für meine Eltern

Hierarchische Fehlersimulation mit Nicht-Standard-Fehlermodellen

Diplomarbeit, vorgelegt von
Björn Schieffer
nach einem Thema von Prof. Dr. Günter Hotz
im Fachbereich 14.1, Informatik,
der Universität des Saarlandes

Hiermit versichere ich an Eides Statt, daß ich diese Arbeit nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe.

Saarbrücken, im Dezember 1991

Inhaltsverzeichnis

Einleitung **ii**

1 Definitionen **1**

2 Fehlermodelle **10**
2.1 Das stuck-at-Modell 10
2.2 Das kombinatorische Zellenfehlermodell 11
2.3 Das sequentielle Zellenfehlermodell 13
2.3.1 Der Automat über Zustandsmengen 18
2.3.2 Eine obere Schranke für die maximale Länge der kürzesten blind homing sequence 19
2.3.3 Eine untere Schranke für die maximale Länge der kürzesten blind homing sequence 21
2.3.4 Kriterien für die Existenz einer blind homing sequence . 26
2.3.5 Erwartete Länge einer blind homing sequence bei gewürfelter Eingabe 29

3 Fehlersimulation **35**
3.1 Nicht-hierarchische Schaltkreise 36
3.1.1 stuck-at-Modell 36
3.1.2 kombinatorische Zellenfehler 38
3.1.3 sequentielle Zellenfehler 40
3.2 Hierarchische Schaltkreise 45
3.2.1 Erwartungen 45
3.2.2 Die intuitive Lösung 45
3.2.3 Topologische Sortierbarkeit 47
3.2.4 Verluste durch Parameterübergabe 50
3.2.5 Schnelle, hierarchische Fehlersimulation 52
3.2.6 Vorteile der Hierarchie 61

Ausblick **65**

Einleitung

Man wird das 20. Jahrhundert später vielleicht einmal das Zeitalter der Miniaturisierung nennen. Ich meine damit weniger die aktuelle politische Situation des Zerfalls größerer Staatskonglomerate. Und ich habe auch nicht im Sinn, über schrumpfende Wohnungsgrößen bei gleichzeitig steigenden Mieten zu sprechen, nein, der Technik möchte ich meine Aufmerksamkeit schenken. Man stelle sich nur einmal die Größe eines Zahnarztbohrers in der Technologie von 1891 vor. Oder als weniger schmerzhaftes Beispiel, wieviele Menschen wären wohl in der Lage, einen programmierbaren Taschenrechner made in republic of Weimar zur Bedienung in die Hand zu nehmen?

Auch und vor allem Computerchips werden immer kleiner und dabei immer leistungsfähiger. So leistungsfähig, daß man sie auch gerne in Bereichen einsetzt, wo das Leben und die Gesundheit von Menschen eine Rolle spielen. Kernkraftwerke, Mondraketen oder Hochgeschwindigkeitszüge sollten doch bitteschön gemäß ihrer Konzipierung funktionieren. Dazu ist es nötig, daß die verwendeten Bauteile in einem einwandfreien Zustand sind. Neben dem festen Glauben daran, sie seien es, wäre es recht beruhigend, man könnte dies überprüfen.

Wohlan, da ist also so ein kleiner schwarzer Chip mit seinen 200 Eingabe - Beinchen, frisch geliefert von irgendeiner japanischen Firma. Um sicherzugehen, daß er korrekt funktioniert, legen wir einfach alle seine 2^{200} Eingaben an, schauen, was an den Ausgängen rauskommt und vergleichen es mit den Sollwerten. Bis wir damit fertig sind, wird die Menschheit eingesehen haben, daß Atomstrom nicht nötig, Urlaub auf dem Mond langweilig und langsamere Züge viel umweltfreundlicher sind. Da diese Erkenntnisse heute noch nicht der allgemeinen Einschätzung entsprechen, müssen wir uns wohl oder übel mit dem Test nicht aller Eingabemuster begnügen.

Dann sollten wir aber wenigstens eine „gute Auswahl“ der Testmuster treffen. Zu diesem Zweck hat die Informatik die Testmustergenerierung und die Fehlersimulation erfunden. Warum denn gleich zwei Dinge? Nun, das eine ist gut, aber teuer, das andere billiger und auch nicht schlecht. Etwas präziser ausgedrückt, Testmuster zu gewinnen, ist ein NP - vollständiges Problem; deshalb behilft man sich durch Würfeln von Eingabemustern und Verifizieren durch Simulation, ob sie zu einer „guten Auswahl“ gehören. Was aber ist dabei eine

„gute Auswahl“? Vielleicht eine Menge von Mustern, die alle möglicherweise auftretenden Fehler entdecken? Nicht ganz, denn leider ist man in vernünftiger Zeit erst in der Lage, Mengen von Mustern zu finden, die *fast* alle möglicherweise auftretenden Fehler entdecken. Für den Rest der Fehler bleibt ja immer noch der feste Glaube, ... (siehe oben).

Wobei sich die Frage aufdrängt, was denn eigentlich „alle möglicherweise auftretenden Fehler“ sind? „Alles“ zu Beschreiben ist unmöglich, beschränken wir uns also mit einem Modell der Wirklichkeit und sprechen von „vielen möglicherweise auftretenden Fehlern“ – mit einem weiteren Appell an den festen Glauben ... In der Wirklichkeit werden die Störungen oft durch Staubkörner verursacht, die sich während einer Belichtung zwischen Maske und Chip verirrt haben. Leiten wir also daraus das Staubkorn - Fehlermodell ab: *Ein Fehler ist ein etwa kreisrunder Fleck in einer Struktur des Chips.* Es gibt dann so ungefähr unendlich viele Fehler auf einem Chip. Ein wenig zu viele, um damit sinnvoll zu arbeiten. Man benötigt also ein einfacheres Modell. Je einfacher, desto leichter zu Handhaben, aber auch desto weiter weg von der Wirklichkeit! Das sehr gebräuchliche stuck-at-Modell läßt sich ganz gut handhaben ... Deshalb beschäftigt sich der erste Teil meiner Arbeit mit den Problemen einer feineren Modellierung von Fehlern.

Der zweite greift etwas auf, das ich schon im ersten Satz dieser Einleitung erwähnt habe, die Miniaturisierung. Die führt nämlich dazu, daß die auf die Chips gebrannten Schaltungen immer umfangreicher werden. Weshalb wiederum die Algorithmen für Testgenerierung und Fehlersimulation immer mehr Zeit beanspruchen. Sie bedürfen daher einer Weiterentwicklung, um mit neuen Generationen von Schaltungen Schritt zu halten. Da Testen eine sehr zeitaufwendige Aufgabe ist, stellt man die Laufzeit der Algorithmen gegenüber ihrem Speicherbedarf oft in den Vordergrund. Deshalb wurde deren hierarchischer Aufbau, der das Design großer Schaltungen erst ermöglicht, nicht genutzt. Die höheren Laufzeitkosten eines hierarchischen Algorithmus stufte man als zu teuer gegenüber dem daraus erhofften Platzgewinn ein. Ich will für die hierarchische Fehlersimulation zeigen, daß die Laufzeitkosten gar nicht so hoch sind, wie bisher angenommen.

Gliederung der Arbeit

Formale Beschreibungen geben eine präzisere Sicht als informale. Sie leiden aber darunter, daß sie die zugrundeliegenden Ideen bestens verschleiern können. Ich habe mich deshalb bemüht, sowohl informal als auch formal zu argumentieren. Um den dabei entstehenden Mischmasch nicht zu übertreiben, habe ich die eher grundlegenden Definitionen in ein eigenes, erstes Kapitel verlagert, das die Freunde des Informalen getrost überspringen können.

Das zweite Kapitel wendet sich der Modellierung von Fehlern zu. Es stellt dazu zuerst das weit verbreitete stuck-at-Modell vor, um dann über das kombinatorische Zellenfehlermodell zu dem mächtigeren sequentiellen Zellenfehler-

modell zu gelangen. Der zentrale Begriff des zweiten Kapitels ist der Begriff der *blind homing sequence*. Es wird gezeigt, daß er die aufwendige Verarbeitung von sequentiellen Fehlern entscheidend vereinfachen kann und Verfahren zur Vorberechnung wichtiger Fehlereigenschaften wie der kürzesten blind homing sequence oder der erwarteten Länge einer blind homing sequence bei gewürfelten Eingaben werden vorgestellt.

Das dritte Kapitel beschäftigt sich intensiv mit der Fehlersimulation. Dabei wird zunächst die Einbindung der im Kapitel 2 vorgestellten Fehlermodelle diskutiert, um dann eine Verallgemeinerung eines Algorithmus für kombinatorische Schaltkreise auf hierarchische Beschreibungen vorzustellen. Es wird gezeigt, daß eine „intuitive“ Vorgehensweise zu einer quadratischen Laufzeit für die Richtigsimulation führt. Die wichtigste Aufgabe des dritten Kapitels wird daher darin bestehen, zu zeigen, daß und wie man dies auch in Linearzeit lösen kann!

Den Abschluß soll ein kurzer Ausblick auf sich neu ergebende Probleme und damit verbundene zukünftige Aufgaben bilden.

Danksagungen

Ich möchte mich an dieser Stelle bei Herrn Prof. Dr. Günter Hotz für das Überlassen dieses spannenden Themas und seine interessanten Anregungen auf Schloß Dagstuhl bedanken. Ebenso gilt mein Dank meinen beiden Betreuern, Herrn Diplominformatiker Joachim Hartmann und Herrn Dr. Uwe Sparmann, für ihr aufopferungsvolles Schaffen in den vielen fruchtbaren Diskussionen und für die Sorgfalt beim Korrekturlesen der Arbeit.

Kapitel 1

Definitionen

Dieses Kapitel ist zum Verständnis der Arbeit nicht zwingend notwendig. Ich habe mich bemüht, die informalen Ideen mehr in den Vordergrund zu stellen. Dennoch erwähne ich des öfteren wenigstens die formalen Hintergründe der getroffenen Aussagen. Da ich die Gedankengänge aber nicht ständig durch Definitionen unterbrechen will, bitte ich den interessierten Leser hier nachzuschlagen.

Definition 1.1 (orientierter Graph)
Ein orientierter Graph G *ist ein Viertupel* (V, E, Q, Z) *über der Menge* V *der Ecken, der Menge* E *der Kanten und den beiden Funktionen* $Q : E \to V$, $Z : E \to V$, *die den Kanten ihre Quelle und ihr Ziel zuordnen.*

Definition 1.2 (endlicher Automat)

1. *Ein* endlicher Automat (mit Ausgabe) *ist ein Siebentupel* $(G, X^*, Y^*, S, F, \alpha, \beta)$, *bestehend aus einem Graphen* $G = (V, E, Q, Z)$, *zwei freien Monoiden* X^* *und* Y^* *über dem Eingabealphabet* X *und dem Ausgabealphabet* Y, *den Startzuständen* $S \subseteq V$, *den Endzuständen* $F \subseteq V$, *sowie den beiden Kantenbeschriftungen* $\alpha : E \to X^*$ *und* $\beta : E \to Y^*$.
2. *Ein endlicher Automat* $(G, X^*, Y^*, S, F, \alpha, \beta)$, *über dem Graphen* $G = (V, E, Q, Z)$ *heißt* deterministisch, *genau dann wenn*
   - $|S| = 1$
   - $\forall s_1, s_2 \in E, Q(s_1) = Q(s_2), \alpha(s_1) = \alpha(s_2) \Rightarrow s_1 = s_2$
   - $\alpha(E) \subseteq X$
3. *Ein endlicher Automat* $(G, X^*, Y^*, S, F, \alpha, \beta)$, *über dem Graphen* $G = (V, E, Q, Z)$ *heißt* vollständig, *genau dann wenn*

$$\forall P \in V \text{ und } \forall x \in X \ \exists s \in E \text{ mit } Q(s) = P \text{ und } \alpha(s) = x$$

4. *Jedes Eingabewort $\omega \in X^*$ eines deterministischen vollständigen endlichen Automaten $(G, X^*, Y^*, S, F, \alpha, \beta)$, mit $G = (V, E, Q, Z)$ definiert eine Zustandsübergangsfunktion $\delta^\omega : V \to V$ durch*

$$\delta^\omega(P) = \begin{cases} P & : \quad \omega = \epsilon \\ P_1 & : \quad \omega = \omega_1 x, \exists s \in E, \alpha(s) = x, Q(s) = \delta^{\omega_1}(P), Z(s) = P_1 \end{cases}$$

Ich benutze auch manchmal die auf Mengen von Zuständen erweiterte Funktion $\delta^\omega : \wp(V) \to \wp(V)$

$$\delta^\omega(\{P_1, \cdots, P_r\}) = \{\delta^\omega(P_1), \cdots, \delta^\omega(P_r)\}$$

Definition 1.3 (Schaltkreis)

1. *Ein* Schaltkreis *C ist ein Tupel $(T, M, S, \tau, \rho, type)$, wobei T, M und S endliche Mengen von Typen, Modulen und Signalen sind. $\tau \in M$ hat eine Sonderstellung unter den Modulen. Es repräsentiert den Rand des Schaltkreises C. $\rho = (\rho^{in}, \rho^{out})$ mit $\rho^{in} : M \to S^*$ und $\rho^{out} : M \to S^*$ verknüpft den Rand eines Modules, das heißt seine Ein- und Ausgänge mit Signalen. $type : M \setminus \{\tau\} \to T$ typt ein Modul. Dabei müssen einige „syntaktische" Bedingungen erfüllt sein:*

   - *Es existieren zwei Funktionen $indegree : T \to \mathbf{N}$ und $outdegree : T \to \mathbf{N}$ mit*

$$\forall m \in M : indegree(type(m)) = |\rho^{in}(m)|$$

$$\forall m \in M : outdegree(type(m)) = |\rho^{out}(m)|$$

     Oft verwende ich auch die auf M erweiterten Funktionen indegree und outdegree, für die gilt:

$$indegree(m) = indegree(type(m))$$

$$outdegree(m) = outdegree(type(m))$$

   - *$\forall s \in S : \exists$ genau ein $m \in M$ und $\mu \in \mathbf{N}$ mit*

$$s = \begin{cases} \rho^{in}(m)_\mu & : \quad m = \tau \\ \rho^{out}(m)_\mu & : \quad sonst \end{cases}$$

     Jedes Signal hat genau einen Treiber.

   - *$\forall s \in S : \exists$ genau ein $m \in M$ und $\mu \in \mathbf{N}$ mit*

$$s = \begin{cases} \rho^{out}(m)_\mu & : \quad m = \tau \\ \rho^{in}(m)_\mu & : \quad sonst \end{cases}$$

     Auf jedes Signal wird genau einmal lesend zugegriffen. Fanouts von Signalen müssen als Fanout-Module definiert werden.

- $\forall s \in S\ \exists m_1, m_2 \in M, \mu, \nu$ *mit* $\rho^{out}(m_1)_\mu = s = \rho^{in}(m_2)_\nu$
  Jedes Signal wird sowohl getrieben, als auch gelesen.

2. *Ein* hierarchischer Schaltkreis H *ist eine Folge von Schaltkreisen* $(C_1, C_2, \cdots, C_k)$ *für die gilt:* $C_i = (T_i, M_i, S_i, \tau_i, \rho_i, type_i)$ *und* $\forall i > 1 : T_i := T_{i-1} \cup \{t_{i-1}\}$. *Das heißt, der Schaltkreis* C_i*, für* $i < k$ *manchmal auch als „Box" bezeichnet, darf Module vom Typ* C_1 *bis* C_{i-1} *enthalten. Um sicherzustellen, daß der „äußere" Anschluß eines Modules vom Typ* t_j *im Schaltkreis* C_i *mit den Ein- und Ausgängen von* C_j *übereinstimmt, muß gelten, daß*

$$\forall i > j : indegree_i(t_j) = |\rho_j^{in}(\tau_j)|$$

$$\forall i > j : outdegree_i(t_j) = |\rho_j^{out}(\tau_j)|$$

Hierarchische Schaltungen werden oft mit ihrer „aufgelösten" Version verglichen. Ich möchte daher den Begriff des Auflösens formalisieren. Die „flache" Schaltung einer hierarchischen Beschreibung H wird durch die Funktion $flat(H)$ erzeugt.

$$flat(C_1, \cdots, C_k) = expand(t_k, expand(t_{k-1}, \cdots, expand(t_1, (C_1, \cdots, C_k))))$$

Es werden alle Teilschaltungen nacheinander „expandiert". Dazu dient die Funktion *expand*, die wie folgt definiert ist:

$$expand(t_i, (C_i, \cdots, C_k)) := (subst(t_i, C_{i+1}), subst(t_i, C_{i+2}), \cdots, subst(t_i, C_k))$$

Alle Module vom Typ der Teilschaltung C_i werden durch die Beschreibung von C_i ersetzt.

$$subst(t_i, C_j) := substitute(t_i, C_j, \{m | m \in M_j, type_j(m) = t_i\})$$

$$substitute(t_i, C_j, \{m_1, m_2, \cdots, m_\nu\}) :=$$
$$substitute(t_i, substitute(t_i, C_j, \{m_1, m_2, \cdots, m_{\nu-1}\}), m_\nu)$$

Um ein Modul m vom Typ t_i in einer Schaltung C_j zu ersetzen, benötigt man alle Module von C_j mit Ausnahme von m und eine namensdisjunkte Kopie aller Module von C_i ohne den Rand τ_i. An Signalen übernimmt man alle aus C_j und eine namensdisjunkte Kopie aller aus C_i, die nicht am Rand τ_i anliegen. Die Beschreibung der Verbindungen ρ muß dies berücksichtigen – Verbindungen an Signale, die an den Rand von C_i führen, müssen nun auf die Signale, die an m heranführen, umgeleitet werden.

$$substitute(t_i, C_j, m) := (T_j, M, S, \tau_j, \rho, type)$$

$$M := (M_j \backslash \{m\}) \cup (m \times (M_i \backslash \{\tau_i\}))$$

$$S := S_j \cup (m \times (S_i \backslash \{s | \exists \mu, s = \rho_i(\tau_i)_\mu\}))$$

$$\rho(n)_\mu := \begin{cases} \rho_j(n)_\mu & : \quad n \in M_j \\ \rho_j(m)_\nu & : \quad n = m \times l, \exists \nu \text{ mit } \rho_i(\tau_i)_\nu = \rho_i(l)_\mu \\ (m \times \rho_i(l)_\mu) & : \quad n = m \times l, \nexists \nu \text{ mit } \rho_i(\tau_i)_\nu = \rho_i(l)_\mu \end{cases}$$

$$type(n) := \begin{cases} type_j(n) & : \quad n \in M_j \\ type_i(l) & : \quad n = m \times l \end{cases}$$

Unter der Größe eines Schaltkreises versteht man üblicherweise die Anzahl seiner Signale. Um einen fairen Vergleich zwischen flacher und hierarchischer Beschreibung zu ermöglichen, setze ich die Größe eines hierarchischen Schaltkreises als die Größe seiner Auflösung fest.

Definition 1.4 (Schaltkreisgröße)
Die Schaltkreisgröße size eines Schaltkreises C und eines hierarchischen Schaltkreises H ist gegeben als:

$$size(C) := |S| \text{ wenn } C = (T, M, S, \tau, \rho, type)$$

$$size(H) := size(flat(H))$$

Definition 1.5 (Instanz)
Eine Instanz *eines Modules m_1 der hierarchischen Schaltung $H = (C_1, C_2, \cdots, C_k)$ ist eine Folge $((C_{i_1}, m_1), (C_{i_2}, m_2), \cdots, (C_{i_r}, m_r))$, für die gilt:*

- $1 \leq j \leq r : m_j \in M_{i_j}$
  Die Module m_j sind jeweils Modul von C_{i_j}.
- $1 < j \leq r : type_{i_j}(m_j) = t_{i_{j-1}}$
  Die Module m_j sind vom Typ der Schaltung $C_{i_{j-1}}$.
- $C_{i_r} = C_k$
  Die Folge spezifiziert ein Modul, indem sie sein Vorkommen in den übergeordneten Schaltungen beschreibt. Die Beschreibung ist vollständig, wenn man auf der „obersten Ebene“ – C_k – angelangt ist.

Definition 1.6 (kombinatorischer Schaltkreis)

1. *Ein* kombinatorischer Schaltkreis *ist ein Schaltkreis $C = (T, M, S, \tau, \rho, type)$, auf dessen Modulen eine topologische Sortierung $(M \backslash \{\tau\}, \leq_{top})$ existiert, für die gilt:*

$$m_1 \leq_{top} m_2 \Rightarrow \forall \mu, \nu : \rho^{in}(m_1)_\mu \neq \rho^{out}(m_2)_\nu$$

2. *Ein hierarchischer Schaltkreis* $H = (C_1, C_2, \cdots, C_k)$ *heißt* kombinatorisch, *wenn* $flat(H)$ *ein kombinatorischer Schaltkreis ist.*

Definition 1.7 (Semantik von kombinatorischen Schaltkreisen)
Die Wertemenge eines Signals sei mit $\mathbf{B} := \{0, 1\}$ *bezeichnet. Die* Semantik *eines kombinatorischen mit* $k = 1$ *aber nicht notwendigerweise hierarchischen Schaltkreises* $H = (C_1, C_2, \cdots, C_k)$ *ist dann definiert durch:*

1. *Das Verhalten eines Basistyps wird durch die Funktion* $behave : T_1 \rightarrow (\mathbf{B}^* \rightarrow \mathbf{B}^*)$ *beschrieben.*

2. *Eine* Signalbelegung *eines Schaltkreises* $C = (T, M, S, \tau, \rho^{in}, \rho^{out}, type)$ *ist eine Funktion* $\beta : S \rightarrow \mathbf{B}$. *Abkürzend schreibe ich auch* $\beta(\omega)$ *für* $\beta(s_1)\beta(s_2)\cdots\beta(s_n)$ *falls* $\omega = s_1 s_2 \cdots s_n$. *Eine Signalbelegung* β *heißt* stabil, *genau dann wenn*

$$\forall m \in M \setminus \{\tau\} : \beta(\rho^{out}(m)) = behave(type(m))(\beta(\rho^{in}(m)))$$

3. *Das Verhalten einer Zelle vom Nicht-Basistyp* t_i *ist definiert durch eine Erweiterung der behave-Funktion auf alle Typen einer hierarchischen Schaltung* $H = (C_1, C_2, \cdots, C_k)$*:*

$$behave : T_k \rightarrow (\mathbf{B}^* \rightarrow \mathbf{B}^*)$$

$$behave(t_i) := f \text{ mit } f(\omega_{in}) = \omega_{out} : \Longleftrightarrow$$

$$\exists \text{ stabile Signalbelegung } \beta_i \text{ mit } \beta_i(\rho_i^{in}(\tau_i)) = \omega_{in}, \beta_i(\rho_i^{out}(\tau_i)) = \omega_{out}$$

4. *Die Semantik der Schaltung* H *ist schließlich eine Erweiterung der behave-Funktion auf* H *selbst.*

$$behave(H) := f \text{ mit } f(\omega_{in}) = \omega_{out} : \Longleftrightarrow$$

$$\exists \text{ stabile Signalbelegung } \beta_k \text{ mit } \beta_k(\rho_k^{in}(\tau_k)) = \omega_{in}, \beta_k(\rho_k^{out}(\tau_k)) = \omega_{out}$$

Beispiel

Zum Abschluß dieses Kapitels möchte ich die formale Beschreibung eines Volladdierers als Beispiel vorstellen.

