtsimplikation beherrschen, haben in der Regel gewisse Schwächen: Sie sind zum einen auf die Roth–Logik festgelegt und zum anderen oft nur für spezielle Gattertypen geeignet (wie z.B. AND, NAND, OR, NOR, INV; nicht aber für EXOR, MUX, FA oder HA).
Diese Einschränkungen gelten für den Testmustergenerator COAT nicht. Ich werde nun den Ablauf einer Gatterimplikation unter Akers–Logik beschreiben. Vorwärts– und Rückwärtsimplikation werde ich dabei

[2] Aus Gründen der Übersichtlichkeit werde ich in den Illustrationen Signalknoten nicht mehr mit ihren Wertemengen, sondern nur noch mit deren Elementen beschriften.

nicht voneinander trennen, sondern eine gemeinsame Darstellung für beide finden. Die Beschreibung ist auf beliebige Gattertypen anwendbar.

Zunächst jedoch ein kurzer Blick auf die Problemstellung. Die Wertebelegung an einem Gatter g wird durch zwei Faktoren bestimmt:

1. *Äußere Umstände.* Die Frage, ob das Gatter ober– oder unterhalb der Fehlerstelle liegt, ob die logische Fixierung irgendeines PI Auswirkungen auf seine Schaltmöglichkeiten hat und ähnliches bestimmen, welche Belegungen an g noch möglich sind. Diese Faktoren finden sich in der *partiellen Belegung* des Gatters vor Implikationsbeginn wieder.

2. Durch die *Schaltfunktion.* Diese (bzw. die durch sie induzierte *Logiktabelle)* legt für jeden Gattertyp fest, welche Schaltkonfigurationen an einem Gatter dieses Typs logisch korrekt *(konsistent)* sind.

Aus der Kombination von Schaltfunktion und gegebener Belegung ergibt sich die Zielbelegung der Implikation.

4.3.2 Ablauf einer Gatterimplikation

Formal kann man die Implikation an einem Gatter wie folgt beschreiben:

Definition 4.3.1:
Sei z Gatter und b_a^z eine partielle Gatterbelegung an z. Es sei $M_{\dot{\leq} b_a^z}$ die Menge aller totalen Einschränkungen von b_a^z und M_K die Menge aller an z konsistenten Belegungen. $\chi_{\dot{\leq} b_a^z}$ und χ_K seien die zugehörigen charakteristischen Funktionen. Dann ist

$$Imp_z(\chi_{\dot{\leq} b_a^z}, \chi_K) := \mathcal{V}(M_{\dot{\leq} b_a^z} \cap M_K))^3 \text{ \textit{die} \textbf{Gatterimplikation} \textit{aus} } \chi_{\dot{\leq} b_a^z} \text{ \textit{und} } \chi_K \text{ \textit{an Gatter} } z.$$

Abbildung 4.2 zeigt ein Beispiel für eine lokale Implikation an einem Gatter. Es sei angenommen, daß die Wertemenge am *select*–Eingang des MUX durch äußere Einflüsse (z.B. eine PI–Entscheidung) von $\{0, 1\}$ auf $\{0\}$ verkleinert wurde.

Um die Auswirkungen dieser Veränderung zu errechnen, bildet man zunächst die Menge aller totalen Einschränkungen der partiellen Ausgangsbelegung, indem man alle möglichen Kombinationen aus den Elementen der gegebenen Wertemengen bildet. Man erhält die Menge $M_{\dot{\leq} b_a^z}$, die in diesem Fall 6 Elemente umfaßt.
Von diesen sechs sind jedoch nur zwei *konsistent,* korrespondieren also zu einem korrekten Schaltzustand des Multiplexers. Diese beiden (in Abbildung 4.2 mit 'konsistent' markierten) Belegungen bilden die Elemente der Menge $M_{\dot{\leq} b_a^z} \cap M_K$. Aus der Vereinigung $\mathcal{V}(M_{\dot{\leq} b_a^z} \cap M_K))$ (vgl. Abschnitt 2.3) ergibt sich die resultierende partielle Gatterbelegung.

Das gerade beschriebene Verfahren ist für eine direkte algorithmische Umsetzung natürlich zu ineffizient. Bei der Realisierung von COAT wurde es daher ein wenig modifiziert, um den Programmablauf zu beschleunigen. Insbesondere erfolgt die Aufzählung der totalen Einschränkungen nicht explizit; stattdessen werden die neuen Wertebelegungen über vorberechnete pinspezifische Implikationsformeln bestimmt. Der Kürze halber will ich auf eine ausführlichere Beschreibung dieser Implementierungsdetails hier verzichten.

Das Ergebnis einer Gatterimplikation kann auch eine *Inkonsistenz* sein. Wäre beispielsweise durch eine Rückwärtsimplikation eine 0 am Ausgang des MUX in Abbildung 4.2 gefordert worden, wäre die resultierende Wertemenge dort *leer* gewesen: Die sich aus der Rückwärtsimplikation ergebende Forderung hätte der Eingangsbelegung des Gatters widersprochen. Das Auftreten einer Inkonsistenz entspricht der Identifizierung einer non–solution area und hat einen *Backtrack* zur Folge.

[3] Die hier gewählte Definition der Gatterimplikation ähnelt stark der Definition der vollständigen Implikation (s.o.). Im Unterschied zu dieser ist die Gatterimplikation *fehlerunabhängig.* Man kann sich bei der Gatterimplikation auf den fehlerfreien Fall beschränken, da am fehlerhaften Gatter nie impliziert wird (die Belegung am fehlerhaften Gatter ist durch den gewählten cep eindeutig gegeben, somit total und nicht weiter einschränkbar).

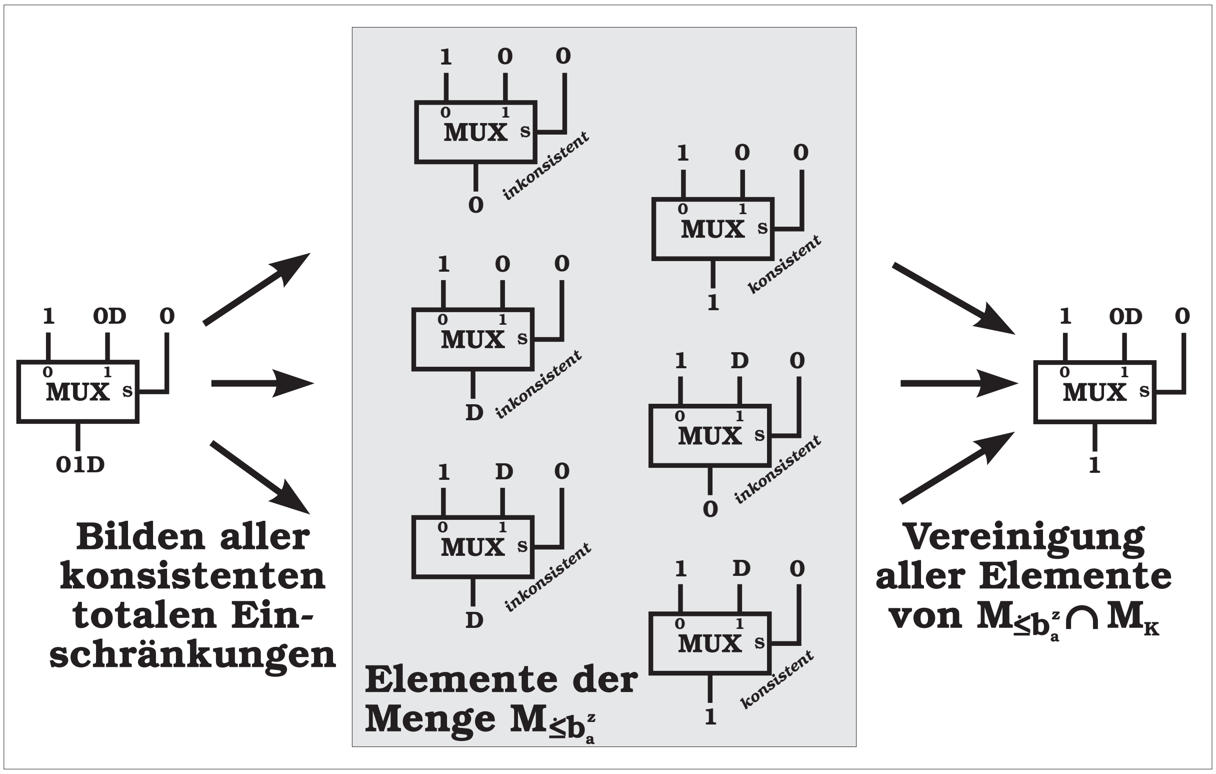
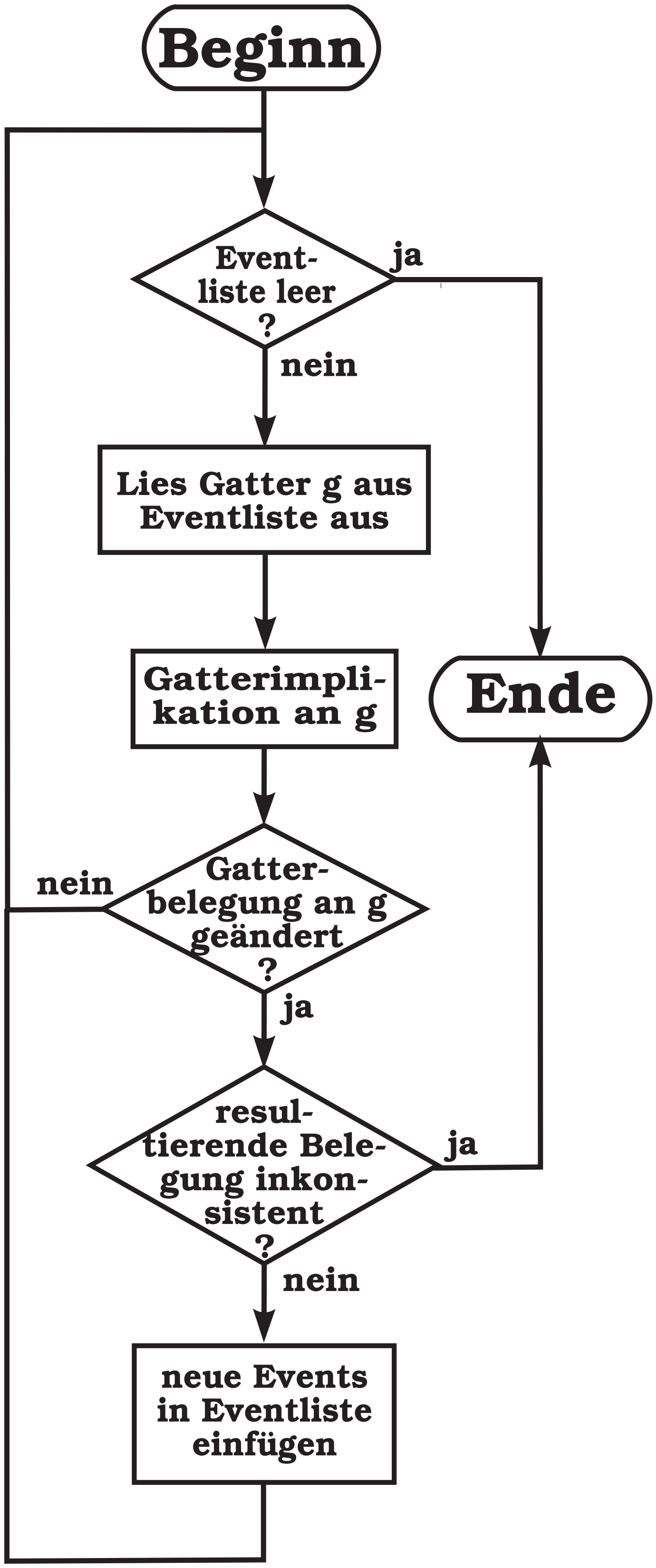


Abbildung 4.2: Beispiel einer Gatterimplikation

4.4 Schaltkreisweite Implikation

Verändert eine Gatterimplikation die Wertebelegung eines Signalknotens s, so wirkt sich dies zunächst auf jene Gatter aus, die direkt mit Signal s verbunden sind. Folglich ist es sinnvoll, auch *diese* einer Gatterimplikation zu unterziehen.

Die Hintereinanderausführung mehrerer Gatterimplikationen an allen Zellen, deren Belegung durch vorangegangene Implikationen geändert wurden, bezeichnet man als *(schaltkreisweite) direkte Implikation* (vgl. nächste Seite):



4.4.1 Schaltkreisweite direkte Implikation

Die direkte Implikation (graphisch dargestellt in nebenstehender Abbildung) verläuft folgendermaßen: Die Verwaltung der zur Implikation anstehenden Gatter übernimmt eine sogenannte *Eventliste.* Vor Beginn der Implikation befinden sich darin — als sogenannte *Startevents* — Zellen wie z.B. neu gesetzte PIs; die zum fehlerhaften Gatter adjazenten Zellen (nach Einpflanzen des ceps zu Beginn der Berechnung) oder *Dominatoren* (→ 5.2). Von dieser Ausgangssituation aus entwickelt sich die Implikation so:

1. *Eventliste leer?* Zu Beginn der Berechnung wird die Eventliste natürlich nicht leer sein, da ohne Startevents die Implikation gar nicht erst aufgerufen wird. Im weiteren Verlauf der Berechnung zeigt eine leere Eventliste aber an, daß alle durch direkte Implikation möglichen Schlußfolgerungen gezogen wurden. Weitere Zusammenhänge können — so sie existieren — höchstens noch von *globalen* Verfahren erkannt werden (→ Kapitel 5).

2. *Gatter auslesen.* Ist die Eventliste nicht leer, wird ihr das den Listenkopf bildende Gatter g entnommen.

3. *Gatterimplikation.* Für das Gatter g wird eine Gatterimplikation durchgeführt, wie sie in Abschnitt 4.3 beschrieben wurde.

4. *Belegung geändert?* Bei *unveränderter* Gatterbelegung ist die Berechnungssituation an g stabil — ein neuer Event kann bearbeitet werden. *Andernfalls* ist mindestens eine Wertemenge an den mit g verbundenen Signalknoten verkleinert worden. Zunächst muß überprüft werden, ob die

5. *resultierende Belegung inkonsistent* ist, ob also eine der veränderten Wertemengen jetzt *leer* ist. Falls ja, so ist auch die Menge der noch erlaubten Schaltkreisbelegungen leer, und es gibt keinen Test im aktuell bearbeiteten Teil des Suchraumes. Die Implikation kann in diesem Fall sofort abgebrochen und ein *Backtrack* durchgeführt werden.

6. *Gatter einfügen.* Wurde *keine* Inkonsistenz festgestellt, so wird die Eventliste um jene zu g adjazenten Gatter erweitert, deren Gatterbelegung sich durch Implikation an g geändert hat. Danach wird die Implikation am nächsten Element der Eventliste fortgesetzt.

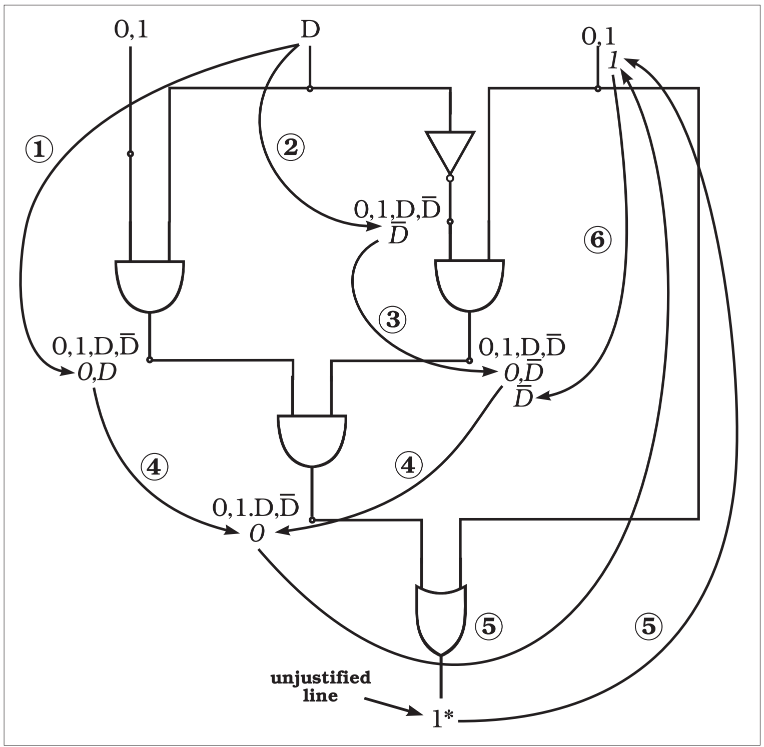


Abbildung 4.4: Beispiel einer Implikation

Beispiel:
Zur Verdeutlichung auch hier ein Beispiel. Im Teilschaltkreis in Abbildung 4.4 seien die Wertemengen am mittleren ‘Primäreingang’ von $\{0, 1, D, \overline{D}\}$ auf $\{D\}$ und die am ‘Primärausgang’ durch vorangegangene Implikationen[4] von $\{0, 1, D, \overline{D}\}$ auf $\{1\}$ eingeschränkt worden. Die Eventliste besteht daher zunächst nur aus diesen beiden Startevents. An den numerierten Pfeilen der Abbildung läßt sich nachverfolgen, wie sich die logischen Veränderungen über den Schaltkreis ausbreiten und an welchen Gattern eine lokale Implikation durchgeführt wird[5]. Dabei finden sowohl Vorwärts– als auch Rückwärtsimplikationen statt (Schritt *5* ist eine Rückwärtsimplikation). Die sich schrittweise verkleinernden Wertemengen an den Signalknoten sind in der Abbildung *untereinander* dargestellt. ■

Aus dem Beispiel ist ersichtlich, daß im Verlaufe einer lokalen Schaltkreisimplikation dasselbe Gatter *mehrfach* betrachtet werden kann (vgl. Schritt 3 und 6 in Abbildung 4.4). Trotzdem steht die Anzahl der Gatterimplikationen im Verlaufe einer direkten Implikation in einer linearen Abhängigkeit zur Schaltkreisgröße.

