UNIVERSITÄT POTSDAM
Institut für Informatik

Optimierung von Fehlererkennungsschaltungen auf der Grundlage von komplementären Ergänzungen für 1-aus-3 und Berger Codes

Dissertation

zur Erlangung des akademischen Grades
Doktor der Naturwissenschaften
(Dr. rer. nat.)
in der Wissenschaftsdisziplin Technische Informatik

eingereicht an der
Mathematisch-Naturwissenschaftlichen Fakultät
der Universität Potsdam

von
Alexei Morozov
Potsdam, im März 2005

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Die mikroelektronische Technik entwickelt sich sehr schnell. Seit dem ersten Tag der Erschaffung der Mikrosysteme sucht man nach Wegen zur Erhöhung der Produktivität. Jedes Jahr vervollkommnen sich die Technologien der Synthese der Mikroprozessorsysteme, nimmt die Anzahl der Transistoren auf dem Kristall und die Dichte ihrer Anordnung zu, werden Taktfrequenz und Effektivität der Ausführung von Befehlen erhöht.

Größe der Oberfläche und Anzahl der Transistoren des Kristalls werden zur hauptsächlichen strategischen Ressource bei der Massenproduktion von Mikroprozessoren im Wettkampf um die Produktivität. Bekannt ist, dass die Integrationsstufe des Mikroschemas von der Größe des Kristalls oder der Anzahl der auf ihm untergebrachten Transistoren abhängt. Der Hauptfaktor, der die Möglichkeit zur Erhöhung der Transistorenanzahl in VLSI (*Very Large Scale Integration*/Sehr hoch integrierte Schaltungen) Schaltungen bestimmt, ist der Entwurf und die Implementierung von Modulen mit minimalem topologischen Umfang. Aus technologischen Gründen ist die Fläche des Kristalls stark beschränkt. Die architektonischen Lösungen sollen die Antwort geben, auf welcher Weise man die maximale spezifische Produktivität in der Berechnung auf die Einheit der Fläche (z.B. auf den einzelnen Transistor) bekommen kann. Auf Abbildung 1.1 ist die von Gordon Moore (einem Mitbegründer von Intel Corporation) vorausgesagte Zunahme der Komplexität und Produktivität digitaler Schaltungen für die letzten zwei Jahrzehnte dargestellt. Nach dem „Gesetz von Moore" entwickeln sich die Technologien seit 1965 auf solche Weise, dass alle 18-24 Monate ein neuer Mikroprozessor auf den Markt kommt, dessen Produktivität doppelt so hoch ist wie bei seinem Vorgänger [1]. Tatsächlich verdoppelten die CPU's bisher alle 18-24 Monate ihre Leistungskapazität. Der Integrationsgrad elektronischer Schaltungen erhöht sich jährlich um ca. 25%. Im Jahre 1999 wurde prognosiertiziert [2], dass in den folgenden ein bis zwei Jahren 100 Millionen Transistoren pro Chip Realität sein würden (z.B. besitzt der HP PA-8700 Mikroprozessor aus dem Jahre 2001 186 Millionen Transistoren). Diese Voraussetzungen geben den Wissenschaftlern eine interessante Aufgabe und Motivation für die Forschung. Eine Aufgabe, die hauptsächlich aus dem Testen von Mikroprozessoren auf alle mögliche Fehler besteht (*soft-errors*, *transient faults*, usw.) [3]. Die Technologie des Testens ist eine der grundlegendsten und kostspieligsten Hindernisse, die den Fortschritt auf dem Gebiet der Entwicklung von VLSI[1] Schaltungen, Mikroprozessoren und digitalen Systemen zurückhalten.

In der Gegenwart erhalten mikroelektronische Geräte immer mehr verantwortungsvolle Aufgaben auf dem Gebiet der Medizin, der Atomenergie, der Flugzeugtechnik, der Konstruktion von Hochgeschwindigkeitszügen oder z.B. auf dem Gebiet der Weltraumtechnik. Es finden sich noch viele weitere Beispiele und Anwendungsbereiche, in denen fehlertolerante Systeme zum Einsatz kommen.

Fehlertoleranz bezeichnet die Fähigkeit eines Systems, auch mit einer begrenzten Anzahl fehlerhafter Komponenten seine spezifische Funktion zu erfüllen. Fehlertoleranz wird benötigt in Systemen, in denen ein Fehler oder ein Ausfall (Anomalien) tödliche Folgen haben (sicherheitskritische Anwendun-

[1] $> 10^5$ Gatter pro Chip, Gatter werden mit 3-5 Transistoren realisier.

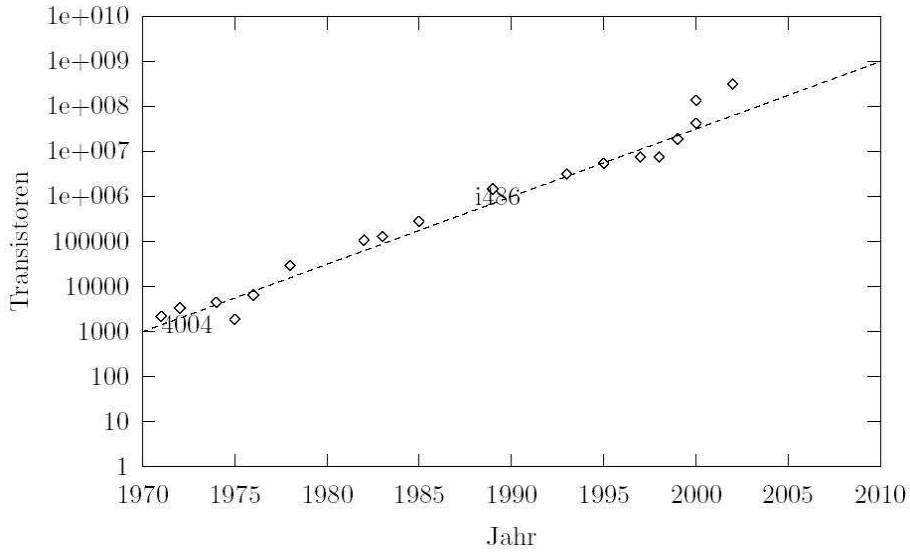

Abbildung 1.1: Potentielle Design-Komplexität digitaler Schaltungen nach dem Gesetz von G. Moore (Quelle: *ITRS Roadmap*)

gen) oder große finanzielle Verluste verursachen kann. Fehlertoleranz wird auch in Systemen benötigt, in denen keine Reparaturen möglich sind, wie z.B. in Satelliten oder Unterwasserstationen. Fehlertoleranz zur Laufzeit wird durch Fehlerentdeckung, Fehlerlokalisierung, Fehlermeldung und Fehlerbehandlung erreicht [4]. Aus diesem Grund ist die Erfindung neuer Methoden, die die korrekte Arbeitsweise solcher Systeme überwachen, besonders wichtig. Die korrekte Funktionalität großer komplexer VLSI Schaltungen lässt sich schwer testen. Deshalb ist die Konstruktion fehlertoleranter Schaltungen mit minimalem Hardware-Aufwand eine weitere Hauptrichtungslinie in der Entwicklung mikroelektronischer Systeme. Beim Entwurf von fehlertoleranten Schaltungen beschäftigt man sich also insbesondere damit, den zusätzlichen Hardware-Aufwand für Fehlererkennung möglichst gering zu halten und den Anteil der erkennbaren Fehler zu maximieren.

Eine der Varianten zum Erreichen von Fehlertoleranz ist die Konstruktion von selbstprüfenden Schaltungen. Ideen über die Möglichkeit der Synthese selbstprüfender Systemen sind in der wissenschaftliche Literatur erstmalig Ende der 60-er, Anfang der 70-er Jahre aufgetaucht. Als selbstprüfend wird ein System bezeichnet, in dem im laufenden Betrieb (*On-Line Mode*) die Überwachung der Korrektheit der inneren Elemente gewährleistet wird, sowie im Falle der Entstehung eines Fehlers ein Signal an seinen Ausgängen oder an einem speziellen Kontrollausgang gebildet wird. Dieses System besitzt einige innere Ressourcen, um seine Arbeit zu kontrollieren und die Richtigkeit der Funktion zu bewerten. Die Eigenschaft der Selbstprüfbarkeit löst das Problem der „Überwachung der Überwachung", so dass der nicht kontrollierbare Anteil des Systems gleich Null wird.

Es wird unterschieden zwischen selbstprüfenden Systemen und Systemen mit periodischem Selbsttest (*Built-In Self-Test*). Diese zwei grundsätzlichen Methoden zur Überprüfung der korrekten Funktionalität von Schaltungen werden heutzutage in einer großen Zahl von Digitalsystemen verwendet. Nach Gössel [5] werden die Methoden zur Fehlererkennung in Schaltungen auf folgende Weise in Gruppen eingeteilt:

- Modifikation von „Verdopplung und Vergleich";
- Anwendung von Codes;
- Algebraische Methoden;
- Überwachung von Ablaufsteuerungen.

Das Ziel der vorliegenden Dissertation besteht darin, eine neue, in [6] erstmals vorgeschlagene Methode zur Konstruktion selbstprüfender kombinatorischer Schaltungen, ihre Effektivität zu verbessern und so optimieren. Dabei soll der neuartige Weg zum Entwurf von Fehlererkennungsschaltungen auf der Basis von komplementären Schaltungen für spezielle Codes untersucht werden, in denen durch maximale Einbeziehung von Unbestimmtheitsstellen eine bessere Optimierung[2] der Fehlererkennungsschaltungen möglich werden soll.

1.2 Die logische Ergänzung als eine neue Methode zur Überwachung kombinatorischer Schaltungen

Alle bekannten Methoden der Online-Fehlererkennung sind auf zusätzliche Funktionen gegründet. Aus Kostengründen ist es zweckmäßig, einen solchen Entwurf für die notwendige zusätzliche Logik von dem Gesichtpunkt aus zu konstruieren, den Hardware-Aufwand möglichst gering zu halten.

In dieser Arbeit wird eine neue Online-Fehlererkennungsmethode für die Überwachung kombinatorischer Schaltungen untersucht werden, die im folgenden Methode der Logischen Ergänzung genannt wird.

In Abbildung 1.2 ist das Prinzip der Überwachung einer Schaltung f im laufenden Betrieb durch eine komplementäre Schaltung g dargestellt. Diese neue Methode der Fehlererkennung durch eine komplementäre Schaltung transformiert die Ausgänge $f_1(x), f_2(x), \cdots, f_n(x)$ der zu überwachenden Schaltung durch komponentenweise XOR-Verknüpfung mit den Ausgängen $g_1(x), g_2(x), \cdots, g_n(x)$ der komplementären (ergänzenden) Schaltung g in die Ausgänge $h_1(x), h_2(x), \cdots, h_n(x)$, die Elemente eines geeigneten Codes sind. Der selbstprüfende Code Checker *CC* überwacht die gesamte Schaltung auf ihre korrekte Funktionsweise.

Dadurch, dass die komponentenweise XOR-Verknüpfung $h_1(x) = f_1(x) \oplus g_1(x), h_2(x) = f_2(x) \oplus g_2(x), \cdots, h_n(x) = f_n(x) \oplus g_n(x)$ ein beliebiges Codewort des betrachteten Codes sein kann, ergeben sich viele Möglichkeiten für die Optimierung der komplementären Ergänzung g. Jeweils n Ausgänge der ursprünglichen, zu überwachenden Schaltung f werden durch n Ausgänge der komplementären Schaltung zu einem 1-aus-n oder Berger-Codewort ergänzt. Dabei erweist es sich, dass die unterschiedliche Einteilung der Ausgange $f_1(x), f_2(x), \cdots, f_n(x)$ der zu überwachenden Schaltung in Gruppen zu k Ausgängen, die durch komplementäre Ausgange zu Codewörtern ergänzt werden, zu ganz unterschiedlichem Aufwand für die Realisierung der komplementären Schaltung g führen.

Ziel der Dissertation ist es, Fehlererkennungsschaltungen auf der Grundlage von komplementären Schaltungen zu untersuchen. Insbesondere soll dabei der Hardware-Aufwand für unterschiedliche fehlererkennende Codes optimiert und verglichen werden. Gleichzeitig soll eine möglichst hohe Fehler-

[2] Das Optimierungsproblem im Schaltungsentwurf auf Gatterebene besteht darin, eine Schaltung A durch geeignete Transformationen in eine günstige Schaltung B umzuformen. Es dürfen aber solche Transformationen durchgeführt werden, die die logische Funktion der Schaltung A nicht ändern.