Typen und Verhalten

$$T_1 := \{nand, and, or, fanout\}$$

$$T_2 := T_1 \cup \{t_1\}$$

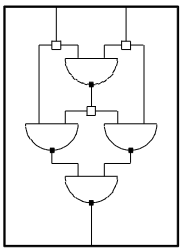

Bild 1: C_1: Xor

$$T_3 := T_2 \cup \{t_2\}$$

$behave(nand)$	$:=$	$\begin{pmatrix} 00\rightarrow 1 \\ 01\rightarrow 1 \\ 10\rightarrow 1 \\ 11\rightarrow 0 \end{pmatrix}$
$behave(and)$	$:=$	$\begin{pmatrix} 00\rightarrow 0 \\ 01\rightarrow 0 \\ 10\rightarrow 0 \\ 11\rightarrow 1 \end{pmatrix}$
$behave(or)$	$:=$	$\begin{pmatrix} 00\rightarrow 0 \\ 01\rightarrow 1 \\ 10\rightarrow 1 \\ 11\rightarrow 1 \end{pmatrix}$
$behave(fanout)$	$:=$	$\begin{pmatrix} 0\rightarrow 00 \\ 1\rightarrow 11 \end{pmatrix}$

Xor – Abbildung 1

$$M_1 := \{\tau_1, nand_1^1, nand_1^2, nand_1^3, nand_1^4, fanout_1^1, fanout_1^2, fanout_1^3\}$$

$$S_1 := \{s_1^1, s_1^2, \cdots, s_1^{12}\}$$

$$\rho_1^{in} := \begin{cases} \tau_1 & \rightarrow s_1^1 s_1^2 \\ fanout_1^1 & \rightarrow s_1^1 \\ fanout_1^2 & \rightarrow s_1^2 \\ nand_1^1 & \rightarrow s_1^4 s_1^5 \\ fanout_1^3 & \rightarrow s_1^7 \\ nand_1^2 & \rightarrow s_1^3 s_1^8 \\ nand_1^3 & \rightarrow s_1^9 s_1^6 \\ nand_1^4 & \rightarrow s_1^{10} s_1^{11} \end{cases} \qquad \rho_1^{out} := \begin{cases} \tau_1 & \rightarrow s_1^{12} \\ fanout_1^1 & \rightarrow s_1^3 s_1^4 \\ fanout_1^2 & \rightarrow s_1^5 s_1^6 \\ nand_1^1 & \rightarrow s_1^7 \\ fanout_1^3 & \rightarrow s_1^8 s_1^9 \\ nand_1^2 & \rightarrow s_1^{10} \\ nand_1^3 & \rightarrow s_1^{11} \\ nand_1^4 & \rightarrow s_1^{12} \end{cases}$$

$$type_1 := \begin{cases} fanout_1^1 & \rightarrow fanout \\ fanout_1^2 & \rightarrow fanout \\ fanout_1^3 & \rightarrow fanout \\ nand_1^1 & \rightarrow nand \\ nand_1^2 & \rightarrow nand \\ nand_1^3 & \rightarrow nand \\ nand_1^4 & \rightarrow nand \end{cases}$$

$$C_1 := (T_1, M_1, S_1, \tau_1, \rho_1, type_1)$$

Der Halbaddierer – Abbildung 2

$$M_2 := \{\tau_2, and_2^1, xor_2^1, fanout_2^1, fanout_2^2\}$$

$$S_2 := \{s_2^1, s_2^2, \cdots, s_2^8\}$$

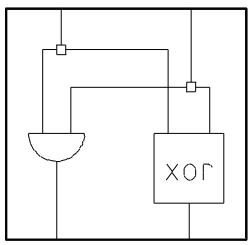

Bild 2: C_2: Halbaddierer

$$\rho_2^{in} := \begin{cases} \tau_2 & \rightarrow s_2^1 s_2^2 \\ fanout_2^1 & \rightarrow s_2^1 \\ fanout_2^2 & \rightarrow s_2^2 \\ and_2^1 & \rightarrow s_2^3 s_2^5 \\ xor_2^1 & \rightarrow s_2^4 s_2^6 \end{cases} \qquad \rho_2^{out} := \begin{cases} \tau_2 & \rightarrow s_2^7 s_2^8 \\ fanout_2^1 & \rightarrow s_2^3 s_2^4 \\ fanout_2^2 & \rightarrow s_2^5 s_2^6 \\ and_2^1 & \rightarrow s_2^7 \\ xor_2^1 & \rightarrow s_2^8 \end{cases}$$

$$type_2 := \begin{cases} fanout_2^1 & \rightarrow fanout \\ fanout_2^2 & \rightarrow fanout \\ and_2^1 & \rightarrow and \\ xor_2^1 & \rightarrow t_1 \end{cases}$$

$$C_2 := (T_2, M_2, S_2, \tau_2, \rho_2, type_2)$$

Der Volladdierer – Abbildung 3

$$M_3 := \{\tau_3, ha_3^1, ha_3^2, or_3^1\}$$

$$S_3 := \{s_3^1, s_3^2, \cdots, s_3^8\}$$

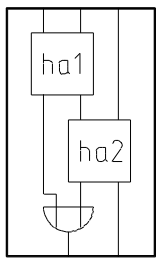

Bild 3: C_3: Volladdierer

$$\rho_3^{in} := \begin{cases} \tau_3 & \rightarrow s_3^1 s_3^2 s_3^3 \\ ha_3^1 & \rightarrow s_3^1 s_3^2 \\ ha_3^2 & \rightarrow s_3^5 s_3^3 \\ or_3^1 & \rightarrow s_3^4 s_3^6 \end{cases} \qquad \rho_3^{out} := \begin{cases} \tau_3 & \rightarrow s_3^8 s_3^7 \\ ha_3^1 & \rightarrow s_3^4 s_3^5 \\ ha_3^2 & \rightarrow s_3^6 s_3^7 \\ or_3^1 & \rightarrow s_3^8 \end{cases}$$

$$type_3 := \begin{cases} ha_3^1 & \rightarrow t_2 \\ ha_3^2 & \rightarrow t_2 \\ or_3^1 & \rightarrow or \end{cases}$$

$$C_3 := (T_3, M_3, S_3, \tau_3, \rho_3, type_3)$$

Damit erhält man die hierarchische Beschreibung $H := (C_1, C_2, C_3)$ eines Volladdierers.

Kapitel 2

Fehlermodelle

Die Abbildung der Wirklichkeit auf einen Computer ist immer mit starken Einschränkungen durch Vereinfachung verbunden. So leidet auch jedes Modell physikalischer Fertigungsfehler eines Chips unter starken Restriktionen. Um dies ein wenig abzumildern, sollte es dem Anwender eines Testsystems freigestellt sein, seine Fehler in einem möglichst weiten Rahmen selbst zu modellieren. Üblicherweise wird ihm heute aber nur das stuck-at-Modell angeboten. Dieses Modell hat aufgrund seiner Einfachheit eine weite Verbreitung gefunden, stößt aber immer wieder auf heftige Kritik. Denn nicht alle physikalisch auftretenden Fehler lassen sich damit beschreiben [BART82, BA83, GCV80, LA83].

Eine exaktere Modellierung verspricht man sich durch Fehlermodellgenerierungstools, die aus Prozeßparametern und statistischen Erfahrungen versuchen, real existierende Fehler vorherzusagen [Mal87, FS88]. Dies kann nicht mehr auf der Ebene von Signalen erfolgen, sondern geschieht vielmehr für die einzelnen Zellen einer Schaltung. Daher rücken kombinatorisches Fehlverhalten von Zellen und vor allem beliebiges Fehlverhalten immer mehr ins Licht des Interesses.

Dieses Kapitel ist in drei Abschnitte gegliedert. In den ersten beiden wird kurz auf das stuck-at-Modell und auf kombinatorische Zellenfehlermodelle eingegangen. Richtig spannend wird es aber erst im dritten Abschnitt bei den sequentiellen Fehlermodellen. Zu diesem Thema gibt es zur Zeit nur Arbeiten, die sich mit Spezialfällen davon befassen. Fehlermodelle mit beliebigem, deterministischen Fehlverhalten wurden bisher noch nicht untersucht!

2.1 Das stuck-at-Modell

Das stuck-at-Modell beschreibt einen Fehler, als das Aufreißen einer Signalleitung. Leitungen in hochintegrierten Schaltungen sind nur noch Bruchteile eines Millimeters dick. Eine kleine Ungenauigkeit im Fabrikationsprozeß kann daher schon dazu führen, daß eine Verbindung nicht korrekt arbeitet. Bei einem Zu-

griff auf eine solche zerstörte Leitung verhält sie sich, als trage sie konstant den logischen Wert Null oder Eins, ein Wechsel ist nicht zu sehen.

Eine Schaltung $C = (T, M, S, \tau, \rho, type)$, die einen stuck-at-$x$-Fehler an einem Signal $s \in S$ hat, wird formal durch die Schaltung $C_{sa}(s, x)$ beschrieben:

$$C_{sa}(s,x) := (T \cup \{t_{err}\}, M \cup \{m_{err}\}, S \cup \{s_{err}\}, \tau, \sigma, type[m_{err} \rightarrow t_{err}])$$

$$\sigma^{in}(m)_\mu = \begin{cases} s & : \quad m = m_{err} \\ s_{err} & : \quad m \neq m_{err},\ \rho^{in}(m)_\mu = s \\ \rho^{in}(m)_\mu & : \quad \text{sonst} \end{cases}$$

$$\sigma^{out}(m) = \begin{cases} s_{err} & : \quad m = m_{err} \\ \rho^{out}(m) & : \quad \text{sonst} \end{cases}$$

$$behave(t_{err}) := \left\{ \begin{array}{l} 0 \rightarrow x \\ 1 \rightarrow x \end{array} \right\}$$

Das Signal s wird also aufgebrochen und ein Modul m_{err} vom Typ t_{err} eingefügt, das den Ausgangswert des Moduls m_{err} und damit auch des Signals s_{err} immer auf dem Wert x hält. Hier macht es sich bezahlt, die Fanouts in eigene Module zu verlagern. Ansonsten müßte man nämlich zwischen Eingangsstämmen und Ausgangsstämmen von Signalen unterscheiden.

Ein Testmuster eines stuck-at-Fehlers ist eine Eingabe an den Schaltkreis, die im Vergleich zum fehlerhaften Schaltkreis eine unterschiedliche Ausgabe hervorruft. Die Menge aller Testmuster TM_{sa} für einen stuck-at-x-Fehler eines Signals s der Schaltung C ergibt sich damit zu:

$$TM_{sa}(C,s,x) := \{tm \in \mathbf{B}^{indegree(\tau)} | behave(C)(tm) \neq behave(C_{sa}(s,x))(tm)\}$$

2.2 Das kombinatorische Zellenfehlermodell

Die Zellbibliotheken moderner Designsysteme enthalten immer komplexere Zellen. Das stuck-at-Modell wird dafür immer unpräziser, da es nur Fehler am Rand solcher komplexen Zellen betrachtet. Nehmen wir als Beispiel das Xor-Gatter aus Abbildung 1 auf Seite 6. Die Zelleingabe 00 führt zu einem unterschiedlichem Verhalten der korrekten Zelle gegenüber den mit stuck-at-1-Fehlern an einem der Eingänge oder dem Ausgang behafteten Zellen. Mit den Zelleingaben 01 und 10 lassen sich die noch fehlenden stuck-at-0-Fehler der Eingänge und des Ausganges finden. Auf der Basis des stuck-at-Modells ist das Xor-Gatter damit vollständig getestet. Den stuck-at-1-Fehler am Ausgang des obersten Nandgatters übersieht man dabei! Um ihn zu testen, muß der Wert am Ausgang des Nandgatters Null betragen und damit die Eingänge beide auf Eins gesetzt sein. Das ist aber bei allen „Testmustern“ aus $\{00, 01, 10\}$ nicht der Fall.

Kombinatorische Zellenfehlermodelle lassen hingegen eine beliebige, funktionale Abweichung im Verhalten einer Basiszelle zu. Damit steht dem Anwender

Schaltkreis	stuck-at	Zellenfehler	Zellenfehler(*)
c1908	94.4 %	80.6 %	79.0 %
c6288	99.6 %	82.6 %	81.9 %
c7552	91.8 %	77.6 %	77.1 %
im16	69.7 %	57.5 %	57.1 %

Tabelle 2.1: Fehlerüberdeckungen

ein wesentlich mächtigeres Werkzeug zur Verfügung. Selbstverständlich lassen sich alle stuck-at-Fehler auch als kombinatorische Zellenfehler beschreiben.

Es wäre allerdings möglich, daß sich viele physikalische Fehler, selbst wenn sie sich nicht korrekt im stuck-at-Modell beschreiben lassen, dennoch durch andere stuck-at-Fehler und deren Testmuster mitentdecken ließen. Dazu möchte ich folgende Ergebnisse aus [SS90] zitieren: An drei Benchmarkschaltungen aus [BF85] und einem Integermultiplizierer für 16-Bit-Zahlen aus [Spa91] wurden die Fehlerüberdeckungen nach 1024 zufälligen, aber jeweils gleichen Testmustern für das stuck-at-Modell und für das vollständige kombinatorische Zellenfehlermodell miteinander verglichen. Vollständig soll hier bedeuten, daß alle Fehlverhalten einer Zelle angenommen werden können. Als dritte Messung (*) wurde die Überdeckung des Zellenfehlermodells nur mit den „erfolgreichen“ Mustern des stuck-at-Modells ermittelt. Sie ist deutlich niedriger und belegt damit, daß eine Testmusterauswahl mithilfe des stuck-at-Modells sinnvolle Muster übersieht.

Formal ist ein kombinatorischer Zellenfehler für ein Modul vom Typ t eine Funktion $f : \mathbf{B}^{indegree(t)} \rightarrow \mathbf{B}^{outdegree(t)}, f \neq behave(t)$. Ein an Modul m mit Fehler f behafteter Schaltkreis $C = (T, M, S, \tau, \rho, type)$ ergibt sich dann als:

$$C_{komb}(m, f) := (T \cup \{t_{err}\}, M, S, \tau, \rho, type[m \rightarrow t_{err}])$$

$$behave(t_{err}) := f$$

Ein Testmuster eines kombinatorischen Fehlers f eines Moduls m ist analog den stuck-at-Fehlern eine Eingabe an den Schaltkreis C, die im Vergleich zum fehlerhaften Schaltkreis eine unterschiedliche Ausgabe hervorruft. Analog ergibt sich die Menge aller Testmuster TM_{komb}.

$$TM_{komb}(C, m, f) :=$$

$$\{tm \in \mathbf{B}^{indegree(\tau)} | behave(C)(tm) \neq behave(C_{komb}(m, f))(tm)\}$$

In hierarchischen Schaltkreisen sind nicht einzelne Module fehlerbehaftet, sondern Instanzen von Basiszellen. Da zur Untersuchung von Fehlermodellen flache Schaltkreise genügen, erspare ich mir die aufwendige, formale Definition.

2.3 Das sequentielle Zellenfehlermodell

Auch wenn kombinatorische Zellenfehler eine Verbesserung gegenüber den stuck-at-Fehlern darstellen, so unterliegen sie dennoch der Einschränkung, nur kombinatorisches Fehlverhalten zu beschreiben. Die nächste Stufe auf dem Weg zu einer möglichst realistischen Fehlermodellierung stellen die sequentiellen Zellenfehler dar, die es auch erlauben, Fehlverhalten in Abhängigkeit von zeitlich früheren Eingaben zu definieren. Genauso wie das kombinatorische Zellenfehlermodell ein Obermodell der stuck-at-Fehler ist, können durch sequentielle Zellenfehler alle kombinatorischen Zellenfehler beschrieben werden.

Bevor ich auf die sequentiellen Fehler näher eingehe, möchte ich diese durch ein Beispiel motivieren: Wie so oft, muß wieder der Volladdierer dran glauben – dieses Mal erhält er aber einen Platz unter den Basiszellen.

Was passiert, wenn die Leiterbahn eines der drei Eingänge eine leitende Verbindung zum Carry-Ausgang hat? (Solche Fehler nennt man „bridging faults“.) Der Volladdierer ist dann kein kombinatorischer Schaltkreis mehr, er hat einen eingebauten Flip-Flop! Um das Verhalten des fehlerhaften Volladdierers zu untersuchen, muß man klären, wie der Eingang einer Zelle darauf reagiert, wenn er von mehreren Treibern gespeist wird. Da dies technologieabhängig ist, wähle ich als Resultat mehrerer Treiber die Undverknüpfung ihrer Werte. Dann wird bei allen Zelleingaben, bei denen der „richtige“ Treiber den Wert Null liefert, sich dieser gegen die fehlerhafte Rückkoppelung vom Carry-Ausgang durchsetzen. Die Zelle verhält sich korrekt und es läßt sich der Wert jedes Signals eindeutig bestimmen. Ebenso harmlos wie eine Null am rückgekoppelten Eingang ist eine Eins am Carry-Ausgang. Bei anderen Zelleingaben kann es aber sein, daß die Eins des richtigen Treibers von der Null des Carry-Ausgangs überschrieben wird. Damit bleiben, wenn man den fehlerhaften Eingang als die linke Position betrachtet, nur noch die Eingaben 101 und 110 als kritisch übrig. Wechselt die Zelleingabe von 111 auf 101, dann verhält sich die Zelle korrekt. Denn durch die Eingabe 111 wird der Carry-Ausgang auf Eins gezwungen. Diese Eins stört die Eingabe 101 nicht. Eine andere Situation ergibt sich beim Wechsel der Zelleingabe von 001 auf 101: Dann wird die Null des Carry-Ausgangs, die sich durch 001 einstellt, der Zelle auch während der Eingabe 101 der Zelle weiterhin die Eingabe 001 „vortäuschen“.

Das Verhalten des fehlerhaften Volladdierers wird in Tabelle 2.2 beschrieben. Mögliche „Zustände“ sind dabei Kombinationen aus Zelleingabe und dem Wert des Carry-Ausgangs. Dieser kann nicht nur Null oder Eins, sondern auch „unbekannt“ (*) sein. Denn wenn bei einem Wechsel der Zelleingabe mehr als ein Eingang kippt, geschieht dies nur idealisiert gleichzeitig. Man muß dann alle möglichen Reihenfolgen beachten. Daher kann es vorkommen, daß man den Folgezustand nicht sicher kennt. Da einige Kombinationen nicht vorkommen und bei manchen Eingaben die Vorgeschichte keine Rolle spielt, sind nur die kritischen Zustände beschrieben. Ein $\otimes$ in Zeile i und Spalte j bedeutet, daß vom Zustand i mit der Eingabe, die zu j korrespondiert, zum Zustand j gewechselt

	101,0	101,1	101,*	110,0	110,1	110,*
000,0	⊗			⊗		
001,0	⊗					⊗
010,0			⊗	⊗		
011,1			⊗			⊗
100,0	⊗			⊗		
101,0	⊗					⊗
101,1		⊗				⊗
101,*			⊗			⊗
110,0			⊗	⊗		
110,1			⊗		⊗	
110,*			⊗			⊗
111,1		⊗			⊗	

Tabelle 2.2: Volladdierer mit bridging-Fehler

wird.

Durch Verschmelzen der Zustände mit dem gleichen Ausgangsverhalten erhält man Tabelle 2.3. Kennt man den Wert des carry-Ausgangs nicht genau, muß man immer davon ausgehen, daß der richtige Wert anliegt, die Zelle also unter diesem Muster nicht testbar ist. Dann lassen sich aber die Zustände mit unbekanntem Carry-Ausgang mit denen des korrekten Wertes verschmelzen, den auch das Ausgangsverhalten ist gegenüber allen anderen Zuständen gleich. Damit erhält man Tabelle 2.4, einen endlichen Automaten mit vier Zuständen.

In dem Beispiel des Volladdierers klingt schon an, daß endliche Automaten mit Ausgabe eine vernünftige Beschreibungsmöglichkeit für sequentielle Fehler sind. Die Ein- und Ausgabe des Automaten entspricht dabei der Ein- Ausgabe der Zelle. Der Folgezustand des Automaten muß für jede Zelleingabe definiert und eindeutig sein. Dies entspricht fast einem deterministischen vollständigen Automaten. Dieser besitzt zusätzlich noch einen eindeutigen Startzustand. Und genau das fehlt einem sequentiellen Fehler! Will man einen gefertigten Chip im Testautomaten prüfen, dann kennt man den Zustand eines Fehlers beim Starten des Tests nicht!

Definition 2.1 (sequentieller Fehler)
Ein sequentieller Fehler $f = (G, X^*, Y^*, V, \emptyset, \alpha, \beta)$ *ist ein endlicher Automat über einem Graphen* $G = (V, E, Q, Z)$*, für den gilt: Durch Einschränken der Startzustände auf einen einzelnen Zustand* $P \in V$ *ergibt sich ein vollständiger deterministischer endlicher Automat* $(G, X^*, Y^*, P, \emptyset, \alpha, \beta)$*. Das Verhalten der fehlerhaften Zelle wird dabei durch die Ausgabe des endlichen Automaten bestimmt.*

	101,0	101,1	101,*	110,0	110,1	110,*
$000,0$ $100,0$	$\otimes$			$\otimes$		
$001,0$ $101,0$	$\otimes$					$\otimes$
$010,0$ $110,0$			$\otimes$	$\otimes$		
$011,1$ $101,*$ $110,*$			$\otimes$			$\otimes$
101,1		$\otimes$				$\otimes$
110,1			$\otimes$		$\otimes$	
111,1		$\otimes$			$\otimes$	

Tabelle 2.3: Volladdierer mit bridging-Fehler

	101,0	$101,1$ $101,*$	110,0	$110,1$ $110,*$
$000,0$ $100,0$	$\otimes$		$\otimes$	
$001,0$ $101,0$	$\otimes$			$\otimes$
$010,0$ $110,0$		$\otimes$	$\otimes$	
$011,1$ $101,*$ $110,*$ $101,1$ $110,1$ $111,1$		$\otimes$		$\otimes$

Tabelle 2.4: Volladdierer mit bridging-Fehler

Ein erfolgreicher Test für einen kombinatorischen Zellenfehler erfordert eine bestimmte Eingabe an den Zelleingängen und das Propagieren der fehlerhaften Zellausgänge. Ein Test für sequentielle Zellenfehler ist nun nicht mehr ein einzelnes Muster, das diese Aufgaben löst, sondern eine ganze Folge davon. Dabei muß sichergestellt werden, daß der Fehler unabhängig des Zustandes zu Beginn der Musterfolge gefunden wird.