Zum Beweis vergegenwärtige man sich, daß nur dann ein Startevent für eine weitere Gatterimplikation entsteht, wenn die Belegungsmenge an mindestens einem Signal des Schaltkreises um mindestens ein Element verkleinert wird. Da jede dieser Mengen maximal 4 Elemente enthält — nämlich $0, 1, D$ und $\overline{D}$ — und jede Menge mindestens ein Element enthalten muß — weil die Implikation sonst wegen Inkonsistenz abgebrochen würde — kann die Anzahl der lokalen Gatterimplikationen maximal $3 \cdot \#S = O(\#S)$ betragen, wobei S die Menge der Signalknoten der Schaltung ist.

[4] Wie man sieht, ist die 1 am Ausgang des OR_2 anfangs noch nicht durch die Inputbelegung gerechtfertigt. Der zugehörige Signalknoten ist also eine *unjustified line.*
[5] In der Abbildung sind nur die Implikationen erfaßt, die tatsächlich zu einer Veränderung der Schaltkreisbelegung führen.

Durch die größere Auflösung der Akers–Logik sind Zusammenhänge lokal erkennbar, die die Roth–Logik bestenfalls durch globale Betrachtungen gefunden hätte. Man kann sich beispielsweise leicht überlegen, daß unter 5–wertiger Logik in Abbildung 4.4 (außer am Inverter) keine Schlußfolgerung möglich gewesen wäre.
Unter Akers–Logik hingegen sind zahlreiche, teilweise sogar eindeutige Einschränkungen möglich; eine unjustified line konnte gerechtfertigt *(justified)* werden, und auch die Belegung am rechten 'Eingang' der Teilschaltung konnte eindeutig bestimmt werden — alles Folgerungen, die für die Testmustergenerierung auf dem Gesamtschaltkreis wesentlich sein können.

Kapitel 5

Indirekte Implikation

5.1 Theoretische Vorbemerkungen

Das direkte Implikationsverfahren kann für bestimmte Klassen von Schaltkreisen durchaus ausreichend sein. So ist z.B. allgemein bekannt, daß die direkte Implikation auf korrekt arbeitenden Baumschaltkreisen vollständig ist [Fuji83].
Leider läßt sich diese Aussage nicht auf beliebige Schaltkreise verallgemeinern. Das folgende Beispiel aus [SA88] zeigt, daß es bei nicht baumartigen Schaltkreisen Zusammenhänge gibt, an denen direkte Implikationsverfahren versagen.

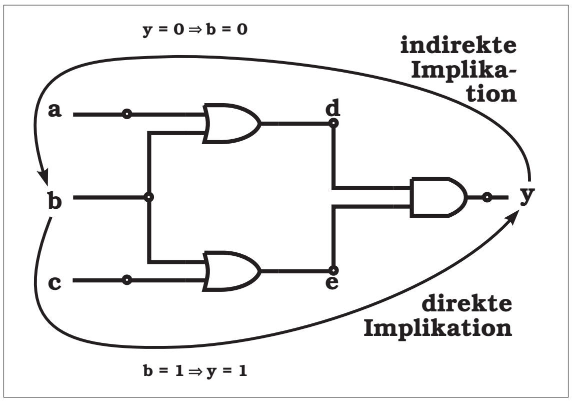


Abbildung 5.1: Indirekte Implikation

Beispiel:
Man betrachte die Schaltung C in Abbildung 5.1. Die von ihr berechnete Funktion ist $\tau_C : y := b \vee (a \wedge c)$.
Es besteht offensichtlich folgender logischer Zusammenhang:

$$b = 1 \Rightarrow y = 1 \qquad \textbf{(1)}$$

Wird also in einer Schaltkreisbelegung die Wertemenge an Signal b auf 1 eingeschränkt, kann man eine ebensolche Einschränkung an Leitung y implizieren. Diese Folgerung stellt für die Implikationsroutinen aus Abschnitt 4 keine Schwierigkeit dar; drei Gatterimplikationen genügen, um eine 1 an b nach y zu propagieren. Bei (1) handelt es sich folglich um eine *direkte Implikation.*

Anders die Situation im folgenden Fall. Nach dem *Gesetz der Umkehrung* $((A \Rightarrow B) \Leftrightarrow (\overline{B} \Rightarrow \overline{A}))$ ist folgende Aussage zwar zu (1) äquivalent:

$$y = 0 \Rightarrow b = 0 \qquad \textbf{(2)}$$

Direkte Implikationsverfahren sind jedoch leider nicht in der Lage, Zusammenhang (2) zu erkennen. Warum dies so ist, zeigt folgende nähere Betrachtung.

Zur Schaltung in Abbildung 5.1 sei folgende Startsituation gegeben: Die Belegung an y sei durch Rückwärtsimplikation auf $\{0\}$ eingeschränkt worden, alle anderen Signale seien mit $\{0,1\}$ markiert. Aufgrund von (2) wäre auch eine Einschränkung an Signal b auf $\{0\}$ möglich.
Kann dies die direkte Implikation aus Abschnitt 4.4 implizieren ? — Die Veränderung an y ist ihr *Startevent;* das direkte Implikationsverfahren beginnt also mit einer Gatterimplikation an dem dieses Signal treibenden AND_2. Aus $y = 0$ läßt sich direkt jedoch keine Veränderung an den Gatterinputs folgern. Da die Gatterbelegung des AND_2 unverändert bleibt, gibt es keine Events für weitere Gatterimplikationen an den beiden OR_2, und die direkte Implikation terminiert. Der Zusammenhang (2) bleibt unentdeckt. ■

Es gibt also nicht–direkte logische Zusammenhänge, die direkte Implikationsverfahren nicht (oder nicht vollständig) in Wertemengeneinschränkungen umsetzen können. Um Folgerungen dieser Art entdecken zu können, benötigt man spezielle, sogenannte *indirekte* Implikationsverfahren. In den folgenden Abschnitten werde ich zwei wichtige Vertreter dieser Gattung vorstellen.

5.2 Dominatoren

Ein wichtiges in Polynomialzeit berechenbares Verfahren zum Erkennen globaler Zusammenhänge ist die *Dominatorenberechnung.* Ich möchte den Begriff des Dominators, den ich in Abschnitt 3.1 schon kurz angesprochen habe, nun exakt definieren.
Ich unterscheide dabei zwei Typen von Dominatoren: *Statische* Dominatoren, die dem Dominatorbegriff aus FAN [Fuji83] entsprechen, und *dynamische* Dominatoren, die eine Verallgemeinerung des ersten Typs darstellen und in SOKRATES [SA88] verwendet werden.

Definition 5.2.1:
Sei $C = (G, F)$ mit $G = (Z \cup S)$ Schaltkreis, $f := (z_f, c_f)$ elementarer Zellenfehler an Zelle $z_f \in Z$, c_f cep zu f. Sei $b : S \to L_A$ die aktuelle Belegung.

- *Eine Zelle $d \in Z$ heißt* **(statischer) Dominator** *bezüglich f, wenn jeder Pfad von z_f zu einem primären Output d enthält.*
- *Ein Pfad in G von z_f zu einem Primären Ausgang heißt* **potentieller Propagationspfad,** *wenn für alle Signale s des Pfades gilt: $b(s) \cap \{D, \overline{D}\} \neq \emptyset$.*
- *Eine Zelle $d \in Z$ heißt* **(dynamischer) Dominator** *bezüglich f, wenn jeder* potentielle Propagationspfad *von z_f zu einem primären Output d enthält.*

Beispiel:
Ein Beispiel soll den Begriff des Dominators erläutern und den Unterschied zwischen statischen und dynamischen Dominatoren verdeutlichen.

Statischer Dominator Abbildung 5.2 a) zeigt einen *statischen* Dominator. Wie die Skizze zeigt, führen sämtliche Pfade vom fehlerhaften Gatter z_f zu den primären Ausgängen über die Zelle d. Der Verlauf dieser Pfade ist allein von der Schaltkreisstruktur abhängig — daher die Bezeichnung 'statisch'. Statische Dominatoren können schon vor Beginn der eigentlichen Testmustergenerierung durch eine Vorberechnung ermittelt werden.

Dynamischer Dominator Im Gegensatz zu den statischen sind *dynamische* Dominatoren von der aktuellen Berechnungssituation abhängig; so daß sie während der Testmustergenerierung bei fortschreitender Spezifizierung der Schaltkreisbelegung entstehen.

In Abbildung 5.2 b) ist das Gatter d kein statischer Dominator, denn der Pfad p führt vom fehlerhaften Gatter z_f zu einem PO, ohne d zu enthalten. Andererseits gilt aber für das auf p liegende Signal s, daß $b(s) \cap \{D, \overline{D}\} = \{0\} \cap \{D, \overline{D}\} = \emptyset$. Die Belegung von Signal s ist damit in jedem Fall fehlerunabhängig und p somit kein potentieller Propagationspfad mehr. Alle verbleibenden potentiell fehlerpropagierenden Pfade führen damit über d : d wird so zum *dynamischer Dominator.*

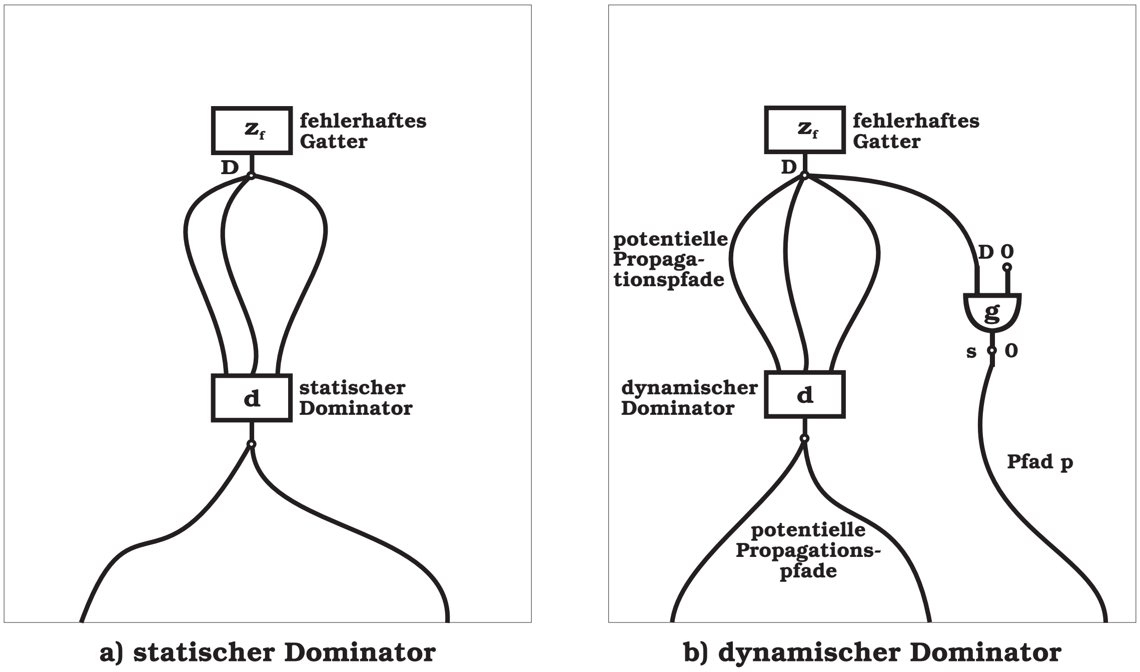


Abbildung 5.2: Dominatorvarianten

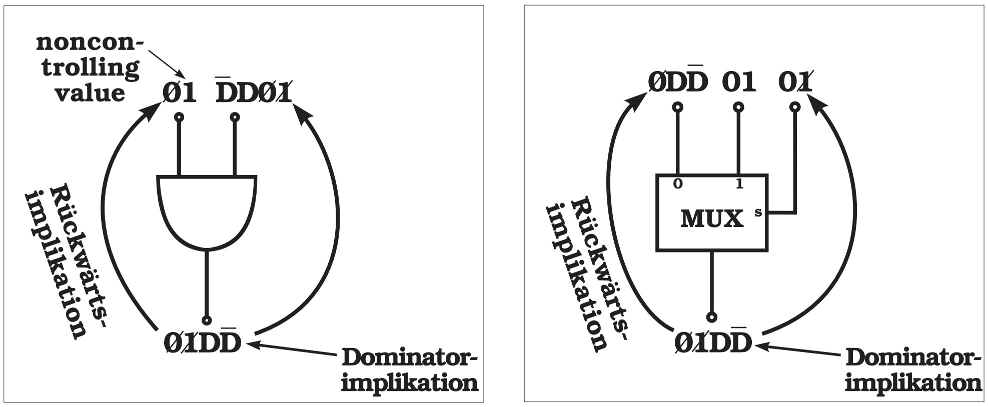
Ausnutzung von Dominatoren

Welche für direkte Implikationsroutinen nicht erkennbaren Schlußfolgerungen lassen sich nun aus der Existenz von Dominatoren ziehen ? — Dominatoren haben die besondere Eigenschaft, *sicher* Propagationspunkte zu sein, da sich alle potentiell fehlerpropagierenden Pfade an ihnen treffen.
Ein Fehler wird an einem Signal jedoch nur dann sicher propagiert, wenn fehlerunabhängige Signalwerte (also 0 und 1) dort ausgeschlossen werden können. Die Existenz eines Dominators erlaubt daher folgenden Schluß:

> *Die Wertemenge $b(d)$ am Gatteroutput eines Dominators d kann (sofern d nur einen Output besitzt) unabhängig von der Belegung seiner Inputs auf $b(d) \cap \{D, \overline{D}\}$ eingeschränkt werden. Der Output von d wird durch diese Einschränkung (falls es sich um eine echte Einschränkung handelt) zu einer unjustified line.*

Für multi–output–gates, bei denen der Fehlereffekt über mehrere Ausgänge propagiert werden kann, gilt diese Aussage nicht, weil die Propagation dann an keinem dieser Outputs unmittelbar zwingend ist.

Die Einschränkung der aktuellen Belegung am Ausgang eines Dominators wird in der Regel direkte Implikationen nach sich ziehen, wie die folgenden Beispiele beweisen.
Betrachte Abbildung 5.3. Sowohl das AND_2 in Bild a) als auch der Multiplexer in b) seien als Dominatoren identifiziert worden. Die Wertemengen, mit denen die Signalknoten in der Abbildung markiert sind, geben zunächst den Zustand vor dem Erkennen der Dominatoreigenschaft wieder. Die durchgestrichenen Werte verdeutlichen, welche Einschränkungen die Entdeckung des Dominators ermöglicht.
Das Bild zeigt zwei verschiedene Arten von Einschränkungen: Die Einschränkung der Wertemenge am Ausgang des Dominators auf $\{D, \overline{D}\}$ folgt direkt aus dem oben genannten Kriterium und wird in der Abbildung als *Dominatorimplikation* bezeichnet. Alle anderen Zustandsveränderungen sind daraus durch *Rückwärtsimplikation* lokal ableitbare Folgerungen.
In Abbildung 5.3 a) folgert die Rückwärtsimplikation aus der Dominatoreigenschaft, daß auch der rechte Input des AND_2 sicher fehlerpropagierend sein muß, während der linke Input auf den sogenannten *non-controlling value* festgelegt werden kann, der ein Weiterleiten des am anderen Pin anliegenden logischen Wertes ermöglicht (bei AND–Gattern ist dies die 1, bei OR–Gattern beispielsweise die 0).



a) Standardzelle b) komplexere Basiszelle

Abbildung 5.3: **Rückwärtsimplikation an Dominatoren**

Beim Multiplexer in Abbildung 5.3 b) folgen analog Werteeinschränkungen am linken Dateneingang des MUX und am *select*–Signal.

Dominatoren unter Roth– und Akers–Logik

Was sich unter Akers–Logik so einfach und natürlich ergibt, ist unter Roth–Logik wesentlich schwieriger darzustellen. Zwar gibt es den Dominatorenbegriff auch dort; das Ausnutzen desselben ist jedoch komplizierter und zudem mit höherem Programmieraufwand verbunden.
Das Akers–Kriterium (Einschränken der Wertemenge am Dominatorausgang auf $\{D, \overline{D}\}$) ist unter Roth–Logik nicht möglich: Mehrelementige Wertemengen wie $\{D, \overline{D}\}$ lassen sich lediglich als X darstellen, wodurch die entscheidende Information (zwingende Fehlerpropagation; an diesem Signal keine fehlerunabhängigen logischen Werte mehr möglich) verloren geht.

Roth–Testmustergeneratoren müssen daher andere Wege suchen, um die Dominatoreigenschaft auszunutzen. Verwendet wird üblicherweise das folgende Kriterium:

Fehlerunabhängige Inputs eines Dominators d können auf den noncontrolling value (z.B. 1 bei AND/NAND, 0 bei OR/NOR) eingeschränkt werden, sofern ein solcher für typ(d) existiert.

An diesem Kriterium fallen zwei Dinge besonders auf.

Erstens ist das Kriterium auf spezielle Gattertypen zugeschnitten. Noncontrolling values gibt es zwar bei den AND/NAND, OR/NOR–Standardgattern; für komplexe Gattertypen wie den MUX (vgl. 5.3 b) hingegen existieren sie nicht. Folge: Unter Roth–Logik muß für *jeden* komplexen Datentyp eine eigene Dominator–Routine geschrieben werden.
Anders bei Verwendung der 16–wertigen Logik (s.o.): Das Akers–Kriterium betrifft direkt nur die Outputbelegung des Dominators; alle weiteren Folgerungen ergeben sich automatisch und unmittelbar aus der Gatterimplikation. Gattertypabhängige Zusatzuntersuchungen sind nicht nötig.

Der zweite Punkt betrifft die Fehlerabängigkeit von Signalen. Ein Beispiel: Um den select–Eingang des MUX in Abbildung 5.3 b) fehlerpropagierend schalten zu können, muß bekannt sein, welcher der beiden Dateninputs den Fehlereffekt transportiert. Da die Roth–Logik aber sowohl $\{0, D, \overline{D}\}$ als auch $\{0, 1\}$ als X codiert, läßt sich das bei ihr nicht ohne weiteres feststellen. Auch hier braucht man daher eine Zusatzroutine (einen sogenannten *x_path_test*), um die fehlerabhängigen Pfade von den –unabhängigen zu unterscheiden.
Unter 16–wertiger Logik stellt sich dieses Problem, wie die Rückwärtsimplikation in Abbildung 5.3 b) zeigt, nicht.