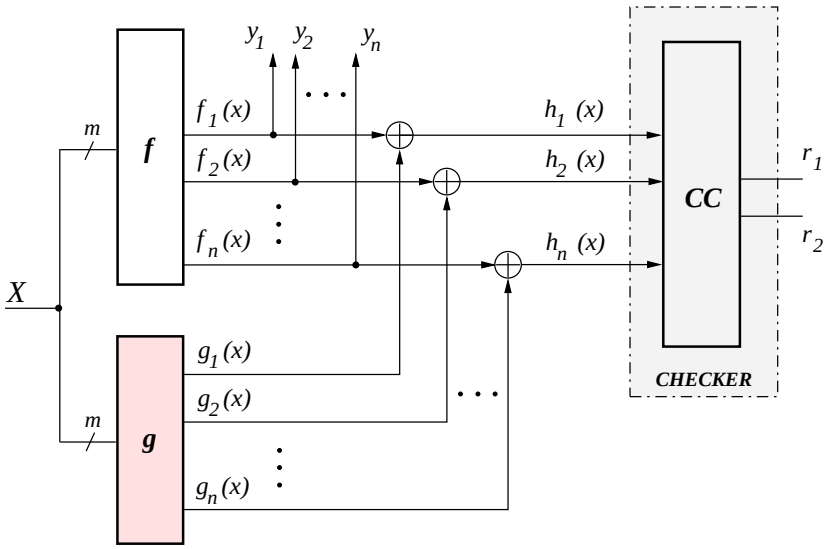

Abbildung 1.2: Neue Online-Fehlererkennungsmethode

überdeckung erreicht werden. Dieses Ziel soll insbesondere durch die Anpassung der zu bestimmenden komplementären Funktion an die Struktur der zu überwachenden Schaltung erreicht werden.

Neben den separierbaren Berger-Codes sollen auch nicht-separierbare m-aus-n Codes, insbesondere ein 1-aus-3 Code untersucht werden. Zu den wichtigen fehlererkennenden Codes gehören die Berger-Codes und 1-aus-n Codes [7]. In der Arbeit werden diese zwei Klassen fehlererkennender Codes für den Beweis der Effektivität der neue Methode verwendet.

Um den Aufwand für die Fehlererkennung zu reduzieren, sollen

- intelligente gemeinsame Realisierungen von ursprünglicher Schaltung und komplementärer Schaltung,
- abgeschwächte Anforderungen an die Fehlererkennung

untersucht werden.

Neben dem Nachweis der Möglichkeiten und Grenzen des neuen Verfahrens für den Entwurf von Fehlererkennungsschaltungen auf der Grundlage von komplementären Schaltungen sollen auch mathematische Aufgabenstellungen für die Optimierung kombinatorischer Schaltungen präzisiert werden, für die Nebenbedingungen als Relationen formuliert werden können. Es ist klar, dass der Entwurf von neuartigen Fehlererkennungsschaltungen mit geringem Hardware-Aufwand bei zunehmender Schaltungskomplexität von großer praktischer Bedeutung ist.

Der zusätzliche Hardware-Aufwand für Fehlererkennungsschaltungen ist bei den bekannten Verfahren relativ hoch. Die typische(klassische) Struktur einer bekannten Fehlererkennungsschaltung für eine kombinatorische Schaltung f zeigt Abbildung 1.3, vgl. [4]. Die ursprüngliche Schaltung f mit den Ausgängen $y_1 = f_1(x), \cdots, y_n = f_n(x), x = (x_1, \cdots, x_m)$ wird durch einen Coder C mit den $c_1(x), \cdots, c_l(x)$ Ausgängen so ergänzt, dass $f_1(x), \cdots, f_n(x), c_1(x), \cdots, c_l(x)$ Elemente eines Codes sind. $c_1(x), \cdots, c_l(x)$ sind die Kontrollbits des betrachteten Codes, die aus den Werten der Eingangssignale X berechnet werden. Der Generator G bestimmt aus den Schaltungsausgängen y die zugehörigen

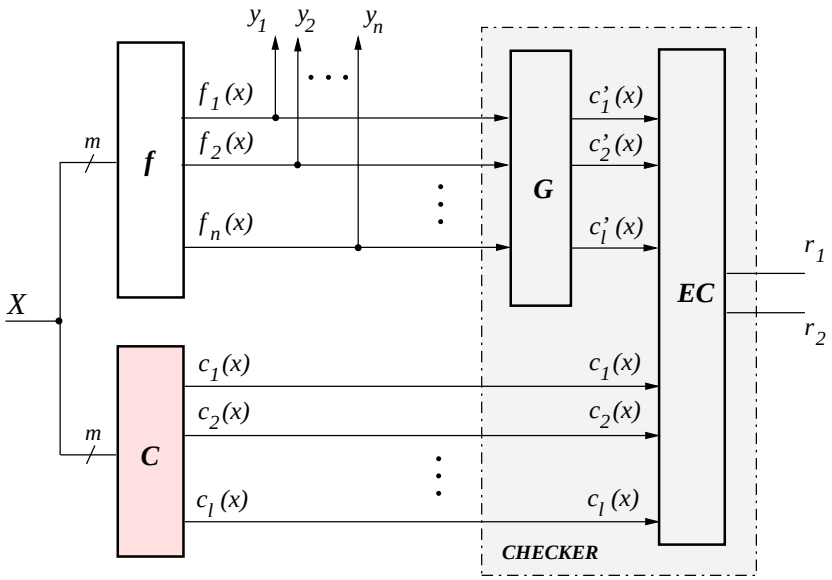

Abbildung 1.3: Klassische Online-Fehlererkennungsmethode

Kontrollbits $c'_1(x), \cdots, c'_l(x)$, die die Kontrollbits zu den von der Schaltung f erzeugten Ausgabenwerte $y_1(x), \cdots, y_n(x)$ sind. Stimmen $c_1(x), \cdots, c_l(x)$ und $c'_1(x), \cdots, c'_l(x)$ nicht überein, so wird ein Fehler in der Schaltung erkannt. Es kann hierfür ein beliebiger fehlererkennender Code verwendet werden, auf dessen Grundlage die Kontrolle organisiert wird. Die oben genannte traditionelle Methode zur Überwachung kombinatorischer Schaltungen kann als Methode der Kontrollstellenberechnung des redundanten Codes bezeichnet werden. In der vorgestellten Methode (Abbildung 1.3) werden aus den Werten der Ausgangsignale der Schaltung f die Kontrollbits des redundanten Codes ausgerechnet. Für die Überwachung der Schaltung werden die berechneten Codevektoren an den Prüfer (Checker) übergeben. Dabei ist die Codewortlänge des Codes immer größer als die Zahl der Ausgänge der ursprünglichen Schaltung f. In dem neuen Verfahren zum Entwurf von Fehlererkennungsschaltungen ist die Wortlänge des fehlererkennenden Codes gleich der Zahl der Ausgänge der funktionalen Schaltung f. Deswegen wird in der neuen Methode zur Überwachung einer Schaltung durch eine komplementäre Schaltung g (Abbildung 1.2) ein einfacherer Code Checker verwendet als in der traditionellen Methode (Abbildung 1.3). Der gegebene Umstand vereinfacht die Aufgabe der Konstruktion des selbstprüfenden Checkers.

1.3 Struktur der Arbeit

In dieser Dissertation wird untersucht, wie die Optimierung digitaler Schaltungen unter Verwendung der neue Methode der Logische Ergänzung verbessert werden kann. Diese Arbeit besteht aus sechs Kapiteln.

In Kapitel 2 wird zunächst ein Überblick über die Grundlagen des Entwurfs und den allgemeinen Ablauf beim Entwurf digitaler Fehlererkennungsschaltungen vermittelt. Weiterhin werden in Kapitel 2 die Prinzipien der Konstrukion selbstprüfender Systeme und ebenso die Eigenschaften, die sie aufwei-

sen müssen, betrachtet. Die neue Methode gehört zur Klasse der Methoden der funktionalen Diagnostik und garantiert die Konstruktion selbstprüfender Schaltungen. Deshalb wird im zweiten Kapitel der Frage nach der Konstruktion selbstprüfender Checker besondere Aufmerksamkeit gewidmet, insbesondere wird der neue selbstprüfende Checker für den 1-aus-3 Code vorgestellt. Die Fehlererkennung digitaler Geräte ist möglich dank der Einführung von Redundanz in die Struktur der Schaltung und der Verwendung verschiedener Fehlererkennungscodes. Aus diesem Grunde werden in Kapitel 2 die klassischen Fehlererkennungsmethoden unter Verwendung von Codes sowie Fehlererkennungscodes beschrieben.

In Kapitel 3 werden auf Grundlage von Berger-Codes und 1-aus-3 Code die neue Methode der Logische Ergänzung ausführlich vorgestellt. Während der Arbeit an diesem Kapitel wurde eine große Zahl an Experimenten mit dem Ziel des Vergleichs der neuen Methode mit den klassischen Methoden der funktionalen Diagnostik durchgeführt. Für einige MCNC-Benchmark-Schaltungen werden die Methoden hinsichtlich der Fehlererkennungswahrscheinlichkeit und der notwendigen Realisationsfläche verglichen. Zusätzlich werden in diesem Kapitel einige Algorithmen zur Zusammenstellung der Gruppen von Schaltungsausgängen vorgestellt, deren Ziel die Gewährleistung der minimalen Fläche der Ergänzungsschaltung ist. Kapitel 3 enthält experimentelle Resultate.

Bei der Optimierung kombinatorischen Schaltungen spielt die zweckmäßige Verwendung der Eigenschaften von Don't Cares in den Funktionen der Ausgänge eine wichtige Rolle. In Kapitel 4 wurde ein Versuch unternommen, Don't Cares in Schaltungen zu verwenden, die mit der neuen Methode der Logischen Ergänzung konstruiert wurden. In diesem Kapitel werden eine große Anzahl von Algorithmen zur Zusammenstellung der Gruppen von Ausgängen vorgestellt. Kapitel 4 enthält den experimentellen Teil.

Kapitel 5 der Beschreibung der notwendigen und hinreichenden Bedingungen für die Existenz vollständig selbstprüfender Schaltungen gewidmet. Die notwendigen und hinreichenden Bedingungen werden für komplementäre Schaltungen, die mit einem 1-aus-n Code überprüft werden, abgeleitet. Bei Entwurf vollständig selbstprüfender Schaltungen ist es erforderlich, dass sowohl der Code Checker, hier der 1-aus-n Checker, als auch die XOR-Gatter, die die entsprechenden Ausgänge der originalen Schaltung und der komplementären Schaltung verknüpfen, vollständig auf single stuck-at Fehler getestet werden. In Kapitel 5 wird bewiesen, dass die neue Methode der Logischen Ergänzung die Konstruktion vollständig selbstprüfender Schaltungen selbst in den Fällen zuläßt, in denen es nicht möglich ist, eine solche Schaltung mit einer der traditionellen Methoden der funktionalen Diagnostik zu konstruieren.

Eine Zusammenfassung in Kapitel 6 listet die im Rahmen dieser Dissertation gewonnenen Ergebnisse auf. Die Dissertation schließt mit einen Ausblick auf mögliche Erweiterungen.

1.4 Danksagung

Die vorliegende Dissertation ist möglich geworden durch eine Kooperation der Arbeitsgruppe „Fehlertolerantes Rechnen” an der Universität Potsdam und Institut für Automatisierungstechnik an der St. Petersburger Staatlichen Universität für Verkehrswesen.

Als erstes möchte ich mich bei meinem Mentor und Betreuer dieser Arbeit Prof. Dr. M. Gössel, Leiter der Arbeitsgruppe „Fehlertolerantes Rechnen”, ganz herzlich bedanken, der mir während der Entstehung dieser Arbeit in vielen ergiebigen Diskussionen mit konstruktiver Kritik, Anregungen und Rat zur Seite stand.

Mein ganz besonderer Dank gilt Herrn Prof. Dr. Vl. V. Saposhnikov, Leiter des Instituts für Automatik und Telemechanik, und Herrn Prof. Dr. Va. V. Saposhnikov, Prorektor für Wissenschaftliche Arbeit an der St. Petersburger Staatliche Universität für Verkehrswesen, die diese Arbeit überhaupt erst möglich machten.

Ich möchte Prof. E.S. Sogomonyan aus der Russischen Akademie der Wissenschaft für ausführliche hilfreiche Diskussionen zum Kapitel 2 danken. Weiter möchte ich Prof. D. K. Das aus der University of Calcutta danken, welcher einen großen Beitrag zur im Kapitel 4 beschriebenen Thematik leistete. Ich möchte meinen Kollegen im Lehrstuhl für Rechnerarchitektur und Fehlertoleranz in der Arbeitsgruppe Fehlertolerantes Rechnen für die Unterstützung meiner Arbeit in der Universität Potsdam danken. Ich möchte Nina Schumer innig danken für die Hilfe beim Niederschreiben und die sorgfältige Korrektur des Manuskripts.