Definition 2.2 (Testmuster)
Ein Testmuster *für einen Fehler* $f = (G, X^*, Y^*, V, \emptyset, \alpha, \beta)$ *über einem Graphen* $G = (V, E, Q, Z)$ *an einem Modul* m *des Schaltkreises* $C = (T, M, S, \tau, \rho, type)$ *ist eine Folge* $(tm_1, tm_2, \cdots, tm_r)$ *von Schaltkreiseingaben* $tm_i \in \mathbf{B}^{indegree(\tau)}$, *wenn mit den durch* $\gamma_1(\rho^{in}(\tau)) = tm_1, \cdots, \gamma_r(\rho^{in}(\tau)) = tm_r$ *definierten stabilen Signalbelegungen* γ_i *und den Moduleingaben* $\omega_1 := \gamma_1(\rho^{in}(m)), \cdots, \omega_r := \gamma_r(\rho^{in}(m))$ *folgendes gilt:*

$$\forall z \in V \ \exists s \in E, 1 \leq i \leq r, e : \mathbf{B}^{indegree(m)} \rightarrow \mathbf{B}^{outdegree(m)} \ \textit{mit}$$

1. $\alpha(s) = \omega_i$
2. $\beta(s) \neq behave(type(m))(\omega_i)$

} *Fehler einpflanzen*

3. $\delta^{\omega_1 \omega_2 \cdots \omega_{i-1}}(z) = Q(s)$
4. $tm_i \in TM_{komb}(C, m, e)$
   $e(\omega_i) = \beta(s)$

} *und propagieren*

Ein Testmuster stellt also für jeden Zustand z des Fehlers f einen Zustand $Q(s)$ ein, der mit der folgenden Zelleingabe $\omega_i = \alpha(s)$ eine fehlerhafte Ausgabe $\beta(s)$ hervorruft, die an die primären Ausgänge der Schaltung propagiert wird. Als Beispiel mag der Fehler f_1 in Abbildung 4 dienen. Er besitzt die zwei Zustände z_0 und z_1. Gilt für eine Kante s, daß $\beta(s) = behave(type(m))(\alpha(s))$ so ist s nur mit $\alpha(s)$ beschriftet. Ansonsten ist die Beschriftung von s gleich $\alpha(s)/\beta(s)$. Bestünde die gesammte Schaltung C nur aus einem einzigen Modul m, für das f_1 einen möglichen sequentiellen Fehler beschreibt, dann ist (c, b, c) ein Testmuster für f_1 an m. Befindet sich der Fehler nämlich zu Beginn des Tests im Zustand z_1, dann wird über das erste c die Zellausgabe zu *err* geändert und damit auch zu den primären Ausgängen der Schaltung propagiert. Befindet er sich jedoch in Zustand z_0 so geht er mit der zweiten Zelleingabe, dem b in den Zustand z_1 über und wird durch das zweite c getestet.

Testmuster zu erzeugen, oder bei gegebenen Eingabefolgen zu entscheiden, ob ein Test vorliegt, ist mit dieser Definition recht aufwendig. Denn man muß sicherstellen, daß der Fehler auch wirklich von jedem Zustand aus mit dem Testmuster gefunden wird!

Betrachtet man sich den Fehler f_2 in Abbildung 5, so sieht man, daß es auch einfacher gehen kann. Hier ist nämlich schon (a, c) ein Testmuster. Durch a als die erste Zelleingabe wird jeder Zustand des Fehlers in den Zustand z_1 überführt. Dadurch muß nur einmal die Propagation der fehlerhaften Zellausgabe an die primären Ausgänge gefordert werden! Eine solche Eingabefolge, die jeden Zustand in einen einzigen überführt, gibt es für den Fehler f_1 leider nicht.

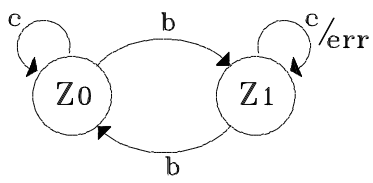

Bild 4: sequentieller Fehler f_1

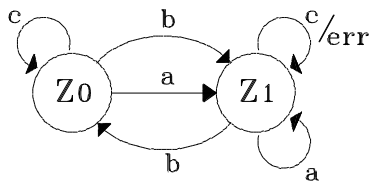

Bild 5: sequentieller Fehler f_2

Definition 2.3 (blind homing sequence)
Ein Eingabewort ω eines sequentiellen Fehlers $F = (G, X^*, Y^*, V, \emptyset, \alpha, \beta)$ über einem Graphen $G = (V, E, S, D)$ mit $\delta^\omega(V) = \{P\}, P \in V$, heißt blind homing sequence.

$$bhs := \{\omega \in X^* \mid \delta^\omega(V) = \{P\}, P \in V\}$$

Der Name blind homing sequence soll die Verwandschaft zum Problem der *homing seqence* ausdrücken [Gil62], welches ebenfalls zustandsdefinierende Eingaben sucht, wobei aber auch die Ausgabe des endlichen Automaten berücksichtigt wird. Bei sequentiellen Fehlern inmitten großer Schaltungen ist die Ausgabe im allgemeinen aber nicht beobachtbar.

Besitzt ein sequentieller Fehler eine blind homing sequence ω, so lassen sich Testmuster, die ω enthalten mit nur einer einzigen Propagationsüberprüfung verifizieren. Damit vereinfachen sich sowohl Fehlersimulation als auch Testmustergenerierung ungemein. Deshalb möchte ich blind homing sequences im folgenden näher untersuchen, um Kriterien zu finden, wann ein sequentieller Fehler eine blind homing sequence besitzt oder nicht, und wie die Länge einer solchen aussieht.

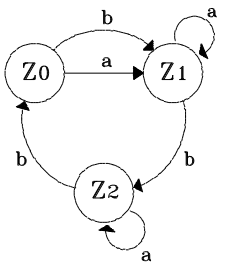

Bild 6: sequentieller Fehler mit drei Zuständen

2.3.1 Der Automat über Zustandsmengen

Ähnlich der Konstruktion eines deterministischen endlichen Automaten aus einem nichtdeterministischen, kann man für einen sequentiellen Fehler $f = (G, X^*, Y^*, V, \emptyset, \alpha, \beta)$ über einem Graphen $G = (V, E, Q, Z)$ einen deterministischen endlichen Automaten $\widetilde{f} = (\widetilde{G}, X^*, Y^*, \widetilde{S}, \widetilde{F}, \widetilde{\alpha}, \epsilon)$ über einem Graphen $\widetilde{G} = (\widetilde{V}, \widetilde{E}, \widetilde{Q}, \widetilde{Z})$ definieren, der eine Eingabe $\omega \in X^*$ genau dann akzeptiert, wenn ω eine blind homing sequence ist.

$$\widetilde{V} := \wp(V)$$

$$\widetilde{E} := \{(K_1, K_2, x) \mid K_1, K_2 \in \widetilde{V}, x \in X \text{ und } \delta^x(K_1) = K_2\}$$

$$\widetilde{Q}(K_1, K_2, x) := K_1,\ \widetilde{Z}(K_1, K_2, x) := K_2,\ \widetilde{\alpha}(K_1, K_2, x) := x$$

$$\widetilde{S} := V \in \widetilde{V},\ \widetilde{F} := \{\{z\} | z \in V\}$$

Die Knoten $\widetilde{V}$ des Graphen $\widetilde{G}$ sind dabei Teilmengen der Knoten des Graphen G. Sie repräsentieren alle die Zustände, in denen sich f unter der aktuellen Eingabe befinden kann. Der Startzustand von $\widetilde{f}$ ist deshalb der Knoten in $\widetilde{G}$, der alle Knoten von G darstellt. Kann sich f nur noch in einem einzelnen Zustand befinden, so ist die Eingabe eine blind homing sequence und der endliche Automat $\widetilde{f}$ hält.

Als Beispiel möge der sequentielle Fehler aus Abbildung 6 mit $n = 3$ Zuständen dienen. Abbildung 7 zeigt den dazugehörenden Automaten $\widetilde{f}$.

Man sieht daran, daß $\omega = babbbabbab$ eine blind homing sequence ist. Sie ist aber sicherlich nicht die kürzeste. Betrachtet man den Automaten etwas genauer, so erkennt man, daß es keine blind homing sequence kürzer als $\omega = abba$, also der Länge vier, gibt.

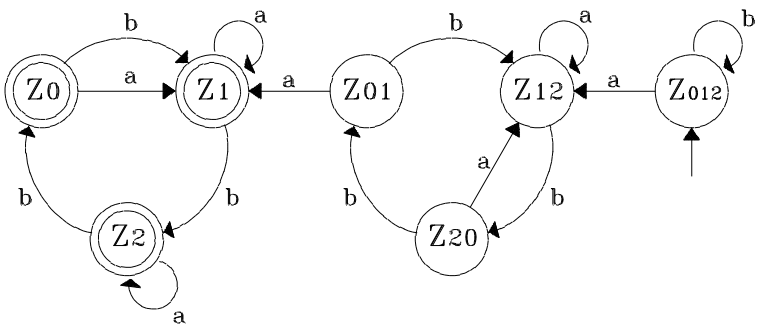

Bild 7: $\widetilde{f}$ zu Bild 6

Wie hängt die Länge der kürzesten blind homing sequence dabei von der Anzahl der Zustände n eines sequentiellen Fehlers ab? Als Antwort auf diese Frage, untersuche ich in den folgenden Abschnitten obere und untere Schranken für die maximale Länge einer kürzesten blind homing sequence eines sequentiellen Fehlers mit n Zuständen. Da die Existenz einer blind homing sequence nicht gesichert ist, erhebe ich sie zur Voraussetzung.

2.3.2 Eine obere Schranke für die maximale Länge der kürzesten blind homing sequence

Eine triviale obere Schranke für die kürzeste blind homing sequence eines sequentiellen Fehlers mit n Zuständen stellt $2^n - (n+1)$ dar. Denn dies ist die Anzahl der mehrelementigen Knoten in $\widetilde{G}$, und wenn es einen längsten Weg von $\widetilde{S}$ zu einem einelementigen Knoten gibt, so kann der nur über maximal $2^n - (n+1)$ Knoten laufen. Diese Schranke ist sogar scharf für Fehler mit bis zu drei Zuständen, wie wir für den Fall $n = 3$ schon am Beispiel aus Abbildung 6 gesehen habe, denn $2^3 - (3+1) = 4$.

Für größere n ist die Schranke aber viel zu unscharf, wie der folgende Satz zeigt.

Satz 1 *Falls ein sequentieller Fehler mit n Zuständen eine blind homing se-*

quence besitzt, so hat er auch eine mit einer Länge von höchstens

$$1 + \frac{n(n-1)(n-2)}{2}$$

Um diesen Satz zu beweisen, betrachte ich für $0 \leq i < \infty$ die Mengen K_i definiert durch

$$K_i := \left\{ \{z_1, z_2\} \middle| z_1 \neq z_2 \in V, \exists \omega, |\omega| \leq i \text{ mit } \delta^\omega(z_1) = \delta^\omega(z_2) \right\}$$

In der Menge K_i werden alle zweielementigen Teilmengen der Zustände aus V zusammengefaßt, deren beide Zustände sich mit einer Eingabe aus nicht mehr als i Zeichen auf den selben Zustand überführen lassen. Mit dieser Konstruktion gilt folgendes Lemma:

Lemma 2.1

1. $\forall i : K_i \subseteq K_{i+1}$
2. $K_i = K_{i+1} \Rightarrow K_i = K_{i+j} \ \forall j \geq 1$
3. $\forall i : |K_i| \leq \binom{n}{2}$
4. $K_{\binom{n}{2}} = \left\{ \{z_1, z_2\} \middle| z_1 \neq z_2 \in V, \exists \omega \ \textit{mit} \ \delta^\omega(z_1) = \delta^\omega(z_2) \right\}$

Beweis:

1. trivial
2. durch Induktion über j:
   Induktionsanfang ($j = 1$): klar
   Induktionsschritt:
   zu zeigen: $\forall j > 1$ und $\forall \{z_1, z_2\} \in K_{i+j}$ gilt: $\{z_1, z_2\} \in K_{i+j-1}$
   $\{z_1, z_2\} \in K_{i+j} \Rightarrow \exists \omega$ mit $|\omega| \leq i + j$ und $\delta^\omega(z_1) = \delta^\omega(z_2)$
   Aus $|\omega| < i + j$ folgt die Behauptung, also bleibt noch $|\omega| = i + j$
   Sei $\omega = \omega_1 x \omega_2$ mit $|\omega_2| = i \Rightarrow \{\delta^{\omega_1}(z_1), \delta^{\omega_1}(z_2)\} \in K_{i+1}$
   Da $K_{i+1} = K_i \Rightarrow \exists \omega_3$ mit $\delta^{\omega_1\omega_3}(z1) = \delta^{\omega_1\omega_3}(z2)$ und $|\omega_3| \leq i$
   Also $\{z_1, z_2\} \in K_{i+j-1}$
3. $|V| = n \Rightarrow \left| \left\{ \{z_1, z_2\} \middle| z_1 \neq z_2 \in V \right\} \right| = \binom{n}{2}$
4. Sei $fix := \min\{i | K_i = K_{i+1}\}$.
   $\Rightarrow K_{fix} = \left\{ \{z_1, z_2\} \middle| z_1 \neq z_2 \in V, \exists \omega \text{ mit } \delta^\omega(z_1) = \delta^\omega(z_2) \right\}$ (Lemma 2.1.2)
   $0 = |K_0| < |K_1| < \cdots < |K_{fix}|$. (Lemma 2.1.1)
   $\Rightarrow fix \leq \binom{n}{2}$ (Lemma 2.1.3)

□

Lemma 2.2 *bhs* $\neq \emptyset \Rightarrow$

1. $\forall M \subseteq V, |M| > 1, \exists \omega \in X^*$ *mit* $|\delta^\omega(M)| < |M|$ *und* $|\omega| \leq \binom{n}{2}$
2. $\exists x \in X$ *mit* $|\delta^x(V)| < |V|$

Beweis:

1. Seien $z_1, z_2 \in M, z_1 \neq z_2, \widehat{\omega} \in bhs$

$$\begin{array}{ll} \Rightarrow \delta^{\widehat{\omega}}(z_1) = \delta^{\widehat{\omega}}(z_2) & \\ \Rightarrow \{z_1, z_2\} \in K_{\binom{n}{2}} & \text{(Lemma 2.1.4)} \\ \Rightarrow \exists \omega \text{ mit } \delta^\omega(z_1) = \delta^\omega(z_2) \text{ und } |\omega| \leq \binom{n}{2} & \\ \Rightarrow |\delta^\omega(M)| < |M| & \end{array}$$

2. Annahme: $\neg\exists x \in X$ mit $|\delta^x(V)| < |V| \Rightarrow \neg\exists \omega \in X^*$ mit $|\delta^\omega(V)| < |V|$ Widerspruch zu $bhs \neq \emptyset$!

□

Damit läßt sich nun Satz 1 zeigen: Es existieren $\omega_0 \cdots \omega_l$ mit der Eigenschaft

$$\begin{array}{ll} \delta^{\omega_0\omega_1\cdots\omega_l}(V) = \{P\}, P \in V & (bhs \neq \emptyset) \\ |\delta^{\omega_0}(V)| < |V| \text{ und } |\omega_0| = 1 & \text{(Lemma 2.2.2)} \\ |\delta^{\omega_0\omega_1\cdots\omega_i}(V)| < |\delta^{\omega_0\omega_1\cdots\omega_{i-1}}(V)| \ (1 \leq i \leq l) \text{ mit } |\omega_i| \leq \binom{n}{2} & \text{(Lemma 2.2.1)} \end{array}$$

Da $n = |\delta^\epsilon(V)| > |\delta^{\omega_0}(V)| > \cdots > |\delta^{\omega_0\omega_1\cdots\omega_l}(V)| = 1$, ergibt sich daraus $l \leq n-2$ und für $\omega := \omega_0\omega_1\cdots\omega_l$, daß

$$\delta^\omega(V) = \{P\}, P \in V \text{ und } |\omega| = \sum_{i=0}^{l} |\omega_i| \leq \sum_{i=0}^{n-2} |\omega_i|$$

$$\leq 1 + (n-2)\binom{n}{2} = 1 + \frac{n(n-1)(n-2)}{2}$$

□

2.3.3 Eine untere Schranke für die maximale Länge der kürzesten blind homing sequence

Der vorherige Abschnitt hat gezeigt, daß die kürzeste blind homing sequence eines sequentiellen Fehlers, falls sie existiert, durch $1 + \frac{n(n-1)(n-2)}{2}$ in ihrer Länge nach oben beschränkt ist. Es bleibt zu klären, ob sie diese Länge wirklich erreichen kann. Einen ersten Schritt in diese Richtung stellt der folgende Satz dar.

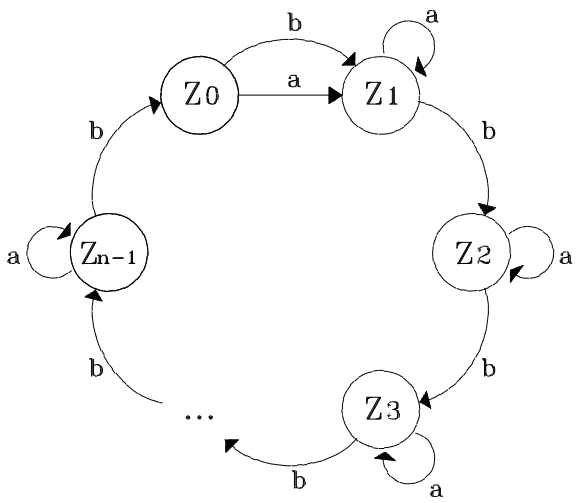

Bild 8: Beispielfehler mit n Zuständen

Satz 2 *Es gibt sequentielle Fehler mit n Zuständen, deren kürzeste blind homing sequence mindestens die Länge* $n(n-2)+1$ *haben.*

Zum Beweis des Satzes betrachte man den sequentiellen Fehler in Abbildung 8. Als hilfreiche Vorstellung mag ein Wettlauf dienen. Auf jedem Zustand startet ein Läufer. Ruft man den Läufern ein b zu, so rennen sie alle zum nächsten Zustand. Ertönt hingegen ein a, so setzt sich nur der Läufer in Zustand Z_0 in Bewegung, die anderen bleiben stehen. Trifft er dabei auf einen Läufer in Z_1, so scheidet dieser aus. Ist nur noch ein Läufer übrig, so bilden die Zurufe eine blind homing sequence. Wie weit muß ein Läufer rennen, um einen anderen einzuholen? Jedes Annäheren an seinen Konkurrenten kostet ihn eine Runde, denn nur im Zustand Z_0 kann er gegenüber seinen Gegnern aufholen. Will ein Läufer seinen Hintermann einholen, muß er also fast n Runden laufen. Der folgende, formale Beweis zeigt, daß es bei jeder blind homing sequence einen Läufer geben muß, der seinen Hintermann einholt.

Die Funktion $abst^{\omega} : V \times V \rightarrow N$ soll den Abstand der Folgezustände zweier Läufer über die b-Kanten nach Zuruf von ω wiederspiegeln, dabei unterscheide ich beim Einholen zwischen Gewinner (Abstand ist Null) und Verlierer des Wett-

laufes (Abstand ist n).

$$\begin{array}{lcl} abst^{\epsilon}(z_i,z_j) & := & (j-i)mod(n) \\ abst^{\omega b}(z_i,z_j) & := & abst^{\omega}(z_i,z_j) \\ abst^{\omega a}(z_i,z_j) & := & \begin{cases} abst^{\omega}(z_i,z_j)-1 & : \ \delta^{\omega}(z_i)=z_0 \neq \delta^{\omega}(z_j) \\ abst^{\omega}(z_i,z_j)+1 & : \ \delta^{\omega}(z_i)\neq z_0=\delta^{\omega}(z_j) \\ abst^{\omega}(z_i,z_j) & : \ sonst \end{cases} \end{array}$$

Das folgende Lemma zeigt den Aufwand des Gewinners zweier Läufer.

Lemma 2.3

1. $\delta^{a\omega}(z_0)=z_0 \Rightarrow |\omega| \geq n-1.$
2. $abst^{\omega}(z_i,z_j)=0 \Rightarrow |\omega| \geq \big((j-i)\mathrm{mod}(n)-1\big)n+1$

Beweis:

1. $\delta^{a}(z_0)=z_1, \delta^{\omega}(z_1)=z_0 \Rightarrow \#_b\omega \geq n-1$
   ($\#_b\omega$ sei die Anzahl der b's in ω.)
2. $abst^{\omega}(z_i,z_j)=0$ und $abst^{\epsilon}(z_i,z_j)=(j-i)mod(n)$
   Wie man in der Definition der Funktion $abst^{\omega}$ sieht, verringert sich deren Wert für feste z_i und z_j nur durch Anhängen von a an ω, wenn $\delta^{\omega}(z_i)=z_0$ ist.
   $\Rightarrow \omega=\omega_0 a\omega_1 a\omega_2 \cdots a\omega_{(j-i)mod(n)}$
   und $\forall 0 \leq l < (j-i)mod(n)$ gilt: $\delta^{\omega_0 a\omega_1 a\omega_2 \cdots a\omega_l}(z_i)=z_0$.
   $\Rightarrow \forall 1 \leq l < (j-i)mod(n) : |\omega_l| \geq n-1$ (Lemma 2.3.1)
   $\Rightarrow |\omega| \geq \big((j-i)mod(n)-1\big)n+1$

□

Das folgende Lemma zeigt die Übereinstimmung der Abstandsfunktion mit der Intuition, die ihr zugrundeliegt. Dabei identifiziere ich die Zustände oft mit den Nummern, die sie tragen.