Fazit: Zwar ist das Roth–Kriterium zur Auswertung von Dominatoren für gewisse Situationen geeignet (vgl. Abbildung 5.3 a); komplexere Fälle können jedoch nur mit einem erheblichen Zusatzaufwand bewältigt werden. Das in COAT verwendete Kriterium hingegen ist für beide in Abb. 5.3 dargestellten Situationen gleichermaßen geeignet, einfach zu handhaben und bietet zudem die konzeptionell schönere Darstellung.

5.3 Lernen

Das indirekte Implikationsverfahren, das ich jetzt beschreiben werde, verdankt seine Entstehung einer Beobachtung aus der Praxis. Bei der Untersuchung redundanter, also nicht testbarer Fehler (→ Abschnitt 1.3) stellte man fest, daß die Ursachen für Redundanzen häufig *lokaler* Art waren: Zu ihrer Auffindung reichte oft schon die Betrachtung eines *Teilschaltkreises* aus.
Genauere Untersuchungen zeigten, daß derselbe Effekt bei indirekten Zusammenhängen beobachtbar ist: Auch zur Erkennung solcher Abhängigkeiten genügt oft die Untersuchung eines kleinen Teiles des Gesamtschaltkreises. Aus Effizienzgründen war es daher naheliegend, die Suche nach indirekten Zusammenhängen, wenn möglich, auf Teilschaltkreise zu konzentrieren.
Doch wie untersucht man Teilschaltungen, die mitten im Schaltungsgraphen liegen? Die Belegung innerer Signalknoten läßt sich im 'normalen' Testablauf nur schwer kontrollieren; der Aufzählungsprozeß des *branch & bound*–Algorithmus beschränkt sich auf PIs, und Signale in Teilschaltkreisen lassen sich nur indirekt beeinflussen.

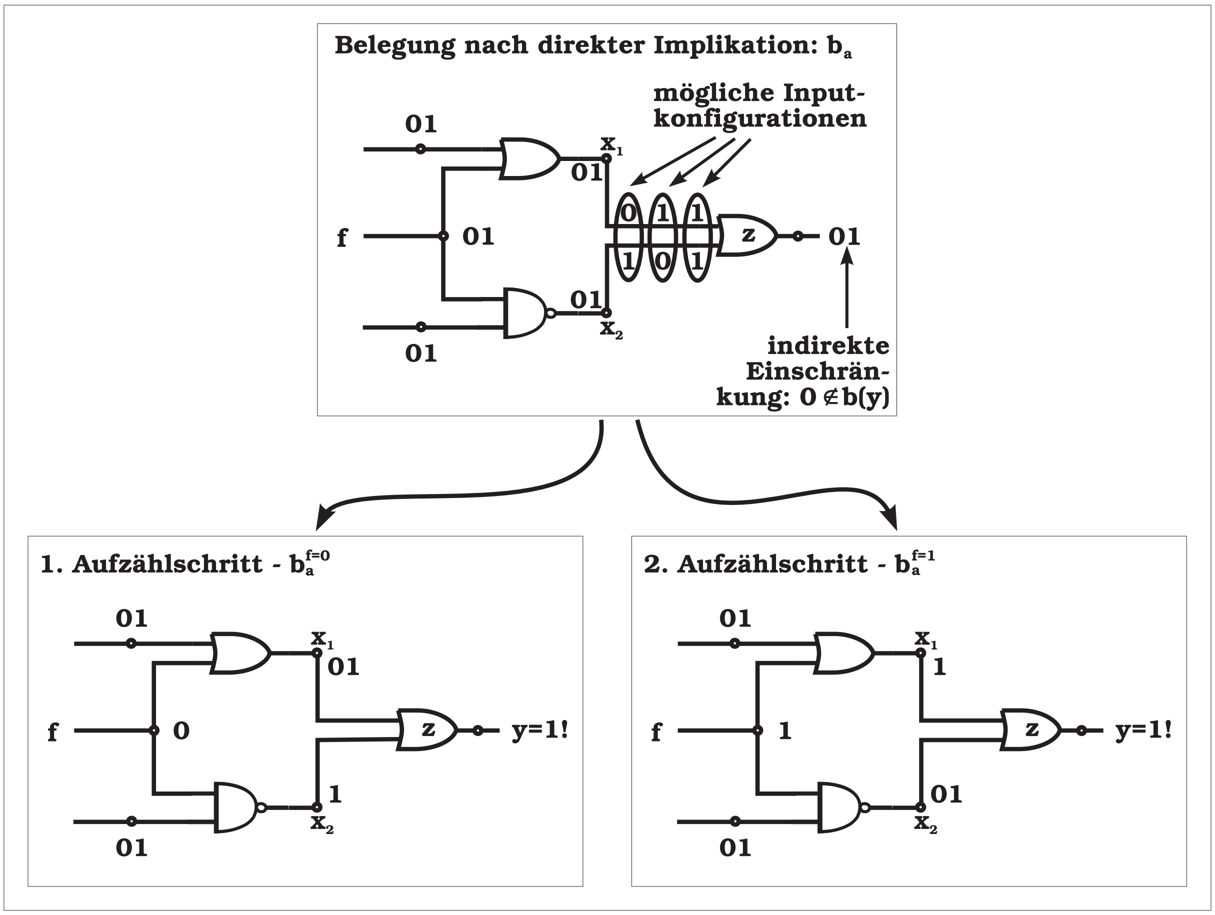


Abbildung 5.4: Indirekte Einschränkung

Aus diesem Problem heraus entstand die Idee, zur Auffindung indirekter Zusammenhänge den Aufzählungsprozeß zu *lokalisieren;* das heißt: ihn auf interne Schaltsignale auszuweiten. M. Schulz prägte für Algorithmen, die auf dieser Idee basieren, den Begriff *Lernen* [STS87].
In den folgenden Kapiteln werde ich diese Technik genauer beschreiben. Meine Beschreibung wird sich etwas von Schulz' Originalarbeit unterscheiden, da ich hier eine auf die Akers–Logik zugeschnittene Form

des Lernens darstellen werde. Neu ist zudem das *Lernen unter der Fehlerstelle* ($\rightarrow$ 5.3.4).

5.3.1 Lernen indirekter Einschränkungen

Ich beginne meine Ausführungen über das Lernen mit der Betrachtung *indirekter Einschränkungen.* Ich will diesen Begriff an einem Beispiel erläutern und zugleich ein passendes Lernverfahren für diese Klasse indirekter Zusammenhänge angeben.

Beispiel:
Der Teilschaltkreis in Abbildung 5.4 sei ein Ausschnitt aus einer größeren Schaltung. Alle Signale seien fehlerunabhängig, ansonsten aber unspezifiziert (d.h. die Belegung aller Signalknoten ist $\{0,1\}$). Die Anfangsbelegung sei b_a.
Durch *direkte* Implikationen ist keine Einschränkung von b_a möglich — die Belegung aller Inputsignale der drei Gatter in der Teilschaltung ist $\{0,1\}$, also unspezifiziert, und folglich kann eine Gatterimplikation auch keine Einschränkungen am Gatterausgang implizieren.

Ich betrachte nun das OR–Gatter z. Obwohl für seine Gatterbelegung gilt: $b_a(x_1) = b_a(x_2) = b_a(y) = \{0,1\}$, ist hier doch eine Einschränkung möglich. Betrachtet man nämlich die an Gatter z möglichen *Eingabekombinationen,* so stellt man fest, daß lediglich die 2–Tupel (0,1), (1,0) und (1,1) erzeugbar sind. Die Eingabe $(0,0)$ hingegen läßt sich — aufgrund der Schaltungsstruktur — an z nicht anlegen. $(0,0)$ wäre aber die einzige Belegung, die eine 0 an Signal y erzeugen kann. Somit gilt also: $0 \notin b(y)$ und damit $b(y) = \{1\}$.

Der direkten Implikation bleibt diese Tatsache verschlossen, weil sie von der *Unabhängigkeit* der Eingangssignale von z ausgeht — obwohl zwischen diesen, ausgelöst durch den FanOut f, eine Abhängigkeit besteht. Da jede Einschränkung, die von der direkten Implikation nicht gefunden wird, per Definition indirekt ist, gilt dies auch für den oben gefundenen Zusammenhang an y.

Die Frage ist nun, wie man diese Information *algorithmisch* gewinnen kann. Durch Ausweitung des Aufzählungsprozesses, wie ich es oben schon angesprochen habe, gelingt das wie folgt:

Ich schränke die partielle Belegung von f auf eindeutige Werte ein und untersuche, welche Schlußfolgerungen sich aus dieser Einschränkung ziehen lassen. Dies tue ich für *alle* an f möglichen logischen Grundwerte und zähle damit explizit alle Belegungsmöglichkeiten an f auf.
Da $b_a(f) = \{0,1\}$ gilt, kommen an f als totale Einschränkungen die Wertemengen $\{0\}$ und $\{1\}$ in Frage. Ich schränke $b_a(f)$ zunächst auf die eine, dann auf die andere Menge ein und führe eine direkte Implikation durch. Ich erhalte zwei Einschränkungen von b_a : $b_a^{f=0}$ und $b_a^{f=1}$. Beide Belegungen sind in Abbildung 5.4 gezeigt.

Die beiden Einschränkungen machen, zusammengenommen, den bisher unentdeckten Zusammenhang zwischen den Signalen x_1 und x_2 sichtbar, denn es gilt am Gatterausgang von z: $b_a^{f=0}(y) = b_a^{f=1}(y) = \{1\}$. Welche Belegung man an f auch wählt; die Wertemenge an Signal y ist stets $\{1\}$. Die Aufzählung erkennt — im Gegensatz zur direkten Implikation — die Korreleation, die der Fan–Out f zwischen den Inputs der *OR*–Gatters erzeugt. Resultat: Eine Einschränkung der Ausgangsbelegung $b_a(y) = \{0,1\}$ ist möglich. Man sagt: $b(y)$ ist *indirekt einschränkbar auf* $\{1\}$. ■

Das Lernprinzip für indirekte Einschränkungen

Aus obigem Beispiel läßt sich folgende allgemeine Strategie zum Aufspüren indirekter Zusammenhänge ableiten, die auf Arbeiten von M. Schulz et. al. [STS87] zurückgeht und im Testmustergenerator SOKRATES erstmals implementiert wurde. Die Grundidee des Schulz'schen Verfahrens kann man im folgenden **Lernprinzip für indirekte Einschränkungen** zusammenfassen[1]:

[1]Eine naheliegende Verallgemeinerung ist es, statt an *einzelnen* Signalen an ganzen Signal*gruppen* aufzuzählen. Solch ein Vorgehen wird u.a. in [KP93] vorgeschlagen und ist auch in COAT implementiert ($\rightarrow$ 5.3.6) . Das prinzipielle Vorgehen ändert sich dadurch jedoch nicht. Um die Beschreibung des Lernalgorithmus nicht unnötig zu komplizieren, konzentriere ich mich zunächst auf die einfachste Variante.

> *Sei b_a eine partielle Schaltkreisbelegung. Führe zum Lernen indirekter Einschränkungen an jedem fehlerunabhängigen Signal s des Schaltkreises einzeln folgende Aktionen durch:*
>
> - *Schränke $b_a(s)$ zunächst auf $\{0\}$, dann auf $\{1\}$ ein. Impliziere direkt. Es entstehen die Belegungen $b_a^{s=0}$ und $b_a^{s=1}$.*
> - *Gibt es in der Ausgangsbelegung b_a eines Signals s' einen Wert w, der weder in $b_a^{s=0}$ noch in $b_a^{s=1}$ an s' angelegt werden kann, dann kann $b_a(s')$ um w eingeschränkt werden. Die Aussage $w \notin b_a(s')$ bezeichnet man als* indirekte Einschränkung.

5.3.2 Lernen logischer Schlüsse

Implikationen finden im Verlauf der ATPG in großer Zahl statt. Jede von ihnen sollte möglichst vollständig sein und daher so viele indirekte Zusammenhänge wie möglich erkennen. Daher wäre es prinzipiell sinnvoll, an jede direkte Implikation ein Lernen indirekter Einschränkungen anzuschließen.
Der Laufzeitaufwand einer solchen regelmäßigen Untersuchung kann jedoch so groß werden, daß er jeglichen aus dem Lernen resultierenden Laufzeitvorteil zunichte macht. Aus Effizienzgründen ist man daher bemüht, den Lernalgorithmus zum einen möglichst selten anzuwenden (beispielsweise nur beim Auftreten einer Inkonsistenz, um non–solution–areas möglichst schnell verlassen zu können [SA88]) und zum anderen aus jedem Lernprozeß möglichst viele Informationen zu gewinnen — auch wenn einige von ihnen nicht unmittelbar zu indirekten Einschränkungen führen.
Wie ein solcher indirekter Zusammenhang, der nicht direkt zu einer logischen Einschränkung führt, aussehen kann, und warum es dennoch sinnvoll sein kann, sich diesen Zusammenhang zu merken, möchte ich am folgenden Beispiel erläutern.

Beispiel:

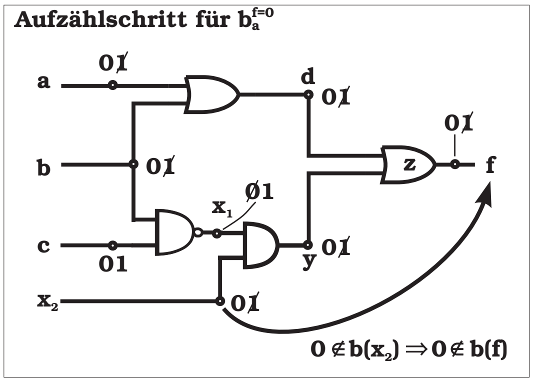


Abbildung 5.5: Lernen logischer Schlüsse

Gegeben sei die Teilschaltung in Abbildung 5.5. Die Anfangsbelegung b_a beschreibe einen fehlerunabhängigen, sonst aber unspezifizierten Teilschaltkreis.
Man beginne einen Lernprozeß an Signal f mit der totalen Wertzuweisung $b_a(f) := \{0\}$. Die aus der direkten Implikation resultierende Belegung sei $b_a^{f=0}$.
(Die Einschränkungen von b_a nach $b_a^{f=0}$ ist in Abbildung 5.5 so dargestellt, daß alle logischen Werte, die in b_a, nicht aber in $b_a^{f=0}$ liegen, durchgestrichen werden.)
Von Interesse ist nun die Wertemengeneinschränkung an Signal x_2. Zunächst ist festzustellen, daß das Lernen einer *indirekten Einschränkung* nach dem oben angegebenen Verfahren an x_2 nicht möglich ist. Grund: Der zweite Lernschritt, also die Betrachtung von $b_a^{f=1} \leq b_a$ mit $b_a^{f=1}(f) := \{1\}$, liefert keine Einschränkung an x_2.

Daß trotzdem ein lernenswerter indirekter Zusammenhang existiert, zeigt eine genauere Betrachtung des ersten Lernschritts bzw. seiner resultierenden Belegung $b_a^{f=0}$.
$b_a^{f=0}$ umfaßt alle totalen konsistenten Einschränkungen $b_t \leq b_a$ von b_a mit $b_t(f) := \{0\}$. Wegen $b_a^{f=0}(x_2) = \{0\}$ folgt damit aber für *alle* $b_t \leq b_a$:

$$b_t(f) = \{0\} \Rightarrow b_t(x_2) = \{0\} \qquad \textbf{(1)}$$

Durch Anwendung des *Gesetzes der Umkehrung* $[(A \Rightarrow B) \iff (\overline{B} \Rightarrow \overline{A})]$ gewinnt man daraus die logisch äquivalente Aussage:

$$b_t(x_2) \neq \{0\} \Rightarrow b_t(f) \neq \{0\} \qquad \textbf{(2)}$$

Daraus wiederum folgt für alle *partiellen* Belegungen $b_p \leq b_a$, daß die Wertemenge an f genau dann um 0 eingeschränkt werden kann, wenn auch die Signalbelegung von x_2 keine 0 mehr enthält:

$$0 \notin b_p(x_2) \Rightarrow 0 \notin b_p(f) \qquad \textbf{(3)}$$

Man überzeugt sich leicht, daß der Schluß (3) indirekt ist. Dazu schränke man in Belegung b_a die Wertemenge des Signals x_2 auf $\{1\}$ ein und impliziere direkt. Nach (3) ist die Einschränkung $b_a^{x_2=1}(f) := \{1\}$ möglich. Sie wird von der direkten Implikation jedoch nicht gefunden. ■

Der logische Schluß in (3) ist ein schönes Beispiel für einen indirekten Zusammenhang, der zwar nicht unmittelbar zu einer Wertemengeneinschränkung führt, im Laufe der Berechnung aber einen erneuten Aufruf der Lernprozedur ersparen kann. Sobald an Signal x_2 eine Einschränkung stattfindet, bei der der Wert 0 entfernt wird, kann auch die Wertemenge an f verkleinert werden, ohne daß nochmal ein expliziter Lernvorgang durchgeführt werden muß.
Dazu muß der Schluß (3) lediglich zwischengespeichert, eben *gelernt,* werden. Von diesem 'Lernen logischer Schlüsse' erhielt der gesamte Algorithmus des lokalen Aufzählens seinen Namen.

5.3.3 Verallgemeinertes Lernen im ATPG–System COAT

Was ich bisher beschrieben habe, ist das Prinzip des Lernens nach Schulz [STS87]. Bevor ich auf Erweiterungen dieses Verfahrens eingehe, möchte ich die bisherigen Ergebnisse kurz zusammenfassen. Das Lernen beschäftigt sich mit zwei Arten indirekter Implikationen:

1. *Indirekte Einschränkungen.* Gemeint sind damit Signalbelegungen, die logisch bereits stärker festgelegt sind, als die direkte Implikation dies feststellen kann. Wissen über indirekte Einschränkungen kann sofort zu einer Anpassung der Signalbelegung genutzt werden. Ein Beispiel für eine indirekte Einschränkung ist die Folgerung '$0 \notin b(y)$' in Abbildung 5.4.