Besondere Worte der Dankbarkeit möchte ich meiner Frau, meinen Eltern und meinem Bruder aussprechen, welche mich in der Zeit der Entstehung und Niederschreibung dieser Arbeit moralisch und seelisch unterstützten.

2 Entwurf selbstprüfender digitaler Schaltungen zur Fehlererkennung

2.1 Kombinatorische Schaltnetzwerke und Boolesche Algebra

Informationen (Zeichen, Text-, Bild-, Audiodaten) können binär verschlüsselt werden (zweiwertig, als Folge von Nullen und Einsen). Solche Informationsstellen werden als Bits (*binary digits*) bezeichnet und speichern jeweils entweder eine 0 oder eine 1. Die Datenverarbeitung wird mit Hilfe von Booleschen Funktionen[1] beschrieben. Boolesche Funktionen rechnen mit binären Werten. Sie werden in elektronischen Schaltungen verwendet und haben eine grundlegende Bedeutung für die binäre (zweiwertige) Digitaltechnik.

Um logische Verknüpfungen mit elektronischen Hilfsmitteln realisieren zu können, müssen folgende Voraussetzungen erfüllt sein [8]:

1. Ein vollständiges System binärer Operatoren muss vorliegen. Zu jedem elementaren Operator muss eine elektronische Schaltung existieren, das heißt, die Operationen müssen technisch realisierbar sein;

2. Es müssen systematische Verfahren existieren, mit denen Analyse und Synthese von Verknüpfungsschaltungen ermöglicht wird. Die Boolesche Algebra[2] ist ein geeignetes Hilfsmittel.

Definition 1:
Es sei B^n ein n-dimensionaler *Boolescher Raum*, dessen Punkte jeweils einer der $|B^n|$ möglichen Wertekombinationen aus B entsprechen. Die Abbildung, $f : B^n \rightarrow B^m$, die jedem Punkt in B^n eindeutig ein Element aus B zuordnet, heißt Boolesche Funktion.

Definition 2:
Die Punkte des Booleschen Raumes, denen der Wert 1 zugeordnet wird, heißen Einsstellenmenge (*On-Set*) von f, die Punkte, denen der Wert 0 zugeordnet wird, heißen Nullstellenmenge (*Off-Set*).

Boolesche Funktionen können auch unvollständig spezifiziert sein, d.h., für einige Punkte in B^n ist nicht festgelegt, auf welches Element von B^n sie abgebildet werden. Man sagt auch, dass diesen Punkten der *Don't-Care* Wert (Schreibweise: „$\times$" oder „$\sim$") zugeordnet ist. Entsprechend heißen diese Punkte auch *Don't-Care* Menge. Wird jeder Punkt des Booleschen Raumes auf ein Element aus B^m abgebildet, ist das durch eine mehrstellige Boolesche Funktion („Bündelfunktion") $f : B^n \rightarrow B^m$ beschreibbar.

[1] Booleschen Funktionen (binäre Schaltfunktionen) - Logische Funktionen und Operationen, benannt nach dem englischen Mathematiker George Boole (1815 - 1864).

[2] Die Boolesche Algebra wurde von C.E. Shannon und E.V. Huntington auf die Analyse und Synthese von logischen Schaltkreisen übertragen.

Eine Unterscheidung zwischen zwei Zuständen ist technisch besonders einfach zu realisieren. Zur technischen Realisierung benötigt man allgemein Schaltelemente, die zwei verschiedene Zustände annehmen können.

Wichtige Klassen Boolescher Funktionen sind [9]: 0-bewahrende, 1-bewahrende, monotone, lineare und selbstduale Boolesche Funktionen. Auf Grundlage dieser Eigenschaften wurden bis zur Gegenwart eine große Anzahl von Methoden für die logische Synthese vorgestellt.

In [10], [11] und [12] sind Beweise für die Effektivität der Nutzung dieser Eigenschaften für den Entwurf selbstprüfender Schaltungen zu finden. In [10] wird eine Methode zur Konstruktion von selbstprüfenden Schaltungen entwickelt, welche auf Suche und Nutzung von monoton unabhängigen Ausgängen des Schaltkreises basiert.

Die Arbeit [11] beschäftigt sich mit der Transformation einer vorliegenden Schaltung in eine selbstduale Schaltung. In [13] wurde die Eigenschaft der Selbstdualität für Fehlererkennung in kombinatorischen Schaltungen erstmals angewendet. Die kombinierte Benutzung der Selbstdualität mit der traditionellen Fehlererkennungsmethode (Abbildung 1.3) ist fähig, den zusätzlichen Hardware-Aufwand der Fehlererkennungsschaltung auf 22% zu verringern [14], [12].

Wie bereits oben erwähnt wurde, wird die Boolesche Algebra für Analyse und Synthese digitaler Schaltungen verwendet. Dabei entsprechen die Variablen einer Booleschen Funktion den Eingangssignalen des digitalen Schaltkreises, während die Funktionswerte den Ausgangssignalen entsprechen.

Die vorliegende Arbeit beschäftigt sich mit kombinatorischen Schaltungen. Kombinatorische Schaltungen benötigen wir als Ausgangspunkt für die Struktursynthese, als Resultat der Verhaltensberechnung einer Schaltungsstruktur und als Gegenstand der Verhaltensanalyse. In der Praxis sind kombinatorischen Schaltungen durch eine Liste von Gattern gegeben. Eine Schaltung $k = (X, Y)$ wird als kombinatorische Schaltung bezeichnet, wenn (nach [15]):

1. X und Y nichtleere Mengen sind (X - Menge von Eingangssignalen, Y - Menge von Ausgangssignalen);

2. φ - eine eindeutige Abbildung ist, die jedem Eingangssignal $x \in X$ eindeutig ein Ausgangssignal $y \in Y$ zuordnet, d.h. $\varphi : X \to Y$.

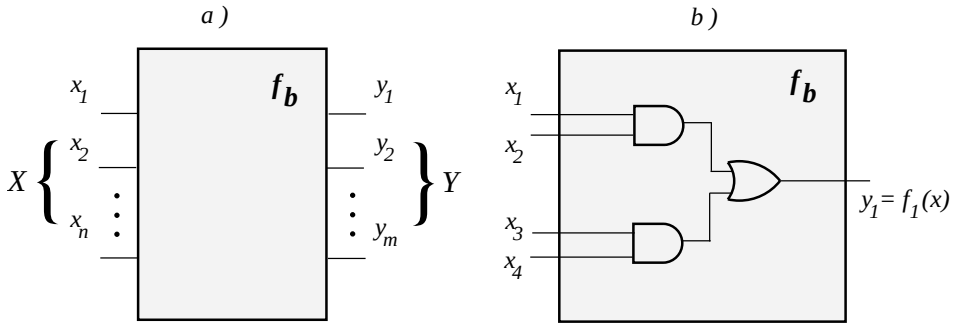

Abbildung 2.1: Kombinatorische Logik-Schaltung

Die Abbildung 2.1 a) zeigt die allgemeine Form eines binären Systems f_b mit n Eingängen $x_1, \cdots, x_n$ und m Ausgängen $y_1, \cdots, y_m$. Die Beispielschaltung f_b besteht aus drei zweistufigen Gattern und hat vier Eingänge x_1, x_2, x_3, x_4. Sie realisiert am Ausgang y_1 die Boolesche Funktion $f_1(x) = x_1x_2 \vee x_3x_4$

(Abbildung 2.1 b)). Es ist bekannt, dass die Anzahl m aller binäre Booleschen Funktionen mit n Variablen $m = 2^{2^n}$ ist (bei zwei Variablen existieren entsprechend $m = 16$ unterschiedliche Funktionen).

2.2 Kombinatorische Fehlererkennungsschaltungen

Der vorliegende Abschnitt behandelt einige grundlegende Prinzipien zur Überwachung digitaler Schaltungen.

Digitalsysteme können eine Vielzahl verschiedener Fehler aufweisen. In den vorhergehenden Abschnitten wurde bereits erwähnt, dass die Zuverlässigkeit beim Betrieb von Digitalsystemen sowie die Diagnose und Wartung von Fehlern durch die Verbreitung der Computer in alle Lebens- und industrielle Bereiche eine immer größere Bedeutung gewinnt.

Es gibt drei Möglichkeiten, das Problem der Fehlererkennung digitaler Schaltkreise zu lösen [16], [17]. Es ist auch möglich, zwei der unten genannten Lösungen gemeinsam zu realisieren [18].

1. External Test;
2. Off-Line Fehlererkennung (BIST-Modus);
3. On-Line Fehlererkennung (normaler Modus).

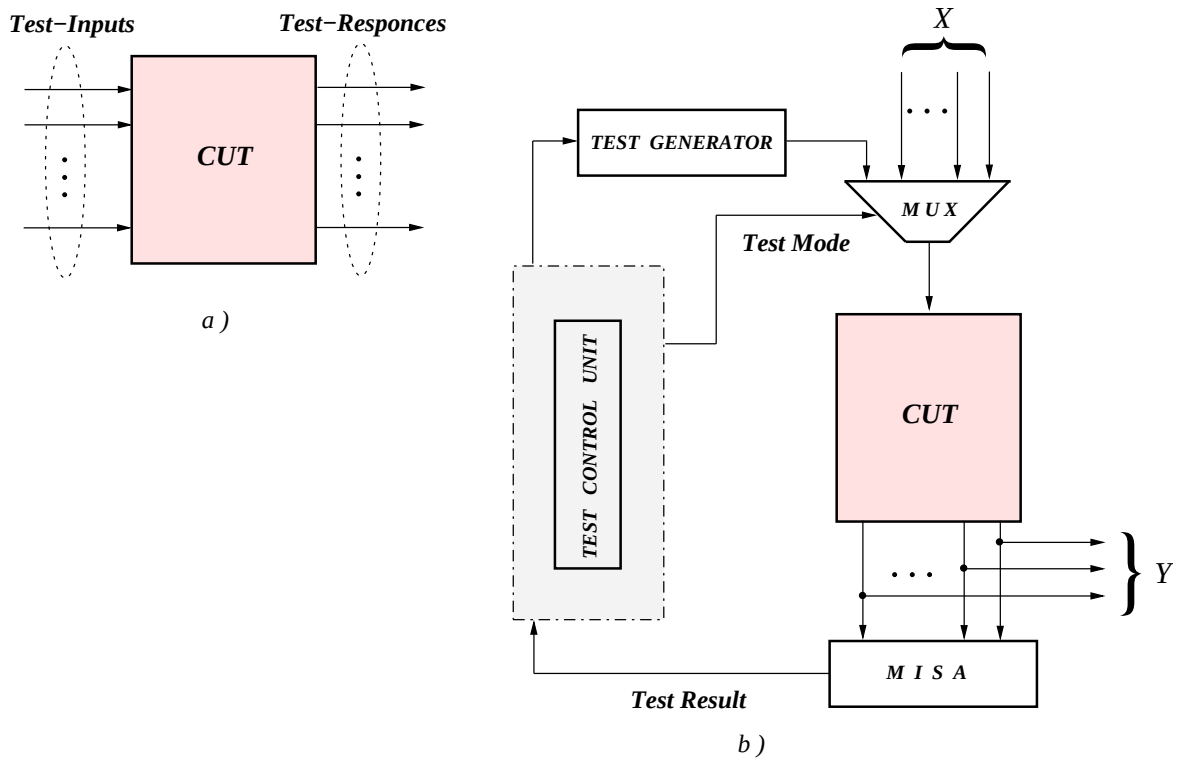

Abbildung 2.2: a) External Test Modus, b) Off-Line-Fehlererkennung (BIST-Modus)

Die Schaltung eines externen Tests ist in Abbildung 2.2 a) dargestellt. Die Idee des externen Tests besteht darin, die Testvektoren (*Test-Inputs*) an die CUT-Schaltung (*Circuit Under Test*) anzulegen. Auf Grundlage der Analyse der Ergebnisse, d.h. der Ausgangsvektoren (*Test-Responces*) schließt man auf die Anwesenheit oder Abwesenheit von Fehlern.