Lemma 2.4

1. $\big(abst^{\omega}(z_i,z_j)=\delta^{\omega}(z_j)-\delta^{\omega}(z_i)\big)\mathrm{mod}(n)$
2. $\delta^{\omega}(z_j)-\delta^{\omega}(z_i)=0\mathrm{mod}(n) \Rightarrow \forall\omega_1 \in X^*$ *gilt:* $\delta^{\omega\omega_1}(z_i)=\delta^{\omega\omega_1}(z_j)$
3. $abst^{\omega}(z_i,z_j)=0\mathrm{mod}(n) \Rightarrow \forall\omega_1 \in X^*$ *gilt:* $abst^{\omega\omega_1}(z_i,z_j)=abst^{\omega}(z_i,z_j)$
4. $\forall\omega \in X^*$ *und* $\forall z_i,z_j \in V$ *gilt:* $0 \leq abst^{\omega}(z_i,z_j) \leq n.$

Beweis:

1. durch Induktion über die Länge von ω:

   Induktionsanfang: $\omega = \epsilon$
   $abst^{\epsilon}(z_i, z_j) = (j-i) \, mod(n) = \big(\delta^{\omega}(z_j) - \delta^{\omega}(z_j)\big) \, mod(n)$
   $\Rightarrow \big(abst^{\epsilon}(z_i, z_j) = \delta^{\omega}(z_j) - \delta^{\omega}(z_i)\big) \, mod(n)$

   Induktionsschluß:
   Fall 1: $\omega = \omega_1 b$

   $$\begin{aligned}
   &\big(abst^{\omega_1 b}(z_i, z_j)\big) \, mod(n) \\
   &= \big(abst^{\omega_1}(z_i, z_j)\big) \, mod(n) && \text{(Definition von } abst\text{)} \\
   &= \big(\delta^{\omega_1}(z_j) - \delta^{\omega_1}(z_i)\big) \, mod(n) && \text{(Induktionsvoraussetzung)} \\
   &= \big((\delta^{\omega_1 b}(z_j) - 1) - (\delta^{\omega_1 b}(z_i) - 1)\big) \, mod(n) \\
   &= \big(\delta^{\omega}(z_j) - \delta^{\omega}(z_i)\big) \, mod(n)
   \end{aligned}$$

   Fall 2: $\omega = \omega_1 a$
   $\big(abst^{\omega_1 a}(z_i, z_j)\big) \, mod(n)$

   $$= \begin{cases}
   \big(abst^{\omega_1}(z_i, z_j) - 1\big) \, mod(n) & : \quad \delta^{\omega}(z_i) = z_0 \neq \delta^{\omega}(z_j) \\
   \big(abst^{\omega_1}(z_i, z_j) + 1\big) \, mod(n) & : \quad \delta^{\omega}(z_i) \neq z_0 = \delta^{\omega}(z_j) \\
   \big(abst^{\omega_1}(z_i, z_j)\big) \, mod(n) & : \quad \text{sonst}
   \end{cases} \quad \text{(Definition von } abst\text{)}$$

   $$= \begin{cases}
   \big(\delta^{\omega_1}(z_j) - \delta^{\omega_1}(z_i) - 1\big) \, mod(n) & : \quad \delta^{\omega}(z_i) = z_0 \neq \delta^{\omega}(z_j) \\
   \big(\delta^{\omega_1}(z_j) - \delta^{\omega_1}(z_i) + 1\big) \, mod(n) & : \quad \delta^{\omega}(z_i) \neq z_0 = \delta^{\omega}(z_j) \\
   \big(\delta^{\omega_1}(z_j) - \delta^{\omega_1}(z_i)\big) \, mod(n) & : \quad \text{sonst}
   \end{cases} \quad \text{(Ind.vor.)}$$

   $$= \begin{cases}
   \big(\delta^{\omega_1 a}(z_j) - (\delta^{\omega_1 a}(z_i) - 1) - 1)\big) \, mod(n) & : \quad \delta^{\omega}(z_i) = z_0 \neq \delta^{\omega}(z_j) \\
   \big(\delta^{(\omega_1 a}(z_j) - 1) - \delta^{\omega_1}(z_i) + 1\big) \, mod(n) & : \quad \delta^{\omega}(z_i) \neq z_0 = \delta^{\omega}(z_j) \\
   \big(\delta^{\omega_1 a}(z_j) - \delta^{\omega_1 a}(z_i)\big) \, mod(n) & : \quad \text{sonst}
   \end{cases}$$

   $$= \big(\delta^{\omega}(z_j) - \delta^{\omega}(z_i)\big) \, mod(n)$$

2. $\delta^{\omega}(z_j) - \delta^{\omega}(z_j) = 0 \, mod(n) \Rightarrow \delta^{\omega}(z_j) = \delta^{\omega}(z_j)$, denn der Beispielfehler hat genau n Zustände.
   $\delta^{\omega\omega_1}(z_i) = \delta^{\omega_1}(\delta^{\omega}(z_i)) = \delta^{\omega_1}(\delta^{\omega}(z_j)) = \delta^{\omega\omega_1}(z_j)$

3. $$\begin{aligned}
   &abst^{\omega}(z_i, z_j) = 0 \, mod(n) \\
   &\Rightarrow \delta^{\omega}(z_j) - \delta^{\omega}(z_j) = 0 \, mod(n) && \text{(Lemma 2.4.1)} \\
   &\Rightarrow \forall \omega_1 \in X^* \text{ gilt:} \\
   &\delta^{\omega\omega_1}(z_i) = \delta^{\omega\omega_1}(z_j) && \text{(Lemma 2.4.2)}
   \end{aligned}$$

   Um $abst^{\omega}$ für feste z_i und z_j zu verändern, müssen sich $\delta^{\omega}(z_i)$ und $\delta^{\omega}(z_j)$ aber unterscheiden.

4. Nach Teil 3 des Lemmas gilt für $abst^{\omega}(z_i, z_j) = 0 \, mod(n)$, daß $\forall \omega_1 \in X^* : abst^{\omega\omega_1}(z_i, z_j) = abst^{\omega}(z_i, z_j)$. Zusammen mit $0 \leq abst^{\epsilon}(z_i, z_j) = (j - i) \, mod(n) \leq n$ folgt induktiv die Behauptung.

□

Um eine Ordnung der Läufer gemäß ihres Erfolges zu erhalten, definiere ich die Relation „holt-mit-ω-ein“.

Definition 2.4 *Die Relation* $\leq_{\omega} \subseteq V \times V$ *ist definiert durch*

$$z_i \leq_{\omega} z_j :\Leftrightarrow i = j \text{ oder } \omega = \omega_1 a \omega_2 \text{ und } \delta^{\omega_1}(z_i) = z_0, \; \delta^{\omega_1}(z_j) = z_1$$

Lemma 2.5 $\leq_{\omega}$ *ist lineare Ordnung auf* V *für* $\omega \in bhs$.

Beweis:

Reflexivität:
$\forall 0 \leq i < n$ gilt: $z_i \leq_{\omega} z_i \Rightarrow \leq_{\omega}$ ist reflexiv.

Antisymmetrie:
Sei $z_i \leq_{\omega} z_j$ und $z_j \leq_{\omega} z_i$. Nehmen wir nun $i \neq j$ an.
Daraus folgt: $\omega = \omega_1 a \omega_2$ und $\delta^{\omega_1}(z_i) = z_0, \delta^{\omega_1}(z_j) = z_1$
sowie $\omega = \omega_3 a \omega_4$ und $\delta^{\omega_3}(z_j) = z_0, \delta^{\omega_3}(z_i) = z_1$
Sei nun o.B.d.A. $|\omega_1| < |\omega_3|$. Aus $\delta^{\omega_1 a}(z_i) = \delta^{\omega_1 a}(z_j)$ folgt nach Lemma 2.4.2 auch $\delta^{\omega_3}(z_i) = \delta^{\omega_3}(z_j)$ im Widerspruch zu $\delta^{\omega_3}(z_j) = z_0 \neq z_1 = \delta^{\omega_3}(z_i)$

Transitivität:
Sei $z_i \leq_{\omega} z_j$ und $z_j \leq_{\omega} z_k$. Zu zeigen ist $z_i \leq_{\omega} z_k$.
Falls $i = j$ oder $j = k$ ist dies klar.
Sei nun also $i \neq j$ und $j \neq k$.
Dann gilt: $\omega = \omega_1 a \omega_2$ und $\delta^{\omega_1}(z_i) = z_0, \delta^{\omega_1}(z_j) = z_1$
sowie: $\omega = \omega_3 a \omega_4$ und $\delta^{\omega_3}(z_j) = z_0, \delta^{\omega_3}(z_k) = z_1$.
Da $\delta^{\omega_1}(z_j) = z_1 \neq z_0 = \delta^{\omega_3}(z_j)$ ist $|\omega_1| \neq |\omega_3|$
Fall 1: $|\omega_1| < |\omega_3|$

$$\begin{aligned} & \delta^{\omega_1 a}(z_i) = \delta^{\omega_1 a}(z_j) \\ \Rightarrow \; & \delta^{\omega_3}(z_i) = \delta^{\omega_3}(z_j) = z_0 \quad (\text{ Lemma 2.4.2}) \\ \Rightarrow \; & z_i \leq_{\omega} z_k \end{aligned}$$

Fall 2: $|\omega_3| < |\omega_1|$

$$\begin{aligned} & \delta^{\omega_3 a}(z_i) = \delta^{\omega_3 a}(z_j) \\ \Rightarrow \; & \delta^{\omega_1}(z_i) = \delta^{\omega_1}(z_j) = z_1 \quad (\text{ Lemma 2.4.2}) \\ \Rightarrow \; & z_i \leq_{\omega} z_k \end{aligned}$$

Vollständigkeit:
Sei $i \neq j$. Da $\omega \in bhs$ gilt: $\delta^{\omega}(z_i) = \delta^{\omega}(z_j)$

$\Rightarrow \omega = \omega_1 a \omega_2$ und $\delta^{\omega_1}(z_i) = z_0, \delta^{\omega_1}(z_j) = z_1$ oder $\delta^{\omega_1}(z_j) = z_0, \delta^{\omega_1}(z_i) = z_1$.
Also $z_i \leq_\omega z_j$ oder $z_j \leq_\omega z_i$. □
Daß diese Relation das Gewinnen und Verlieren korrekt wiedergibt, wird im folgenden Lemma gezeigt.

Lemma 2.6 $z_i \leq_\omega z_j \Rightarrow abst^\omega(z_i, z_j) = 0$

Beweis:
Falls $i = j$ ist die Aussage klar. Für $i \neq j$ gilt:
$\omega = \omega_1 a \omega_2$ und $\delta^{\omega_1}(z_i) = z_0, \delta^{\omega_1}(z_j) = z_1$

$$\begin{aligned}
&\Rightarrow (abst^{\omega_1}(z_i, z_j) = 1) mod(n) && \text{(Lemma 2.4.1)}\\
&\Rightarrow abst^{\omega_1}(z_i, z_j) = 1 && \text{(Lemma 2.4.4)}\\
&\Rightarrow abst^{\omega_1 a}(z_i, z_j) = abst^{\omega_1}(z_i, z_j) - 1 = 0 &&\\
&\Rightarrow abst^{\omega_1 a \omega_2}(z_i, z_j) = 0 && \text{(Lemma 2.4.3)}
\end{aligned}$$

□

Damit sind wir in der Lage, Satz 2 zu beweisen:
Nach Lemma 2.5 ist $\leq_\omega$ für jedes $\omega \in bhs$ eine lineare Ordnung auf V. Es existiert also eine Permutation π mit:

$$z_{\pi(0)} \leq_\omega z_{\pi(1)} \leq_\omega \cdots \leq_\omega z_{\pi(n-1)}$$

$$\begin{aligned}
&\Rightarrow z_{\pi(0)} \leq_\omega z_{(\pi(0)-1) mod(n)} &&\\
&\Rightarrow abst^\omega(z_{\pi(0)}, z_{(\pi(0)-1) mod(n)}) = 0 && \text{(Lemma 2.6)}\\
&\Rightarrow |\omega| \geq ((\pi(0) - 1 - \pi(0)) mod(n) - 1)n + 1 && \text{(Lemma 2.3)}\\
&\qquad = n(n-2) + 1 &&
\end{aligned}$$

□

2.3.4 Kriterien für die Existenz einer blind homing sequence

Da es den Test eines sequentiellen Fehlers f beeinflußt, ob er eine blind homing sequence besitzt oder nicht, wünscht man sich natürlich ein möglichst einfaches Kriterium für deren Existenz. Ein solches Kriterium ist zum Beispiel, ob es einen Weg vom Startzustand zu einem der Zielzustände im endlichen Automaten über den Zustandsmengen $\widetilde{f}$ gibt. Dieses Kriterium ist aber nicht sehr handlich, denn es müssen gegenüber dem Ausgangsfehler f exponentiell viele Zustände bearbeitet werden.

Ein wesentlich einfacheres Kriterium liefert Satz 3. Dazu möchte ich an die in Abschnitt 2.3.2 vorgestellten Mengen K_i erinnern:

$$K_i := \left\{ \{z_1, z_2\} \middle| z_1 \neq z_2 \in V, \exists \omega, |\omega| \leq i \text{ mit } \delta^\omega(z_1) = \delta^\omega(z_2) \right\}$$

Satz 3 *$bhs \neq \emptyset \Leftrightarrow K_{fix} = \binom{V}{2}$ wobei $fix := \min\{i | K_i = K_{i+1}\}$*

Beweis:

„⇒“: Sei $\omega \in bhs$.
$\Rightarrow \forall z_1 \neq z_2 \in V : \delta^\omega(z_1) = \delta^\omega(z_2)$
$\Rightarrow \{z_1, z_2\} \in K_{|\omega|}$
$\Rightarrow K_{|\omega|} = \binom{V}{2}$
nach Lemma 2.1 gilt aber auch: $\binom{V}{2} = K_{|\omega|} \subseteq K_{fix} \subseteq \binom{V}{2}$
$\Rightarrow K_{fix} = \binom{V}{2}$

„⇐“: Seien $\omega_1, \omega_2, \cdots, \omega_i$ gegeben. Falls es zwei unterschiedliche Zustände $z_1 \neq z_2$ gibt mit $z_1 \in \delta^{\omega_1\omega_2\cdots\omega_i}(V)$ und $z_2 \in \delta^{\omega_1\omega_2\cdots\omega_i}(V)$, dann existiert nach Voraussetzung ein ω_{i+1} mit $\delta^{\omega_{i+1}}(z_1) = \delta^{\omega_{i+1}}(z_2)$. Es gilt also, daß $|\delta^{\omega_1\omega_2\cdots\omega_i}(V)| > |\delta^{\omega_1\omega_2\cdots\omega_i\omega_{i+1}}(V)|$. Da $|V| < \infty$ gibt es ein $\omega := \omega_1\omega_2\cdots\omega_k$ mit $\delta^\omega(V) = \{P\}, P \in V$. ω ist damit eine blind homing sequence .

□

Um die Existenz einer blind homing sequence des sequentiellen Fehlers f zu untersuchen, genügt es also einen Graphen $\widehat{G} = (\widehat{V}, \widehat{E})$ zu betrachten, definiert durch:
$\widehat{V} := \binom{V}{2}$
$\widehat{E} := \{(z_{12}, z_{34}) | z_{12}, z_{34} \in \widehat{V}, \exists x \in X \text{ mit } \delta^x(z_{12}) = z_{34}\}$
Der Graph $\widehat{G}$ beschreibt den Übergang aller zweielementigen Teilmengen der Zustände des Fehlers f.

Lemma 2.7 *$k \in K_{fix} \Leftrightarrow \exists$ Weg von k zu $k_1 \in K_1$ im Graphen $\widehat{G}$.*

Die Korrektheit des Lemmas folgt aus der Konstruktion des Graphen $\widehat{G}$. Durch Bestimmen von K_1 und der Menge K aller Knoten aus $\widehat{V}$, die einen Weg zu K_1 besitzen, läßt sich die Existenz einer blind homing sequence für den Fehler f entscheiden. Nach Satz 3 und Lemma 2.7 ist die Existenz gleichbedeutend damit, daß $K = \widehat{V}$. Die Anzahl der Knoten in $\widehat{V}$ und die Anzahl der Kanten in $\widehat{E}$ (von jedem Knoten gehen maximal $|X|$ viele Kanten aus) ist quadratisch zu der Anzahl der Zustände in f. Damit ist auch die Berechnung von K_1 und K in quadratischer Zeit möglich. Die exponentielle Laufzeit des Kriteriums über dem Graphen $\widetilde{f}$ läßt sich also auf eine quadratische drücken!

Ich möchte Satz 3 an einem Beispiel anwenden, das zeigt, daß schon eine kleine Änderung an einem sequentiellen Fehler die Existenz einer blind homing sequence beeinflußen kann. Dazu definiere ich die sequentiellen Fehler $F(l, r)$ durch Abbildung 9. Sie bestehen aus zwei „Ringen“ und einem „Verzweigungsknoten“ z_0. An z_0 gelangt man über die Eingabe a in den linken Ring

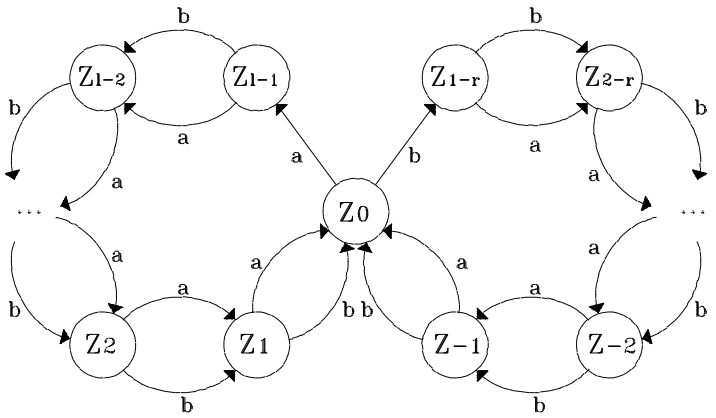

Bild 9: sequentielle Fehler $F(l,r)$

$(z_{l-1},\cdots,z_1,z_0)$ und über die Eingabe b in den rechten Ring $(z_{1-r},\cdots,z_{-1},z_0)$. Bei allen anderen Zuständen ist die Eingabe ohne Bedeutung. Man sieht, daß sich $F(4,6)$ und $F(5,7)$ nur geringfügig unterscheiden. Der linke und rechte Ring in $F(5,7)$ sind jeweils nur um einen Zustand verlängert worden. Man erwartet, daß entweder beide Fehler oder keiner eine blind homing sequence besitzen. Nach Lemma 2.8 existiert aber für $F(5,7)$ im Gegensatz zu $F(4,6)$ eine blind homing sequence.

Lemma 2.8 $bhs_{F(l,r)} \neq \emptyset \Leftrightarrow \gcd(l,r) = 1$

Beweis:
$bhs_{F(l,r)} \neq \emptyset$

$\Leftrightarrow$ $K_{fix} = \binom{V}{2}$ (Satz 3)
$\Leftrightarrow$ $\forall z_j \neq z_1 \in V : \{z_1, z_j\} \in K_{fix}$ $(\forall z_j\ \exists \omega$ mit $\delta^\omega(z_j) = z_1)$
$\Leftrightarrow$ $\forall j\ \exists \omega \in X^*$ mit $\delta^\omega(z_1) = z_{\pm 1}, \delta^\omega(z_j) = z_{\mp 1}$ (Konstruktion von $F(l,r)$)

$\gcd(l,r) = 1$
$\Rightarrow \exists$ ganze Zahlen h,k mit $hl + kr = 1$
Sei o.B.d.A $h > 0$: (Ansonsten gelten alle symmetrischen Aussagen)
Sei $\omega = \omega_1\omega_2\cdots\omega_{hl}$ mit $\omega_i = a :\Leftrightarrow \delta^{\omega_1\omega_2\cdots\omega_{i-1}}(z_1) = z_0$
$\Rightarrow \delta^\omega(z_1) = z_1$, da $|\omega|$ ein Vielfaches von l ist, und ω_i den Folgezustand

$\delta^{\omega_1\omega_2\cdots\omega_{i-1}}(z_1)$ von z_1 immer in den linken Kreis schickt. Ebenso wird der Folgezustand von z_j immer in den rechten Kreis geschickt, da die Folgezustände von z_1 und z_j nie gemeinsam an der Verzweigung z_0 stehen. Deshalb liegt $\delta^{\omega}(z_j)$ im rechten Kreis. Falls sich z_j also im linken Kreis befindet, läßt sich dies durch ω ändern. Also kann ich auch gleich annehmen, z_j läge im rechten Kreis.
$|\omega| = hl = 1 - kr$
Da sich z_j immer im rechten Kreis bewegt, wird es nach $|k|$ Kreisen wieder nach z_j gelangt sein und dann noch einen weiteren Übergang machen.
Durch mehrfache Wiederholung von ω gelangt z_j schließlich nach z_{-1}, und es gilt: $\delta^{\omega\omega\cdots}(z_1) = z_1$, $\delta^{\omega\omega\cdots}(z_j) = z_{-1}$.

$bhs_{F(l,r)} \neq \emptyset$
$\Rightarrow \exists\omega \in X^*$ mit $\delta^{\omega}(z_1) = z_{\pm 1}, \delta^{\omega}(z_{-2}) = z_{\mp 1}$
$\Rightarrow |\omega| = h_0 l + k_0 r$ und $|\omega| = 1 + h_1 l + k_1 r$
$\Rightarrow 1 = (h_0 - h_1)l + (k_0 - k_1)r$
$\Rightarrow 1 = (h_0 - h_1)\gcd(l,r)\widetilde{l} + (k_0 - k_1)\gcd(l,r)\widetilde{r}$
$\Rightarrow 1 = \gcd(l,r)((h_0 - h_1)\widetilde{l} + (k_0 - k_1)\widetilde{r})$
$\Rightarrow \gcd(l,r) = 1$

□

2.3.5 Erwartete Länge einer blind homing sequence bei gewürfelter Eingabe

Um einen sequentiellen Fehler f in einem Fertigungstest einfach testen zu können, sollte man seinen Zustand kennen und dazu eine blind homing sequence an die Eingänge der komplexen Zelle legen. Zur Generierung von Tests sind deshalb kürzeste blind homing sequences von Bedeutung. Bevor für eine Schaltung aber gezielt Testmuster erzeugt werden, versucht man durch gewürfelte Eingabemuster möglichst viele Fehler zu eliminieren. Auch hier sind Tests, die mit einer blind homing sequence beginnen, viel einfacher zu handhaben. Wie lange muß man aber Eingaben würfeln, um eine blind homing sequence zu erhalten? Ich setze im folgenden voraus, daß jeder Zelleingabe eine feste Wahrscheinlichkeit größer Null zugeordnet werden kann, mit der sie bei zufälligen Schaltungseingaben an einer betrachteten Zelle anliegt. Man könnte zum Beispiel davon ausgehen, daß der Kontext einer Zelle keine Rolle spielt, und alle Zelleingaben daher gleichwahrscheinlich sind.

Seien $x_1, x_2, x_3, \cdots \in X$ zufällige Eingaben an den sequentiellen Fehler f. Mit L_i möchte ich das Ereignis $x_1x_2\cdots x_i \in bhs, x_1x_2\cdots x_{i-1} \notin bhs$ bezeichnen. Dann ergibt sich die erwartete Länge einer blind homing sequence, E_{bhs} zu:

$$E_{bhs} = \begin{cases} \sum_{i=1}^{\infty} i \Pr(\mathrm{L}_i) & : \sum_{i=1}^{\infty} \Pr(\mathrm{L}_i) = 1 \\ \infty & : \text{sonst} \end{cases}$$

Bei einem Fehler, dessen E_{bhs} unendlich ist, lassen sich keine durch Würfeln erzeugte Testmuster erhoffen. Folgender Satz beantwortet die Frage, wann dies der Fall ist.