2. *Logische Schlüsse* sind Aussagen der Form '$w_1 \notin b(s_1) \Rightarrow w_2 \notin b(s_2)$'. Logische Schlüsse führen nicht unmittelbar zu einer Einschränkung der Signalbelegung, werden jedoch in geeigneter Weise zwischengespeichert, um später zur Unterstützung der direkten Implikation herangezogen werden zu können. Ein Beispiel für einen indirekten logischen Schluß ist die Folgerung '$0 \notin b(x_2) \Rightarrow 0 \notin b(f)$' in Abb. 5.5.

   Natürlich ist das Lernen eines logischen Schlusses nur dann sinnvoll, wenn er nicht ohnehin von der direkten Implikation gefunden wird.

Schulz erzielte mit dem von ihm vorgestellten Lernverfahren beachtliche praktische Erfolge [SA88]. Durch Einsatz des Lernens gelang es erstmals, auf jedem der bekannten kombinatorischen ISCAS–Benchmarkschaltungen [BF85] eine Fehlerüberdeckung von 100 % zu erreichen.
Trotzdem gibt es auch im Schulz'schen Lernalgorithmus noch einige offene Punkte bzw. Potential für Verallgemeinerungen.

5.3.4 Lernen unterhalb der Fehlerstelle

Einen Ansatzpunkt zur Verallgemeinerung stellt das *Lernen unterhalb der Fehlerstelle* dar. Im Testmustergenerator SOKRATES sind vom Lernen alle die Signale s ausgenommen, zu denen es einen (gerichteten) Pfad von der Fehlerstelle aus gibt.
Dies ist zunächst erstaunlich, da die Grundidee des Lernens allgemeingültig ist. Das Aufzählverfahren kann im Prinzip an *jedem* Signal durchgeführt werden — nur daß an fehlerhaften Signalen außer den Belegungen $\{0\}$ und $\{1\}$ zusätzlich die Wertemengen $\{D\}$ und $\{\overline{D}\}$ berücksichtigt werden müssen. Warum also diese Beschränkung?

Ein Problem liegt sicherlich wieder in der Unschärfe der Roth–Logik. — *Oberhalb* der Fehlerstelle ist deren Codierung eindeutig; '0', '1' bzw. 'X' entsprechen den Wertemengen $\{0\}$, $\{1\}$ bzw. $\{0, 1\}$. *Unterhalb* der Fehlerstelle wird die Roth–Codierung dagegen mehrdeutig, da Wertemengen wie z.B. $\{0, 1\}$ und $\{0, 1, D, \overline{D}\}$ gleichermaßen durch X codiert werden (vgl. Tabelle 2.1).
Soll unter der Fehlerstelle an einem mit 'X' beschrifteten Signal ein Lernprozeß gestartet werden, dann muß im Zweifelsfall die *maximale* Menge $\{0, 1, D, \overline{D}\}$ aufgezählt werden. Folge: Es werden eventuell mehr Werte aufgezählt als nötig; das Lernen wird dadurch nicht nur aufwendiger, sondern auch ungenauer.

Der entscheidende zweite Nachteil der Roth–Logik liegt in der Tatsache, daß die während des Aufzählungsprozesses gelernte *Information* teilweise nicht darstellbar ist. Zur Erläuterung folgendes Beispiel.

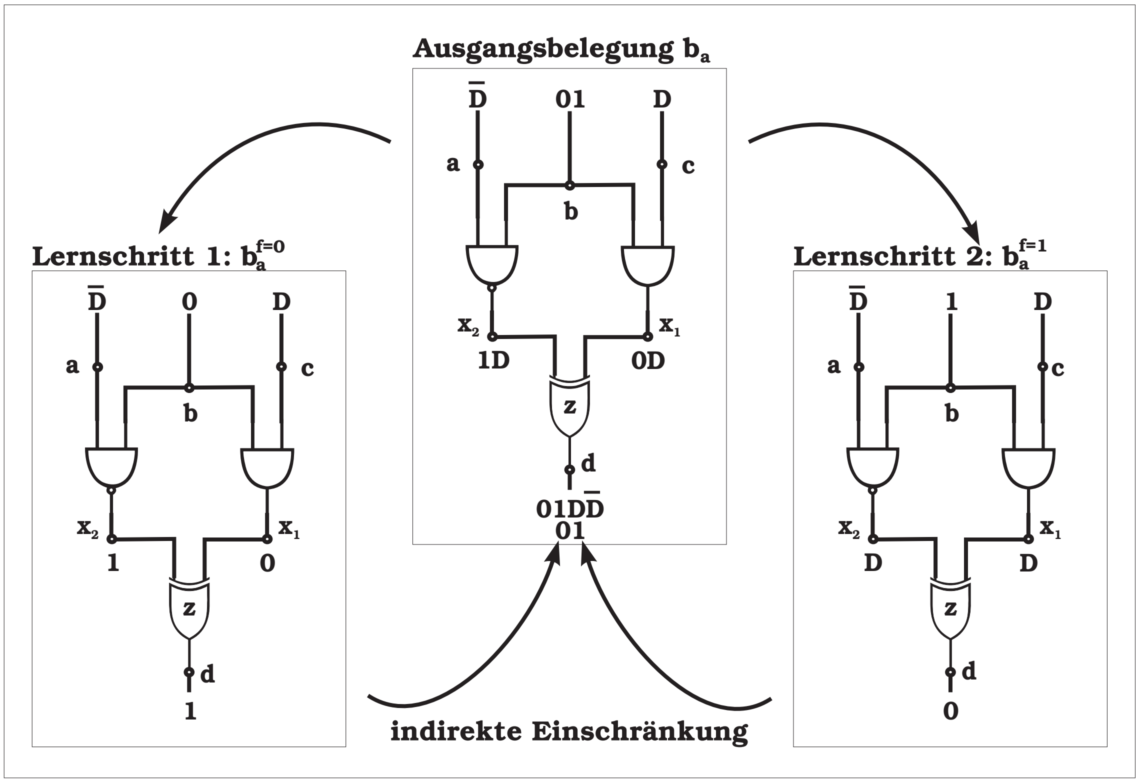


Abbildung 5.6: Lernvorgang unterhalb der Fehlerstelle

Beispiel:
Abbildung 5.6 zeigt einen fehlerabhängigen Teilschaltkreis. Ich erläutere zunächst anhand der Akers–Logik, welche lernenswerte Information sich in ihm verbirgt und erkläre dann, warum die Roth–Logik in dieser Situation versagt.

Die Ausgangsbelegung unter Akers–Logik sei b_a. In ihr ist an den Signalen a und c ein Fehlereffekt beobachtbar. Eine direkte Implikation (oberes Bild) impliziert am Ausgangsignal d des EXOR–Gatters z die Belegung $\{0, 1, D, \overline{D}\}$, also keine Einschränkung der Ausgangsbelegung.

Nun starte ich einen Lernprozeß an Signal b. Da $b_a(b) = \{0, 1\}$ ist, sind zu einer vollständigen Aufzählung hier zwei Lernschritte erforderlich; auch die resultierenden Belegungen $b_a^{b=0}$ und $b_a^{b=1}$ habe ich in Abbildung 5.6 dargestellt.
Wie man sieht, findet an Signal d eine Wertemengeneinschränkung von $\{0, 1, D, \overline{D}\}$ auf $\{0, 1\}$ statt. Diese indirekte Einschränkung liefert die Information, daß keine Fehlerpropagation über Signal d mehr möglich ist — sicherlich eine wesentliche Aussage!

Was geschähe nun mit diesem Wissen unter Roth–Logik? — Antwort: Die 5–wertige Logik repräsentiert die Belegung vor ($\{0, 1, D, \overline{D}\}$) und nach dem Lernen ($\{0, 1)\}$ beide durch X, und da kein Unterschied darstellbar ist, läßt sich die durch das Lernen gewonnene Information auch nicht nutzen!

Bezeichnenderweise spricht man beim Lernen unter Roth von *uniquely defined values* statt von ‘indirekten Einschränkungen’: Nur indirekte Einschränkungen auf totale Belegungen (und damit auf Wertemengen mit einem *eindeutig bestimmten Signalwert*) sind darstellbar. Die Schwächen der Logik schlagen sich in der Begriffsbildung nieder. ■

Das Beispiel zeigt zweierlei. Erstens: Es gibt auch unterhalb der Fehlerstelle lernenswerte indirekte Zusammenhänge. Zweitens: Die Unschärfe der Roth–Logik macht es sehr schwierig und teilweise sogar unmöglich, dieses Zusatzwissen zu gewinnen. Deswegen wird in Roth–Testmustergeneratoren auf ein Lernen unterhalb der Fehlerstelle verzichtet.

Das Lernverfahren unter Akers–Logik (Zusammenfassung)

Bei Verwendung der Akers–Logik ist Lernen unterhalb der Fehlerstelle nicht nur möglich, es ergibt sich auch ganz natürlich aus dem für den Bereich oberhalb der Fehlerstelle beschriebenen Verfahren. An dem Vorgehen nach Schulz sind nur wenige kleine Änderungen nötig:

1. Eine Fallunterscheidung zwischen fehlerab– und unabhängigen Signalen ist nicht mehr erforderlich, da jetzt an *allen* Signalen gelernt wird.
2. Statt sich auf *uniquely defined values,* also auf *totale* Einschränkungen, beschränken zu müssen, können unter Akers–Logik *beliebige* indirekte Einschränkungen gelernt werden.
3. Oberhalb der Fehlerstelle bleibt der Lernprozeß unverändert. Unterhalb der Fehlerstelle werden in die Aufzählprozedur zusätzlich zu den totalen Belegungen $\{0\}$ und $\{1\}$ noch die fehlerabhängigen Belegungen $\{D\}$ und $\{\overline{D}\}$ miteinbezogen.

Das Lernen *logischer Schlüsse* läßt sich analog zu dem beschriebenen Vorgehen für indirekte Einschränkungen auf den fehlerabhängigen Fall übertragen[2].

5.3.5 Der Lernvorgang

In den vorangegangenen Abschnitten habe ich die Aufgabenstellung und den grundsätzlichen Ablauf des Lernens vorgestellt. Abschließend werde ich nun den Lern*algorithmus* des Testmustergenerators COAT beschreiben. Zur graphischen Veranschaulichung dient das Flußdiagramm in Abbildung 5.7.

/∗ Algorithmische Darstellung des Lernprozesses ∗/

aktuelle Belegung: b_a;

forall $s \in S$:

do

  forall $w \in b_a(s)$:

  do

    w an s einpflanzen. Modifizierung der Ausgangsbelegung b_a durch die totale Einschränkung $b_a(s) := \{w\}$.

    direkte Implikation. Anwendung des in Kapitel 4.4 beschriebenen Algorithmus. Die resultierende Belegung sei $b_a^{s=w}$.

    Lernkriterium erfüllt? Analyse der Belegung $b_a^{s=w}$. An jeder Grundzelle z, an der eine Gatterimplikation stattgefunden hat, wird das Lernkriterium überprüft. Dabei wird getestet, ob eine Implikation an z durchgeführt wurde, die die Wertebelegung eines Signals s' auf w' eingeschränkt hat und deren Umkehrimplikation indirekt ist.

    Zusammenhang lernen. War das Lernkriterium für w' an s' erfüllt, so wird der indirekte Zusammenhang $w' \notin b_a(s') \Rightarrow w \notin b_a(s)$ gelernt.

    Belegung wiederherstellen. Entspricht einer Wiederherstellung der Ausgangsbelegung b_a. Auf ihr kann der nächste Schritt des Aufzählprozesses durchgeführt werden.

  od

  Indirekte Einschränkung möglich? Ist an einem Signal s' in allen Lernschritten *dieselbe* Einschränkung um Wert w' möglich, so ist die Wertemenge an s' indirekt einschränkbar.

[2] Die Bestimmung der *Umkehraussage* beim Lernen logischer Schlüsse wurde von Schulz in einer Weise dargestellt, die sich nicht ohne weiteres auf Akers–Logik und das Lernen unterhalb der Fehlerstelle übertragen ließ. Ich habe die Schulz'sche Darstellung daher in Abschnitt 5.3.2 stillschweigend verallgemeinert. In dieser Form ist das Verfahren jetzt unmittelbar auf den Akers–Fall übertragbar.

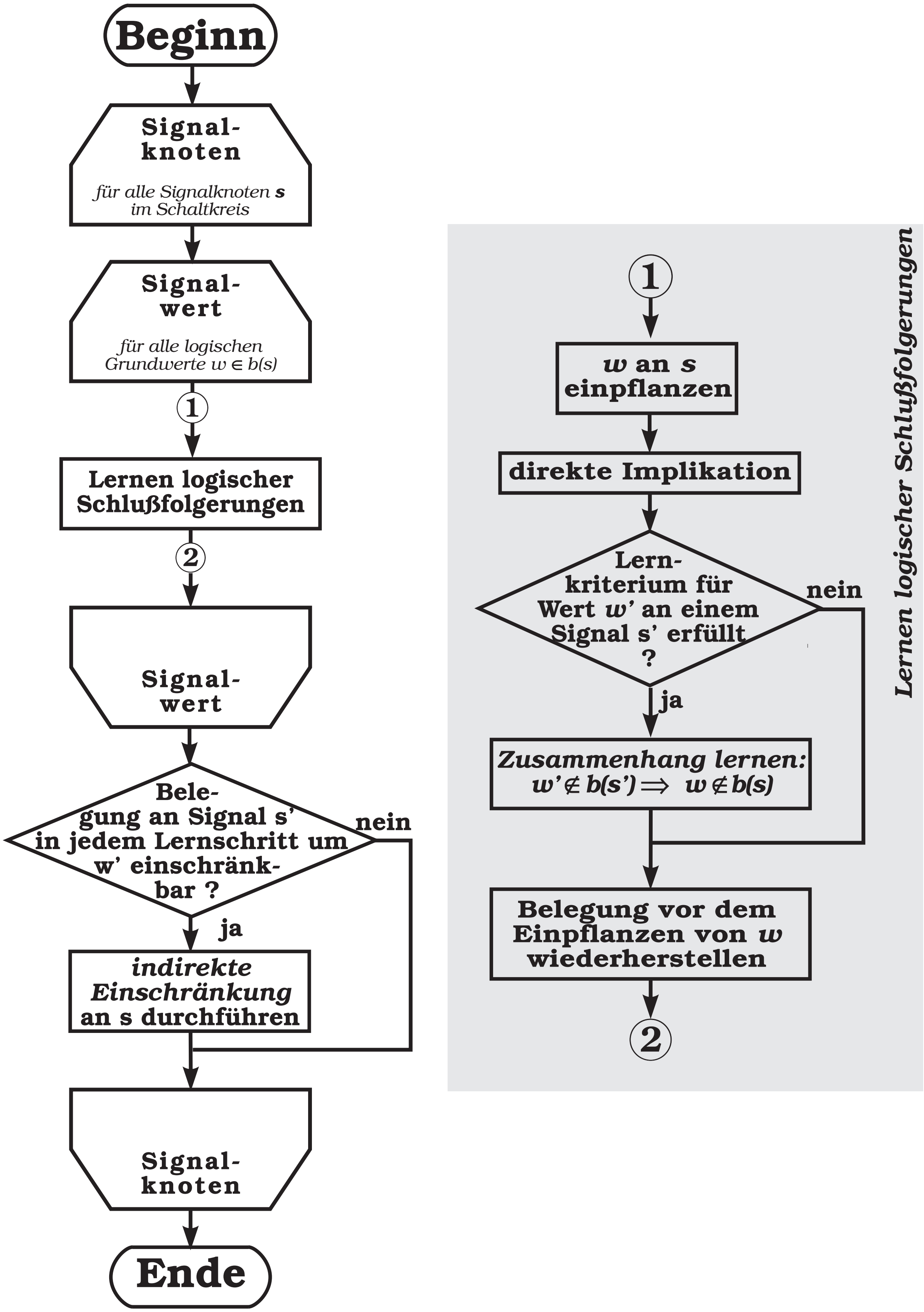


Abbildung 5.7: Flußdiagramm des Lernens

Indirekte Einschränkung durchführen. Modifizierung von b_a durch die totale Einschränkung $w' \notin b_a(s')$. Damit steht die gelernte Implikation der Testmustergenerierung zur Verfügung.

od

5.3.6 Erweiterungen

An dem im letzten Abschnitt beschriebenen Grundalgorithmus sind Erweiterungen möglich, die ich bisher noch nicht besprochen habe. Auch gibt es einige Implementierungseinzelheiten, die ich der Einfachheit halber weggelassen habe. Aus dieser Menge zusätzlicher Details möchte ich kurz ein paar interessante Punkte herausgreifen.

Statisches und dynamisches Lernen

Man kann das Lernen in einer *Vorberechnung* einsetzen (*statisches* Lernen) oder es mit dem direkten Implikationsverfahren kombinieren und so in die eigentliche Testmustergenerierung integrieren (*dynamisches* Lernen). Da das dynamische Lernen zusätzliche Informationen über die aktuelle Belegung nutzen kann, ist es naturgemäß mächtiger als die statische Variante. Der Laufzeitaufwand ist bei dynamischer Anwendung allerdings auch beträchtlich höher.

Lernen an mehreren Signalknoten

Das Schulz'sche Verfahren läßt sich erweitern, indem der Aufzählungsprozeß auf *Signalgruppen* ausgedehnt wird — d.h. man zählt nicht nur totale Belegungen an einem einzigen Signal auf, sondern auch Belegungskombinationen an einer Gruppe von Signalen ([KP93]; vgl. auch Abschnitt 5.3.1).
Ein Extremfall dieser Erweiterung wäre es, alle totalen Schaltkreisbelegungen aufzuzählen. Dies entspräche im Prinzip einer vollständigen Implikation und würde (da dieses Problem NP–hart ist → Satz 4.1.1) zu exponentiellen Laufzeiten führen. Hier muß ein Kompromiß zwischen möglichst hoher Genauigkeit und vertretbarem Zeitaufwand gefunden werden.

Einschränkung des Aufzählraumes

Um den (gerade beim dynamischen Lernen erheblichen) Laufzeitaufwand zu verringern, gibt es Ansätze, den Lernprozeß nur an Signalen durchzuführen, an denen die Wahrscheinlichkeit, indirekte Implikationen zu finden, besonders groß ist. Kandidaten dafür sind *Fan–Out–Punkte,* also Signalknoten mit Ausgangsgrad > 1 (siehe Beispiel in Abbildung 5.4) oder *Rekonvergenzgatter,* an denen sich zwei an einem Fan–Out–Punkt auseinandergehende Pfade wieder treffen (wie im Beispiel in Abbildung 5.5). Genaueres über dieses Verfahren des *orientierten Lernens* findet man in [KP93].