Für die Off-Line-Fehlererkennung (BIST-Modus) wird ein spezieller Testgenerator benötigt, der Tests für die Schaltung erzeugt (Abbildung 2.2 b)). Die Kontrolle der Schaltung erfolgt durch Eingabe dieser Testmuster an den Eingängen der *CUT*-Schaltung (*Test Mode*). Ein weiteres wichtiges Element in diese Struktur ist der Multiplexer *MUX*. Der Multiplexer führt das Umschalten des Systems in den Testbetrieb durch. Die erste Aufgabe des *MISA (Multiple-Input Signature Analyser)* ist es, die Ausgaben des *CUT* zu analysieren und dabei eventuelle Fehler zu erkennen. Seine zweite Aufgabe besteht darin, Informationen über aufgetretene Fehler an das *TCU (Test Control Unit)* weiterzugeben.

On-Line Fehlererkennungsschaltungen überprüfen kombinatorische Schaltungen im laufenden Betrieb auf in ihnen auftretende Fehler [19]. Durch das Überwachungssystem werden alle in der Schal-

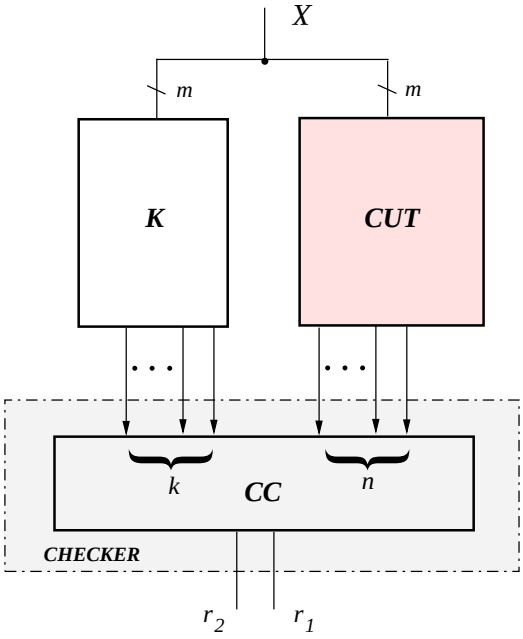

Abbildung 2.3: Fehlererkennungsschaltung im normaler Modus

tung entstehenden nicht redundanten Fehler entdeckt, sobald sie sich auf den Ausgang der Schaltung auswirken. Hier besteht die fehlererkennende Schaltung aus drei Teilen: aus der ursprünglichen kombinatorischen *CUT*-Schaltung , der Kontrollschaltung K und dem Checker CC (vergleicht die Ausgangssignale zweier Schaltungen) (Abbildung 2.3). Normalerweise bilden die Ausgänge der ursprünglichen Schaltung mit den Ausgängen der Kontrollschaltung die Codewörter eines Codes. Beim Auftreten eines Fehlers werden die Ausgänge der Schaltungen verfälscht. Die Aufgabe des Code Checkers ist es nun, diesen Fehler durch die verfälschten Ausgangsignale zu erkennen. In [5] wurden die Grundlagen für Fehlererkennungsschaltungen beschrieben.

Aufgrund der sich ständig verkleinernden Schaltungsabmessungen entstehen zunehmend flüchtige Fehler, die nur im laufenden Betrieb, eben genau dann, wenn sie auftreten, durch Fehlererkennungs-

schaltungen, erkannt werden können. Der zusätzliche Hardware-Aufwand für Fehlererkennungsschaltungen ist bei den bekannten Verfahren relativ hoch. Eine selbstprüfende Schaltung (*Self-Checking Circuit*) ist die Schaltung, welche im laufenden Betrieb alle Fehler eines bestimmten Fehlermodels der ursprünglichen Schaltung und der Kontrollschaltung erkennt. Selbstprüfende Schaltungen erkennen dabei sowohl permanente, vorübergehende und periodisch auftretende Fehler. Das Fehlermodell hat in diesem Zusammenhang eine wichtige Bedeutung. Dieser Begriff wird im Abschnitt 2.5.1 näher beschrieben.

2.3 Statische Redundanz als Fehlererkennungsmittel

Zum Erreichen der Fehlertoleranz ist Redundanz notwendig. Redundanz umfasst alle Betriebsmittel (Hardware bzw. Software), die für die eigentliche Funktion im fehlerfreien Fall nicht benötigt werden.

In Abbildung 2.4 ist das prinzipiell einfachste Verfahren für den Entwurf von Fehlererkennungsschaltungen dargestellt. Das Verfahren besteht aus der Verdopplung der ursprünglichen Schaltungen f und dem Vergleich der Ausgangssignale $\{f_1, \cdots, f_n\}$ beider Schaltungen. Zwei Komponenten führen gleichzeitig dieselbe Funktion aus. Die Ausgangssignale dieser Komponenten werden einem Vergleichselement, dem sogenannten „Comparator" oder „Vergleicher" zugeführt. Im Falle $\{f_1, \cdots, f_n\} = \{f_1, \cdots, f_n\}$ gibt der Vergleicher den Wert 0 (kein Fehler) und im Falle $\{f_1, \cdots, f_n\} \neq \{f_1, \cdots, f_n\}$ den Wert 1 (Fehler) aus.

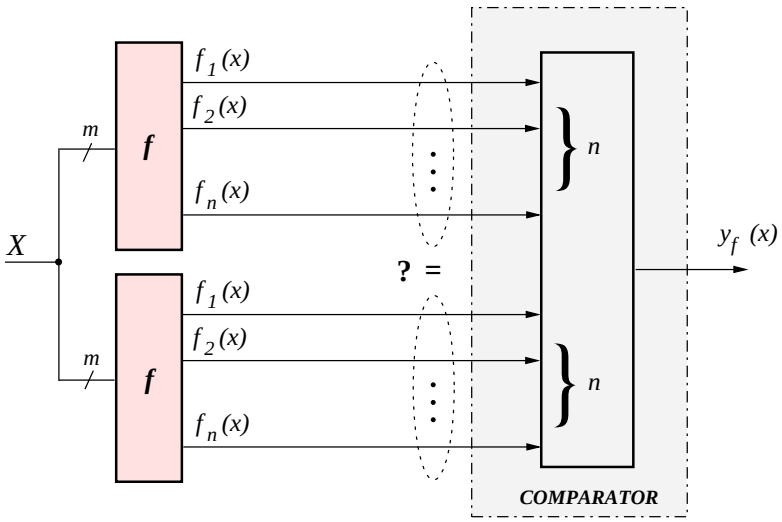

Abbildung 2.4: Verdopplung und Vergleich

„Verdopplung und Vergleich" ist die direkte, universelle Implementierung der statischen oder strukturellen Redundanz[3]. Dieses Verfahren ist praktisch ohne Schwierigkeiten für kombinatorische und sequentielle Schaltungen anwendbar.

[3] Strukturelle Redundanz bezeichnet die Erweiterung eines Systems um zusätzliche (gleich- oder andersartige) für den Nutzbetrieb entbehrliche Komponenten. Strukturell redundante Hardware verursacht im allg. erhebliche Zusatzkosten, weshalb beim Entwurf von Fehlertoleranz-Verfahren häufig versucht wird, den zusätzlichen Hardwareaufwand gering zu halten bzw. zusätzliche Hardware auch zur Leistungssteigerung zu nutzen [20].

Bei diesem Verfahren wird jeder Fehler erkannt, der nur eine der beiden Schaltungen betrifft. Neben dem Vorteil der Einfachheit und Universalität hat dieses Verfahren einige Nachteile. Der Hardware-Aufwand für Fehlererkennung ist aber in den meisten Anwendungsfällen zu hoch (mehr als 200% der ursprünglichen Schaltung) [14].

Weiterhin kann eine Fehlererkennungsschaltung, die nach der beschriebenen Methode konstruiert wurde, nicht erkennen, wenn sowohl in der ursprünglichen Schaltung als auch in der zweiten identischen Schaltung ein Fehler auftrat. Design-Fehler in der ursprünglichen Schaltung können deshalb ebenfalls nicht entdeckt werden. Außerdem führt ein Fehler im Vergleicher zu einer Störung des gesamten Systems.

2.4 Fehlererkennung mit redundanten Codes

Computersysteme verwenden für sämtliche Operationen Daten, die in einer Folge von Bits verschlüsselt sind. Durch die rasante Entwicklung von Computersystemen und die Zunahme der internen Systemkomplexität werden immer höhere Anforderungen an fehlertolerante Systeme gestellt. Die Kodierungstheorie beschäftigt sich u.a. mit dem Entwurf und der Anwendung von fehlererkennenden Codes[4] sowie den zugehörigen Kodierungsalgorithmen. In den meisten heute erhältlichen selbstprüfenden Schaltungen bilden die Schaltungsausgänge Codewörter eines fehlererkennenden Codes. Beeinflusst ein Fehler in einer Schaltung mit codierten Ausgängen einen Ausgang und somit eine Stelle des Codeworts, so kann dieser leicht erkannt werden. Durch die Verfälschung der Codewörter können Fehler durch einen selbstprüfenden Checker erkannt werden. Es ist klar, dass für die Zuverlässigkeit des Gesamtsystems die Zuverlässigkeit jedes Teilsystems gewährleistet sein muss.

Im folgenden werden die Hauptbegriffe und wesentliche Definitionen der Codierungstheorie erklärt. Unter einem Code ist eine Menge von Wörtern über einem Alphabet zu verstehen, die eindeutig den Elementen einer Inputmenge zugeordnet sind. Dabei kann die Inputmenge eine Menge von Symbolen oder auch eine Menge von Wörtern über einem Alphabet sein. Die Elemente eines Codes heißen Codewörter. Kann aus jeder Folge von Codewörtern wieder eindeutig auf die Inputfolge zurückgeschlossen werden, aus der durch den Kodierungsprozess die vorgelegte Folge von Codewörter entsteht, so heißt der Code dekodierbar.

Mit einer m-Bit langen Zeichenfolge können Codeworte realisiert werden. Wenn nicht alle Möglichkeiten genutzt werden, so besitzt der Code eine Redundanz[5] ($R_d > 0$). Mit $R_d > 0$ kann eine Fehlererkennung durchgeführt werden.

Definition 3:
Als *Gewicht* g[6] eines Codewortes wird die Anzahl der Stellen mit dem Wert "1" bezeichnet.

Definition 4:
Die kleinste Distanz d zwischen zwei beliebigen Wörtern eines Codes wird *Hamming-Abstand* h genannt.

[4]Der Begriff eines Codes ist laut DIN 44300 folgendermaßen definiert: Ein Code ist eine Vorschrift für die eindeutige Zuordnung der Zeichen eines Zeichenvorrats zu denjenigen eines anderen Zeichenvorrats.

[5]Die Redundanz berechnet sich aus: $R_d = log_2(N_{max} - log_2(N_{code}) > 0$, mit $N_{max} = 2^m$ und N_{code}- Anzahl der Codewörter.

[6]Beispiel: $g(0101) = 2,\ g(0000) = 0$.

Diese kleinste Distanz ergibt sich aus der Anzahl der Binärstellen, die mindestens verfälscht werden müssen, um ein gültiges Codewort in ein anderes gültiges Codewort zu überführen. Für zwei Codewörter $a = (a_1, a_2, \cdots, a_n)$ und $b = (b_1, b_2, \cdots, b_n)$ kann h mit Hilfe der XOR-Funktion bestimmt werden.

Codierung ermöglicht das Erkennen und Korrigieren von Fehlern, die bei der Nachrichtenübertragung im gestörten Datenkanal eingetreten sind. Ein linearer systematischer Binärcode enthält Codewörter, die aus der Nachricht und Kontrollzeichen bestehen. Die Codewörter werden mit einer Generatormatrix erzeugt. Falls jedes Paar von Codewörtern eines Codes sich in mindestens d Stellen unterscheidet, kann dieser Code $[1/2(d-1)]$ Fehler korrigieren und $d/2$ Fehler erkennen.