Satz 4 $E_{bhs} < \infty \Leftrightarrow bhs \neq \emptyset$

Dazu zeige ich zuerst das auch noch später nützliche, folgende Lemma:

Lemma 2.9

$$0 < p < 1 \Rightarrow \sum_{i=1}^{\infty} ip^i = \frac{p}{(1-p)^2}$$

Beweis:

$$\begin{aligned}
&\tfrac{d}{dp}(\textstyle\sum_{i=0}^{n-1} p^i) = \textstyle\sum_{i=0}^{n-1} ip^{i-1} \\
&= \\
&\tfrac{d}{dp}(\tfrac{p^n-1}{p-1}) = \ fracnp^{n-1}(p-1) - (p^n-1)(p-1)^2 \\
\Rightarrow &\sum_{i=1}^{\infty} ip^i = p \lim_{n\to\infty} \frac{np^{n-1}(p-1)-(p^n-1)}{(p-1)^2} = \frac{p}{(1-p)^2}
\end{aligned}$$

□

zum Beweis des Satzes ist folgendes zu zeigen:

1. $$\sum_{i=1}^{\infty} \Pr(\mathrm{L}_i) = 1 \Leftrightarrow bhs \neq \emptyset$$

2. $$\sum_{i=1}^{\infty} i \Pr(\mathrm{L}_i) < \infty, \text{ falls } bhs \neq \emptyset$$

Zu Teil 1:

„⇒“: klar

„⇐“: $bhs \neq \emptyset$
$\Rightarrow \exists \omega \in bhs, k := |\omega|$

$$x_1x_2\cdots x_i \notin bhs \quad \rightarrow \quad x_1x_2\cdots x_i \neq \omega_l\omega\omega_r$$

$$\rightarrow \quad \begin{cases} \omega \neq x_1x_2\cdots x_k & \wedge \\ \omega \neq x_{k+1}x_{k+2}\cdots x_{2k} & \wedge \\ \cdots & \\ \omega \neq x_{(\lfloor \frac{i}{k} \rfloor -1)k+1} x_{(\lfloor \frac{i}{k} \rfloor -1)k+2} \cdots x_{\lfloor \frac{i}{k} \rfloor k} \end{cases}$$

mit $q := \Pr(x_s x_{s+1} \cdots x_{s+k-1} \neq \omega) < 1$ und $1 > \widetilde{q} > q^{\frac{1}{k}}$

$\Rightarrow \Pr(x_1 x_2 \cdots x_i \notin bhs) \leq q^{\lfloor \frac{i}{k} \rfloor} \leq \frac{q^{\frac{i}{k}}}{q} \leq \frac{\widetilde{q}^i}{q}$

$$\Rightarrow \sum_{i=1}^{n} \Pr(\mathrm{L}_i) = \Pr(x_1 x_2 \cdots x_n \in bhs) = 1 - \Pr(x_1 x_2 \cdots x_n \notin bhs) \geq 1 - \frac{\widetilde{q}^n}{q}$$

Für $n \to \infty$ folgt die Behauptung.

Zu Teil 2:
Falls Ereignis L_i eintrifft, muß gelten, daß $x_1 x_2 \cdots x_{i-1} \notin bhs$. Nach nach Teil 1 folgt deshalb:

$$\sum_{i=1}^{\infty} i \Pr(\mathrm{L}_i) \leq \frac{1}{q} \sum_{i=1}^{\infty} i \widetilde{q}^{i-1} \leq \frac{1}{q(1-\widetilde{q})^2} \leq \infty \text{ (Lemma 2.9)}$$

□

Will man den Wert von E_{bhs} genau wissen, müßte man die in der Summe $\sum_{i=1}^{\infty} i \Pr(\mathrm{L}_i)$ auftretenden Wahrscheinlichkeiten ausrechnen und den Grenzwert der Summe bestimmen. Für alle Fehler f, bei denen immer nur das letzte Zeichen einer Eingabe $\omega \in X^*$ entscheidet, ob $\omega \in bhs$, leicht möglich. Dies trifft zum Beispiel für die in der Technologie Cmos auftretenden stuck-open-Fehler zu. Denn hier gilt mit $q := \Pr(x \in X, x \in bhs)$:

$\Pr(\mathrm{L}_i) = \Pr(x_i \in bhs) \prod_{j=1}^{i-1} \Pr(x_j \notin bhs) = (1-q)^{i-1} q.$

$$\Rightarrow E_{bhs} = \sum_{i=1}^{\infty} i(1-q)^{i-1} q = \frac{1}{q} \quad \text{(Lemma 2.9)}$$

Im allgemeinen ist das Ausrechnen der Wahrscheinlichkeiten und die Grenzwertbildung aber sehr schwierig. Ich möchte hier deshalb einen anderen Weg aufzeigen. Ich betrachte dazu zu einem sequentiellen Fehler seinen Automaten über den Zustandsmengen wie in Abschnitt 2.3 beschrieben. Die durchschnittliche Länge einer blind homing sequence entspricht hierin der durchschnittlichen Länge eines Weges vom Start- zu einem Endzustand. Spendiere ich dem Automaten noch einen weiteren Zustand, wobei man von jedem Endzustand mit jeder Eingabe zu diesem neuen Zustand gelangt, dann interessiert mich nun also die Frage, wie lange man in einem endlichen Automaten braucht, um von einem Zustand y_1 zu einem Zustand y_2 zu gelangen. Dabei soll y_1 dem Startzustand entsprechen und y_2 dem neu eingeführten Zustand. Die durchschnittliche Länge einer blind homing sequence ist dann genau diese Länge minus eins, da man ja schon in einem der Endzustände sein Ziel erreicht hat, und nicht erst in y_2.

Definition 2.5 *Vom Zustand y_1 komme ich über die Eingabe ω zum ersten Mal beim Zustand y_2 vorbei ($y_1 \xrightarrow{\omega}_1 y_2$) genau dann, wenn $\delta^\omega(y_1) = y_2$ und für keinen echten Präfix u von ω gilt: $\delta^u(y_1) = y_2$.*

Vom Zustand y_1 komme ich über die Eingabe ω nie beim Zustand y_2 vorbei ($y_1 \stackrel{\omega}{\rightarrow}_{nie} y_2$) genau dann, wenn für jeden Präfix u von ω gilt: $\delta^u(y_1) \neq y_2$.

Die erwartete Länge eines Weges vom Zustand y_1 zum Zustand y_2, $\mathrm{E}(y_1 \rightarrow_1 y_2)$ ergibt sich dann durch:

$$\mathrm{E}(y_1 \rightarrow_1 y_2) = \begin{cases} \sum_{i=0}^{\infty} i \Pr(\omega \in X^i, y_1 \stackrel{\omega}{\rightarrow}_1 y_2) & : \sum_{i=0}^{\infty} \Pr(\omega \in X^i, y_1 \stackrel{\omega}{\rightarrow}_1 y_2) = 1 \\ \infty & : \text{ sonst} \end{cases}$$

Lemma 2.10

1. $\sum_{i=0}^{\infty} \Pr(\omega \in X^i, y_1 \stackrel{\omega}{\rightarrow}_1 y_2) = 1 \Leftrightarrow$
   $\forall \omega \in X^*$ *mit* $y_1 \stackrel{\omega}{\rightarrow}_{nie} y_2$ $\exists u \in X^*$ *mit* $\delta^{\omega u}(y_1) = y_2$
   Das heißt man kann sich von y_1 aus nicht verlaufen, sondern findet immer wieder einen Weg nach y_2, falls man nicht schon dort war.

2. $E(y_1 \rightarrow_1 y_1) = 0$.

3. *Sei* $V(y_1) := \{z \in V | \exists x \in X, \delta^x(y_1) = z\}$
   und $p_z := \Pr(x \in X, \delta^x(y_1) = z)$.
   Dann gilt, falls $y_1 \neq y_2$ und man sich von y_1 aus nicht verlaufen kann, daß:
   $$E(y_1 \rightarrow_1 y_2) = 1 + \sum_{z \in V(y_1)} p_z E(z \rightarrow_1 y_2)$$

Beweis:

1. „⇒“: Nehmen wir an, es gäbe ein $\omega \in X^*$ mit $y_1 \stackrel{\omega}{\rightarrow}_{\text{nie}} y_2$, für das kein $u \in X^*$ existiert mit $\delta^{\omega u}(y_1) = y_2$. Es gibt dann eine Wahrscheinlichkeit q echt größer Null, mit einer unendlichen Eingabefolge nie von y_1 nach y_2 zu gelangen, denn mit allen Folgen, die mit ω beginnen, läßt sich dies nie erreichen. $\sum_{i=0}^{\infty} \Pr(\omega \in X^i, y_1 \stackrel{\omega}{\rightarrow}_1 y_2)$ weicht aber um q von 1 ab.

   „⇐“: Es gibt nach Voraussetzung zu jedem Zustand z zu dem ich von y_1 aus gelangen kann, einen Weg zu y_2. Daher läßt sich ein k finden, so daß man von jedem dieser Zustände in weniger als k Schritten zu y_2 gelangen kann. Also gibt es auch ein q mit:

   $$1 > q > \Pr(\omega \in X^k, z \stackrel{\omega}{\rightarrow}_{\text{nie}} y_2) \ \forall z \text{ erreichbar von } y_1$$

   Damit läßt sich folgern, daß:

   $$\Pr(\omega \in X^n, y_1 \stackrel{\omega}{\rightarrow}_{\text{nie}} y_2) \leq q^{\lfloor \frac{n}{k} \rfloor}$$

   Zieht man nun noch in Betracht, daß

   $$\sum_{i=0}^{n} \Pr(\omega \in X^i, y_1 \stackrel{\omega}{\rightarrow}_1 y_2) = 1 - \Pr(\omega \in X^n, y_1 \stackrel{\omega}{\rightarrow}_{\text{nie}} y_2)$$

   dann ergibt sich für $n \rightarrow \infty$ die Behauptung.

2. klar
3.

$$\begin{aligned}
\mathrm{E}(y_1 \rightarrow_1 y_2) &= \sum_{i=1}^{\infty} i \Pr(\omega \in X^i, y_1 \xrightarrow{\omega}_1 y_2) \\
&= \sum_{z \in V(y_1)} \sum_{i=1}^{\infty} i \Pr(\omega \in X^{i-1}, z \xrightarrow{\omega}_1 y_2) p_z \\
&= \sum_{z \in V(y_1)} \underbrace{\sum_{i=1}^{\infty} (i-1) \Pr(\omega \in X^{i-1}, z \xrightarrow{\omega}_1 y_2)}_{\mathrm{E}(z \rightarrow_1 y_2)} p_z \\
&\quad + \sum_{z \in V(y_1)} \underbrace{\sum_{i=1}^{\infty} \Pr(\omega \in X^{i-1}, z \xrightarrow{\omega}_1 y_2)}_{=1 \ \forall z \in V(y_1) \text{ nach Teil 1}} p_z \\
&= 1 + \sum_{z \in V(y_1)} \mathrm{E}(z \rightarrow_1 y_2) p_z
\end{aligned}$$

□

Betrachtet man alle k Zustände auf Wegen zwischen y_1 und y_2, dann liefert das Lemma k Gleichungen für die k unbekannten mittleren Weglängen zu y_2. Durch Lösen des sich daraus ergebenden linearen Gleichungssystems erhält man den gesuchten Wert.

Zum Abschluß dieses Kapitels möchte ich die erwartete Länge einer gewürfelten blind homing sequence für den sequentiellen Fehler aus Abbildung 6 berechnen. Ich nehme dabei für die Wahrscheinlichkeiten der einzelnen Zelleingaben jeweils einfach den Wert $\frac{1}{2}$ an. Gesucht ist also der Wert von $\mathrm{E}(z_{012} \rightarrow_1 z_1)$ im Automaten aus Abbildung 7. Lemma 2.10 liefert dazu folgende Gleichungen:

$\mathrm{E}(z_{012} \rightarrow_1 z_1) = 1 + \frac{1}{2}\mathrm{E}(z_{012} \rightarrow_1 z_1) + \frac{1}{2}\mathrm{E}(z_{12} \rightarrow_1 z_1)$

$\mathrm{E}(z_{12} \rightarrow_1 z_1) = 1 + \frac{1}{2}\mathrm{E}(z_{12} \rightarrow_1 z_1) + \frac{1}{2}\mathrm{E}(z_{20} \rightarrow_1 z_1)$

$\mathrm{E}(z_{20} \rightarrow_1 z_1) = 1 + \frac{1}{2}\mathrm{E}(z_{12} \rightarrow_1 z_1) + \frac{1}{2}\mathrm{E}(z_{01} \rightarrow_1 z_1)$

$\mathrm{E}(z_{01} \rightarrow_1 z_1) = 1 + \frac{1}{2}\mathrm{E}(z_1 \rightarrow_1 z_1) + \frac{1}{2}\mathrm{E}(z_{12} \rightarrow_1 z_1)$

$\mathrm{E}(z_1 \rightarrow_1 z_1) = 0$

Löst man das sich hieraus ergebende lineare Gleichungssystem, so erhält man unter anderem den gesuchten Wert:

$\mathrm{E}(z_1 \rightarrow_1 z_1) = 0$
$\mathrm{E}(z_{01} \rightarrow_1 z_1) = 8$
$\mathrm{E}(z_{20} \rightarrow_1 z_1) = 12$
$\mathrm{E}(z_{12} \rightarrow_1 z_1) = 14$
$\mathrm{E}(z_{012} \rightarrow_1 z_1) = 16$

Kapitel 3

Fehlersimulation kombinatorischer Schaltkreise

Das Generieren von Tests ist ein sehr zeitaufwendiger, da NP-vollständiger Prozeß. Man versucht ihn daher nach Möglichkeit zu vermeiden. Dies gelingt bei Fehlern, die gut „zufallstestbar“ sind, das heißt, daß ein (eventuell geschickt [Wun87, Har90, Har91]) gewürfeltes Schaltkreis-Eingabemuster mit nicht unbedeutender Wahrscheinlichkeit ein Testmuster für sie ist. Dann befindet sich unter einer Menge von einigen Tausend gewürfelten Eingabemustern mit großer Wahrscheinlichkeit auch ein Testmuster. Das muß man aber erst überprüfen, bevor man den Fehler als getestet betrachtet. Genau dies leistet die Fehlersimulation.

Ich beschäftige mich in diesem Kapitel zuerst mit der Fehlersimulation nichthierachischer Schaltkreise für die in Kapitel 2 vorgestellten Fehlermodelle. Für kombinatorische Zellenfehler stelle ich eine Optimierung der Fehlereinpflanzung vor und diskutiere ihre Güte. Zur Fehlersimulation sequentieller Zellenfehler, zu der es noch keine Algorithmen gibt, schlage ich vier Verfahren vor und betrachte ihre Vor- und Nachteile.

Der zweite Abschnitt dieses Kapitels ist der Fehlersimulation hierarchischer Schaltungen gewidmet. Um dabei die Übersichtlichkeit zu wahren, beschränke ich mich nur auf einen einzigen, der für nicht-hierarchische Schaltkreise bekannten Ansätze. Eine intuitive Erweiterung auf hierarchische Schaltungen würde zu quadratischen Laufzeiten schon alleine für die Richtigsimulation führen. Dies motiviert die Suche nach einem geschickteren Algorithmus. Ich werde einen vorstellen, der die Unzulänglichkeiten – mit Ausnahme des Problems des „virtuellen Feedbacks“, um das ich mich gesondert kümmere – beseitigen wird und zudem in der Lage sein, einen Schaltkreis in linearer Zeit zu simulieren!

3.1 Nicht-hierarchische Schaltkreise

3.1.1 Fehlersimulation für das stuck-at-Modell

Fehlersimulatoren für single-stuck-at-Fehler in nicht-hierarchischen Schaltungen gibt es mit sehr verschiedenen Ansätzen [KS85, AS87, DG87, BHKS91]. Ich möchte mich in den folgenden Kapiteln mit Weiterentwicklungen von [KS85, DG87] beschäftigen und den Algorithmus deshalb kurz beschreiben:

gegeben

- ein Schaltkreis $C = (T, M, S, \tau, \rho, type)$
- eine Fehlerliste $F \subseteq \{(s, x) | s \in S, x \in \mathbf{B}\}$
- eine Menge von Schaltkreiseingabemustern $E \subseteq \mathbf{B}^{indegree(\tau)}$

gesucht

- die Fehler $\widetilde{F} \subseteq F$ des Schaltkreises C aus der Fehlerliste F, die mit den Schaltkreiseingabemustern E getestet werden können.
- die Muster $\widetilde{E} \subset E$ mit denen Fehler aus F getestet werden können.

In einer Vorbereitungsphase wird beim compilergesteuerten Ansatz eine schaltungsspezifische Simulationsroutine erzeugt. Dies soll den zeitaufwendigen Simulationsprozeß gegenüber einer interpretierten Lösung beschleunigen. Da sich kombinatorische Schaltungen topologisch sortieren lassen, ist die erzeugte Routine schleifenfrei. Eine Änderung eines Signalwertes wird daher nicht wieder überschrieben. Man kann Fehler deshalb einfach durch Überschreiben von Leitungswerten einpflanzen und in der Simulationsroutine ab der entsprechenden Stelle fortfahren. Die Werte der einzelnen Signale werden in einem Feld gespeichert und ersparen dadurch die Simulation der Schaltung bis zu jeder einzelnen Fehlerstelle. In der eigentlichen Arbeitsphase wird für jedes Eingabemuster der Schaltkreis einmal komplett durchsimuliert. Dadurch sind die Werte der einzelnen Signale und vor allem auch der primären Ausgänge bekannt. Nun kann jeder Fehler durch Invertieren eines Signalwertes in die Schaltung eingepflanzt werden. Nach erneuter Simulation, diesmal aber nur der restlichen Schaltung, kann man die korrekten Werte der primären Ausgänge mit den aktuellen vergleichen und den Fehler als getestet markieren, wenn sie differieren. Beachtet man die Reihenfolge der Fehler und pflanzt sie aus Sicht der Simulationsroutine „rückwärts“ ein, so stehen für jeden Fehler die benötigten Signalwerte aus der Korrektsimulation zur Verfügung. Eine Sicherheitskopie oder gar eine erneute Richtigsimulation vor jeder Simulation des Fehlers kann man sich damit ersparen.

Für jede Zelle wird ihr Verhalten in einer Zellbibliothek beschrieben. Dies kann zum Beispiel als Ausdruck einer Programmiersprache geschehen, indem Platzhalter für die Ein- und Ausgänge bei der Erzeugung der Simulationsroutine

durch reale Adressen ersetzt werden. Als Speicher für die Werte der Signale wird ein genügend großes Feld angelegt. Ein Bibliothekseintrag für ein Nand-Gatter und eine erzeugte Simulationsroutine könnten etwa wie folgt aussehen:

```
code(nand2x1):  Feld[o] = not( Feld[i1] and Feld[i2] );

---

const Feld_laenge = ...;
long Feld[ Feld_laenge ];

void simulation( ab_wo )
int ab_wo;
{
 switch( ab_wo )
 {
  case 0: Feld[0] = ...;
  case 1: Feld[1] = ...;
  ...
  case 8: Feld[8] = not( Feld[0] and Feld[1] );
  ...
 }
}
```

Der komplette Algorithmus in Kurzschreibweise:

- bestimme eine topologische Sortierung $(S, \leq_{top})$ der Signale S der Schaltung
- erzeuge die Simulationsroutine $simulation(int)$
- $\widetilde{E} \leftarrow \emptyset; \widetilde{F} \leftarrow \emptyset;$
- for each $m \in E$ do
  - berechne Signalbelegung $\beta : S \rightarrow \mathbf{B}$ mit $\beta(\rho^{in}(\tau)) = m$ durch $simulation(0)$
  - $correct \leftarrow \beta(\rho^{out}(\tau))$
  - while $F \neq \emptyset$ do
    - $F \leftarrow F \backslash (s_j, x)$
      mit $(s_j, x) \in F$ und $s \leq s_j \forall (s, y) \in F$
    - $\beta(s_j) \leftarrow x$
    - berechne durch $simulation(j+1)$ die restliche Signalbelegung β neu
    - if $correct \neq \beta(\rho^{out}(\tau))$ then $\widetilde{F} \leftarrow \widetilde{F} \cup \{(s_j, x)\}; \widetilde{E} \leftarrow \widetilde{E} \cup \{m\}$ fi

od

od

3.1.2 Fehlersimulation für kombinatorische Zellenfehler

Eine Verallgemeinerung des im vorigen Abschnitt vorgestellten Algorithmus auf kombinatorische Zellenfehlermodelle wird in [HSS91] beschrieben. Ich möchte daher an dieser Stelle nicht näher darauf eingehen, sondern mich nur mit einer möglichen Verbesserung beschäftigen. Bei Zellen mit mehr als einem Ausgang können sich verschiedene Fehler unter der gleichen Zelleingabe unterschiedlich an den Zellausgängen auswirken. Das führt zu mehreren Simulationsläufen für eine Zelle pro Schaltkreiseingabe. Schnelle Fehlersimulatoren bearbeiten immer mehrere Eingabemuster gleichzeitig. Sie nutzen dabei die in Computern gegebenen, wortbreiten Operationen aus. Jede Bitposition repräsentiert eine Eingabe. Damit erhält man an der zu untersuchenden Zelle an jeder Bitposition eine Menge von einpflanzbaren, fehlerhaften Auswirkungen. Die Kardinalität der größten Menge bestimmt dann die Anzahl der Simulationen, die für die Zelle zu erfolgen hat.

Dies ändert sich aber, wenn eine Simulation erfolgreich verläuft, wenn man erkennt, daß ein Fehler durch eine der Eingaben getestet wird. Dadurch können nämlich an anderen Bitpositionen fehlerhafte Auswirkungen wegfallen, wenn sie nur nach dem nun gefundenen Fehler gefahndet haben! Damit kann sich die Anzahl der Simulationsläufe senken, wenn die Simulation für diese Auswirkung nicht schon erfolgt ist. Die Reihenfolge der eingepflanzten Auswirkungen an den einzelnen Bitpositionen spielt also eine Rolle. Hier stellt sich natürlich die Frage, ob man diese Reihenfolge irgendwie optimieren kann? Wobei man zunächst klären muß, was hier „optimal“ bedeutet. Eine optimale Fehlereinpflanzung ist eine Auswahl von maximal *cost* Wirkungen an jeder Bitposition, mit denen alle Fehler gefunden werden können und *cost* dabei minimal ist. Dabei wird die Möglichkeit eines Mißerfolges einer Wirkung beim Simulieren völlig außer Acht gelassen! Aber selbst unter dieser vereinfachenden Vorraussetzung kann man eine optimale Fehlereinpflanzung nur sehr schwer finden.

Um dies zu zeigen, formalisiere ich das Problem: Gegeben ist eine $M \times F$ Matrix A. Die Spalten der Matrix sollen dabei die Fehler einer Zelle und die Zeilen die Bitpositionen eines Maschinenwortes repräsentieren. Die Einträge a_{ij} seien dabei aus der Menge $\{0, 1, \ldots, k\}$. Der Wert in Zeile i und Spalte j soll dabei die Nummer der durch die Zelleingabe an Bitposition i und den Fehler j eindeutig definierten Wirkung angeben, die man an der Bitposition i einpflanzen muß um den Fehler j zu testen – falls es keine solche Wirkung gibt, soll hier eine Null stehen. Gesucht sind nun M Funktionen

$$choose_1, \ldots, choose_M : \{1, \ldots, cost\} \rightarrow \{1, \ldots, k\}$$

$$\text{so daß } \forall j \; \exists i \text{ mit } a_{ij} \in \{choose_i(1), \ldots, choose_i(cost)\}$$

die *cost* minimieren. Da kombinatorische Zellenfehler eine große Vielfalt bieten, kann man umgekehrt aus einer $M \times F$ Matrix auch eine Zelle mit entsprechendem Fehlverhalten konstruieren.