Kapitel 6

Vorgaben: Ausnutzung von Entwerferwissen

Obwohl sich die bisher vorgestellten ATPG–Verfahren sowohl bei der Testgenerierung als auch bei der Redundanzidentifizierung bewährt haben [SA88], läßt sich mit ihnen bei sehr großen Schaltungen häufig nicht die gewünschte Fehlerüberdeckung erzielen.
So erreichte z.B. das ATPG–System SOKRATES [STS87], dem immerhin eine 100%–ige Fehlerüberdeckung auf allen ISCAS–Benchmarkschaltungen [BF85] gelang, bei der Untersuchung eines Schaltkreisentwurfs für einen 32–Bit Gleitkommamultiplizierer nach einer Laufzeit von über 5 Tagen lediglich eine Fehlerüberdeckung von 97.68 % [Spa91].

Die Praxis zeigt, daß es in Entwürfen von der Größe des genannten Multiplizierers meist eine kleine Zahl extrem schwer testbarer Fehler gibt. Die Generierung eines Tests für diese letzten 2 bis 3 % aller Fehler stellt besondere Anforderungen an das Testgenerierungssystem. Ähnliche Probleme ergeben sich beim Redundanznachweis für nicht testbare Fehler.

Auf der Suche nach neuen Verfahren, die eine erfolgreiche Testmustergenerierung auch in Extremsituationen ermöglichen, wurde die Idee geboren, dem Testmustergenerator COAT Zusatzwissen des Schaltkreisentwerfers zur Verfügung zu stellen.
Der Schaltkreisentwerfer verfügt nämlich gerade bei sehr großen Schaltungen über *high–level*–Informationen, die dem auf Gatterebene operierenden ATPG–Algorithmus nicht ohne weiteres zur Verfügung stehen. Der Entwerfer kennt beispielsweise an Bussen geltende Wertebereichseinschränkungen oder entwickelt spontan aussichtsreiche Teststrategien; z.B. eine einfache Vorgehensweise, um einen Fehlereffekt von einem eingebetteten Modul der Schaltung zu den POs zu propagieren. Wäre dieses Wissen für das ATPG–System COAT nutzbar, so die Idee, dann würde es die Testgenerierung deutlich vereinfachen.

Das zentrale Problem dabei ist natürlich die Lösung der Frage, wie man dieses menschliche Wissen einem Testalgorithmus zur Verfügung stellen kann. Im Testmustergenerator COAT werden dazu sogenannte *Vorgaben (constraints)* verwendet, die es erlauben, die Menge der möglichen Belegungen an beliebigen internen Leitungen und Bussen der Schaltung zu beschränken.
Ein ähnliche Vorgehensweise wird mit der Verwendung sogenannter *Restriktoren* in [KLG93] verfolgt. Das dort vorgestellte System basiert jedoch auf dem PODEM–Algorithmus (→ Abschnitt 3.1) und erlaubt daher nur Beschränkungen an den primären Eingängen, also eine partielle Vordefinition des erwarteten Testmusters. Wie ich im folgenden zeigen werde, können aber gerade mit Vorgaben an *inneren* Signalen, auf die Restriktoren keinen unmittelbaren Einfluß haben, interessante Ergebnisse erzielt werden.

Genutzt werden die Vorgaben des Testmustergenerators COAT in einem System zur interaktiven Testmustergenerierung, welches voll in das CADIC–Entwurfssystem (vgl. [Hotz65], [BHKM+87] und [Bur95]) integriert und 1994 auf der International Test Conference [BHHK+94] vorgestellt wurde.
Eine graphische Oberfläche zur Steuerung der Testalgorithmen stellt die *Testshell* von M. Krallmann [Kra94] dar. Die Testshell erlaubt nicht nur einen bequemen und einheitlichen Aufruf aller Testwerkzeuge von der graphischen Oberfläche aus, sondern dient darüber hinaus zur Visualisierung diverser

Datenstrukturen bzw. Endergebnisse des Testvorgangs (Schaltkreisbelegungen, Fehlerüberdeckung,...). Sie ermöglicht insbesondere eine graphische Eingabe der Vorgaben — eine für den Testingenieur sehr komfortable Art der Spezifikation.

6.1 Einführende Beispiele

Zwei Beispiele sollen den Einsatz von Vorgaben motivieren und zugleich deren Wirkungsweise erläutern.

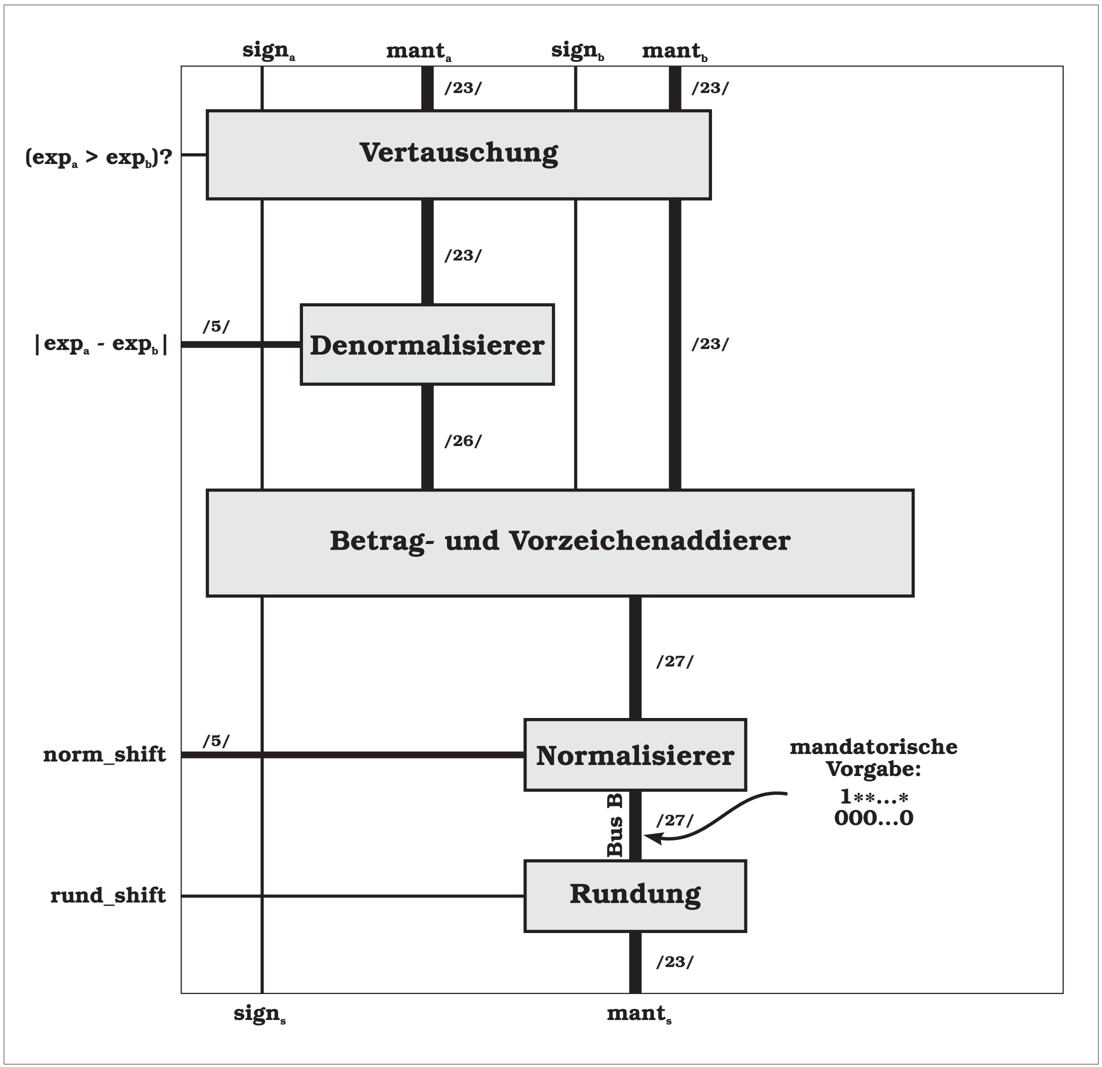


Abbildung 6.1: Mandatorische Vorgabe an 32–Bit Gleitkommaaddierer

Beispiel:
Abbildung 6.1 zeigt den Mantissenteil eines Gleitkommaaddierers. Zum besseren Verständnis will ich die Funktion der Schaltung kurz skizzieren.

Die primäre Schaltkreiseingabe stellen zwei vorzeichenbehaftete Mantissen, also $(sign_a, mant_a)$ und $(sign_b, mant_b)$ dar. Das *Vertauschung*–Modul vertauscht diese beiden Operanden, falls für die zugehörigen Exponenten[1] gilt: $exp_a > exp_b$. Dies soll sicherstellen, daß zur *Denormalisierung* stets die Mantisse

[1] Die Kommunikation mit dem Exponententeil des Gleitkommaaddierers geschieht über die in Abbildung 6.1 am linken Bildrand dargestellten Leitungen.

mit dem *kleineren* Exponenten ausgewählt wird. Beim *Denormalisierungsshift* wird die betroffene Mantisse um $\Delta_{ex} := |exp_a - exp_b|$ nach rechts geshiftet, die herausgeshifteten Bits werden abgeschnitten und die links freiwerdenden Bitpositionen mit Nullen aufgefüllt.
Nachdem die Exponenten durch Denormalisierung angeglichen wurden, werden die beiden Operanden im *Betrag– und Vorzeichenaddierer* addiert. Das Ergebnis wird vom *Normalisierer*–Modul durch Linksshift in die normalisierte Darstellung überführt. Die *Rundung* reduziert das Ergebnis auf die durch die Eingabeoperanden vorgegebene Mantissenlänge.

Der Schaltkreisentwerfer, mit dem Aufbau des Schaltkreises vertraut, verfügt über die high–level–Information, daß die Signalbelegung am Ausgang des Normalisierers (Bus B) im korrekten Schaltkreis der Darstellung einer noch ungerundeten, normalisierten Mantisse entspricht.
Diese Normalisierung hat eingeschränkte Belegungsmöglichkeiten an Bus $\mathcal{B}$ zur Folge, denn in normalisierter Darstellung ist das führende Bit der Mantisse stets 1 — außer bei der Darstellung der Zahl 0, die als '000...0' definiert ist.
An $\mathcal{B}$ sind im korrekten Schaltkreis folglich nur Belegungen aus folgenden beiden Mengen erlaubt:

$$\{1\} \times \{0,1\} \times \{0,1\} \times ... \times \{0,1\}$$
oder
$$\{0\} \times \{0\} \times \{0\} \times ... \times \{0\};$$

wobei die Menge der Busbelegungen als Kreuzprodukt der an den Einzelleitungen erlaubten Belegungen beschrieben ist. Aus Gründen der einfacheren Darstellung führe ich zur Vorgabenspezifikation eine abkürzende Schreibweise ein:

Schreibweise:
Bei der Angabe von Busbelegungen wird statt der Mengen $\{0\}$, $\{1\}$ bzw. $\{0,1\}$ kurz 0, 1 bzw. $*$ geschrieben. Das Kartesische Produkt dieser Mengen wird durch eine Konkatenation der neuen Symbole ersetzt.

Die abkürzende Darstellung für die beiden oben genannten Belegungsmuster ist damit

1 ... *** (statt $\{1\} \times \{0,1\} \times \{0,1\} \times ... \times \{0,1\}$)
bzw.
000 ... 0 (statt $\{0\} \times \{0\} \times \{0\} \times ... \times \{0\}$).

Die auf diese Weise spezifizierten Muster beschreiben alle am Bus $\mathcal{B}$ im korrekten Schaltkreis möglichen Belegungen. Fehler, die mit diesen Mustern an $\mathcal{B}$ nicht getestet werden können, sind damit von vorneherein redundant.

Dem Entwerfer erscheint diese Tatsache aus seiner high–level–Sicht offensichtlich. Der Testgenerierungsalgorithmus muß dieses Wissen jedoch aus der Schaltkreisdarstellung auf Gatterebene extrahieren, was ein schwieriges Problem darstellt — man bedenke nur die riesige Zahl von Schaltmöglichkeiten im oberhalb von $\mathcal{B}$ gelegenen Teil der Schaltung.

Die Information, daß an $\mathcal{B}$ nur noch diesen Mustern entsprechende Belegungen zugelassen sind, bezeichne ich als *Vorgabe (constraint)*. Wie diese Vorgabe an den ATPG–Algorithmus übergeben bzw. von diesem verarbeitet wird, erkläre ich später; zunächst geht es mir um die Motivation des Konzeptes. Graphisch repräsentiere ich Vorgaben vorläufig durch die zugehörigen Belegungsmuster (vgl. Abbildung. 6.1).

Die obige Vorgabe ist übrigens von praktischer Bedeutung: Mit ihr ist es dem Testmustergenerator COAT möglich, schwierige Redundanzen im Rundungsmodul des Addierers, die bei automatischer Vorgehensweise selbst mit erheblichem Laufzeitaufwand nicht zu klassifizieren waren, innerhalb von einer Minute zu identifizieren [BHHK+94]. ■

Zusammenfassend sind folgende Bemerkungen zu obigem Beispiel festzuhalten:

1. Die im Beispiel betrachtete Vorgabe ist *verbindlich (mandatorisch)*, d.h. an Bus $\mathcal{B}$ sind *nur* Belegungen erlaubt, die den obigen Mustern entsprechen.

2. Durch die Einschränkung der Belegungsmöglichkeiten am betroffenen Bus verkleinern verbindliche Vorgaben insbesondere auch den Suchraum, denn man kann die Suche nach einem Test auf die Belegungen (bzw. die zugehörigen Eingabemuster der Schaltung) beschränken, die die Vorgabe respektieren. Dadurch sind verbindliche Vorgaben besonders nützlich beim Auffinden von *Redundanzen.*

Neben den 'mandatorischen' Vorgaben zur Identifizierung von Redundanzen gibt es noch einen weiteren wichtigen Vorgaben–Typ: die 'optionalen' Vorgaben. Diese sind insbesondere dafür geeignet, den Suchprozeß zu *steuern* und zu helfen, einen *Test* zu finden, sofern ein solcher existiert. Auch dazu gebe ich ein Beispiel.

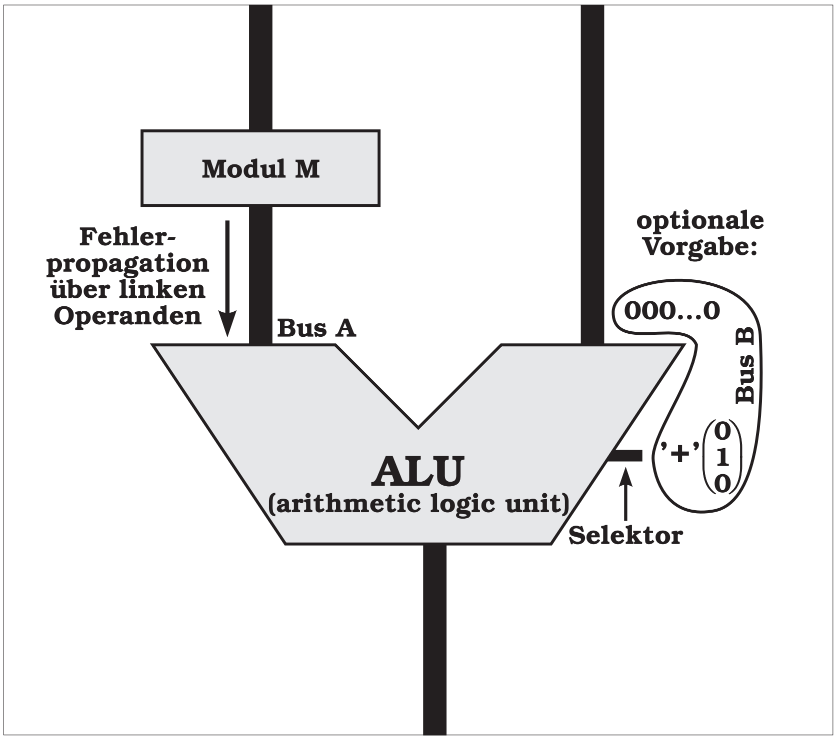


Abbildung 6.2: Optionale Vorgabe an einer ALU

Beispiel:
Man betrachte Abbildung 6.2. Es sei angenommen, daß im Rahmen der ATPG ein Defekt im Modul M untersucht wird. Am Ausgang von M sei ein Fehlereffekt bereits beobachtbar; die Aufgabe besteht nun darin, die Propagation über die ALU *(arithmetic logic unit)* sicherzustellen.

Offensichtlich gibt es dafür verschiedene Möglichkeiten, die je nachdem, welche Operation an der ALU ausgewählt wird, unterschiedlich erfolgversprechend sein können.
Wählt man beispielsweise die bitweise AND–Funktion, so kann es zu einer Fehlermaskierung kommen, wenn das den Fehlereffekt tragende Signal des linken Busses A mit einer am rechten Operanden anliegenden 0 verundet wird. Ähnliches gilt für das bitweise OR und auf 1 gesetzte Signale am rechten Operanden.
Stellt man die ALU auf '+' ein, ist eine Fehlerpropagation in fast jedem Fall sichergestellt (addiert man zwei positive, sich auf mindestens einer Bitposition unterscheidende Binärzahlen a und a' zu demselben Wert b, so unterscheiden sich auch die Ergebnisse c und c'). Lediglich beim Auftreten eines Overflow wäre noch eine Maskierung denkbar (in dem Fall, daß der Fehler sich nur auf das vorderste Bit von a' auswirkt und dieses bei der Addition dadurch abgeschnitten wird, daß die ALU keinen Overflow darstellt). Ein Overflow kann jedoch vermieden werden, indem man den rechten Operanden auf '000...0' setzt.