Eine sehr effektive Möglichkeit zur Erkennung und Maskierung von Fehlern stellt die Verwendung redundanter Codes dar. Das Konzept redundanter Codes basiert darauf, dass die Menge der gültigen Codewörte nur eine Teilmenge T der Menge M aller möglichen Bitkombinationen bildet. Der redundante Code heißt separierbar, wenn jedes Codewort aus zwei Teilen besteht, den ursprünglichen Datenbits und den angehängten Prüfbits oder Kontrollbits. Kann zudem jedes Prüfbit eindeutig als eine Linearkombination einiger der Datenbits errechnet werden, dann liegt ein linearer Code vor. Als

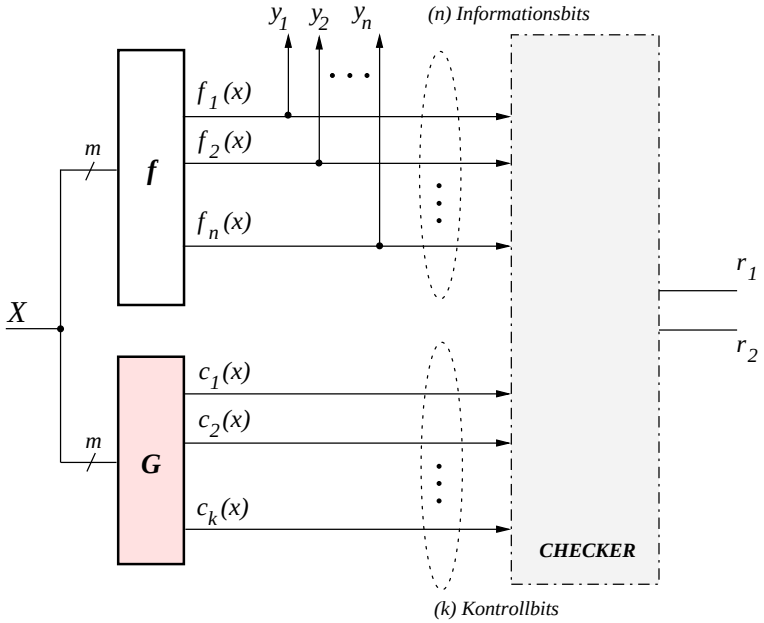

Abbildung 2.5: Klassische Struktur für Fehlererkennungsschaltungen

wichtige Fehlererkennungscodes sind die folgenden Code-Klassen zu nennen: Hamming-Codes, BCH (Bose-Chaudhuri-Hocquenghem) Codes, Paritätscodes und Berger-Codes.

Die klassische Struktur für Fehlererkennungsschaltungen [21], die systematische Codes verwenden, ist in der Abbildung 2.5 vorgestellt. Die funktionale Logik (ursprüngliche Schaltung) mit den n Ausgängen $y_1(x), \cdots, y_n(x)$ wird durch einen Kontrollbitgenerator G mit den k Ausgängen $c_1(x), \cdots, c_k(x)$ so ergänzt, dass $y_1(x), \cdots, y_n(x)\ c_1(x), \cdots, c_k(x)$ Elemente eines Codes sind. $c_1(x), \cdots, c_k(x)$ sind die Kontrollbits des betrachteten Codes, die aus den Werten der Eingangsignale X berechnet wer-

den. Der Checker entscheidet über die Zugehörigkeit einer erhaltenen Bitfolge zum verwendeten Code. Bei Auftreten eines Fehler in der Schaltung gibt der Checker ein Fehlersignal aus.

In Abbildung 2.6 ist der prinzipielle Aufbau eines Checkers für systematische Codes dargestellt. Der Checker besteht aus dem zusätzlichen Kontrollbitgenerator *RCCG (replicated code check generator)* und aus dem Comparator C. An den Eingängen des *RCCG* liegen die n Informationsbits $y_1(x), \cdots, y_n(x)$ der funktionalen Logik an. Die Aufgabe des *RCCG* besteht darin, die n Informationsbits aus den Schaltungsausgängen der funktionalen Logik in die k zugehörigen Kontrollbits $z'_1(y), \cdots, z'_k(y)$, $k < n$ zu transformieren. Die vom *RCCG* ermittelten Kontrollbits $z'_1(y), \cdots, z'_k(y)$ werden durch den Comparator C mit den durch den Kontrollbitgenerator ermittelten Kontrollbits $z_1(x), \cdots, z_k(x)$ verglichen. Stimmen $z_1(x), \cdots, z_k(x)$ und $z'_1(y), \cdots, z'_k(y)$ nicht überein, so wird ein Fehler im System erkannt.

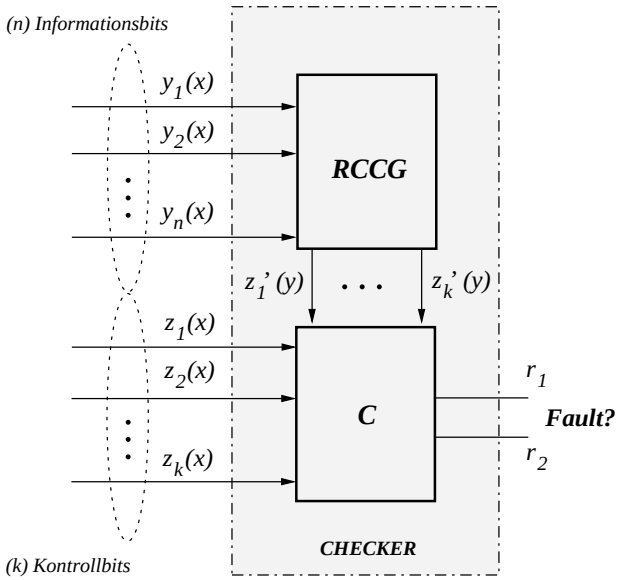

Abbildung 2.6: Checker für systematischen Code

Als erstes Beispiel sei ein System gegeben, das auf dem *Dual-Rail* Code basiert. In einem Dual-Rail System repräsentieren zwei Leitungen mit komplementären Werten eine Variable. Eine Variable f_1 wird dann durch die Werte f_1 und $\overline{f}_1$ dargestellt. Im Dual-Rail Code sind die Signale $f_1\overline{f}_1$=11 und $f_1\overline{f}_1$=00 nicht gültig und repräsentieren einen Fehler. Die sogenannte *Dual-Rail-Logik* [5] stellt eine spezielle Variante von Fehlererkennungsschaltungen durch Verdopplung der Schaltungen und Vergleich der Outputs dar. Diese Logik findet eine sehr einfache Implementierung. Die Fehlererkennungsschaltung verarbeitet an den entsprechenden Stellen jeweils das negierte Signal der funktionalen Logik. Der erforderliche Hardware-Aufwand geht durch diese Eigenschaften gegenüber der Ursprungsform von Verdopplung und Vergleich etwas zurück, ohne dass sich die Fehlererkennungseigenschaften verschlechtern. Diese Verbesserung beruht darauf, dass sich alle Negationen als Leitungskreuzung zwischen der funktionalen Logik und der zweiten Schaltung mit den invertierten verdoppelten Leitungen realisieren lassen.

Nach diesen einführenden Bemerkungen werden nun wichtige Beispiele für die Anwendung ver-

schiedener fehlererkennender Codes in digitalen Schaltungen gegeben.

2.4.1 Paritätscode. Anwendung von Paritätscodes zur Fehlererkennung

Der Paritätscode ist ein systematischer separierbarer Code und hat die Hamming-Distanz zwischen zwei beliebigen Wörtern gleich 2 (d_{min}=2).

Der Paritätscode besteht aus zwei Teilen, den eigentlichen Daten und einem Kontrollbit, dem sogenannten Paritätsbit P. Durch Hinzufügen des Paritätsbits unterscheiden sich zwei gültige Codewörter in mindestens zwei Binärstellen voneinander. Das bedeutet, dass sich ein Übertragungsfehler in einem Bit des Codeworts sicher erkennen lässt. Der Paritätscode hat die Besonderheit, dass unter allen Fehlererkennungscodes die Kontrollstellenanzahl des Paritätscodes am geringsten ist. Das Kontrollbit ist die lineare Summe der Informationsbits der Codewörter. Die Überprüfung dieser Bedingung für ein Codewort lässt sich durch eine Baumstruktur aus XOR Gattern realisieren.

Das Paritätsbit P eines Codeworts X wird aus dem Gewicht $\omega(X)$, der Anzahl der Einsen im Codewort, bestimmt. Für die Berechnung des Paritätsbits (für gerade Parität und für ungerade Parität) gilt:

$$\begin{aligned} P_{gerade} &= \omega(X)\ mod\ 2, \\ P_{ungerade} &= [\omega(X)+1]\ mod\ 2. \end{aligned} \tag{2.1}$$

Bei ungerader Parität (Gleichartigkeit) ist der Wert des Paritätsbits so zu wählen, dass die Zahl der Einsen im Codewort ungerade ist, während bei gerader Parität das Datenwort auf eine gerade Anzahl von Einsen ergänzt wird. Mit Hilfe von Paritätscodes werden Einzelfehler erkannt, der Code ist daher fehlererkennend. Treten gleichzeitig zwei Fehler auf (Doppelfehler), so heben sie sich gegenseitig auf und werden nicht erkannt. In Tabelle 2.1 ist ein Beispiel des Paritätscodes mit gerader Anzahl von Einsen zu sehen.

y_1	y_2	y_3	y_4	y_5	y_6	y_7	P
0	1	0	1	1	0	0	1
1	1	0	1	1	1	1	0
0	0	1	0	0	0	0	1
1	0	1	0	0	1	1	0

Tabelle 2.1: Paritätscode mit gerader Anzahl von Einsen

Für die Bildung des Paritätsbits wird eine logische Schaltung benötigt, die als Paritätsgenerator (*parity generator*) bekannt ist. Die Prüfung empfangener Zeichen auf die Einhaltung der vorgegebenen Zeichenparität erfolgt mit Hilfe von Paritätsprüfschaltungen (*parity checker*). Da beide Funktionen logisch sehr ähnlich sind, genügt in der Praxis eine Standardschaltung. Die Verwendung von Paritätscodes in einer Fehlererkennungsschaltung ist zum ersten Mal in (FFSB68) beschrieben. Abbildung 2.7 zeigt die entsprechende Fehlererkennungsschaltung.

In dieser Struktur hat die originale funktionale Schaltung m Eingänge $x_1, \ldots, x_m$ und $n+1$ Ausgänge $y_1, \cdots, y_{n+1}$. Die funktionellen Ausgänge $y_1, \cdots, y_{n+1}$ realisieren Boolesche Funktionen $f_1(x)$, $\cdots, f_n(x), P(y)$. An den funktionellen Ausgängen ist ein Baum von XOR-Gattern mit je zwei Eingängen implementiert. Dadurch wird gesichert, dass die Ausgänge der originalen Schaltung durch den Paritätscode kodiert wurden.

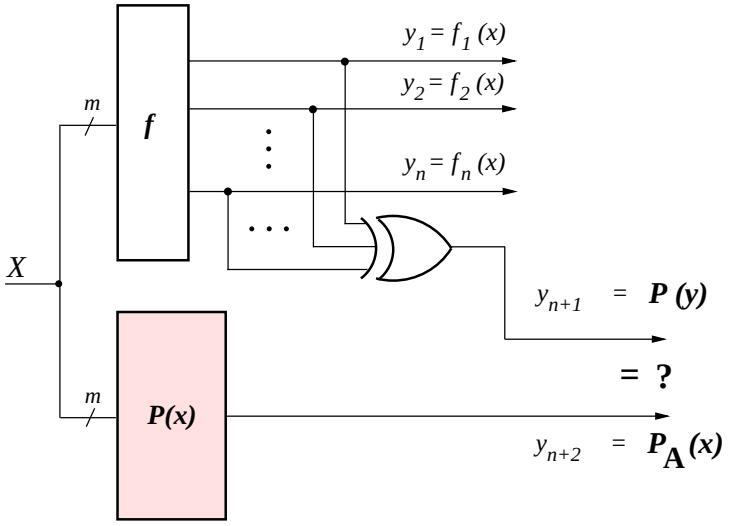

Abbildung 2.7: Allgemeine Struktur der fehlererkennenden Schaltung unter Verwendung von Paritätscodes

Die Paritätsfunktion $P(y)$ auf dem Ausgang y_{n+1} wird folgendermaßen berechnet:

$$P(y) = y_1 \oplus y_2 \oplus \cdots \oplus y_n. \tag{2.2}$$

Der zweite Teil dieses Systems ist die Paritätsschaltung $P(x)$, welche nur einen Ausgang y_{n+2} hat, der die Paritätsfunktion $P_A(x)$ realisiert. Die Paritätsschaltung $P(x)$ ist die optimierte Realisierung der Paritätsfunktion $P(y)$. Im Falle der fehlerfreien Arbeitsweise der originalen Schaltung f und der Paritätsschaltung $P(x)$ gibt die Fehlererkennungsschaltung an den Ausgängen y_{n+1} und y_{n+2} die Belegungen 00 oder 11 aus. Eine andere Ausgabe zeigt einen Fehler an.