Satz 5 *Die Entscheidung, ob es eine Fehlereinplanzung mit Kosten cost gibt, ist NP-vollständig.*

Zeigen möchte ich hier jedoch nur die NP-Härte des Problems, ich reduziere zu diesem Zweck 3-SAT darauf: Gegeben sind die Klauseln $K_1, \ldots, K_F$ über den Variablen $X_1, \ldots, X_M$. Die einzelnen Klauseln bestehen dabei aus genau drei Literalen. 3-SAT zu lösen heißt nun, zu klären, ob es eine Belegung der Variablen gibt, so daß $(K_1 \wedge K_2 \wedge \ldots \wedge K_F)$ wahr ist. Daraus erzeuge ich nun die $M \times F$ Matrix A mit

$$a_{ij} = \begin{cases} 0 & : \quad X_i \notin K_j \text{ und } \neg X_i \notin K_j \\ 1 & : \quad X_i \in K_j \\ 2 & : \quad \neg X_i \in K_j \end{cases}$$

Ich behaupte, daß A eine optimale Fehlereinpflanzung mit $cost = 1$ genau dann hat, wenn $(K_1 \wedge K_2 \wedge \ldots \wedge K_F)$ erfüllbar ist.
Beweis:

$\Rightarrow$: Nimm eine optimale Fehlereinpflanzung von A. Setze $X_i :=$ wahr, wenn $choose_i(1) = 1$, ansonsten $X_i :=$ falsch . Da für jedes j ein i existiert mit $a_{ij} = choose_i(1)$, gibt es in jeder Klausel K_j auch ein Literal X_i oder $\neg X_i$, das erfüllt ist und somit ist 3-SAT erfüllbar.

$\Leftarrow$: Nimm eine Belegung der Variablen, die $(K_1 \wedge K_2 \wedge \ldots \wedge K_F)$ erfüllt. Für jedes i setze $choose_i(1) := 1$ wenn $X_i =$ wahr, ansonsten $choose_i(1) := 2$. Ist in einer Klausel K_j das Literal X_i erfüllt, dann ist $a_{ij} = 1 = choose_i(1)$. Ebenso gilt $a_{ij} = 2 = choose_i(1)$, falls $\neg X_i$ in K_j erfüllt ist. Da in jeder Klausel K_j ein Literal X_i oder $\neg X_i$ erfüllt ist, ist $choose_i(1) = a_{ij}$ und somit gibt es eine optimale Fehlereinpflanzung mit $cost = 1$.

Es ist also nicht zu erwarten, daß die Anzahl der Simulationsdurchläufe auf das Optimum reduziert werden kann. Man sollte aber denoch versuchen, mit möglichst einfachen Mitteln nahe an dieses Optimum heranzukommen. Dazu einige Vorschläge:

- Ich habe bisher vorausgesetzt, daß sich jede eingepflanzte Wirkung bis zu den primären Ausgängen fortpflanzen kann. Da dies normalerweise nicht der Fall ist, sollte nicht die gesamte Auswahl auf einmal, sondern nur für die nächsten einzupflanzenden Wirkungen, vor jedem Simulationslauf, getroffen werden.

- Wähle der Reihe nach für jede Bitposition eine Wirkung aus. Versuche dabei möglichst viele Fehler möglichst gleichmäßig zu überdecken. Ordne

Muster	im16	gm16	gm32
32	896 - 448 = 448	479 - 307 = 172	2.112 - 1.524 = 588
224	1.031 - 583 = 448	1.362 - 1.185 = 177	7.551 - 6.917 = 634
1024	1.381 - 933 = 448	4.223 - 4.046 = 177	26.468 - 25.830 = 638
30016	14.071 - 13.623 = 448	30.451 - 30.247 = 204	508.944 - 508.218 = 726

Tabelle 3.1: Nutzen der optimierten Fehlereinpflanzung

> dazu jedem Fehler einen Überdeckungsgewinn zu, der nach jeder Wirkungsauswahl aktualisiert wird. Wähle als die nächste Wirkung diejenige aus, die den größten Gewinn über allen Fehlern erzielen kann.

Wieviele Simulationsläufe lassen sich durch diese Heuristik konkret sparen? Bei den ersten, zu untersuchenden Eingabemustern bei einer noch geringen Fehlerüberdeckung sicherlich mehr, als bei späteren, wenn schon eine hohe Fehlerüberdeckung erreicht ist. Zur Verdeutlichung habe ich die Heuristiken in einen Fehlersimulator für kombinatorische Zellenfehler eingebaut und die Anzahl der Simulationsläufe mit denen der ursprünglichen Version verglichen. Deren Vorgehensweise war folgende: Man betrachtet nacheinander alle nützlichen Wirkungen und pflanzt sie auf allen Bitpositionen ein, an denen sie Fehler finden können und die noch nicht von anderen Wirkungen besetzt sind. Die Resultate für einen 16-Bit Integer- und zwei 16 und 32-Bit Gleitkommamultiplizierer aus [Spa91] finden Sie in Tabelle 3.1. Darin wird deutlich, daß eine Optimierung der Fehlereinpflanzung nur zu Anfang erfolgreich ist. Später wird man hingegen den durch die Optimierung verursachten Mehraufwand teuer bezahlen, wenn er nutzlos verpufft!

3.1.3 Fehlersimulation für sequentielle Zellenfehler

Fehlersimulatoren für sequentielle Zellenfehler gibt es noch nicht! Allenfalls Sonderfälle, wie das stuck-open-Modell werden unterstützt. Was ist also zu tun?

Kennt man den Zustand eines sequentiellen Fehlers bei Erreichen einer bestimmten Schaltkreiseingabe in der gesammten Folge, mit der die Schaltung getestet werden soll, so kann man mit einem Fehlersimulator für kombinatorische Zellenfehler entscheiden, ob der Fehler mit der Schaltkreiseingabe entdeckt wird oder nicht.

Das Problem besteht also in einer geschickten Berechnung der Folgezustände aller sequentiellen Fehler. Auch hier sollte man die wortparallele Verarbeitung eines Computers ausnützen, und es ermöglichen, die Zustände für mehrere Zelleingaben zu berechnen. Ich möchte im folgenden vier verschiedene Lösungsansätze vorstellen und ihre Vor- und Nachteile diskutieren.

Die intuitive Lösung

Ein Lauf des Fehlersimulators ist in der Lage M Muster zu verarbeiten. Also berechnet man sich nacheinander die nächsten M Zustände eines Fehlers und speichert sich den letzten Zustand ab. Er steht dann beim nächsten Lauf als Startzustand für die nächsten M Muster wieder zur Verfügung. Nur ganz zu Beginn muß dieser Zustand bis zum Erreichen einer blind homing sequence im Automaten über Zustandsmengen berechnet werden.

Vorteil dieser Lösung ist die Einfachheit – dies sorgt für eine schnelle Realisierung – und der geringe Speicherbedarf, nämlich nur ein Zustand pro sequentiellem Fehler. Wobei hierbei die Hoffnung zum Ausdruck kommt, die erwartete Länge einer blind homing sequence läge deutlich unter dem Wert von M, und es daher nicht nötig ist, einen Zustand des exponentiell größeren Automaten über Zustandsmengen abzuspeichern. Als Nachteil darf die benötigte Rechenzeit für M Zustandsberechnungen angesehen werden.

Parallele Präfixberechnung

Diese Lösung knüpft an der intuitiven Lösung an. Sie versucht den Nachteil der M Zustandsberechnungen zu eliminieren. Die Idee stammt aus einem Vorschlag zur schnellen Realisierung von endlichen Automaten auf Hardwareebene [LF80]. Man ordnet dabei jeder Eingabe des Automaten eine Funktion von dessen Zuständen wieder auf dessen Zustände zu. Diese Funktion beschreibt den Übergang, den der endliche Automat unter der Eingabe macht. Interessiert man sich für eine Folge von Eingaben, so kann man die Funktionen einfach mittels Hintereinanderausführung verknüpfen. Da diese Verknüpfung assoziativ ist und man mehrere davon parallel abarbeiten kann, benötigt man dazu nur logarithmisch zur Anzahl der Eingaben viel Zeit. Angewendet auf den Startzustand liefert die erhaltene Funktion den gesuchten Zustand. Abbildung 10 zeigt ein mögliches Berechnungsschema. Zuerst werden alle benachbarten Stellen miteinander verknüpft, dann alle mit Abstand zwei, vier, acht usw.

Wie läßt sich dies nun auf Softwareebene realisieren? Man hat ja nur parallele Operationen zwischen irgendwelchen Registern und nicht innerhalb! Kopiert man sich das Register und shiftet dann die Kopie um einen oder zwei, vier, acht usw. Stellen, dann kann man die Verknüpfung dennoch mit den bitweisen Operationen realisieren. Man muß lediglich darauf achten, daß beim Shiften in die freigewordenen Stellen der Wert der Idenditätsfunktion geschrieben wird.

Leider ist der Aufwand einer Funktionsverknüpfung wesentlich größer als die Berechnung eines Nachfolgezustandes. Deshalb ist diese Lösung der intuitiven Lösung nur für sehr große M überlegen. Für Einzelfälle, bei denen sich eine besonders leichte Funktionsverknüpfung ergibt, könnte sie aber auch bei den heute üblichen Maschinenwortbreiten konkurrenzfähig sein.

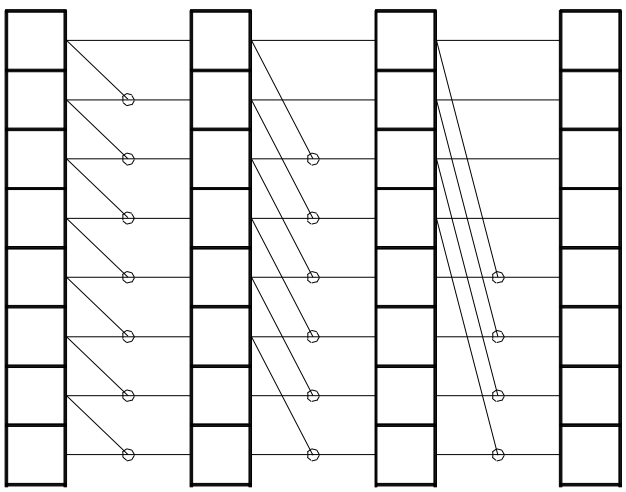

Bild 10: Parallele Präfixberechnung

Die optimistische Lösung

Bei der intuitiven Lösung werden die M Zustände nacheinander berechnet. Bei jeder dieser Berechnungen steht aber noch Platz in den Registern zur Verfügung, der nicht genutzt wird. Man könnte doch einfach an diesen Stellen auch einen Übergang berechnen. Zu Beginn steht an der Stelle eins der Startzustand, alle anderen Zustände sind unbekannt, enthalten also „Schrott“. Nun wird parallel an jeder Stelle der Nachfolgezustand berechnet und um eine Position verschoben. An die freiwerdende Stelle eins kann nun wieder der Startzustand geschrieben werden. Nach einer solchen Berechnung stehen an der ersten und zweiten Stelle korrekte Werte, und an den übrigen Stellen die Nachfolgezustände vom ursprünglichen Schrott. Interessant wird das Ganze, wenn irgendwo sich ein korrekter Wert bildet und nicht mehr verschwindet. Ab dort werden sich dann nämlich nur noch korrekte Werte propagieren. Eine solche Stelle findet sich immer am Ende einer blind homing sequence. Das Verfahren hofft also auf deren häufiges Vorkommen. Der eigentliche Rechenaufwand ist nicht wesentlich höher als bei der intuitiven Lösung, abgesehen von der Notwendigkeit eines Abbruchkriteriums. Der Vergleich der Zustände vor und nach einer Berechnung ist aber nicht teuer und zeigt die Korrektheit aller Zustände bei Gleichheit an.

Satz 6 *Sei ω eine kürzeste blind homing sequence, $k := |\omega|$ sowie $q := \Pr(u \in X^k, u \notin bhs)$. Dann benötigt die optimistische Lösung durchschnittlich nicht mehr als*

$$k(\frac{1}{1-q} - 1 + \log_{\frac{1}{q}} \frac{M}{k})$$

Zustandsberechnungen.

Ich möchte diesen Satz der Einfachheit halber nur für den Fall $k = 1$ beweisen. Die allgemeine Form ergibt sich daraus durch Unterteilen der Eingabefolge in Blöcke der Länge k und scharfes Hinsehen.

Die Anzahl der benötigten Zustandsberechnungen ist kleiner als der größte Abstand zweier blind homing sequences. Würfelt man an jeder Position von 1 bis M mit der Wahrscheinlichkeit q eine Null ($\Pr(x_i = 0) = q$) und mit $1 - q$ eine Eins ($\Pr(x_i = 1) = (1 - q)$), so muß man die Länge der längsten Nullfolge in $x_1, x_2, \cdots, x_M$ berechnen. Für $q = \frac{1}{2}$ entspricht dies gerade der mittleren Laufzeitabschätzung eines von Neumannaddierwerkes in [Hot72]. Die folgende Rechnung stellt nur eine Verallgemeinerung des dort vorgestellten Beweises dar.

Ereignis $E_{i,n}$: $\exists$ Nullfolge der Länge $\geq i$ in $x_1, x_2, \cdots, x_n$
Ereignis $E'_{i,n}$: $x_1, x_2, \cdots, x_{n-1}$ hat keine Nullfolge der Länge $\geq i$
und $x_1, x_2, \cdots, x_n$ hat Nullfolge der Länge $\geq i$
$E_{i,n} = E_{i,n-1} \cup E'_{i,n}$
$\Pr(E'_{i,n}) \leq q^i$
$\Pr(E_{i,n}) \leq \Pr(E_{i,n-1}) + q^i \leq \cdots \leq (n - i + 1)q^i \leq nq^i$
$\Pr(E_{i,n}) \leq \min(1, nq^i)$

Damit läßt sich der gesuchte Erwartungswert μ_n abschätzen:

$$\mu_n = \sum_{i=1}^{n} i(\Pr(E_{i,n}) - \Pr(E_{i+1,n})) = \sum_{i=1}^{n} \Pr(E_{i,n})$$

$$\leq \sum_{i=1}^{n} \min(1, nq^i) = \sum_{i=1}^{\lfloor \log_{\frac{1}{q}} n \rfloor} 1 + \sum_{i=\lfloor \log_{\frac{1}{q}} n \rfloor + 1}^{n} nq^i$$

$$= \lfloor \log_{\frac{1}{q}} n \rfloor + \sum_{i=1}^{n - \lfloor \log_{\frac{1}{q}} n \rfloor} nq^i \underbrace{q^{\lfloor \log_{\frac{1}{q}} n \rfloor}}_{\leq \frac{1}{n}} \leq \lfloor \log_{\frac{1}{q}} n \rfloor + \sum_{i=1}^{n} q^i$$

$$\leq \lfloor \log_{\frac{1}{q}} n \rfloor + \frac{1 - q^{n+1}}{1 - q} - 1 \leq \frac{1}{1-q} - 1 + \log_{\frac{1}{q}} n$$

□

Die Güte der optimistischen Lösung hängt also entscheidend von der Länge der

kürzesten blind homing sequence und der Wahrscheinlichkeit q ihres Auftretens in einer zufälligen Eingabefolge ab. Für sehr kleine sequentielle Fehler gibt es aber meistens schon blind homing sequencen der Länge eins und die Hälfte aller Eingaben sind solche. Damit erwartet man also $1 + \log_2 M$ Zustandsberechnungen, wesentlich weniger also, als die M der intuitiven Lösung.

Zerhacken der Musterfolge

Alle bisher betrachteten Lösungen gehen davon aus, daß die M Eingaben an der Zelle nacheinander anliegen. Wesentlich einfacher ist es aber, davon auszugehen, die Eingaben wären voneinander unabhängig. Dann ließen sich die M benötigten Zustände nämlich in einem Schritt aus den abgespeicherten, letzten M berechnen. Dazu muß man nur die Folge von Schaltkreiseingaben in M kleinere Folgen „zerhacken“ und dann Fehlersimulation für jede dieser Folgen parallel leisten. Es ergibt sich dadurch jedoch ein Problem: Nun ist nicht mehr nur der Startzustand des Fehlers für die Gesamtfolge unbekannt, sondern für jede Einzelfolge! Man muß nun für jede Einzelfolge eine blind homing sequence finden.

Das kann aber dennoch vernünftig sein, wenn die erwartete Länge einer blind homing sequence deutlich unter der Länge der Einzelfolgen liegt. Dann ist es nämlich durchaus vertretbar, nach der Fehlersimulation für die Einzelfolgen - nach denen die anfangs unbekannten Zustände vorliegen - den Beginn der Einzelfolgen nochmals zu bearbeiten. Da dies nur für die nichtgefundenen Fehler erfolgen muß, ist das nicht zu teuer.

Nachteil dieser mit Abstand schnellsten Lösung ist der hohe Speicherplatzverbrauch, denn nun müssen M Zustände abgespeichert werden - im Gegensatz zu nur einem bei den drei vorherigen Lösungen. Zusätzlich muß man bedenken, daß man dort hoffen konnte, nach den ersten M Simulationen schon eine blind homing sequence gefunden zu haben und damit nur einen Zustand des sequentiellen Fehlers, nicht einen des exponentiell größeren Automaten über Zustandsmengen, abspeichern zu müssen. Bei der Lösung durch Zerhacken der Eingabefolge muß man die Zustände nach jeder Simulation abspeichern, anfangs werden dies die platzfressenden des Automaten über Zustandsmengen sein.

Fazit

Je nach primärer Zielsetzung kann man sich für eine der drei etwas langsameren, platzsparenden Lösungen oder die schnelle platzaufwendigere Lösung entscheiden. Die intuitive Lösung ist am einfachsten zu implementieren, die parallele Präfixberechnung mag in Einzelfällen bevorzugt werden, die optimistische verbindet Platz- und Zeitgewinn.

Ein „guter“ Fehlersimulator könnte eine Analyse der auftretenden Fehler vorschalten und sich erst dann für eine der Lösungen entscheiden!

3.2 Hierarchische Schaltkreise

Die nachfolgenden Betrachtungen zur Fehlersimulation hierarchischer Schaltkreise beziehen sich immer auf kombinatorische Zellenfehler. Der Übergang zu sequentiellen Zellenfehlern geschieht völlig analog dem bei nicht-hierarchischen Schaltkreisen.

3.2.1 Erwartungen

Die compilergesteuerte Fehlersimulation stößt bei den heute üblichen Schaltkreisgrößen an ihre Grenzen. Compiler werden für menschliche Progammierer entwickelt und sind daher nicht immer für maschinell erzeugten Code geeignet. So bereitete die in [HSS91] beschriebene Implementierung große Schwierigkeiten durch Begrenzungen in der Anzahl von case-Statements innerhalb einer switch-Anweisung. Hierarchische Beschreibungen von Schaltkreisen sind im Vergleich zu der aufgelösten Version wesentlich kleiner. Dadurch kann man hoffen, die technischen Schranken der verwendeten Compiler einzuhalten.

Fehlermodelle, wie zum Beispiel das stuck-at-Modell, legen eine Betrachtung von Schaltkreisen auf der Ebene des gate-level nahe. Die Entwicklung großer Schaltungen kann aber nur durch Modularisierung effizient und einigermaßen korrekt erfolgen. Moderne Designsysteme wie zum Beispiel [BHK+87] bieten darüber hinaus noch die Möglichkeit, Schaltungsteile rekursiv zu beschreiben. Daraus entsteht eine hierarchische Beschreibung des Schaltkreises, die üblicherweise erst zum Testen auf gate-level aufgelöst wird. Dadurch verliert man aber an Information, die den Algorithmen vielleicht die Arbeit erleichtern könnte.

Von einem hierarchischen Fehlersimulator erhofft man sich eine Einsparung an Speicherplatz, weil eine flache Schaltung Informationen redundant halten muß, die in einer hierarchischen Darstellung von gleichen Schaltungsteilen nur einmal abgelegt zu werden braucht. Aber auch die Compilezeit, ein oft verschwiegenes Kapitel der compilergesteuerten Fehlersimulation, sollte im hierarchischen Fall aus dem gleichen Grund deutlich abnehmen. Befürchtet werden muß aber, diese Vorteile durch eine deutliche Zunahme der Laufzeit zu bezahlen.

Ich möchte zunächst einmal die intuitive Version eines hierarchischen, compilergesteuerten Fehlersimulators vorstellen und dann über die von ihr aufgeworfenen Probleme und Kosten sprechen. Den Abschluß dieses Kapitels soll die Beschreibung eines wesentlich effizienteren Verfahrens bilden.

3.2.2 Die intuitive Lösung

Ein Teilproblem der Fehlersimulation ist die eigentliche Simulation des Schaltkreises. Dazu wird beim compilergesteuerten Ansatz eine eigene Routine für jede zu untersuchende Schaltung erzeugt. Die Vorstellung der Simulation hierarchischer Schaltungen führt zu der Idee von Prozeduren für jede Box des Schaltkreises. Ein Vorkommen der Box wird dann durch einen Prozeduraufruf erledigt.

Als Beispiel, wie das aussehen könnte, möchte ich solche Prozeduren für den Volladdierer aus Kapitel 1 auflisten, der aus zwei Halbaddierern und einem Odergatter aufgebaut ist. Die Halbaddierer bestehen ihrerseits aus einem And- und einem Xor-Gatter, das seinerseits mit vier Nand-Gattern realisiert wird.

```
procedure FA( value i1, i2, i3; reference sum, carry )
{
 variable sum1, carry1, carry2;
 HA( i1, i2, sum1, carry1 );
 HA( i3, sum1, sum, carry2 );
 carry = carry1 or carry2;
}

procedure HA( value i1, i2; reference sum, carry )
{
 XOR( i1, i2, sum );
 carry = i1 and i2;
}

procedure XOR( value i1, i2; reference o1 )
{
 variable s1, s2, s3;
 s1 = not( i1 and i2 );
 s2 = not( i1 and s1 );
 s3 = not( s1 and i2 );
 o1 = not( s2 and s3 );
}
```

Formal stellt sich das Problem der hierarchischen Fehlersimulation wie folgt:

gegeben

- ein hierarchischer Schaltkreis $H = (C_1, C_2, \cdots, C_k)$
- eine Fehlerliste $F \subseteq \{(I, err) | I$ ist Basiszelleninstanz von $H, err : \mathbf{B}^{indegree(I)} \rightarrow \mathbf{B}^{outdegree(I)}\}$
- eine Menge von Schaltkreiseingabemustern $E \subseteq \mathbf{B}^{indegree(\tau_k)}$

gesucht

- die Fehler $\widetilde{F} \subseteq F$, die mit den Schaltkreiseingabemustern aus E getestet werden können.
- die Muster $\widetilde{E} \subset E$, mit denen Fehler aus F getestet werden können.