Der Schaltkreisentwerfer besitzt in diesem Fall also das Wissen, daß sich eine Fehlerpropagation von Bus A zu den POs für jede beliebige Belegung von A erreichen läßt, wenn z.B. der rechte Operand auf '000...0' und der Selektor für das Rechenwerk der ALU auf '+' eingestellt wird. Diese Wahl stellt nicht die *einzige* Propagationsmöglichkeit dar, aber eine sehr aussichtsreiche — und erleichtert die Testgenerierung damit erheblich. ■

Im Vergleich zum Beispiel des Mantissenaddierers ergeben sich am Beispiel der ALU folgende neue Gesichtspunke:

1. Die ALU–Vorgabe ist *optional,* denn es gibt auch andere Möglichkeiten, eine Propagation des linken Operanden zu erreichen (z.B. Setzen des rechten Operanden auf '111...1' bei Selektion der bitweisen AND–Funktion). Ein Test ist auch mit anderen als den durch die Vorgabe festgelegten Belegungen denkbar. Führt eine optionale Vorgabe zu einer Inkonsistenz, kann also nicht ohne weiteres auf Redundanz geschlossen werden.
2. Die Vorgabe sollte mit einer *Fehlerbereichsselektion* kombiniert werden, da die Vorgabe auf Fehlereffekte im Modul M zugeschnitten und in anderen Situationen (z.B. dann, wenn eine Fehlerpropagation über den *rechten* Operanden erfolgen soll) eventuell nicht sinnvoll ist.
3. Die Vorgabe erleichtert die Propagation des Fehlereffektes. Damit unterstützt sie die ATPG bei der Generierung eines *Tests.*

Man beachte, daß die Constraint–Belegungen am rechten Operanden und am Selektor der ALU eine Einheit bilden, d.h. ihre Wirkung ergibt sich erst durch Kombination der Einschränkung an *beiden* Stellen. Die bei der Spezifikation von Vorgaben zusammengehörenden Leitungsstränge müssen also nicht notwendigerweise mit Bussen in der technischen Realisierung des Schaltkreises übereinstimmen.

6.2 Begriffsbildung

In den vorangegangenen Beispielen habe ich das Vorgabenkonzept von COAT vorgestellt und zugleich motiviert. Ich will die dabei informal eingeführten Begriffe nun genauer definieren.

Zuvor möchte ich noch kurz darauf hinweisen, daß es nötig ist, zwischen Belegungen des *korrekten* Schaltkreises (auf die sich die Vorgaben beziehen) und den *parallelen* Darstellungen von korrektem und inkorrektem Schaltkreis (die die Berechnungsgrundlage des ATPG–Algorithmus bilden) zu unterscheiden.

Parallele Belegungen unter Akers–Logik habe ich in Abschnitt 2.2 als Abbildungen der Form $b : S \rightarrow L_A$ definiert. Belegungen des *korrekten* Schaltkreises können analog — da die fehlerabhängigen Grundwerte D und $\overline{D}$ entfallen — durch Abbildungen der Form $\beta : S \rightarrow \{\{0\}, \{1\}, \{0,1\}\}$ dargestellt werden.

Um dem Leser die Unterscheidung zwischen Belegungen des korrekten Schaltkreises und der Paralleldarstellung zu erleichtern, werde ich Belegungen des korrekten Schaltkreises in der Folge mit 'β' und Akers–Belegungen wie gewohnt mit 'b' bezeichnen.

Ich definiere nun die Begriffe 'Bus' und 'Menge erlaubter Busbelegungen'.

Definition 6.2.1:
Sei C ein (korrekter) Schaltkreis, S die Menge seiner Signale.

1. *Ein n–Tupel von Signalen $\mathcal{B} := (s_1, s_2, ..., s_n)$; $s_i \in S \; \forall\, i$ heißt* **Bus der Breite n.** *Es muß nicht notwendigerweise zu einem Bus in der technischen Realisierung korrespondieren.*
2. *Eine* Vorgabe (Constraint) *$v_{\mathcal{B}}$ an Bus $\mathcal{B}$ beschreibt eine* **Menge erlaubter Busbelegungen $M_{v_{\mathcal{B}}}$** *an Bus $\mathcal{B}$.*
3. *Eine totale Belegung $\beta : S \rightarrow \{\{0\}, \{1\}\}$ des (korrekten) Schaltkreises C heißt* **konsistent** *zu einer Vorgabe $v_{\mathcal{B}}$, wenn die durch β gegebene Belegung des Busses $\mathcal{B}$ in der Menge erlaubter Busbelegungen $M_{v_{\mathcal{B}}}$ enthalten ist.*

Ich habe in den obigen Beispielen den *mandatorischen* und den *optionalen* Vorgabentyp schon vorgestellt. Den Unterschied zwischen beiden beschreibt folgende Definition:

Definition 6.2.2:

1. *Eine Vorgabe* $v_{\mathcal{B}}^{m}$ *heißt* **mandatorisch (verbindlich),** *wenn sie eine* zwingende *Beschränkung des Testsuchraumes beschreibt. Kann der Testmustergenerator in dem durch* $v_{\mathcal{B}}^{m}$ *gegebenen Teil des ATPG–Suchraumes* keinen *Test zu einem elementaren Zellenfehler* f *finden, so markiert er* f *als redundant*[2].

2. *Eine Vorgabe* $v_{\mathcal{B}}^{o}$ *heißt* **optional,** *wenn sie eine* nicht zwingende *Einschränkung des Testsuchraumes beschreibt. Kann der Testmustergenerator in dem durch* $v_{\mathcal{B}}^{o}$ *vorgegebenen Teil des Suchraumes* keinen *Test zu einem elementaren Zellenfehler* f *finden, so markiert er* f *als unklassifiziert.*

Sinn der Unterscheidung zwischen mandatorischen und optionalen Vorgaben ist es, Entwerferwissen unterschiedlicher Ausprägung adäquat beschreiben zu können.

Verbindliche Vorgaben setzen ein *exaktes* Wissen über den Schaltkreis voraus, denn der Testingenieur garantiert ja mit der Eingabe mandatorischer Constraints, daß außerhalb des durch die Vorgabe beschriebenen Teilsuchraumes keine Tests möglich sind.

Optionale Vorgaben hingegen dienen dazu, *vages* Wissen zu beschreiben; z.B. Schaltkreisbelegungen, die nach Ansicht des Testingenieurs das Finden eines Tests *vereinfachen* (vgl. dazu das ALU–Beispiel im vorangegangenen Abschnitt). Optionale Vorgaben garantieren *nicht,* daß außerhalb des von ihnen vorgegebenen Suchteilraumes keine Tests existieren. Eine Redundanzerkennung ist mit optionalen Vorgaben nicht möglich.

Die Unterscheidung soll es dem Testingenieur ermöglichen, sein high–level–Wissen möglichst flexibel in Vorgaben umsetzen zu können.

Der Einfachheit halber will ich im folgenden annehmen, daß entweder nur mandatorische oder nur optionale Vorgaben verwendet werden[3].

6.3 Spezifikation von Vorgaben

Im Abschnitt 6.2 habe ich eine Vorgabe abstrakt als Menge von Busbelegungen definiert. *Nicht* gesagt habe ich bisher, wie diese Menge vom Testingenieur tatsächlich angegeben (also *spezifiziert)* werden soll.

An eine gute Vorgaben–Spezifikation werden verschiedene Anforderungen gestellt: Sie sollte *kurz* und *übersichtlich* sein, d.h. den Eingabeaufwand klein halten und von der Datenmenge her überschaubar bleiben; und zugleich *mächtig* in dem Sinne, daß sich jede beliebige Wertemengeneinschränkung durch sie beschreiben läßt.
Im Testmustergenerator COAT wurde ein Mittelweg zwischen komfortaber Eingabemöglichkeit für den Benutzer und algorithmusgerechter Spezifikation gewählt.Das System stellt dazu verschiedene Spezifikationstechniken zur Verfügung, die ich jetzt vorstellen will.

[2] Als *redundant* bezeichnet man Fehler, für die es keine Tests gibt, weil der Defekt das Schaltverhalten des Gesamtschaltkreises nicht beeinflußt (→ Abschnitt 1.3).

[3] Die gleichzeitige Verwendung von optionalen und mandatorischen Vorgaben bringt insofern keine Vorteile, als sich eine Testmustergenerierung mit mandatorischen *und* optionalen Vorgaben stets durch mehrere 'reine' Programmläufe simulieren läßt.

6.3.1 Vorgaben und Vorgabevektoren

Eine triviale Möglichkeit der Spezifikation wäre es, alle erlaubten (oder alternativ: untersagten) totalen Busbelegungen explizit aufzuzählen. Da deren Anzahl jedoch exponentiell mit der Busbreite wächst, ist dieses Verfahren für den Benutzer zu aufwendig.

Darum werden in COAT Belegungs*muster* verwendet, um Wertemengen kompakt beschreiben zu können (diese Vorgehensweise wurde ja schon implizit auf Seite 50 vorgestellt). Ich bezeichne solche Belegungsmuster von nun an als *Vorgabevektoren.*

Definition 6.3.1:
Sei $\mathcal{B} := (s_1, ..., s_n)$ ein Bus der Länge n.

Ein **Vorgabevektor** φ *ist ein n–Tupel* $\varphi := (x_{s_1}, ..., x_{s_n})$ *mit* $x_{s_i} \in \{0, 1, *\}\ \forall\, i$.

Die **Menge der von φ spezifizierten Belegungen M_φ** *ist wie folgt gegeben: Sei* $\beta_\mathcal{B} := (\beta(s_1), ..., \beta(s_n))$ *eine totale Belegung von* $\mathcal{B}$*. Dann gilt:*

$$\beta_\mathcal{B} \in M_\varphi \quad \Leftrightarrow \quad \forall\, i \text{ mit } (x_{s_i} \neq `*'): \quad \beta_\mathcal{B}(s_i) = \{x_{s_i}\}.$$

Beispiel:
Sei $\mathcal{B}$ ein Bus der Länge 4. Die folgende Menge erlaubter Busbelegungen

$$M := \{(0,0,1,0), (0,1,1,0), (1,0,1,0), (1,1,1,0), (0,1,1,1), (1,1,1,1)\}$$

läßt sich kompakt durch die beiden Vorgabevektoren

$$\varphi_1 := (*, *, 1, 0) \qquad \textit{und} \qquad \varphi_2 := (*, 1, 1, 1)$$

darstellen. ■

Im System COAT werden Vorgaben über drei Mengen von Vorgabevektoren spezifiziert:

Definition 6.3.2:
Sei $\mathcal{B} := (s_1, ..., s_k)$ ein Bus der Länge k. Eine **Vorgabe (Constraint) $v_\mathcal{B}$** *an $\mathcal{B}$ ist gegeben durch*

$v_\mathcal{B} := \{v_\mathcal{B}^f, v_\mathcal{B}^r, v_\mathcal{B}^e\}$ *mit*

$$\begin{array}{lcll} v_\mathcal{B}^f & := & \{\varphi^f\}, & \textit{die Menge des } \text{fixed–Vektors;} \\ v_\mathcal{B}^r & := & \{\varphi_1^r, ... \varphi_m^r\}, & \textit{die Menge der } \text{restrict–Vektoren } \textit{sowie} \\ v_\mathcal{B}^e & := & \{\varphi_1^e, ... \varphi_n^e\}, & \textit{die Menge der } \text{exclude–Vektoren.} \end{array}$$

$v_\mathcal{B}^f$, $v_\mathcal{B}^r$ *und* $v_\mathcal{B}^e$ *beschreiben folgende Belegungsmengen:*

$$M_{v_\mathcal{B}^f} := \begin{cases} \{0,1\}^k & , \textit{falls } v_\mathcal{B}^f = \emptyset \\ M_{\varphi^f} & , \textit{sonst} \end{cases} \qquad M_{v_\mathcal{B}^r} := \begin{cases} \{0,1\}^k & , \textit{falls } v_\mathcal{B}^r = \emptyset \\ \bigcup\limits_{i=1}^{m} M_{\varphi_i^r} & , \textit{sonst} \end{cases}$$

$$M_{v_\mathcal{B}^e} := \begin{cases} \{0,1\}^k & , \textit{falls } v_\mathcal{B}^e = \emptyset \\ \{0,1\}^k \setminus \bigcup\limits_{i=1}^{n} M_{\varphi_i^e} & , \textit{sonst} \end{cases}$$

Die zu $v_\mathcal{B}$ gehörende Belegungsmenge $M_{v_\mathcal{B}}$ ist: $M_{v_\mathcal{B}} := M_{v_\mathcal{B}^f} \cap M_{v_\mathcal{B}^r} \cap M_{v_\mathcal{B}^e}$.

Um die Unterschiede zwischen den einzelnen Vektortypen zu verdeutlichen, werde ich fixed–, restrict– und exclude–Vektoren jetzt im einzelnen betrachten, ihre jeweiligen Vorteile und Einsatzgebiete nennen und sie an Beispielen erläutern.

Fixed–Vorgaben

Die fixed–Vorgabenmenge $v^f_{\mathcal{B}} := \{\varphi^f\}$ enthält nur einen *einzigen* Vorgabevektor. Die Möglichkeiten, Busbelegungen in $M_{v^f_{\mathcal{B}}}$ zu spezifizieren, sind damit zwar beschränkt, dafür lassen sich fixed–Vorgaben vom ATPG–Algorithmus besonders effizient auswerten (→ Abschnitt 6.4).
Fixed–Constraints werden verwendet, wenn Teile des betrachteten Busses auf eine ganz bestimmte Belegung *fixiert* werden sollen. Sie können die Vorgabenspezifikation an sehr breiten Bussen, bei denen an bestimmten Leitungen immer das gleiche Muster auftaucht, erleichtern.

Um Fixed–Vorgaben von den anderen beiden Typen unterscheiden zu können, werde ich den Vorgabenvektor φ^f im folgenden durch die Markierung 'f:' kennzeichnen.

Beispiel:
Der Constraint an der ALU in Abbildung 6.2 (Seite 51) ist eine (optionale) fixed–Vorgabe. Sie wird spezifiziert durch den Vorgabevektor

$$f : 000...0\,010.$$

■

Restrict–Vorgaben

Die restrict–Vorgabenmenge $v^r_{\mathcal{B}} := \{\varphi^r_1, ...\varphi^r_m\}$ ist — im Gegensatz zu $v^f_{\mathcal{B}}$ — mehrelementig. In $M_{v^r_{\mathcal{B}}}$ lassen sich damit *beliebige* Belegungsmengen beschreiben.

Restrict–Vorgaben beschreiben genau wie fixed (aber im Gegensatz zu exclude, s.u.) *erlaubte* Busbelegungen. Die φ^r_i–Vorgabevektoren von restrict–Constraints markiere ich im folgenden mit 'r:'.

Beispiel:
Die (mandatorische) Vorgabe in Abbildung 6.1 (Seite 49) ist vom Typ restrict, denn ihre Belegungsmenge wird durch *zwei* Vorgabevektoren beschrieben:

$$r : 1**...*$$
$$r : 000...0$$

■

Exclude–Vorgaben

geben — im Gegensatz zu den beiden vorangegangenen Typen — *nicht* erlaubte Busbelegungen an ($M_{v^e_{\mathcal{B}}}$ beschreibt das *Komplement* der von den exclude–Vorgabevektoren beschriebenen Belegungsmengen, s.o.). Sie sind besonders dann hilfreich, wenn sich die Einschränkung einer Busbelegung durch restrict–Vorgaben nur mit großem Aufwand realisieren ließe.

Die φ^e_i–Vorgabevektoren von exclude–Constraints markiere ich im folgenden mit 'e:'.

Beispiel:
Will man an einem Bus der Breite n alle Belegungen außer '000...0' zulassen, so wären bei reiner Verwendung von restrict–Vorgaben n Vorgabevektoren zu ihrer Spezifikation nötig:

$$r : 1**...**$$
$$r : *1*...**$$
$$r : **1...**$$
$$...$$
$$r : ***...1*$$
$$r : ***...*1$$

Bei Verwendung von exclude–Vorgaben reicht jedoch ein einziger Vektor aus:

$$e : 000...00$$

■

6.4 Realisierung des Constraint–Konzeptes

Ich habe den Begriff der Vorgabe bisher aus Benutzersicht gesehen; habe also geschildert, wie high–level–Wissen des Entwerfers aussehen kann und wie man es in eine Vorgabendarstellung übersetzt. Nun wende ich mich der Frage zu, wie das ATPG–System die ihm so zugänglich gemachten Informationen verarbeitet.

Es wäre natürlich schön, wenn man die in den Vorgaben übergebene high–level–Information *optimal* nutzen könnte, wenn man also aus den Vorgabevektoren die Menge der zu ihnen konsistenten totalen Busbelegungen tatsächlich exakt bestimmen könnte. Leider erweist sich dies als sehr schwierig.

Lemma 6.4.1:
Sei wieder $\mathcal{B} := (s_1, ..., s_k)$ ein Bus der Breite k. An $\mathcal{B}$ sei die Vorgabe $v_\mathcal{B} := \{v_\mathcal{B}^f, v_\mathcal{B}^r, v_\mathcal{B}^e\}$ spezifiziert über folgende Mengen von Vorgabenvektoren:

$$\begin{aligned} v_\mathcal{B}^f &:= \{\varphi^f\}, \\ v_\mathcal{B}^r &:= \{\varphi_1^r, ...\varphi_m^r\} \quad \textit{sowie} \\ v_\mathcal{B}^e &:= \{\varphi_1^e, ...\varphi_n^e\}. \end{aligned}$$

Das Problem, ob es eine zu $v_\mathcal{B}$ konsistente Busbelegung von $\mathcal{B}$ gibt ('$M_{v_\mathcal{B}} \neq \emptyset$?'), ist NP–hart.