Die angegebene Methode zur Fehlererkennung durch Paritätscodes hat folgende Eigenschaften:

1. Für eine getrennte[7] Implementierung der beiden Teilsysteme in Fehlererkennungsschaltungen, die den Paritätscodes verwenden, wird ein zusätzlicher Hardware-Aufwand von ca. 56,0% benötigt. Werden die beiden Teilsysteme jedoch gemeinsam implementiert, ergibt sich ein zusätzlicher Hardware-Aufwand von ca. 28,0%;

2. Bei der Paritätsbitprüfung werden im On-Line Betrieb etwa 78% der Fehler erkannt;

3. Der Entwurf des Checker ist sehr einfach (da nur zwei Signale verglichen werden müssen);

4. Die Paritätsbitprüfung erkennt das Auftreten genau eines Fehlers, allgemein - einer ungeraden Anzahl von Fehlern. Dagegen wird eine gerade Anzahl von Fehlern nicht erkannt;

5. Da der Paritätscode ein minimale Hamming-Distanz von $d_{min} = 2$ besitzt, kann das Verfahren keine Fehler korrigieren.

[7] Das schließt das gleichzeitige Auftreten eines Haftfehlers in beiden Schaltungen unter der Annahme von Einzelfehlern aus.

Es gibt viele Möglichkeiten, die Eigenschaften von Booleschen Funktionen für eine Fehlererkennung unter Verwendung des Paritätscodes mit relativ geringem Schaltungsaufwand zu nutzen. Zum Beispiel, wird in [12] eine noch relativ einfache und aufwandgünstige Variante von Paritätsprüfungsschaltungen unter Verwendung der Selbst-Dualität der Ausgangsfunktion beschrieben. In diesem Verfahren wird bei getrennter Implementierung der Teilsysteme für die Fehlererkennung ein zusätzlicher Schaltungsaufwand von ca. 25,0% und für eine gemeinsame Implementierung ca. 18,0% benötigt.

2.4.2 Berger-Codes. Anwendung von Berger-Codes zur Fehlererkennung

Ein weiterer redundanter Code ist der Berger-Code. Der vorliegende Abschnitt behandelt einige grundlegende Eigenschaften des Berger-Codes.

Berger-Codes sind systematische Codes und gehören zur Klasse der separierbaren Codes. Systematisch heißt ein Code dann, wenn die Informationsfolge nach wie vor Bestandteil des Codewortes ist. Der Vorteil systematischer Codes ist, dass die Informationen zumindest nach fehlerloser Übertragung ohne weitere Berechnungen direkt abgelesen werden können. Der Berger-Code gehört zur Klasse der *All Unidirectional Error Detection Codes*.

Definition 5:
Ein Fehler heißt *unidirektional*, wenn die fehlerhafte Veränderung der Signalwerte nur in eine Richtung stattfindet. Dabei werden entweder Nullen zu Einsen, oder Einsen zu Nullen gestört.

Gegenwärtig stellen unidirektionale Fehler den dominanten Fehlertyp in den VLSI Systemen dar. Aus diesem Grunde ist die Suche nach neuen Fehlererkennungsmethoden, die den Berger-Code verwenden, sehr wichtig [4]. Berger-Codes finden eine sehr breite praktische Anwendung. Zum Beispiel, wird in [22] ein vollständig selbstprüfender Prozessor sowie eine vollständig selbstprüfende ALU vorgestellt [23]. In der Arbeit [24] wird eine neue BCP On-Line Technik (*Berger-Code Prediction*) vorgestellt, die für komplexere Strukturen wie FPU (*Floating Point Unit*) verwendet werden kann.

Man bezeichnet diese Codes als optimale Codes, da sie eine minimale Anzahl von Kontrollbits bei gegebener Anzahl von Informationsbits erfordern [25]. Zusätzlich kann ein Berger-Code alle monotonen Multibitfehler eines Codewortes erkennen [10].

Zwischen der Kontrollstellenanzahl und der Informationsstellenanzahl besteht logarithmische Abhängigkeit (im Gegensatz zu dem m-aus-n Code).

Definition 6:
Der Code A heißt *Berger-Code* [26], wenn A ein separierbarer Code ist und die Anzahl der Codewörter auf folgende Weise gebildet wird: $[a : a] = [I_a \;\; P_a]$, wobei P_a die binäre Darstellung der Anzahl der Nullen in dem Informationsteil des Wortes I_a ist.

Die Länge eines Codewortes ergibt sich also aus $L = k + [log_2(k+1)]$, wobei k die Anzahl der Informationsbits ist. Man unterscheidet die verschieden Berger-Codes als Codes mit maximaler und nicht maximaler Länge. Wenn $k - 2^m - 1,\ m \geq 1$ gilt, bezeichnet man diesen Berger-Code als einen Code mit maximaler Länge. Andernfalls bezeichnet man den Code als Code mit nicht maximaler Länge. Solche Codes, für die diese Beziehung nicht gilt, sind zum Beispiel die Berger-Codes 2S2, 7S3, 15S4, 19S15.

Beispiel 1:

Falls ein Wort sieben Informationsbits besitzt (z.B. $k = 0101000$), dann ist die Anzahl der Kontrollbits des Codewortes $|P_a| = [log_2(7+1)] = 3$ und die Länge eines Codewortes ist gleich 10 $(L = k + [log_2(k+1)] = 10)$. In diesem Beispiel entsprechen die Kontrollbits dem binären Äquivalent der Anzahl der Nullen (hier fünf) in den sieben Informationsbits. Es ergibt sich ein Codewort des 3S7 Berger-Codes: $0101000\{101\}$.

Definition 7:
Der Berger-Code A heißt vollständiger Code, falls alle $2^{(n-I_a)}$ möglichen Kontrollwörter in den $(n - I_a)$ - Kontrollstellen der Codewörter erscheinen.

Ein weiteres Beispiel für den Berger-Code mit drei Informationsbits und zwei Kontrollbits ist in der Tabelle 2.2 dargestellt. Der vorgestellte vollständig separierbare Code wird in der Arbeit im weiteren für die Konstruktion von Fehlererkennungsschaltungen nach der neuen Methode verwendet.

y_1	y_2	y_3	y_4	y_5
0	0	0	1	1
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	0	0

Tabelle 2.2: Codewörter des 5S2 Berger-Codes

Es existiert eine große Anzahl an Berger-Code Unterklassen. Eine mögliche Modifikation des Codes besteht in der Veränderung der Codewortlänge, und zwar in der Verkleinerung der Anzahl der Kontrollbits, z.B. [27]. Für praktische Anwendungsfälle ist dieser Berger-Codes sehr nützlich. Der Grund hierfür ist die Tatsache, dass die Struktur des Checkers bedeutend vereinfacht wird.

2.4.3 M-aus-N Codes. Anwendung von m-aus-n Codes zur Fehlererkennung

In der Praxis wichtige Codes sind die m-aus-n Codes, die in dem folgenden Abschnitt betrachtet werden. Neben der Erkennung von konstanten Fehlern haben diese Codes vor allem Vorteile bei der Erkennung von unidirektionalen Fehlern.

Bei einem m-aus-n Code haben alle $(\frac{m}{n})$ Wörter das gleiche Gewicht. Unter Gewicht wird die Anzahl der Einsen in einem Codewort verstanden. M-aus-n bedeutet, dass von den n Gesamtstellen stets m Stellen mit 1 besetzt sind. In der Literatur sind dafür auch häufig die Bezeichnung „*m-hot*" oder „gleichgewichtige" Codes [28] aufzutreffen.

Dies sind Blockcodes mit der Wortlänge L, bei denen in jedem Codewort genau (m) Einsen und dementsprechend $(L - m)$ Nullen vorkommen. Bei gegebenen (m) und (n) gibt es offenbar genau verschiedene Codewörter. Die Anzahl D der möglichen Zeichen eines m-aus-n Codes wird mit Hilfe

des binomischen Satzes berechnet:

$$D = \left(\frac{m}{n}\right) = \frac{n!}{m!(n-m)!} \quad . \tag{2.3}$$

Beispielsweise ist die Anzahl der Codewörter des 2-aus-5 Codes (die Wörter sind $\{11000\}, \{10100\}, \{10010\}, \cdots, \{00011\}$ gleicht $D = 10$. Da in allen Codewörter dieselbe Anzahl von Einsen enthalten ist, müssen sich zwei verschiedene Codewörter in mindestens zwei Stellen unterscheiden, so dass die Hamming-Distanz von m-aus-n Codes gleich zwei ist. Damit sind Ein-Bit-Fehler immer erkennbar. M-aus-n Codes gehören zur Klasse der nicht-separierbaren Blockcodes.

Es existiert eine große Menge spezieller Fälle der m-aus-n Codes. Man klassifiziert in [4]:

1. **k-aus-2k Codes**, mit (k) ist eine ganze Zahl (Beispiel: 0011, 0101, 1001, 0110, 1010, 1100). Ist die Anzahl der Nullen (m) ungleich der Anzahl der Einsen (n), liegt mindestens ein Ein-Bit-Fehler vor;

2. **k-pair Dual-Rail Codes**. Es existieren nur 2^k Codewörter von den $2k!/k!k!$ möglichen der k-aus-2k Codes. (Beispiel: 0011, 1001, 0110, 1100). Ist die Anzahl der Nullen (m) ungleich der Anzahl der Einsen (n), liegt mindestens ein Ein-Bit-Fehler vor;

3. **1-aus-n Codes**. Anzahl der Einsen (m) in den Codewörtern ist immer gleich eins $(m = 1)$, $d_{min} = 2$.

2.5 Self-Checking System

Heutige Digitalsysteme übernehmen zunehmend Schutz- und Leitfunktionen in Industrie, Verkehr und Handel. Es existiert derzeit kein System, dass vollständig vor Defekten, Fehlern und Störungen in der Arbeit geschützt wäre. Der Begriff des Fehlers ist im allgemeinen sehr vielfältig interpretierbar: Fehler können auftreten in der Spezifikation, im Design, in der Implementierung oder während der Ausführung eines Systems. Fehler will man aber eigentlich vermeiden, oder wenigstens erkennen und/oder ihre Auswirkungen tolerieren können. Der Umgang mit Fehlern im Rahmen einer Systementwicklung ist dennoch kaum systematisiert.

Aber ein beliebiges Digitalsystem, das ebenso ein Teil anderen Systems sein kann, kann unkorrekte Ergebnisse aus zwei Hauptgründen generieren:

- das Vorhandensein eines internen Fehlers;
- das Vorhandensein eines Fehlers am Eingang der Schaltung.

Dieser Abschnitt behandelt wichtige Aspekte des Fehlerbegriffs und gibt eine Fehler-Klassifikation. Weiterhin wird das Fehlermodell als Beschreibungsmittel der Fehlermöglichkeiten eines Systems definiert.

Diese Dissertation beschäftigt sich mit der Transformation einer vorliegenden kombinatorischen Schaltung in eine selbstprüfende Schaltung. Die Abschnitte 2.5 und 2.6 führen den Begriff der selbstprüfenden Schaltung ein. Schließlich wird ein neuer vollständig selbstprüfender Checker für einen 1-aus-3 Code vorgestellt, der im Vergleich mit bisherigen Checkern vorteilhafte Parameter (Abschnitt 2.6.6) aufweist.

2.5.1 Fehlermodell

Digitalsysteme können eine Vielzahl verschiedener Fehler aufweisen. Man unterscheidet physikalische Fehler und systematische Fehler. Physikalische oder Hardwarefehler[8] treten als Folge von Alterung, Verschleiß oder äußeren Einwirkungen auf (abweichende geometrische oder chemische Veränderung). Darunter fallen z.B. ausgefallene Transistoren. Systematische Fehler treten als Folge eines mangelhaften Entwurfs auf.

Bei der Konstruktion mikroelektronischer Systeme werden einige Informationen über die Fehler e_i benötigt, die in der Struktur des Systems entstehen können und sich möglicherweise auf die Ausgänge auswirken. Diese Informationen gibt das Fehlermodell, die den Satz der Fehlerwahrscheinlichkeiten $P(y, e)$ bestimmt. Die Fehlerwahrscheinlichkeit bestimmt die Wahrscheinlichkeit, dass der Fehler e auf den Ausgang y des logischen Elementes/der Schaltung einwirken wird. Das Fehlermodell beschreibt den logischen Fehler, der durch physikalische Fehler verursacht wird. Welche Fehler wie häufig auftreten, hängt im allgemeinen insbesondere von der zugrunde gelegten Technologie (CMOS, TTL, usw.) und den Einsatzbedingungen der jeweiligen Schaltung ab. Eine ausführliche Beschreibung der Darstellung physikalischer Defekte als logische Defekte wird in [29] gegeben.

Drei wichtige Eigenschaften von Fehlermodellen sind:

- Vereinfachung komplexer physikalischer Fehler;
- Beschreibung verschiedener physikalischer Ursachen;
- Eventuelle Technologieabhängigkeit.