Der Algorithmus zur Fehlersimulation könnte dann grob etwa so aussehen:

- for $C_i = C_1$ to C_k do
  - sortiere die Module M_i der Schaltung C_i topologisch
  - erzeuge die Simulationsroutine $simulation_{C_i}()$
- $\widetilde{E} \leftarrow \emptyset; \widetilde{F} \leftarrow \emptyset;$
- for each $m \in E$ do
  - berechne die Belegung der primären Ausgänge durch $simulation_{C_k}$: $correct \leftarrow behave(C_k)(m)$
  - for each $(I, err) \in F$ do
    - berechne Belegung der primären Ausgänge $behave(C_k)(m)$ unter der Annahme $behave(I) := err$ neu.
    - if $correct \neq behave(C_k)(m)$ then $\widetilde{F} \leftarrow \widetilde{F} \cup \{(I, err)\}$; $\widetilde{E} \leftarrow \widetilde{E} \cup \{m\}$ fi

    od

od

Die Berechnung der Belegung der primären Ausgänge unter der Annahme, das Verhalten einer Basisinstanz unterscheide sich von der korrekten, ist dabei nicht einfach, denn es gibt nun im Gegensatz zur nicht-hierarchischen Lösung keine zu der Instanz korrespondierende Codestelle. Man müßte etwa bei jeder Berechnung einer Basiszelle prüfen, ob die gesuchte Instanz erreicht ist. Doch dies ist leider nicht das einzige Problem. Deshalb läßt sich die intuitive Idee so einfach nicht realisieren.

3.2.3 Topologische Sortierbarkeit

Der in Abschnitt 3.2.2 vorgestellte Algorithmus braucht zur Erzeugung der Simulationsroutine für eine Schaltung C_i die topologische Sortierbarkeit ihrer Module M_i. Bei flachen Schaltkreisen stellt dies kein Problem dar, denn topologisch sortierbar und kombinatorisch sind dort äquivalente Begriffe. Bei hierarchischen Schaltkreisen kann es aber durchaus vorkommen, daß sich ein Ausgang eines Moduls bis zu einem Eingang des selben Moduls fortpflanzt und sich trotzdem nach Auflösen der Hierarchie eine flache, kombinatorische Schaltung ergibt. Dies ist immer dann möglich, wenn ein Ausgang eines Moduls nicht von allen Eingängen abhängt. Als Beispiel mag Abbildung 11 dienen.

Definition 3.1 *Ein hierarchischer Schaltkreis* $H = (C_1, C_2, \cdots, C_k)$ *hat ein* virtuelles Feedback *genau dann, wenn es für eine seiner Teilschaltkreise* C_i *keine topologische Sortierung gibt, seine Auflösung* $flat(H)$ *aber ein kombinatorischer Schaltkreis ist.*

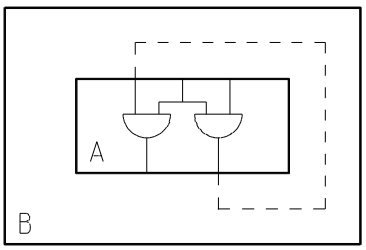

Bild 11: virtuelles Feedback

Zur Lösung des Problems des virtuellen Feedbacks zur Erzeugung von Simulationsroutinen möchte ich im folgenden zwei Vorschläge vorstellen.

erste Lösung: Auflösen von Boxinstanzen

Wenn die flache Darstellung einer Schaltung keine Feedbacks hat, wohl aber die hierarchische, so liegt es nahe, die hierarchische Schaltung so weit aufzulösen, bis die Feedbacks alle verschwunden sind. Diese Lösung ist schon etwas älter, sie geht auf [Ser88] zurück. Leider wird in [Ser88] auch gezeigt, daß man nicht nur gezwungen werden kann, die Schaltung komplett aufzulösen, das heißt auf Hierarchie vollständig zu verzichten, sondern daß auch das Auffinden einer möglichst kleinen Auflösung als NP - vollständiges Problem nicht ganz leicht fällt. Dies motiviert die Suche nach einer besseren Lösung

zweite Lösung: erhöhter Simulationsaufwand

Nach der Simulation der Zelle A in Abbildung 11 ist der Wert am rechten Ausgang korrekt, wenn der mittlere und rechte Eingang bekannt sind. Wir können ihn nun den Modulen auf dem Pfad zum linken Eingang von Modul A weiterreichen. Dadurch wird auch der Wert an diesem Eingang bekannt. Eine erneute Simulation des Modules A führt zu einer korrekten Simulation für das Modul B. Dabei wurde das Modul A allerdings zweimal simuliert, das heißt der Simulationsaufwand nimmt zu.

Um die Güte dieser Lösung beurteilen zu können, möchte ich folgendes Beispiel einführen: Der Schaltkreis C_0 habe gemäß Abbildung 12 zwei Ein- und zwei Ausgänge. Dabei soll der „obere“ Ausgang nur vom „oberen“ Eingang abhängen und ebenso der „untere“ Ausgang nur vom „unteren“ Eingang. Der Schaltkreis C_i bildet sich für $i \geq 1$ jeweils, wie in Abbildung 12 gezeigt, aus dem Schaltkreis C_{i-1}. Da auch hier der obere Ausgang nur vom oberen Eingang so-

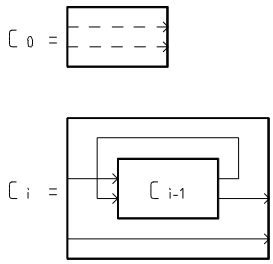

Bild 12: Schaltung mit virtuellen Feedbacks

wie der untere Ausgang nur vom unteren Eingang abhängen, folgt induktiv, daß der Feedback in der Schaltung C_i virtuell ist, das heißt, nach Auflösen aller Teilschaltkreise C_i ergibt sich ein flacher, kombinatorischer Schaltkreis.

Will man die hierarchische Schaltung $H = (C_0, C_1, \cdots, C_k)$ mit Hilfe der ersten Lösung simulieren, so kommt man zu folgenden Simulationskosten: Die Kosten der Schaltung C_0 sind unabhängig von i und können deshalb als Konstante c_0 betrachtet werden. Um die Schaltung C_i simulieren, muß man zuerst die sich darin befindliche Instanz der Schaltung C_{i-1} simulieren – dann hat man den Wert des oberen Ausgangs dieser Instanz – und dann diese Instanz nochmal, um auch den Wert des unteren Ausgangs zu erhalten. Der restliche Aufwand in C_i, der unabhängig von i ist, werde durch eine Konstante c_1 abgedeckt.

$cost(C_0) := c_0$
$cost(C_i) = c_1 + 2cost(C_{i-1})$

$$\Rightarrow cost(C_k) = c_1{+}2(c_1{+}2(\cdots{+}2(c_1{+}2c_0)\cdots)) = 2^k c_0 + \sum_{i=0}^{k-1} 2^i c_1 = 2^k(c_0{+}c_1){-}c_1$$

Gäbe es eine Lösung, die mit nur einer einfachen Simulation der Instanz C_{i-1} in der Schaltung C_i auskommt, dann käme man zu folgender Rechnung:

$einfach(C_0) := c_0$
$einfach(C_i) = c_1 + einfach(C_{i-1})$

$$\Rightarrow einfach(C_k) = c_1 + c_1 + \cdots + c_1 + c_0 = kc_1 + c_0$$

Gegenüber einer solchen Lösung, die sicherlich immer noch teurer ist als Simulation auf dem flachen Schaltkreis, hat man sich also einen erhöhten Simulationsaufwand in exponentieller Größe des flachen Schaltkreises eingehandelt!

Fazit

Nachdem die zweite Lösung durch ihre hohe Laufzeit nicht in Frage kommt, muß man sich wohl oder übel auf die Nachteile der ersten Lösung einlassen. Das heißt, man kann dazu gezwungen werden, Hierarchie vollständig aufzugeben. Dies mag allerdings ein sehr unrealistischer Fall sein, dennoch wird man nur sehr schwer eine „optimale", das heißt möglichst kleine Teilauflösung finden. Man darf aber sicherlich hoffen, daß eine Vielzahl der in der Praxis auftretenden virtuellen Feedbacks durch sogenannte Durchläufer entstehen. Das sind Signale, die nur durch eine Schaltung durchgereicht werden, um das Design der Gesammtschaltung einfacher zu gestalten. Diese lassen sich aber vor der Fehlersimulation „herausziehen" und damit den Feedback auflösen!

3.2.4 Verluste durch Parameterübergabe

Ich möchte in diesem Abschnitt die Kosten der hierarchischen Simulation untersuchen. Dazu muß ich zunächst einmal klären, was ich unter den Kosten genau verstehe.

Zeitraubend bei Computern sind die Zugriffe des Zentralrechners auf seinen Speicher, da dieser in der Regel nicht auf den gleichen Chip integriert ist. Sprungbefehle stören die Befehlscaches eines Rechners und verlangsamen daher den Ablauf eines Programmes. Als Kosten einer Simulationsroutine will ich deshalb die Summe ihrer Speicherzugriffe und ihrer Sprungbefehle ansehen. Operationen, wie die Undverknüpfung, oder Negationen spielen keine Rolle, ihre Rechenzeit wird mit den dafür benötigten Speicherzugriffen abgegolten.

Für Instanzen von Basiszellen ergeben sich daraus als Simulationskosten die Größe ihres Randes, das heißt Eingangsgrad plus Ausgangsgrad. Denn für jeden Eingang benötigt man einen lesenden Speicherzugriff und für jeden Ausgang einen schreibenden. Für Module muß man die Simulationskosten der Schaltung rechnen, von deren Typ das Modul ist, plus die Größe seines Randes plus zwei. Die Größe seines Randes entspricht der Anzahl der Parameter, die der Prozedur zur Auswertung des Moduls übergeben werden und damit Speicherzugriffe darstellen. Die additive Zwei steht für den Hin- und Rücksprung zu der Prozedur. Zur Verdeutlichung hier die Kosten des Volladdierers aus Kapitel 1:

Ein Xor-Gatter besteht aus vier Nand-Gattern mit jeweils zwei Eingängen und einem Ausgang.

$$\Rightarrow cost(\text{Xor}) = 4(2+1) = 12$$

Ein Halbaddierer besteht aus einem Xor-Gatter und einem And-Gatter.

$$\Rightarrow cost(\text{Ha}) = \underbrace{cost(\text{Xor}) + (2+1) + 2}_{\text{Xor}} + \underbrace{(2+1)}_{\text{And}} = 20$$

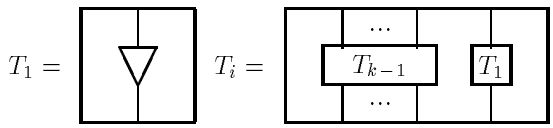

Bild 13: Treiberschaltung – hierarchisch

Ein Volladdierer besteht aus zwei Halbaddierern und einem Or-Gatter.

$$\Rightarrow cost(\text{Fa}) = \underbrace{cost(\text{HA}) + (2+2) + 2}_{\text{erster Ha}} + \underbrace{cost(\text{HA}) + (2+2) + 2}_{\text{zweiter Ha}} + \underbrace{(2+1)}_{\text{Or}} = 55$$

Wie hoch sind die Simulationskosten einer hierarchischen Schaltung im Vergleich zu einer Simulationsroutine für den aufgelösten, flachen Schaltkreis? Beim Volladdierer, der aus 11 Basiszellen mit jeweils zwei Eingängen und einem Ausgang besteht, wären das 33. Daß die hierarchische Version teurer ist, war zu erwarten. Wie „schlimm“ kann der Unterschied aber denn nun werden? Um dies zu zeigen, möchte ich folgenden Schaltkreis vorstellen: Es handelt sich um die rekursive Beschreibung einer Treiberkette. Seien dazu T_1 und T_2 bis T_k gemäß Abbildung 13 definiert. Die flache Version $flach_k$ des Schaltkreises findet sich in Abbildung 14.

$cost(flach_k) = 2k$

$cost(T_1) = 2$
$cost(T_i) = cost(T_{i-1}) + 2(i-1) + 2 + cost(T_1) + 2 + 2 = 2(i+3) + cost(T_{i-1})$

$$\Rightarrow cost(T_k) = 2(\sum_{i=2}^{k}(i+3)) + 2 = 2(3(k-1) + \frac{k(k+1)}{2} - 1) + 2 = k(k+1) + 6(k-1)$$

Hierarchische Simulation für die Treiberkette hat also quadratische Laufzeit in der Größe der Schaltung, wenn man diese als die Anzahl der Signale ansieht, gegenüber linearer Laufzeit bei flacher Simulation!

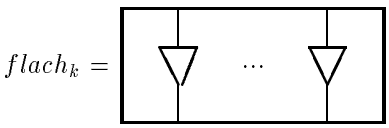

Bild 14: Treiberschaltung – flach

Ist das auch eine obere Schranke, oder kann es gar noch schlimmer werden? Falls ein Schaltkreis Boxen enthalten darf, die nur aus einer weiteren Box bestehen, so lautet die Antwort: ja. Dann kann man den Simulator nämlich dazu zwingen, ein Signal beliebig oft zu kopieren, bis es endlich benutzt wird. Schließt man solche „Verpackungsboxen“ allerdings aus, so ist die quadratische Laufzeit der schlimmste aller Fälle. Dies sieht man wie folgt ein. Eine Box enthält mindestens eine Basiszelle oder aber mindestens zwei weitere Boxen. Damit ist die Anzahl der Boxen auf die zweifache Anzahl der Basiszellen begrenzt (Vergleiche die Anzahl der Knoten in einem binären Baum). Da die Anzahl der Basiszellen und die Größe des Randes einer Box durch die Anzahl der Signale beschränkt sind, wächst die Laufzeit „nur“ mit dem Quadrat der Signale.

3.2.5 Schnelle, hierarchische Fehlersimulation

Es hat sich gezeigt, daß die intuitive Lösung zu sehr großen Laufzeiten führen kann. Wie es auch schneller gehen kann, will ich in diesem Abschnitt zeigen. Dazu zunächst einige Anforderungen an die Simulationsroutine. Denn bisher wurde nur von reiner Simulation gesprochen, Fehlersimulation verlangt aber noch einige weitere Aspekte.

1. Die Simulationszeit soll nicht wesentlich schlechter sein, als bei flacher Simulation. Insbesondere sollte die Komplexität nicht quadratisch wie bei der intuitiven Lösung, sondern linear wie bei der flachen Simulation sein.

2. Zum Einpflanzen der Fehler müssen die Belegungen der Ein- und Ausgänge jeder Basiszelleninstanz erreichbar sein.

3. Zur Berechnung der Ausgabe der fehlerhaften Schaltung muß man nach dem Einpflanzen des Fehlers die restliche Schaltung simulieren. Der Kellerzustand an der Stelle einer Instanz beim Simulieren muß daher herstellbar sein, um ab der Instanz weiterrechnen zu können.

Die hohen Kosten der intuitiven Lösung entstehen durch den Transport der Signalwerte vom Ort ihrer Erzeugung zu einem lesenden Zugriff über viele Hierarchiegrenzen hinweg. Das Überqueren einer jeden solchen Grenze bedeutet das Kopieren des Wertes. Sinnvoll kann diese Vorgehensweise sein, wenn man innere Signale einer Schaltung nach ihrer Simulation löscht, und nur die Werte der Ausgänge weiterreicht und deshalb den Speicherplatz reduzieren kann. Nun fordert aber gerade Punkt 2, daß auch innere Signalwerte erreichbar sein müssen, um im Innern einer Teilschaltung Fehler einzupflanzen. Dann ist die ganze Kopiererei aber überflüssig, denn Speicherplatz läßt sich nur sparen, wenn man die Werte der inneren Signale lokal hält. Ein Verweis vom Ort des Zugriffs auf ein Signal zu seinem Entstehungsort ist jedoch wesentlich schneller als der Transport durch Kopieren! Spendieren wir also den Speicherplatz für ein Feld mit Platz für den Rand jeder Basiszelleninstanz. An die Ausgänge können wir dann die Werte während einer Simulation schreiben, und an die Eingänge Zeiger auf den entsprechenden Ausgang der den Eingang treibenden Basiszelleninstanz. Damit bleibt nur noch, den dritten Punkt zu klären. Der Zustand eines Prozedurenkellers läßt sich in einer Hochsprache nicht restaurieren. Und Fehlersimulation auf Assemblerebene widerspricht der Idee der Portabilität. Deshalb sollten wir uns um die Verwaltung des Kellers selber kümmern und keine vorgefertigten Prozeduren benutzen. Damit das ganze auch noch effizient geschehen kann, schlage ich vor, die Kellerinformation mit in das Feld der Basiszelleninstanzen einzubinden. Sie steht dort immer zur Verfügung, und es muß deshalb nicht jedesmal ein Keller restauriert werden, sondern nur ein Zeiger auf die richtige Stelle in dem Feld gesetzt werden. Dies ist möglich, da die Aufrufstruktur des Programmes schon vor seiner Ausführung bekannt ist!

rekursive Beschreibung des Feldes

Das Feld muß Platz bieten für alle Instanzen einer hierarchischen Schaltung. Basiszelleninstanzen benötigen Platz für die Belegung der Ausgänge und für Zeiger der Eingänge. Instanzen von Schaltkreisen benötigen Platz um Kellerinformationen abzuspeichern. Der Aufbau des Feldes läßt sich rekursiv am einfachsten beschreiben:

Für eine Basiszelleninstanz m:

$$\text{Feld}(m) = \underbrace{\boxed{\bullet}\cdots\boxed{\bullet}}_{\text{Ausgänge}}\underbrace{\boxed{\leftarrow}\cdots\boxed{\leftarrow}}_{\text{Eingänge}}$$

Für eine Instanz m einer Schaltung C_i aus den Modulen m_1 bis m_n:

$$\text{Feld}(m) = \underset{\uparrow Kp}{\boxed{\text{RSP}}\boxed{\text{RSA}}} \circ \text{Feld}(m_1) \circ \text{Feld}(m_2) \circ \cdots \circ \text{Feld}(m_n)$$

Für die Instanz m der Schaltung C_k, die die hierarchische Schaltung $H = (C_1, C_2, \cdots, C_k)$ mit i primären Inputs und o primären Outputs repräsentiert:

$$\text{Feld}(H) = \underbrace{\boxed{\bullet}\cdots\boxed{\bullet}}_{i \text{ Eingänge}}\underbrace{\boxed{\leftarrow}\cdots\boxed{\leftarrow}}_{o \text{ Ausgänge}} \circ \underset{\uparrow start}{\text{Feld}(m)}$$

Während der Simulation der Schaltung C_i zeigt ein Kellerpegel Kp auf die Adresse der zu simulierenden Instanz im Feld. Nach Beenden der Simulation für die Schaltung C_i muß der Kp auf die aufrufende Instanz gesetzt werden. Ebenso muß in die aufrufende Routine zurückgesprungen werden. Dazu zeigt der Rücksprungpegel RSP auf die Feldadresse der aufrufenden Instanz und die Rücksprungadresse RSA auf die Programmstelle der aufrufenden Routine nach Beenden von C_i.

Codeschema

Wie sieht nun die Simulationsroutine aus? Auch hier ist es angebracht, das Codeschema rekursiv zu definieren.

Für Basiszellinstanzen ändert sich nicht viel gegenüber der Codegenerierung flacher Simulation. Die Adressierung erfolgt jedoch nicht mehr global, sondern lokal zum Kellerpegel. Außerdem werden die Eingänge nun indirekt adressiert. Zum Vergleich dazu noch einmal das Codeschema aus Abschnitt 3.1.1:

flaches Codeschema für Nandgatter:

```
code(nand2x1): Feld[o] = not( Feld[i1] and Feld[i2] );
```

hierarchisches Codeschema für Nandgatter:

```
code(nand2x1): Kp[o] = not( Feld[Kp[i1]] and Feld[Kp[i2]] );
```

Beim Aufruf der Simulationsroutine für eine Instanz m einer Schaltung C_i muß zunächst der Kellerpegel auf die neue Feldadresse gesetzt werden. Gemäß dem Aufbau des Feldes kann dies durch Addition einer Konstanten auf den Kp erfolgen. Nun kann die gewünschte Routine angesprungen werden. Um den Rücksprung muß man sich an dieser Stelle noch nicht kümmern, denn der Kellerpegel

zeigt nun auf die Instanz m, in der durch den Wert von RSA für einen korrekten Rücksprung gesorgt wird.

$$\text{code}(m) = \begin{array}{l} \text{Kp} = \text{Kp} + address_m; \\ \text{goto } label_{C_i}; \end{array}$$

Die Simulationsroutine einer Schaltung C_i besteht aus dem Code für die einzelnen Module und Befehlen für einen korrekten Rücksprung. Dazu muß der Kellerpegel restauriert und zur Rücksprungadresse RSA gesprungen werden.

$$\begin{array}{rll} & label_{C_i}: & \text{code}(m_1); \\ & & \text{code}(m_2); \\ & & \cdots \\ \text{code}(C_i) = & & \text{code}(m_n); \\ & & help = \text{Kp}[1]; \\ & & \text{Kp} = \text{Kp}[0]; \\ & & \text{goto } help; \end{array}$$

Damit bleibt nur noch das Codeschema für die gesammte, hierachische Schaltung $H = (C_1, C_2, \cdots, C_k)$ anzugeben:

$$\begin{array}{rl} & \text{Kp} = start; \\ & \text{code}(C_k); \\ \text{code}(H) = & \text{code}(C_{k-1}); \\ & \cdots \\ & \text{code}(C_1); \end{array}$$

Zur Verdeutlichung mag das Feld, dargestellt in Tabelle 3.2 und der dazugehörende Code für eine Schaltung bestehend aus einem Volladdierer des Kapitels 1 dienen.