Beweis:
Obige Aussage läßt sich leicht durch Reduktion von SAT auf das gegebene Problem beweisen. Ich konzentriere mich auf die exclude–Vorgabe $v_\mathcal{B}^e$ und bilde die charakteristische Funktion der von ihr beschriebenen Belegungsmenge $M_{v_\mathcal{B}^e}$. Man betrachte dazu folgendes Beispiel:

Beispiel: An dem Bus $\mathcal{B} := \{s_1, s_2, s_3, s_4, s_5\}$ sei die exclude–Vorgabe $v_\mathcal{B}^e$ wie folgt gegeben:

$$v_\mathcal{B}^e \left\{ \begin{array}{cccccc} & s_1 & s_2 & s_3 & s_4 & s_5 \\ e: & 1 & 0 & 0 & * & * \\ e: & 0 & 1 & * & 1 & * \\ e: & * & * & 0 & 1 & 1 \end{array} \right.$$

Die charakteristische Funktion von $M_{v_\mathcal{B}^e}$ ist eine boolesche Funktion in 5 Variablen, die sich wie folgt ergibt:

$$\begin{aligned} \sigma(M_{v_\mathcal{B}^e}) &:= \overline{(x_1\overline{x}_2\overline{x}_3) \vee (\overline{x}_1 x_2 x_4) \vee (\overline{x}_3 x_4 x_5)} \\ &= (\overline{x}_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \overline{x}_2 \vee \overline{x}_4) \wedge (x_3 \vee \overline{x}_4 \vee \overline{x}_5); \end{aligned}$$

was offensichtlich eine boolesche Formel in konjunktiver Normalform (KNF) mit Termlänge ≥ 3 Literalen ist.

Umgekehrt läßt sich nun in dieser Weise zu jeder booleschen Formel σ in KNF eine exclude–Vorgabe an einem Bus geeigneter Länge konstruieren, zu der es *genau dann* eine konsistente Busbelegung gibt, wenn σ erfüllbar ist.

Damit ist das Erfüllbarkeitsproblem für boolesche Formeln in KNF (SAT) auf das Problem reduziert, ob die exclude–Vorgabevektoren aus $v_\mathcal{B}^e$ die leere Menge beschreiben. ■

Da eine vollständige algorithmische Ausnutzung des in einer Vorgabe $v_\mathcal{B}$ enthaltenen high–level–Wissens aufgrund des Lemmas nicht realisierbar erscheint, muß man sich notgedrungen auf Teillösungen mit vertretbarem Laufzeitaufwand beschränken. Die in COAT eingesetzten Heuristiken werde ich in den nun folgenden Abschnitten vorstellen. Zuvor aber noch ein paar Worte zu einem technischen Problem.

6.4.1 Vorgaben und Akers–Logik

Ein Problem, welches es vor der algorithmischen Nutzung von noch Vorgaben zu lösen gilt, ist das der *Schaltkreisbelegung.* Wie schon in Abschnitt 6.2 erläutert wurde, beschreiben Vorgaben Belegungen des *korrekten* Schaltkreises. Der ATPG–Algorithmus arbeitet aber auf parallelen Beschreibungen der 16–wertiger Logik (→ Kapitel 2). Bei der Auswertung der Vorgaben muß daher die eine Belegungsdarstellung in die andere übertragen werden.
Diese Konvertierung ist leicht, wenn man sich in Erinnerung ruft, wie in den parallelen Belegungen korrekter und fehlerhafter Schaltkreis codiert wurden:

$$\begin{aligned}\mathbf{0} &:= 0/0\\ \mathbf{1} &:= 1/1\\ \mathbf{D} &:= 1/0\\ \overline{\mathbf{D}} &:= 0/1,\end{aligned}$$

wobei der Wert vor dem Querstrich die Belegung des *korrekten* und der danach die des *fehlerhaften* Schaltkreises beschreibt. Man konvertiert eine Belegung β des korrekten Schaltkreis also in die Akers–Darstellung b_β, indem man sich auf die *erste* Komponente der obigen Codierung konzentriert:

Belegung β des korrekten Schaltkreises		**Belegung b_β in 16–wertiger Logik**
	wird konvertiert in $\longrightarrow$	
$\{0\}$		$\{0, \overline{D}\}$
$\{1\}$		$\{1, D\}$
$\{0, 1\}$		$\{0, 1, D, \overline{D}\}$

Formal läßt sich Zusammenhang zwischen Vorgaben und Belegungen unter Akers–Logik so beschreiben:

Definition 6.4.2:
Sei $\mathcal{B}$ ein Bus; $v_\mathcal{B}$ eine Vorgabe an $\mathcal{B}$ und $M_{v_\mathcal{B}} := \{\beta_1(\mathcal{B}), ..., \beta_m(\mathcal{B})\}$ die zugehörige Menge erlaubter totaler Belegungen des korrekten Schaltkreises.

1. *Die zu einem $\beta_i \in M_{v_\mathcal{B}}$ durch Konversion (s.o.) gewonnene Akers–Belegung $\boldsymbol{b}_{\beta_i}$ heißt das* **Äquivalent** *zu β_i unter 16–wertiger Logik.*
2. *$M^*_{v_\mathcal{B}} := \{b_{\beta_1}, ..., b_{\beta_m}\}$ heißt* **die Menge der unter Akers–Logik erlaubten totalen Busbelegungen** *von $v_\mathcal{B}$.*
3. *$\boldsymbol{b}_{v_\mathcal{B}}(\boldsymbol{\mathcal{B}}) := \boldsymbol{\mathcal{V}}(\boldsymbol{M}^*_{v_\mathcal{B}})$ heißt* **die von $\boldsymbol{v_\mathcal{B}}$ induzierte Busbelegung** *unter 16–wertiger Logik an $\mathcal{B}$.*

Mit folgendem Beispiel will ich den Zusammenhang zwischen Vorgaben und Belegungen des korrekten Schaltkreises sowie den Akers–Belegungen aus Kapitel 2 illustrieren.

Beispiel:
Sei dazu $v_\mathcal{B} := \{\emptyset, \{\varphi_1^r, \varphi_2^r\}, \emptyset\}$ mit

$$\begin{aligned}\varphi_1^r &:= 100*\\ \varphi_2^r &:= 1*10\end{aligned}$$

eine Vorgabe an einem Bus $\mathcal{B}$ der Breite 4. $v_\mathcal{B}$ besteht nur aus zwei restrict–Vektoren, denen folgende zwei partiellen Belegungen des korrekten Schaltkreises entsprechen:

$$\begin{aligned}\beta_1(\mathcal{B}) &:= (\{1\}, \{0\}, \{0\}, \{0,1\})\\ \beta_2(\mathcal{B}) &:= (\{1\}, \{0,1\}, \{1\}, \{0\})\end{aligned}$$

Das 16–wertige Äquivalent zu den beiden genannten Restrict–Vektoren erhält man durch die oben beschriebene Art der Konvertierung:

$$\begin{aligned} b_{\beta_1}(\mathcal{B}) &:= (\{1, D\}, \{0, \overline{D}\}, \{0, \overline{D}\}, \{0, 1, D, \overline{D}\}) \\ b_{\beta_2}(\mathcal{B}) &:= (\{1, D\}, \{0, 1, D, \overline{D}\}, \{1, D\}, \{0, \overline{D}\}) \end{aligned}$$

... und die Menge der unter Akers–Logik erlaubten totalen Busbelegungen sowie die von $v_{\mathcal{B}}$ induzierte Gesamtbelegung ergeben sich zu:

$$M^*_{v_{\mathcal{B}}} := \{b_t \mid (b_t \dot{\leq} b_{\beta_1}) \lor (b_t \dot{\leq} b_{\beta_2})\}$$

$$b_{v_{\mathcal{B}}}(\mathcal{B}) := \mathcal{V}(M^*_{v_{\mathcal{B}}}) = \{(\{1, D\}, \{0, 1, D, \overline{D}\}, \{0, 1, D, \overline{D}\}, \{0, 1, D, \overline{D}\})$$

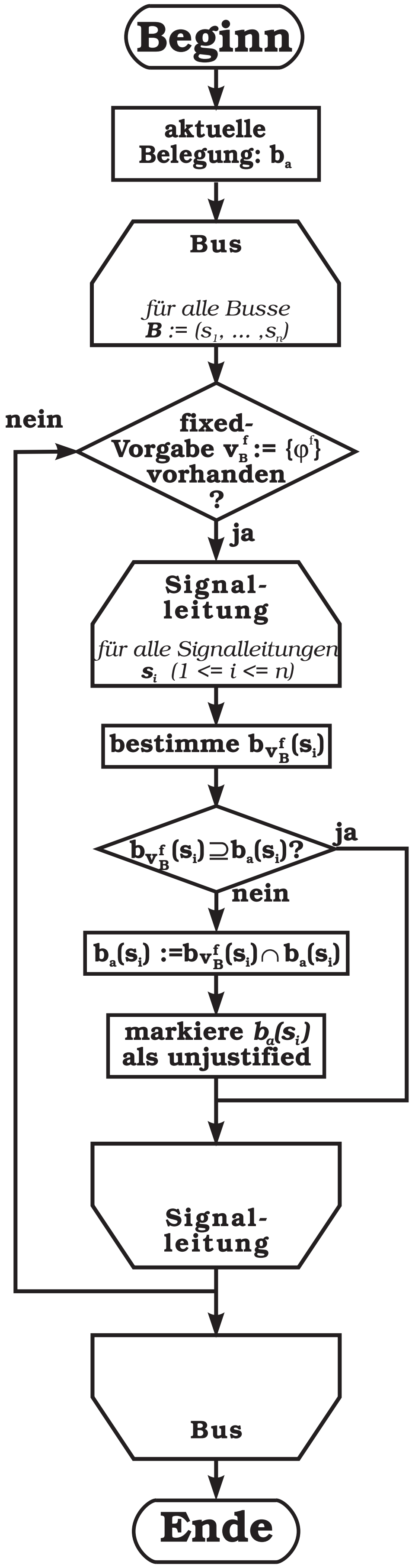
■

Mit Hilfe der in diesem Abschnitt eingeführten Begriffe werde ich als nächstes die Implementierung des Vorgabenkonzeptes in COAT beschreiben. Ich beginne mit den fixed–Vorgaben, deren Auswertung einfacher ist als die von restrict– und exclude–Constraints.

Fixed–Vorgaben werden in der Initialisierungsphase der ATPG ausgewertet. Sie verändern dadurch die Startbelegung der Berechnung, verändern den eigentlichen ATPG–Algorithmus jedoch nicht.

Dadurch können fixed–Vorgaben effizienter ausgewertet werden als restricts und excludes, die eine *dynamische* Verarbeitung und so einen größeren Aufwand erfordern.

Der Auswertungsalgorithmus für fixed–Vorgaben ist graphisch auf der nächsten Seite dargestellt:



Auswertung von fixed–Vorgaben

Die Auswertung von fixed–Vorgaben erfolgt in COAT im Rahmen der *Initialisierung* (→ S. 23).

Es handelt sich damit um eine einmalige Aktion; *dynamisch,* also innerhalb des eigentlichen ATPG–Algorithmus, wird keine Rechenzeit benötigt.Der nebenstehende Programmablaufplan beschreibt den Algorithmus graphisch.

Der Algorithmus betrachtet an jedem Bus $\mathcal{B}$ mit einer fixed–Vorgabe $v^f_{\mathcal{B}}$ die zugehörigen Signalleitungen *einzeln* (vgl. Schleife *'Signalleitung'* in der Abbildung links).
Fixed–Vorgaben beschreiben *feste* Einschränkungen, d.h. an jedem s_i sind nur *solche* Signalwerte zugelassen, die $b_{v^f_{\mathcal{B}}}$, die von der fixed–Vorgabe induzierte Belegung, beschreibt.
Es wird daher an jedem s_i der *Schnitt* aus aktueller Belegung $b_a(s_i)$ und der induzierten Belegung $b_{v^f_{\mathcal{B}}}(s_i)$ gebildet und die alte Signalbelegung durch den Schnitt ersetzt.

Es ist zu beachten, daß sich die daraus resultierende Einschränkung aus der *Vorgabe* ergibt und im allgemeinen noch nicht durch die PI–Belegung *gerechtfertigt* ist. Um dies anzuzeigen, wird das Signal s_i als *unjustified line* markiert (zum Begriff 'unjustified line' vgl. Abschnitt 4.3).

6.4.2 Vorgaben im Suchprozeß

Ich habe schon erwähnt, daß die Auswertung von restrict– und exclude–Vorgaben etwas aufwendiger ist als die der fixed–Constraints. Warum ist das so?

Wie in Abschnitt 3.2 erläutert, untersucht der ATPG–Algorithmus schrittweise einzelne Teilbereiche des Gesamtsuchraumes. In einer gegebenen Berechnungssituation können einige Vorgabenvektoren dadurch *irrelevant* werden, wenn sie Beschränkungen in Belegungsbereichen beschreiben, die zu diesem Zeitpunkt nicht untersucht werden. Bei einem *Backtrack* (→ 3.3) und dem anschließenden Wechsel des Teilsuchraumes können diese 'desaktiven' Vorgabenvektoren ihre Bedeutung aber wiedergewinnen.

Vorgaben sind damit demselben Mechanismus unterworfen, der auch die Schaltkreisbelegung und interne Listenstrukturen an den gerade bearbeiteten Suchraum anpaßt und diese Änderungen im Falle eines Backtracks wieder rückgängig macht (wie in Abschnitt 3.3 unter *'Berechnungssituation wiederherstellen'* beschrieben).
Wie unterscheidet man nun auf den verschiedenen Berechnungsebenen effizient zwischen relevanten und irrelevanten Vorgabevektoren? Um die restrict– und exclude–Constraints nicht bei jedem Übergang zu einer neuen Berechnungsebene kopieren zu müssen, werden in COAT alle Vorgabevektoren mit Markierungen versehen, die anzeigen, ob der Vektor auf der jetzigen Berechnungsstufe *aktiv* oder *desaktiv* ist.
Nur diese Markierungen werden bei jedem *branch*–Schritt gesichert und bei einem nachfolgenden *bound* wieder rekonstruiert. Die Vorgabevektoren selbst sind nur einmal vorhanden.

Auswertung von restrict– und exclude–Vorgaben

Nun zu den eigentlichen Auswertungsalgorithmen. Abbildung 6.4 zeigt sowohl den Ablauf der restrict– als auch der exclude–Auswertung in COAT .

Bei *restrict*–Vorgaben werden zunächst alle aktiven Vorgabevektoren bestimmt (Schleife *'Restrict–Vektoren'* in der Abbildung). Ein Vorgabevektor φ_i^r ist desaktiv, wenn er eine Belegungsmenge beschreibt, die derzeit nicht betrachtet wird; wenn also die von ihm beschriebene Belegungsmenge $M^*_{\varphi_i^r}$ und die durch die aktuelle Belegung b_a gegebene Menge totaler Einschränkungen $M_{\dot{\leq} b_a}$ keine gemeinsamen Elemente besitzen.

Ist der Aktivitätsstatus aller φ_i^r geklärt, folgen zwei Auswertungsschritte:

1. sind *alle* φ_i^r *aus* $v^r_{\mathcal{B}}$ *desaktiv* (d.h. keines der von der restrict–Vorgabe erlaubten Belegungsmuster paßt auf die aktuelle Belegung b_a), so liegt eine *Inkonsistenz* vor, die zu einem *Backtrack* (oder, falls kein Backtrack mehr möglich, zum *Abbruch* der Berechnung) führt.
2. fordern *alle aktiven* φ_i^r *dieselbe Einschränkung an einem* s_i (d.h. gleich, welches Belegungsmuster gewählt wird: Die Einschränkung an s_i ist dieselbe), so kann die Belegung an s_i auf $b_a(s_i) \cap b_{v^r_{\mathcal{B}}}(s_i)$ eingeschränkt werden — analog zur Wertemengeneinschränkung bei fixed–Vorgaben.

Zur Auswertung von exclude–Vorgaben wird in COAT eine recht einfache Heuristik verwendet. Als Kriterium beschrieben, läßt sie sich etwa wie folgt zusammenfassen (auch hier befindet sich wieder eine graphische Darstellung in Abbildung 6.4):

> *Sei* φ_i^e *ein exclude–Vektor,* b_a *aktuelle Schaltkreisbelegung. Ist* $M^*_{\varphi_i^e}$ *eine echte Obermenge von* $M_{\dot{\leq} b_a}$*, d.h. wird* b_a *voll von einem* φ_i^e *überdeckt*
> → Inkonsistenz.

Das Kriterium erkennt genau dann auf Inkonsistenz, wenn alle gemäß b_a noch zulässigen Busbelegungen an $\mathcal{B}$ von φ_i^e als 'verboten' klassifiziert werden. Dadurch werden exclude–Vorgaben in COAT vornehmlich zum Aufspüren von non–solution–areas eingesetzt.

Auch exclude–Vorgaben können im Verlauf der Berechnung des– und wieder reaktiviert werden; das Aktivitätskriterium sowie die Verwaltung der Aktivierungs–/Desaktivierungsinformation sind dieselben wie bei den restrict–Vorgaben.

Zusammengenommen stellen fixed–, restrict– und exclude–Vektoren eine komfortable und leicht zu handhabende Technik zur Spezifikation und Auswertung von Vorgaben dar.