Die Fehlermodellierung für die Simulation und die Testsatzgenerierung sind leicht handhabbar. Die Hauptidee der Fehlermodellierung besteht darin, dass die generierten Testsätze eine Prüfung für die logischen Fehler ermöglichen, die zur Entdeckung physikalischer Defekte führen kann. Und das Ziel hierbei ist es, ein Modell zu entwickeln, das eine möglichst große Anzahl der denkbaren physikalischen Fehler erfasst. Auf diese Weise erleichtert das abstrakte Fehlermodell auf logischem Niveau die Aufgabe der Modellierung einer großen Anzahl möglicher Defekte in den Digitalsystemen und vereinfacht ebenso die Aufgabe ihrer Prüfung. Ein Fehlermodell, das sich ziemlich weit für den industriellen Gebrauch durchgesetzt hat, ist das Haftfehlermodell (*Stuck-at-Fault Model*) [30]. Das ist verständlich, weil die Wahrscheinlichkeit des Auftretens eines Mehrfach-stuck-at-Fehler geringer ist als die Wahrscheinlichkeit des Auftretens eines Einfach-stuck-at-Fehler.

Definition 8:
Ein *stuck-at-1/0 Fehler* (deutsch: Haftfehler bzw. konstante Fehler) liegt vor, wenn eine Leitung (Signal) im Schaltkreis fälschlicherweise immer den logischen Wert 1 (stuck-at-1) bzw. immer den logischen Wert 0 (stuck-at-0) annimmt.

Das *Stuck-at-Fehlermodell* wird bei der Experimentdurchführung in dieser Arbeit verwendet. Im folgenden werden entscheidende Vorteile genannt:

- Implementierung auf logischem Niveau ist einfach;

[8] Defekte auf dem Wafer, Löcher im Gateoxid der Transistoren, Kurzschlüsse zum Beispiel in den metallischen Zuleitungen, Kurzschlüsse bei der Diffusion, abgebrochene Verbindungen, Potentialverschiebungen aufgrund von Verschmutzungen durch bestimmte Ionen, Kurzschlüsströme.

- Gemäßigte Gesamtzahl $2r$ der möglichen stuck-at-1/0 Fehler, wobei r die Anzahl der Verbindungen zwischen den Elementen der Schaltung ist;
- Verhalten einer fehlerhaften Schaltung ist logisch beschreibbar, es kann durch veränderte Boolesche Gleichungen beschrieben werden;
- mit Hilfe dieses Fehlermodells es ist möglich, kompliziertere Fehler zu modellieren und Testsätze für sie zu finden;
- andere in der Synthese digitaler Schaltungen verwendete Fehlermodelle (*stuck-open/bridging faults*) können teilweise durch das Stuck-at-Fehlermodell ersetzt werden;
- Forschungen zeigen auf, dass einfache konstante Fehler etwa 90% der allgemeinen Defektanzahl in industriellen CMOS-Schaltungen einnehmen.

Ein Beispiel eines Stuck-at-Fehlermodells für eine kombinatorische Schaltung f_b mit drei logischen Gattern wird in Abbildung 2.8 gezeigt. Die Struktur enthält eine funktionale Schaltung mit einem stuck-at-0 Fehler a) bzw. einem stuck-at-1 Fehler b) am Eingang x_3. Durch diese zwei Stuck-at-Fehler wird die fehlerfreie Funktion $z = x_1x_2 \vee x_3x_4$ in folgende Funktionen transformiert:

$$z(x_1, x_2, x_3, x_4)^{3/0} = x_1x_2 \text{ und } z(x_1, x_2, x_3, x_4)^{3/1} = x_1x_2 \vee x_4.$$

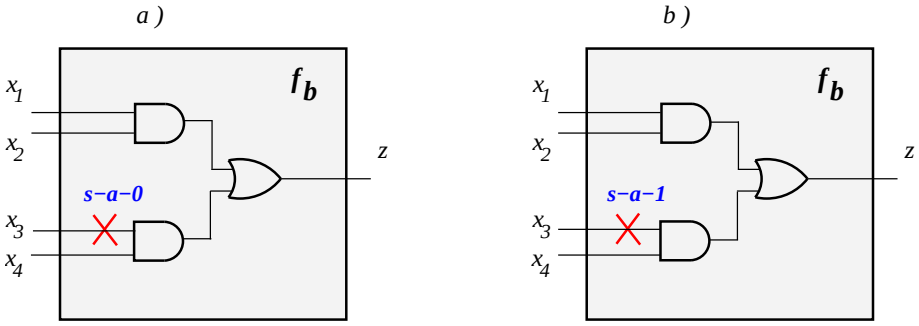

Abbildung 2.8: Stuck-at-Fehlermodell a) stuck-at-$f^{3/0}(x_3)$, b) stuck-at-$f^{3/1}(x_3)$

2.5.2 Selbstprüfende Systeme

Systeme mit der Möglichkeit der internen Prüfung können nach der Qualität der Fehlererkennung klassifiziert werden. Die kombinatorische Schaltung f mit dem internen Fehler φ kann drei Typen der Ausgangssignale generieren:

1. $F(X, \varphi) = F(x)$: für X[9] wird das korrekte Ausgangssignal erzeugt trotz des Fehlers φ. Man sagt, der Fehler φ ist **maskiert** für X;

[9] $X = \{x_1, \cdots, x_n\} \subseteq \{0, 1\}^n$-Menge der Eingangs-Codewörter (*input code space*)

2. $F(X,\varphi) \neq F(x)$ und $F(X,\varphi) \notin Y_{code}$[10]: Bei Auftreten des Fehlers φ wird für X als Ergebnis der Schaltung f ein inkorrektes Ausgangssignal (kein Codewort) generiert, durch das der Fehler erkannt wird. Man sagt, der Fehler φ wird **erkannt** für X.

3. $F(X,\varphi) \neq F(x)$ und $F(X,\varphi) \in Y_{code}$: Bei Auftreten des Fehlers φ wird für X als Ergebnis der Schaltung f ein inkorrektes Codewort generiert, durch das der Fehler nicht erkannt wird; Man sagt, der Fehler φ ist **nicht erkennbar** für X.

Die Aufgabe der Korrektheitskontrolle einer beliebigen kombinatorischen Logik kann gelöst werden mit Hilfe der Implementierung der besonderen Eigenschaft der Selbstprüfung in ein System. Allgemein kann man diese Eigenschaft als die Fähigkeit zur automatischen Verifizierung eines Fehlers sowohl in der funktionalen Logik als auch in der Kontrollschaltung ohne zusätzliche Test-Generierung bezeichnen. Offensichtlich kann ein System nur für einen vermuteten Fehlersatz selbstprüfend sein. Ein solcher Satz enthält die unidirektionalen konstanten Fehler ($F = \varphi_i$, stuck-at-1/0) [4].

Für kombinatorische Schaltungen, die im fehlerfreien Betrieb Codewörter in Codewörter abbilden, ist der Begriff der vollständig selbstprüfenden Schaltung eingeführt worden. Hier werden die drei wichtigsten Bergriffe von [28], [31] für kombinatorische Schaltungen angegeben. Zunächst soll eine binäre kombinatorische Schaltung f betrachtet werden. Die Eingänge und Ausgänge der Schaltung f werden binär kodiert. Jedem Eingang $x \in X$ wird ein n-stelliger Binärvektor $(x_1, \cdots, x_n)$ und jedem Ausgang $y \in Y$ ein m-stelliger Binärvektor $\{y_1, \cdots, y_m\}$ eineindeutig zugeordnet.

Die Schaltung f realisiert die Funktion $F: \{0,1\}^n \rightarrow \{0,1\}^m$. $F(x) = f_1(x_1, \cdots, x_n), \cdots, f_m(x_1, \cdots, x_n)$ ist ein m-Tupel von n-stelligen Booleschen Funktionen. Die Funktion $F(x,\varphi_i)$ ist die Funktion, die beim Vorliegen des Fehlers $\varphi_i \in \Phi$ von der Schaltung f realisiert wird.

Definition 9 [28]:
Eine Schaltung f heißt **selbsttestend** (*self-testing*) bezüglich einer Menge von Fehlern Φ, wenn es für jeden Fehler φ_i aus Φ mindestens ein Input-Codewort gibt, so dass der zugehörige Output der Schaltung kein Codewort ist.

$$(\forall \varphi \in \Phi)\ (\exists X \in X_{code} \mid F(X,\varphi) \notin Y_{code}) \tag{2.4}$$

Definition 10 [31]:
Eine Schaltung f heißt **fehlersicher** (*fault-secure*) bezüglich einer Menge von Fehlern Φ, wenn es für jeden Fehler φ_i aus Φ kein Input-Codewort gibt, so dass die fehlerhafte Schaltung bei Eingabe dieses Input-Codewort ein falsches Codewort ausgibt.

$$(\forall \varphi \in \Phi)\ (\forall X \in X_{code})(F(X,\varphi) = F(X))\ \ oder\ \ (F(X,\varphi) \notin Y_{code}) \tag{2.5}$$

Definition 11 [31]:
Eine Schaltung f heißt **vollständig selbstprüfend** (*Totally Self-Checking / TSC*), wenn sie gleichzeitig **selbsttestend** und **fehlersicher** ist.

Definition 12 [31]:
Eine Schaltung f heißt **codetrennend** (*code-disjoint*), wenn sie die Menge der Eingangs-Codewörter

[10] $Y_{code} = \{y_1, \cdots, y_n\} = \{y \mid y = F(x) \wedge x \in X\}$ - Menge der Ausgang-Codewörter (*output code space*).

auf die Menge der Ausgangscodewörter abbildet und die Menge der Eingangswörter, die keine Codewörter sind, auf eine Menge von Ausgangswörtern abbildet, die ebenfalls keine Codewörter sind.

$$(\forall x \in X_{code})\,(F(X) \in Y_{code}) \;\; und \;\; (\forall x \notin X_{code})\;(F(x) \notin Y_{code}) \tag{2.6}$$

Eine ausführliche Übersicht über Designmethoden selbstprüfender Schaltungen unter Verwendung verschiedener fehlererkennenden Codes kann in [32], [33], [34], [25], [35] gefunden werden.

Ein typisches TSC System besteht aus drei Teilen: der originalen Schaltung, der zusätzlichen Logik und dem TSC Checker. Dabei übernimmt der TSC Checker die Aufgabe eines Beobachters, um eventuelle Fehler im gesamten TSC System zu erkennen, was eine überaus verantwortungsvolle Aufgabe ist. Im nächsten Abschnitt wird deshalb besondere Aufmerksamkeit dem Entwurf von vollständig selbstprüfenden Checkern gewidmet.

2.6 Self-Checking Checkers

Die Konzeption der Konstruktion vollständig selbstprüfender Checker ist in [28] vorgestellt und in [31] formell niedergeschrieben. In diesem Abschnitt sind verschiedene Checkerdesigns für die oben beschriebenen fehlererkennenden Codes dargestellt.

Im Rahmen dieser Arbeit wurden verschiedene Algorithmen für die Logiksynthese der Checker untersucht. Für den Entwurf des Checkers wurden in dieser Dissertation die folgenden Anforderungen zu Grunde gelegt:

1. Die Struktur des Checkers soll die Kontrolle der Eingangskombinationen gewährleisten. Am Ausgang des Checkers entsteht ein Codewort ($z_1 \neq z_2$), wenn die Eingangskombination zur Menge X_{code} der Codewörter gehört. Andernfalls gilt an den Ausgängen $z_1 = z_2$.

2. Die Struktur des Checker soll selbsttestend sein. Für jeden Fehler φ_i aus der Fehlermenge Φ gibt es mindestens ein Eingangs-Codewort, so dass die zugehörigen Ausgänge der Checker gleich sind, also $z_1 = z_2$ (00 oder 11).

Offensichtlich ist der Checker mit den erwähnten Eigenschaften fähig, die Kontrolle einer gesamten Schaltung auf den Vergleich zweier Signale an den Ausgängen zu reduzieren. Als besonderer Fall wird in dieser Arbeit (Abschnitt 2.6.7) ein vollständig selbstprüfender Checker mit nur einem Ausgang betrachtet.

Der Checker entscheidet über die Zugehörigkeit einer erhaltenen Bitfolge zum verwendeten Code. Die Funktionalität des selbstprüfenden Checkers besteht darin, ein Codewort des Codes Y in ein Codewort des 1-aus-2 Code (Abbildung 2.9) zu transformieren.

Die Effektivität des Checkersdesigns wird in der Praxis nach den folgenden Kriterien bestimmt: nach Komplexität (Flächebedarf), nach Schnelligkeit und nach minimaler Anzahl von Testwörtern.

1. Unter der Komplexität eines Checkers wird im allgemeinen die Gesamtanzahl der Eingänge aller logischen Elemente verstanden.