0					in_1			•
1					in_2			•
2					in_3			•
3					out_1	→		42
4					out_2	→		45
5	Fa				RSP	→		0
6					RSA	→		l_{23}
7		Ha_1			RSP	→		5
8					RSA	→		l_4
9			And		out			•
10					in_1	→		1
11					in_2	→		2
12			Xor		RSP	→		7
13					RSA	→		l_{13}
14				$Nand_1$	out			•
15					in_1	→		1
16					in_2	→		2
17				$Nand_2$	out			•
18					in_1	→		1
19					in_2	→		14
20				$Nand_3$	out			•
21					in_1	→		14
22					in_2	→		2
23				$Nand_4$	out			•
24					in_1	→		17
25					in_2	→		20
26		Ha_2			RSP	→		5
27					RSA	→		l_6
28			And		out			•
29					in_1	→		0
30					in_2	→		23
31			Xor		RSP	→		26
32					RSA	→		l_{13}
33				$Nand_1$	out			•
34					in_1	→		0
35					in_2	→		23
36				$Nand_2$	out			•
37					in_1	→		0
38					in_2	→		33
39				$Nand_3$	out			•
40					in_1	→		33
41					in_2	→		23
42				$Nand_4$	out			•
43					in_1	→		36
44					in_2	→		39
45		Or			out			•
46					in_1	→		9
47					in_2	→		28

Tabelle 3.2: Feld für Schaltung aus Volladdierer

```
l1 :          Kp = 5;
l2 :  Fa :    Kp = Kp + 2;
l3 :          goto Ha;
l4 :          Kp = Kp + 21;
l5 :          goto Ha;
l6 :          Kp[40] = Feld[Kp[41]] or Feld[Kp[42]];
l7 :          help = Kp[1];
l8 :          Kp = Kp[0];
l9 :          goto help
l10 :  Ha :   Kp[2] = Feld[Kp[3]] and Feld[Kp[4]];
l11 :         Kp = Kp + 5;
l12 :         goto Xor;
l13 :         help = Kp[1];
l14 :         Kp = Kp[0];
l15 :         goto help
l16 :  Xor :  Kp[2] = not( Feld[Kp[3]] and Feld[Kp[4]] );
l17 :         Kp[5] = not( Feld[Kp[6]] and Feld[Kp[7]] );
l18 :         Kp[8] = not( Feld[Kp[9]] and Feld[Kp[10]] );
l19 :         Kp[11] = not( Feld[Kp[12]] and Feld[Kp[13]] );
l20 :         help = Kp[1];
l21 :         Kp = Kp[0];
l22 :         goto help
l23 :
```

optimierte Version

Betrachtet man sich das Beispiel des Volladdierers im vorigen Paragraphen, so sieht man, daß sich noch einige Kleinigkeiten verbessern lassen. So setzt das Programm nach Beenden der Simulation des Xor-Gatters des ersten Halbaddierers den Kellerpegel auf den Halbaddierer und springt zu Label l_{13}. Dort wird aber nun das Ende der Simulationsroutine für den Halbaddierer vorbereitet, das heißt, der Kellerpegel wird auf den übergeordneten Volladdierer gesetzt und zu Label l_4 gesprungen. Doch damit nicht genug, denn jetzt muß zuerst der Kellerpegel auf den zweiten Halbaddierer gesetzt und zu ihm hin gesprungen werden. Diese Odysse sollte man natürlich durch einen einzigen Sprung ersetzen. Möglich wird dies durch das Setzen der Rücksprungpegel und Rücksprungadressen im Feld. Man sollte sie derart setzen, daß zwischen der Auswertung zweier Basiszellen maximal ein Sprung liegt!

Die indirekte Adressierung der Eingänge von Basiszellen wurde eingeführt, um das Kopieren von Signalwerten an Hierarchiegrenzen zu vermeiden. Da die indirekte Adressierung doppelt so teuer im Sinne des Kostenmaßes aus Abschnitt 3.2.4 ist, sollte sie – wo dies möglich ist – vermieden werden. Eine direkte

Adressierung ist immer dann möglich, wenn der Treiber des gesuchten Signales „innerhalb“ der Schaltung, für die die Routine erzeugt wird, liegt, das heißt, wenn das Signal nicht über einen primären Eingang importiert wird. Im Volladdierer lassen sich beide Eingänge des Oder-Gatters direkt adressieren, während dies beim Und-Gatter eines Halbaddierers nicht möglich ist.

Hier nun noch einmal Code und Feld (Tabelle 3.3) einer Schaltung aus einem Volladdierer, nun aber optimiert.

```
l_1 :          Kp = 7;
l_2 :          goto Ha;
l_3 :          Kp[40] = Kp[4] or Kp[23];
l_4 :          help = Kp[1];
l_5 :          Kp = Kp[0];
l_6 :          goto help
l_7 :  Ha :    Kp[2] = Feld[Kp[3]] and Feld[Kp[4]];
l_8 :          Kp = Kp + 5;
l_9 :  Xor :   Kp[2] = not( Feld[Kp[3]] and Feld[Kp[4]] );
l_10 :         Kp[5] = not( Feld[Kp[6]] and Kp[2] );
l_11 :         Kp[8] = not( Kp[2] and Feld[Kp[10]] );
l_12 :         Kp[11] = not( Kp[5] and Kp[8] );
l_13 :         help = Kp[1];
l_14 :         Kp = Kp[0];
l_15 :         goto help
l_16 :
```

Simulationskosten

In Abschnitt 3.2.4 hat sich gezeigt, daß die Laufzeit für hierarchische Simulation bei der intuitiven Lösung quadratische Kosten in der Anzahl der Signale als der Größe der Schaltung haben kann. Wie hoch sind die Laufzeitkosten der vorgestellten, schnellen Lösung?

- Für Basiszellinstanzen muß man die Summe der Ausgänge und maximal die zweifache Summe der Eingänge rechnen. Diese Kosten entstehen durch Speicherzugriffe.
- Boxzellinstanzen verursachen, abgesehen von den nötigen Hin- und Rücksprüngen, die ich gesondert betrachten will, keine weitere Kosten zu der Summe der Kosten der in ihnen enthaltenen Module.
- Die Kosten eines Sprunges sollen nun auch die Verwaltung des Kellerpegels berücksichtigen, sie seien als Konstante c gegeben. Die Anzahl aller Sprünge ist durch die Anzahl der Basiszellinstanzen begrenzt.

$$\Rightarrow cost(H) \leq \sum_{\text{Basiszellinstanz } I} (2\, indegree(I) + outdegree(I) + c) = \mathrm{O}(size(H))$$

0					in_1		•
1					in_2		•
2					in_3		•
3					out_1	→	42
4					out_2	→	45
5	Fa				RSP	→	0
6					RSA	→	l_{16}
7		Ha_1			RSP		
8					RSA		
9			And		out		•
10					in_1	→	1
11					in_2	→	2
12			Xor		RSP	→	26
13					RSA	→	l_7
14				$Nand_1$	out		•
15					in_1	→	1
16					in_2	→	2
17				$Nand_2$	out		•
18					in_1	→	1
19					in_2	→	14
20				$Nand_3$	out		•
21					in_1	→	14
22					in_2	→	2
23				$Nand_4$	out		•
24					in_1	→	17
25					in_2	→	20
26		Ha_2			RSP		
27					RSA		
28			And		out		•
29					in_1	→	0
30					in_2	→	23
31			Xor		RSP	→	5
32					RSA	→	l_3
33				$Nand_1$	out		•
34					in_1	→	0
35					in_2	→	23
36				$Nand_2$	out		•
37					in_1	→	0
38					in_2	→	33
39				$Nand_3$	out		•
40					in_1	→	33
41					in_2	→	23
42				$Nand_4$	out		•
43					in_1	→	36
44					in_2	→	39
45		Or			out		•
46					in_1	→	9
47					in_2	→	28

Tabelle 3.3: Feld für Schaltung aus Volladdierer – zu optimiertem Code

praktische Anwendung

Leider waren die Entwerfer von Hochsprachen nicht gewillt, dem Anwender die Verwaltung von Sprungadressen zu übertragen. Es mag ja sein, daß man damit einigen Unfug machen kann, aber *muß man denn gleich die Messer verbieten, nur weil man sich damit schneiden kann* (Prof. Hotz)? Es ist daher mit einigen Effizienzverlusten zu rechnen.

Ich schlage vor, die erzeugten Routinen innerhalb einer Fallunterscheidung zu übersetzen und jede Zeile durch einen Fall zu trennen. Erste Meßergebnisse einer Implementierung in der Sprache C ergeben Laufzeitkosten von nur etwa dem Doppelten der Simulation flacher Schaltkreise.

Zum Abschluß die erzeugte Routine für einen Volladdierer. Sie unterscheidet sich ein wenig von dem früher vorgestellten Beispiel, weil hier in den Halbaddierern das Xor-Gatter topologisch kleiner als das And-Gatter ist.

```
/*
   Simulationsroutine fuer hierarchische Fehlersimulation
   Version vom 8.08.91
*/

long SIG[48];

void simulation( Kp, ab )
     long *Kp;
     long ab;
{
 while( ab )
 {
  switch( ab )
  {
   XOR:
   case 1:
        Kp[2] = ~( SIG[Kp[3]] & SIG[Kp[4]] ) ;
   case 2:
        Kp[5] = ~( Kp[2] & SIG[Kp[7]] ) ;
   case 3:
        Kp[8] = ~( SIG[Kp[9]] & Kp[2] ) ;
   case 4:
        Kp[11] = ~( Kp[8] & Kp[5] ) ;
   case 5:
        break;
   case 6:
        Kp[16] = SIG[Kp[17]] & SIG[Kp[18]] ;
   case 7:
```

```
        break;
   case 8:
        Kp[45] = ( Kp[23] | Kp[42] ) ;
   case 9:
        break;
  }
  ab = Kp[1];
  Kp = &SIG[*Kp];
 }
}

void correct()
{ simulation( &SIG[9], 1); }
```

3.2.6 Vorteile der Hierarchie

Bisher ist die Hierarchie eher schlecht weg gekommen. Fehlersimulation auf hierarchischen Schaltungen wurde mit flacher Simulation verglichen und immer war von Kosten die Rede. In diesem Abschnitt will ich mich deshalb den Vorteilen widmen, die man durch die Hierarchie erzielen kann. Sie rühren von drei Eigenschaften einer hierarchischen Schaltungsbeschreibung her.

1. Teilschaltungen einer hierarchischen Schaltung können mehrfach als Instanzen vorkommen. Hierarchische Beschreibungen liefern also Informationen über Musterwiederholungen, die in einer flachen Beschreibung eventuell nur sehr schwer sichtbar sind.

2. Zwei Module einer hierarchischen Schaltung, die sich gegenseitig nicht beeinflußen, kann man als parallel erkennen. In einer flachen Schaltungsbeschreibung kann diese Parallelität verwischen.

3. Hierarchische Beschreibungen können „gute Schnitte“ aufzeigen. Das heißt, es können Module auftreten, die nur kleine Ränder, aber ein großes Innenleben haben. Solche Schnitte in flachen Schaltungsbeschreibungen zu finden, ist sehr schwierig.

Ich möchte diese Eigenschaften und mögliche Auswirkungen auf die Fehlersimulation im Folgenden näher erläutern.

Musterwiederholungen

Den Effekt der Musterwiederholung habe ich schon bei der Motivation für hierarchische Fehlersimulation erwähnt. So sollte damit Speicherplatz und Compilezeit durch eine kürzere Simulationsroutine eingespart werden. Der Grund für die Verkürzung der Simulationsroutine liegt aber gerade in der Wiederkehr von Mustern und daß man nur einmal Code für alle Vorkommen erzeugen muß.

Interessant wird die Musterwiederholung aber auch, um Aussagen über die Rentabilität von Vorberechnungen zu machen. Dazu ein kleines Beispiel. Nehmen wir an, die zu untersuchende Schaltung beinhaltet viele Instanzen ein und derselben Teilschaltung. Mit einem geometrisch regelmäßigen Design kommt es durchaus vor, daß in all den Instanzen nur noch nach einem Fehler – und zwar immer dem gleichen – gesucht wird. Durch das häufige Auftreten der Instanz bieten sich folgende Effizienzverbesserungen an, die im allgemeinen viel zu teuer sind:

- Die Simulationsroutine für die Teilschaltung ist für alle darin vorkommenden Basiszellen geschrieben. Da aber nur noch ein Fehler an einer einzigen Basiszelle existiert, kann man dafür eine eigene, viel schnellere Routine erzeugen, die nur noch Zellen auswertet, die auch von der Fehlerzelle aus erreicht werden können.

- Der gesuchte Fehler „vereinfacht“ vielleicht die Teilschaltung. Mit Hilfe von Algorithmen aus der Logikminimierung [BRSVW87] kann es dann möglich sein, eine einfachere Simulationsroutine, mit weniger Gatterberechnungen, zu erzeugen.

- Falls der Rand der Instanzen nicht zu groß ist, kann man sich die Simulation der fehlerhaften Teilschaltung ganz sparen, wenn man sich einmal das Ein- / Ausgabeverhalten vorberechnet.

Anzeigen von Parallelität

Ich möchte hierfür zwei Beispiele geben, die den Begriff der Parallelität ein wenig unterschiedlich interpretieren.

1. Hat man einen Fehler eingepflanzt, so muß man die restliche Schaltung simulieren. Dazu muß man aber nicht unbedingt zu der Zelle mit der nächsthöheren topologischen Sortierungsnummer springen, sondern es genügt die Zelle mit der kleinsten topologischen Sortierungsnummer, die von der fehlerhaften Zelle abhängt. Dadurch spart man Simulationskosten. In hierarchischen Schaltungen kann man diesen Effekt bei dem Rücksprung aus jeder Instanz, die den Fehler beinhaltet, ausnutzen. In flachen Schaltkreisen nur einmal, nach dem Fehlereinpflanzen.

   Betrachten wir als Beispiel den Baum aus Und-Gattern in den Abbildungen 15 und 16. Unabhängig von den vielen Möglichkeiten einer topologischen Sortierung benötigt man in der flachen Schaltung nahezu $\frac{1}{2}n^2$ Gatterauswertungen, um von jeder Zelle aus den Rest der Schaltung zu simulieren. Die hierarchische Routine braucht aber für jedes Gatter nur den Pfad zum Ausgang zu durchlaufen, wenn sie nach jedem Verlassen einer Box die dazu Parallele überspringt. Damit fallen aber höchstens $n \log n$ Gatterauswertungen an.

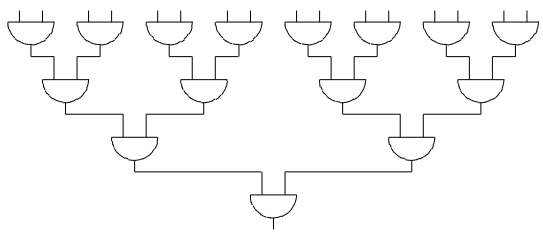

Bild 15: Und-Gatter – Baum, flach

Durch dieses Beispiel sollte allerdings nicht die Hoffnung geweckt werden, die Komplexität der Fehlersimulation von n^2 senken zu können (vergleiche [HK87]), denn auf Bäumen ist Fehlersimulation mit anderen Methoden sogar in Linearzeit möglich [AS87].

2. Verwischen von Parallelität kann man aber auch als das Verwischen von Grenzen ansehen. Wenn in der hierarchischen Schaltung ein Addierer als Teilschaltung vorkommt, so ist er in der flachen Schaltung vielleicht mit seiner Umgebung vermengt. Dann ist es aber nicht mehr möglich, eine Semantiksubstitution vorzunehmen. Dies ist aber sehr wohl in der hierarchischen Version möglich. Eine Realisierung, die einen schnellen Addierer liefert, zum Beispiel einen Carry-look-ahead, ist nicht unbedingt die schnellste Möglichkeit einen Addierer zu simulieren. Man denke nur an den guten alten ripple-carry-Addierer. Man spart ungemein an Rechenzeit, wenn man für alle Fehler außerhalb des Addierers, die sich durch ihn hindurch propagieren, die schnellere Simulationsroutine einsetzt.

Gute Schnitte

Das Schönste, was Hierarchie liefern kann, sind gute Schnitte. Obwohl man es natürlich am liebsten sieht, wenn sowohl die Anzahl der Eingänge als auch die Anzahl der Ausgänge von Teilschaltungen sehr klein ist, möchte ich dies im Folgenden unterscheiden. Denn manche Effekte benötigen nicht beides.

wenig Eingänge Will man sich das Ein- /Ausgabeverhalten einer Teilschaltung C für alle ihre Fehler abspeichern, und sich damit alle Simulationen dieser Schaltung in Zukunft sparen, so benötigt man pro Fehler

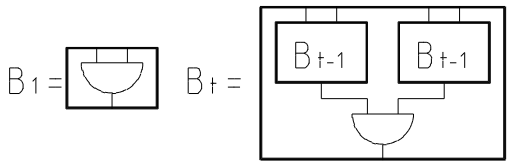

Bild 16: Und-Gatter – Baum, hierarchisch

$2^{indegree(C)}outdegree(C)$ Speicherplatz. Deshalb ist dies nur möglich, wenn die Anzahl der Eingänge von C sehr klein ist.

Ist das Innere einer Teilschaltung C sehr groß im Verhältnis zu der Anzahl ihrer Eingänge, dann ist es ratsam, vor Beginn ihrer Simulation für einen Fehler der sich möglicherweise durch sie durchpropagiert, zu überprüfen, ob sich der Wert der Eingänge gegenüber der Richtigsimulation geändert hat. Wenn nicht, kann man sich die Simulation und damit viel Rechenzeit schenken.

wenig Ausgänge Wenn eine Teilschaltung nur wenige Ausgänge besitzt, so ist es sehr wahrscheinlich, daß sich viele Fehler aus ihrem Inneren an ihren Ausgängen gleich verhalten. Es kann erfolgversprechend sein, alle Fehler nur bis zum Ende der Teilschaltung zu simulieren, alle vorkommenden Ausgabemuster aufzusammeln, wobei mehrfache Vorkommen zusammenfallen und dann die restliche Schaltung für diese Muster zu simulieren. So kann man teure Simulationen mit den selben Werten verhindern.

Fehlersimulatoren nutzen die in einem Computer vorgegebene Parallelität aus. Sie pflanzen an verschiedene Bitpositionen verschiedene Fehler ein und simulieren sie simultan. Aber oft „verlaufen“ sich Fehler sehr schnell und die von ihnen genutzten Positionen werden frei. Nach der Simulation einer Teilschaltung läßt sich dies an den Ausgängen überprüfen. Dabei entstehen aber Kosten in der Anzahl der Ausgänge. Die freigewordenen Positionen lassen sich mit anderen Fehlern besetzen und sich somit Simulationen sparen.

Ausblick

Das Kapitel über Nicht-standard-Fehlermodelle stellt einen ersten Einstieg in die Thematik dar. Bis zu den ersten lauffähigen Versionen von Testalgorithmen, die sie nutzen können, ist es noch ein weiter Weg.

Dagegen sind die Ergebnisse des Kapitels über hierarchische Fehlersimulation durchaus einsetzbar. Erste Implementierungen sind erfolgreich abgeschlossen und ermutigen dazu, die Ideen des Abschnittes 3.2.6 auszuprobieren.

Ich hoffe, beide Richtungen weiterverfolgen zu können und mithilfe einiger Fortgeschrittenenpraktika irgendwann einen „cleveren, hierarchischen Fehlersimulator für Nicht-standard-Fehlermodelle“ vorzustellen!

Literaturverzeichnis

[AS87] Kurt J. Antreich and Michael H. Schulz. Accelerated fault simulation and fault grading in combinational circuits. *IEEE Transactions on Computer-Aided Design*, CAD-6(5):704–712, September 1987.

[BA83] P. Banerjee and J.A. Abraham. Generating tests for physical failures in MOS logic circuits. In *Proceedings of the 1983 International Test Conference*, pages 554–559, 1983.

[BART82] C.C. Beh, K.H. Aray, C.E. Radke, and K.E. Torku. Do stuck-at fault models reflect manufacturing defects? In *Proceedings 1982 International Test Conference*, pages 35–42, 1982.

[BF85] F. Brglez and H. Fujiwara. A neutral netlist of 10 combinational benchmark circuits and a target translator in fortran. In *Proceedings IEEE International Symposium on Circuits and Systems*, 1985.

[BHK+87] B. Becker, G. Hotz, R. Kolla, P. Molitor, and H.G. Osthof. Hierarchical design based on a calculus of nets. In *Proceedings of the 24th ACM/IEEE Design Automation Conference (DAC87)*, pages 649–653, June 1987.

[BHKS91] B. Becker, R. Hahn, R. Krieger, and U. Sparmann. Structure based methods for parallel pattern fault simulation in combinational circuits. In *Proceedings of the 2nd European Design Automation Conference (EDAC91)*, 1991.

[BRSVW87] R. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang. MIS: Multiple-level interactive logic minimization. *IEEE Transactions on CAD of Integrated Circuits and Systems*, CAD-6(6):1062–1081, November 1987.

[DG87] Wilfried Daehn and Manfred Geilert. Fast fault simulation for combinational circuits by compiler driven single fault propagation.

In *Proceedings 1987 International Test Conference*, pages 286–292, 1987.

[FS88] F.J. Ferguson and J.P. Shen. Extraction and simulation of realistic CMOS faults using inductive fault analysis. In *Proceedings 1988 International Test Conference*, pages 475–484, 1988.

[GCV80] J. Galiay, Y. Crouzet, and M. Vergniault. Physical versus logical fault models MOS LSI circuits: Impact on their testability. *IEEE Transactions on Computers*, C-29(6):527–531, June 1980.

[Gil62] A. Gill. *Introduction to the theory of finite-state machines.* McGraw - Hill, 1962.

[Har90] J. Hartmann. Analyse zweier Techniken zur Erhöhung der Zufallstestbarkeit. In *Tagungsband des ITG/GI-Workshops 'Testmethoden und Zuverlässigkeit von Schaltungen und Systemen'*, 1990. 6 Seiten.

[Har91] J. Hartmann. The random testability of the n-input AND gate. In *Proceedings of the 8th Annual Symposium on Theoretical Aspects of Computer Science (STACS91)*, February 1991.

[HK87] D. Harel and B. Krishnamurthy. Is there hope for linear time fault simulation. In *Fault Tolerant Computing Symposium*, pages 28 – 33, 1987.

[Hot72] G. Hotz. *Rechenanlagen*, pages 79+. Teubner, 1972.

[HSS91] J. Hartmann, B. Schieffer, and U. Sparmann. COFS – a cell oriented fault simulator. In *Kurzfassungen der Beiträge zum 3. ITG/GI-Workshop: Testmethoden und Zuverlässigkeit von Schaltungen und Systemen*, 1991.

[KS85] S. Köppe and C.W. Starke. Logiksimulation komplexer Schaltungen für sehr große Testlängen. In *NTG-Fachberichte*, volume 87, pages 73 – 80, Baden-Baden, 1985. NTG-Fachtagung Großintegration.

[LA83] P. Lamoureux and V.K. Agarwal. Non-stuck-at fault detection in NMOS circuits by region analysis. In *Proceedings 1983 International Test Conference*, pages 129–137, 1983.

[LF80] R. E. Ladner and M. J. Fischer. Parallel prefix computation. *Journal of the Association for Computing Machinery*, 27(4):831–838, October 1980.

[Mal87] W. Maly. Realistic fault modeling for VLSI testing. In *Proceedings 24th ACM/IEEE Design Automation Conference*, pages 173–180, 1987.

[Ser88] B. Serf. Probleme bei der Simulation kombinatorischer Schaltkreise unter Ausnutzung von Hierarchie. Master's thesis, Fachbereich Informatik, Universität des Saarlandes, Im Stadtwald, W–6600 Saarbrücken 11, FRG, 1988. 49 Seiten.

[Spa91] U. Sparmann. *Strukturbasierte Testmethoden für arithmetische Schaltkreise*. PhD thesis, Fachbereich Informatik, Universität des Saarlandes, Im Stadtwald, W–6600 Saarbrücken 11, FRG, 1991.

[SS90] E. Schanck and B. Schieffer. Dokumentation des fsim – Fehlersimulators. Projekt im Rahmen eines Fortgeschrittenenpraktikums, 1990.

[Wun87] Hans-Joachim Wunderlich. On computing optimized probabilities for random tests. In *Proceedings 24th ACM/IEEE Design Automation Conference (DAC87)*, pages 392–398, 1987.