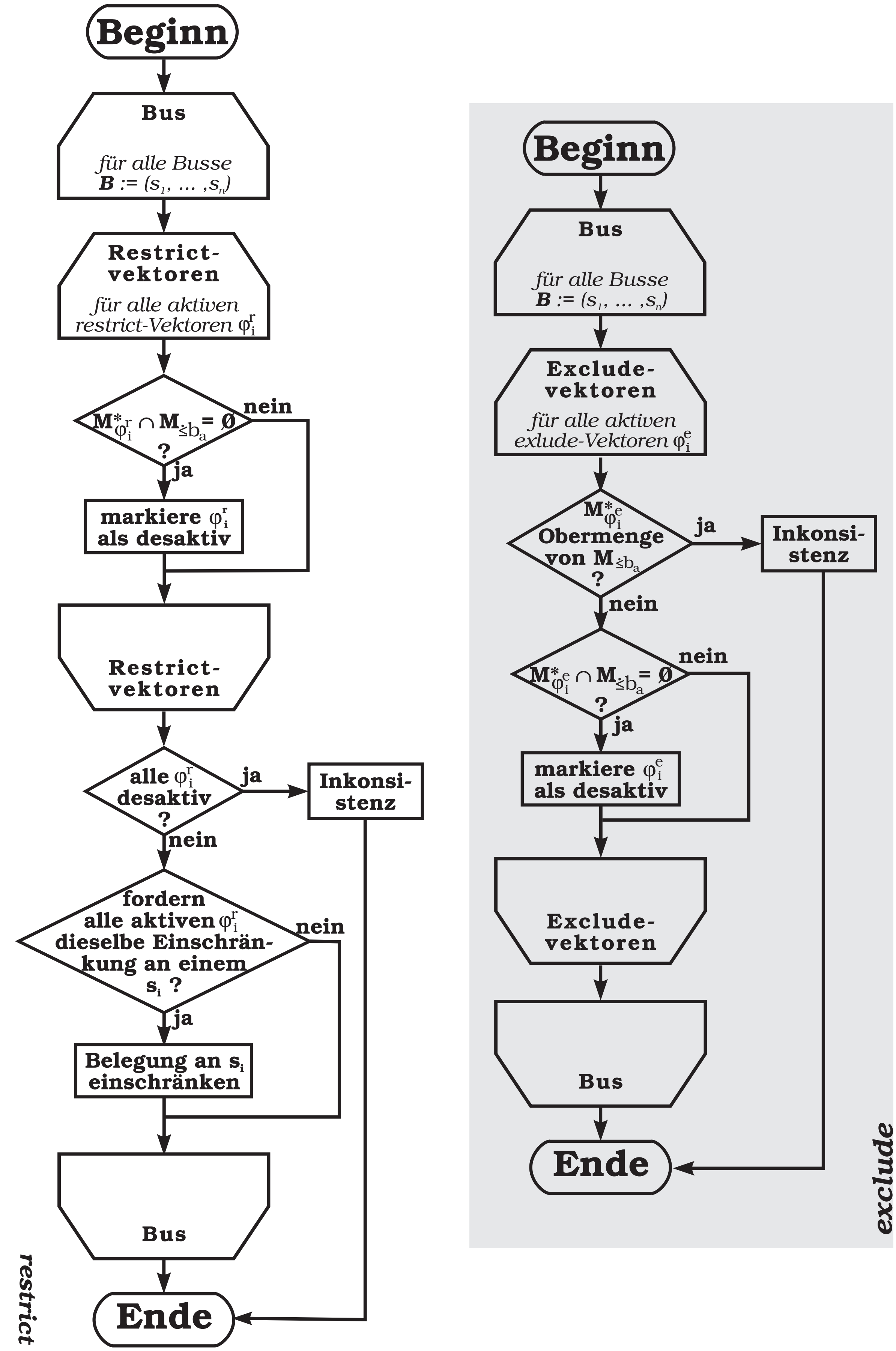


Abbildung 6.4: Auswertung von restrict– und exclude–Vorgaben

Kapitel 7

Praktische Ergebnisse

Aus den Erfahrungen, die im praktischen Einsatz mit COAT gemacht wurden, möchte ich zum Abschluß zwei Resultate herausgreifen, die im Hinblick auf die Zielsetzung meiner Arbeit von besonderem Interesse sind.

COAT und das SB–PRAM–Projekt

Der Testmustergenerator COAT wurde am Lehrstuhl von Prof. W. J. Paul (Fachbereich Informatik, Universität des Saarlandes) zur Testmustergenerierung auf einem realen Chipentwurf eingesetzt. Es handelte sich dabei um den Prozessorchip der SB–PRAM [ADKP+93].

Für diese Aufgabe erwies sich der Testmustergenerator COAT aufgrund seiner komplexen Basiszellen und der Verwendung des Zellenfehlermodells als besonders geeignet:

1. Beim Entwurf des Prozessorchips waren in der Tat Gattertypen wie z.B. Voll– und Halbaddierer verwendet worden; also solche, deren Komplexität die der 2–Input–Standardgatter (AND/NAND, OR/NOR) überschreitet und die deshalb von vielen ATPG–Systemen nicht unterstützt werden.
2. Bei der Fehlermodellierung hatte man sich bewußt für das Zellenfehlermodell entschieden, da ein reiner stuck_at–Test des Prozessorchips von der PRAM–Gruppe als nicht ausreichend betrachtet wurde.

Mit Unterstützung des Testmustergenerators COAT und des Fehlersimulators COFS [HSS92] sowie unter Ausnutzung der Resultate aus [Spa91] gelang es Robert Knuth [Knu95], für den Prozessorchip der SB–PRAM eine Fehlerüberdeckung von **100 %** auf der Basis des Zellenfehlermodelles zu erreichen.

COAT im HIT–System

Zur Untersuchung des in Kapitel 6 beschriebenen Vorgabenkonzepts wurde der Testmustergenerator COAT mit dem Fehlersimulator COFS und der graphischen Oberfläche *Testshell* [Kra94] zum integrierten Testwerkzeug HIT [BHHK+94] zusammengefaßt. Die Testshell ermöglichte so ein steuerndes Eingreifen sowohl in den Fehlersimulations– als auch in den Testmustergenerierungsprozeß.
Am Beispiel einer 32–Bit Gleitkommaarithmetik (Mantissenteil eines Gleitkommaadierers) wurde untersucht, wie sich der Einsatz von Vorgaben auf den Testgenerierungsablauf auswirkt.
Zunächst wurde ein rein automatischer Programmlauf durchgeführt; die Ergebnisse dieser Untersuchung sind in Tabelle 7.1 dargestellt. Wie man sieht, waren die rein automatischen Verfahren auch nach 27 Stunden Laufzeit nicht in der Lage, eine Fehlerüberdeckung von über 97.77 % zu erreichen.
Problematisch an den Ergebnissen in der Tabelle ist jedoch vor allem der exponentielle Anstieg der Rechenzeit: Beim Anheben der Laufzeitschranke (beschrieben durch die Anzahl der Backtracks) von 1000 auf 10.000 stieg die Fehlerüberdeckung nur noch um $0,11\%$, während sich die Laufzeit fast verzehnfachte! Aufgrund dieser Entwicklung ist auch bei einer weiteren Anhebung der Laufzeitschranke und Laufzeiten von mehreren Tagen oder sogar Wochen keine wesentliche Ergebnisverbesserung mehr zu erwarten (was experimentelle Ergebnisse im übrigen bestätigen).

eingesetztes Programmodul	Anzahl Backtracks	erreichte Fehler–überdeckung	benötigte Laufzeit
COFS	—	94,93 %	6,7 min
COAT	10	97,32 %	6,8 min
COAT	100	97,46 %	22,4 min
COAT	1000	97,66 %	2 h 57 min
COAT	10000	97,77 %	27 h

Tabelle 7.1: Laufzeitergebnisse ohne Steuervorgaben

Ganz anders die Situation beim Einsatz der interaktiven Steuermöglichkeiten des HIT–Systems. Um einen direkten Vergleich zum Einsatz der automatischen Verfahren (s.o.) zu ermöglichen, wurden die HIT–Tools auf dasselbe Ausgangsproblem angesetzt (gleicher Schaltkreis, gleiche Fehlerliste).
Ich will auf die Einzelheiten dieser Untersuchung hier nicht näher eingehen; Vorgehen und Meßergebnisse sind detailliert in [BHHK+94] beschrieben. Ich beschränke mich auf das wesentliche Resultat: Schon durch einfache Steuervorgaben an den Testgenerierungsprozess war es möglich, eine *100 %–ige Fehlerüberdeckung bei weniger als einer Stunde Rechenzeit des Algorithmus* zu erzielen.

Bei diesem Ergebnis gilt es natürlich zu berücksichtigen, daß der Gesamtzeitaufwand für diesen Testvorgang über die reine Rechenzeit hinausgeht. Der Testingenieur benötigt z.B. zusätzliche Zeit, um die Ergebnisse des ATPG–Prozesses zu analysieren, Strategien zur Benutzersteuerung zu entwickeln und diese in Vorgaben umzusetzen.
Es zeigte sich aber, daß sich diese Arbeiten von einem im Umgang mit den HIT–Tools erfahrenen Testingenieur ohne weiteres innerhalb eines Tages durchführen lassen. Damit benötigt das interaktive Verfahren wesentlich weniger Zeit als das rein automatische Verfahren.

Insgesamt gesehen kann mit steuernden Vorgaben eine wesentliche Beschleunigung des Testgenerierungsprozesses erreicht werden.

Kapitel 8

Zusammenfassung

Ich habe in der vorliegenden Arbeit den Testmustergenerator COAT beschrieben. Die zentralen Punkte meiner Arbeit fasse ich im folgenden kurz zusammen.

Bei der Entwicklung des Testmustergenerators COAT war ein vordringliches Ziel die **Verallgemeinerung des ATPG–Konzeptes.** Ich setzte dabei folgende Schwerpunkte:

1. Das *beliebige kombinatorische Zellenfehlermodell* wurde auf seine Eignung zur automatischen Testmustergenerierung untersucht. Die Überlegenheit des Zellen– gegenüber dem stuck_at–Fehlermodell wurde gezeigt. Ferner wurde erläutert, daß die Verwendung des Zellenfehlermodelles nicht zwangsläufig zu einem höheren Laufzeitaufwand führen muß.

2. COAT *unterstützt komplexe Basiszellen.* Auch komplexere Gattertypen, wie etwa Multiplexer oder Volladdierer, lassen sich als Basiszellen definieren und in den ATPG–Ablauf einbinden. Man erspart sich dadurch die Konvertierung des Schaltung in Standardgatter–Darstellung, deren Nachteile diskutiert wurden.

3. Zur Darstellung von Schaltkreisbelegungen wurde in COAT die *16–wertige Logik nach Sheldon B. Akers* verwendet. Ihre Vorteile gegenüber der üblicherweise verwendeten 5–wertigen Roth–Logik wurden eingehend besprochen. Ausführlich wurde dargestellt, welche Auswirkungen die Verwendung der Akers–Logik auf den ATPG–Algorithmus hat.

Durch das veränderte Konzept von COAT wurde eine **Anpassung der ATPG–Algorithmen** an die andere Logik, das Zellenfehlermodell usw. nötig. Folgende Punkte sind dabei zu nennen:

1. Ein Großteil der ATPG–Heuristiken und Prozeduren aus bekannten Testmustergeneratoren wie PODEM [Goel81], FAN [Fuji83] oder SOKRATES [STS87] wurden an das verallgemeinerte ATPG–Konzept angepaßt und in COAT integriert.

2. Am Beispiel der Implikationsalgorithmen wurde ausführlich gezeigt, wie sich Algorithmen durch obige Anpassung verändern und welche Auswirkungen das auf den ATPG–Prozeß hat. Es wurde gezeigt, daß insbesondere die Verwendung der Akers–Logik Effizienzsteigerungen ermöglicht, eine harmonischere Darstellung der Theorie erlaubt und teilweise sogar die Leistungsfähigkeit der Algorithmen steigert. Letzteres gilt insbesondere für die Prozeduren zur Gatterimplikation, zum Auffinden (vor allem dynamischer) Dominatoren und zum Lernen (unterhalb der Fehlerstelle).

Der dritte Schwerpunkt meiner Ausführungen lag auf der **Ausnutzung von high–level–Wissen des Schaltkreisentwerfers.** An mehreren Beispielen habe ich gezeigt, warum die Ausnutzung dieses Wissen für die Testmustergenerierung von Vorteil sein kann.

Ausführlich habe ich erläutert, wie das high–level–Wissen des Schaltkreisentwerfers mittels *Vorgaben* an den ATPG–Algorithmus übergeben werden kann. Das Vorgabenkonzept von COAT wurde beschrieben und durch mehrerer Anwendungsbeispiele verdeutlicht. Verschiedene Spezifikationsmöglichkeiten und deren Implementierung wurden beschrieben. — Anhand von Laufzeitmessungen wurde erläutert, in welcher Weise Vorgaben den ATPG–Prozeß beschleunigen.

Literaturverzeichnis

[ADKP+93] F. Abolhassan, R. Drefenstedt, J. Keller, W. J. Paul, D. Scheerer. *On the Physical Design of PRAMs.* Comput. J. Dezember 1993

[ABF90] M. Abramovici, M. A. Breuer, A. D. Friedman *Digital Systems. Testing and Testable Design.* Computer Science Press, New York 1990

[Ake76] S. B. Akers, Jr. *A Logic System for Fault Test Generation.* IEEE Transactions on Computers, Juni 1976

[BHKM+87] B. Becker, G. Hotz, R. Kolla, P. Molitor und H. G. Osthof. *Hierarchical design based on a calculus of nets.* ACM/IEEE Design Automation Conference (DAC) 1987

[Blum85] N. Blum. *Fehlererkennung in kombinatorischen Schaltkreisen — revidierte Fassung.* Technischer Bericht des SFB 124, Fachbereich 10, Universität des Saarlandes Saarbrücken 1985

[BF85] F. Brglez, H. Fujiwara. *A Neutral Netlist of 10 Combinational Benchmark Circuits and a Target Translator in Fortran.* IEEE International Symposium on Circuits an Systems; Special Session on ATPG and Fault Simulation, Juni 1985

[BPH84] F. Brglez, P. Pownall, P. Hum. *Applications of testability analysis: From ATPG to critical path tracing.* International Test Conference 1984

[Bur95] T. Burch. *Eine graphische Arbeitsumgebung für das VLSI–Entwurfssystem CADIC.* Dissertation, Universität des Saarlandes, Saarbrücken 1995

[BHHK+94] T. Burch, J. Hartmann, G. Hotz, M. Krallmann, U. Nikolaus, S. M. Reddy, U. Sparmann. *A Hierarchical Environment for Interactive Test Engineering.* International Test Conference 1994

[CP89] S. J. Chandra, J. H. Patel. *Experimental Evaluation of Testability Measures for Test Generation.* IEEE Transactions on Computer–Aided Design, Januar 1989

[FS88] F. J. Ferguson, J. P. Shen. *Extraction and simulation of realistic CMOS faults using inductive fault analysis.* International Test Conference 1988

[Fuji83] H. Fujiwara, T. Shimono. *On the Acceleration of Test Generation Algorithms.* IEEE Transactions on Computers, Dezember 1983

[Goel81] P. Goel. *An Implicit Enumeration Algorithm to Generate Tests for Combinational Logic Circuits.* IEEE Transactions on Computers, März 1981

[Gol79] L. H. Goldstein. *Controllability/Observability Analysis of Digital Circuits.* IEEE Transactions on Circuits and Systems, September 1979

[GMOD90] T. Gruening, U. Mahlstedt, C. Oeczan, W. Daehn. *Accelerated test pattern generation by cone-oriented circuit partitioning.* Proc. European Design Automation Conference 1990

[Har92] J. Hartmann. *Analyse und Verbesserung der probabilistischen Testbarkeit kombinatorischer Schaltungen.* Dissertation. Universität des Saarlandes, Saarbrücken, April 1992

[HSS92] J. Hartmann, B. Schieffer, U. Sparmann. *Cell oriented fault simulation.* Proceedings of the European Simulation Multiconference 1992

[HWA95] M. Henftling, H. Wittmann, K. J. Antreich. *A Formal Non-Heuristic ATPG-Approach.* EURO-DAC 1995

[Hotz65] G. Hotz. *Eine Algebraisierung des Syntheseproblems für Schaltkreise.* EIK Journal of Information Processing and Cybernetics 1965

[KLG93] M. H. Konijnenburg, J. TH. von der Linden, A. J. van de Goor. *Test Pattern Generation with Restrictors.* International Test Conference 1993

[Knu95] R. Knuth. *Produktionstest des Prozessors der SB–PRAM.* Diplomarbeit. Universität des Saarlandes, Saarbrücken, Februar 1995

[Kra94] M. Krallmann. *TestShell – Eine Benutzer– und Entwicklungsumgebung zur interaktiven, hierarchischen Testmustergenerierung.* Diplomarbeit. Universität des Saarlandes, Saarbrücken, September 1994

[Kre93] S. Kremer. *Fehlersimulation in sequentiellen Schaltkreisen.* Diplomarbeit. Universität des Saarlandes, Saarbrücken, Februar 1993

[KP92] W. Kunz, K. Pradhan. *Recursive Learning: An attractive alternative to the decision tree for test generation in digital circuits.* International Test Conference 1992

[KP93] W. Kunz, K. Pradhan. *Accelerated Dynamic Learning for Test Pattern Generation in Combinational Circuits.* IEEE Transactions on Computer–Aided Design of Integrated Circuits and Systems, Mai 1993

[MA93] U. Mahlstedt, J. Alt. *Simulation of non–classical Faults on the Gate Level — The Fault Simulator COMSIM .* International Test Conference 1993

[Mil85] B. Milne. *Testability, 1985 technology forecast.* Electronic Design 1985

[Muth76] P. Muth. *A Nine–Valued Circuit Model for Test Generation.* IEEE Transactions on Computers, Juni 1976

[Nik95] U. Nikolaus. *COAT – Ein automatischer Testmustergenerator unter 16–wertiger Logik mit variabler Fehlermodellierung.* Diplomarbeit. Universität des Saarlandes, Saarbrücken, August 1995

[RC90] J. Rajski, H. Cox. *A Method to Calculate Necessary Assignments in Algorithmic Test Pattern Generation.* International Test Conference 1990

[Roth66] J. P. Roth. *Diagnosis of automata failures: A calculus & a method.* IBM Research Development, Juli 1966

[SMTS89] M. Sarfert, R. Markgraf, E. Trischler, M. H. Schulz. *Hierarchical Test Pattern Generation Based on High–Level Primitives.* International Test Conference 1989

[Sch91] B. Schieffer. *Hierarchische Fehlersimulation mit Nicht-Standard-Fehlermodellen.* Diplomarbeit. Universität des Saarlandes, Saarbrücken, 1991

[Schu88] M. H. Schulz. *Testmustergenerierung und Fehlersimulation in digitalen Schaltungen mit hoher Komplexität.* Springer Verlag 1988

[STS87] M. H. Schulz, E. Trischler, M. Sarfert. *SOKRATES: A Highly Efficient Automatic Test Pattern Generator.* International Test Conference 1987

[SA88] M. H. Schulz, E. Auth. *Advanced Automatic Test Pattern Generation and Redundancy Identification Techniques.* 18th Symposium on Fault–Tolerant Computing 1988

[Spa91] U. Sparmann. *Strukturbasierte Testmethoden für arithmetische Schaltkreise.* Dissertation. Universität des Saarlandes, Saarbrücken 1991

[Wun91] H.-J. Wunderlich. *Hochintegrierte Schaltungen: Prüfgerechter Entwurf und Test.* Springer Verlag 1991