2. Die Schnelligkeit eines Checkers wird nach der Anzahl der Niveaus ausgerechnet. Unter dem Niveau einer beliebigen kombinatorischen Schaltung wird die maximale Anzahl der Gatter, welche die Eingänge mit den Ausgängen verbindet, verstanden.

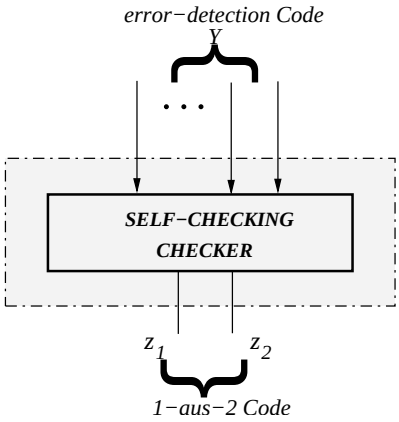

Abbildung 2.9: Checkerübersetzer $Y \Rightarrow 2C1$

3. Das Kriterium der Anzahl der Testvektoren. Die Verringerung der Anzahl der Testvektoren, die für die vollständige Prüfung des Checker notwendig sind, bewirkt, dass die Wahrscheinlichkeit des Entdeckens eines Fehlers erhöht wird.

2.6.1 Checker für Dual-Rail Codes

In [28] wurde eine Struktur für den selbstprüfenden Dual-Rail Checker (*Two-Rail Checker*) entwickelt. Diese war der erste Schritt für die Entwicklung selbstprüfender Systeme. Abbildung 2.10 zeigt diesen allgemein bekannten Checker für den Dual-Rail Code mit zwei Eingangsgruppen $c_1, c_2, c\,'_1, c\,'_2$ ($c_1 = \bar{c}\,'_1, c_2 = \bar{c}\,'_2$) und zwei Ausgängen z_1, z_2. Der Checker verfügt über die Fähigkeit der Kontrolle der

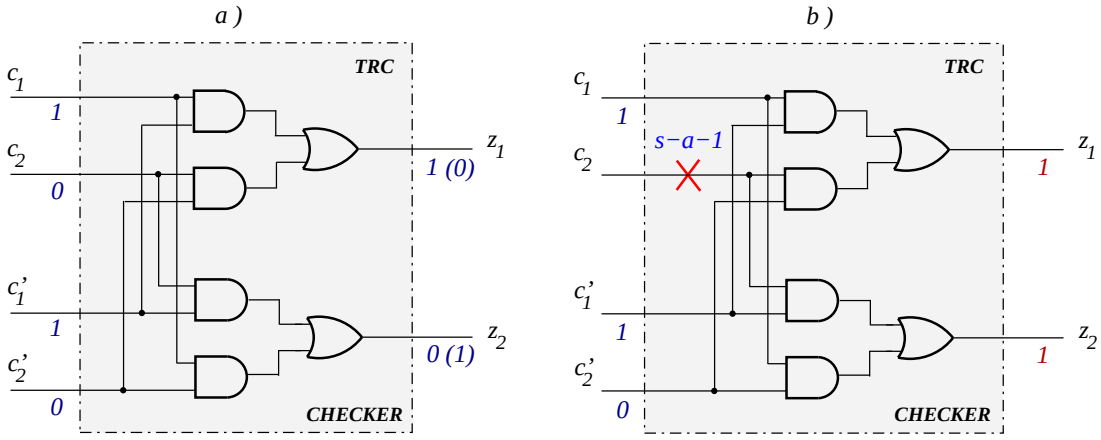

Abbildung 2.10: Ein *Dual-Rail* Checker

Eingangskombinationen durch die Verwendung der Dual-Rail Logik. Bei Abwesenheit von Fehlern, wenn $c_1, c\,'_1 = \{11\}, c_2, c\,'_2 = \{00\}$ (entgegengesetzte, invertierte Werte) wird an den Ausgängen z_1, z_2 des Checker die Kombination $\{01\}$ erzeugt.

Falls der Dual-Rail Comparator fehlerhaft ist, dann nehmen die Ausgänge z_1, z_2 den gleichen Wert an. Bei der Entstehung eines beliebigen Fehlers in der Struktur des Dual-Rail Checkers, werden an den Ausgängen die nicht-Codewörter des 1-aus-2 Codes erzeugt.

An den Eingängen dieses Checkers wird das Anliegen von sechs Codewörtern gewährleistet, was ausreichend für das Entdecken aller konstanten stuck-at-1/0 Fehler ist (Abschnitt 2.5.1). Liegt zum Beispiel auf dem Eingang c_2 ein stuck-at-1 Fehler vor (Abbildung 2.10 b)), so ergibt sich an den Ausgängen z_1, z_2 des Checker das Signal $\{11\}$. Tabelle 2.3 zeigt die Tests für die Schaltung in Abbildung 2.10. Im fehlerfreien Fall sind die Ausgaben $z_1, z_2 = \{01\}$ oder $z_1, z_2 = \{10\}$. Für den Fall, dass die

c_1	c_2	c'_1	c'_2	z_1	z_2
0	1	0	1	0	1
0	1	1	0	1	0
1	0	0	1	1	0
1	0	1	0	0	1

Tabelle 2.3: Testfälle für die Schaltung von Abbildung 2.10

Anzahl n der Eingänge des selbstprüfenden Checkers für den Dual-Rail Code größer als vier ist, kann der Checker als ein „Baum" miteinander verbundener Checker mit vier Eingängen und zwei Ausgängen realisiert werden (siehe Abbildung 2.11).

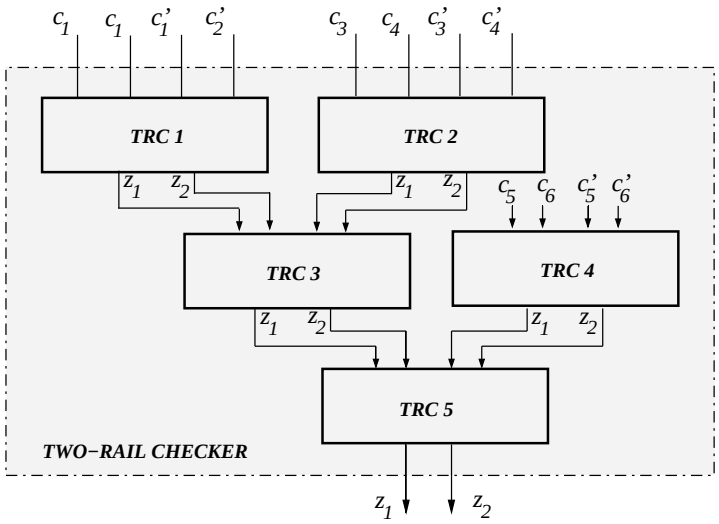

Abbildung 2.11: Baumstruktur eines selbstprüfenden Checkers für den Dual-Rail Code [4]

In dieser Arbeit werden derartige Strukturen oft verwendet. Grundsätzlich wird der Dual-Rail Checker als Endelement in Fehlererkennungsstrukturen, die nach der Methode „Verdopplung und Vergleich" aufgebaut sind, verwendet. Die neue Methode der Logischen Ergänzung wird in allen durchgeführten Experimenten mit der Methode „Verdopplung und Vergleich", welche die einzige Methode mit 100%-iger Fehlerüberdeckung ist, verglichen. Teilweise wird dieser Checker für die Konstruktion des neuen Checkers verwendet, der später beschrieben wird.

2.6.2 Checker für Paritätscodes

Ein wesentlicher Vorteil der Konstruktion selbstprüfender Schaltungen unter Verwendung des Paritätscodes sind die minimalen Kosten für die Realisierung des Checkers. Sie überprüfen die ungerade Anzahl der Einsen in einem Codewort, was durch ein Paritätsbit gewährleistet wird. Als Checker für diese selbstprüfende Schaltungen ist es möglich, das elementare logische XOR-Element zu benutzen (Kapitel 2.4.1). In diesem Fall ist der Checker nicht selbsttestend, da immer einer der Haftfehler des einzigen Ausganges nicht entdeckt wird.

Im weiteren wird ein Beispiel für das Prinzip des Checkersdesigns für den ungerade Paritätscode aufgeführt. Wie zuvor bereits beschrieben wurde, enthält jedes Codewort des ungerade Paritätscodes eine ungeraden Anzahl von Einsen. In diesem Entwurf wurde folgende Besonderheit genutzt: Bei der Teilung eines Codewortes in zwei gleiche Teile werden in einer Worthälfte eine gerade Anzahl von Einsen, in der zweiten Hälfte des Wortes eine ungerade Anzahl von Einsen verfügbar. Für die Konstruktion des Checkers werden beide Codeworthälften benötigt, und jede Hälfte wird durch einen XOR-Baum dargestellt. In einer solchen Checker-Struktur werden an zwei Ausgängen der Schaltung die Ausgaben $\{01\}$ oder $\{10\}$ erzeugt. Im Falle der Verwendung eines geraden Paritätscodes gibt es in der Konstruktion des Checkers einen einzigen Unterschied, und zwar das Invertieren der beiden Baumausgänge.

Eine mögliche Realisierung des Checkers für den ungeraden Paritätscodes ist in der Abbildung 2.12 a) illustriert. In Abbildung 2.12 a) ist ein Checker für den Paritätscode abgebildet, der ebenfalls nicht

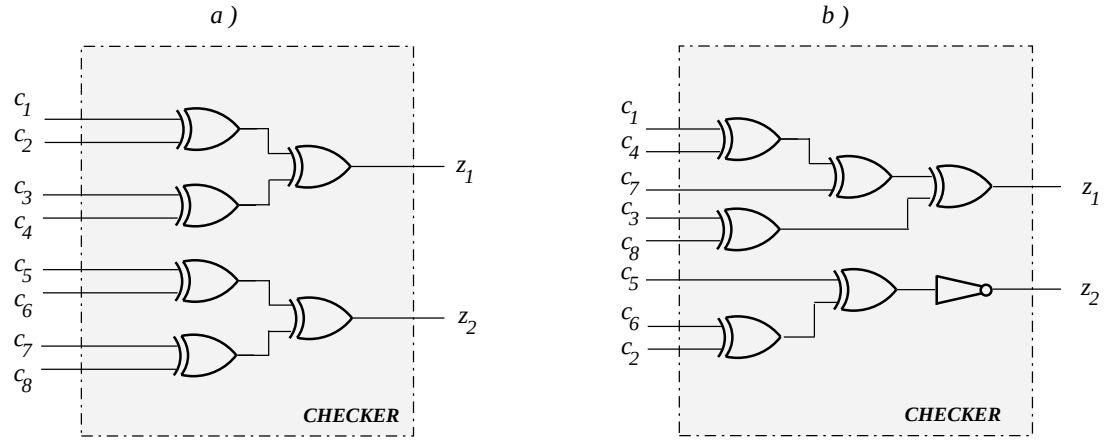

Abbildung 2.12: a) Möglicher Aufbau eines Checkers für den ungeraden Paritätscode, b) Vollständig selbstprüfender Checker für die Paritätscodewörter aus Tabelle 2.4

selbsttestend ist [36]. Das liegt darin begründet, dass nicht alle Wörter eines Testsatzes Eingänge für den Checker darstellen. Ein Nachteil dieses Konstruktionsentwurfs ist die Abhängigkeit der Baumstrukturen von der ursprünglichen Teilung der Codewörter in zwei Teile.

Andererseits existiert eine einfache und wirksame Methode für die Synthese eines vollständig selbstprüfenden Checkers für Paritätscodes. Die Methode ist darauf gegründet, dass für das Entdecken aller Haftfehler vier Testwörter als Eingang für den Checker ausreichend und notwendig sind [37]. Aus diesem Grunde wird der Parity-Checker auf der Grundlage der vorhandenen Codewörter konstruiert.

Der Konstruktionsalgorithmus analysiert die Menge der Paritätscodewörter einer bestimmten Länge n. Falls diese Menge eine "$4 \times n$"-Matrix hat, d.h. sie besteht aus vier Codewörtern des geraden

Paritätscodes, und jede Spalte der Matrix besitzt zwei Nullen und zwei Einsen, so ist es möglich, einen vollständig selbstprüfenden Checker zu konstruieren. Ein Beispiel einer "4×8"-Paritätscodematrix zeigt Tabelle 2.4.

inputs	c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8
$4 \setminus 8$	1	2	3	4	5	6	7	8
1	0	1	0	1	1	0	0	1
2	1	1	0	1	1	1	1	0
3	0	0	1	0	0	0	0	1
4	1	0	1	0	0	1	1	0

Tabelle 2.4: "4×8"-Matrix von Paritätscodewörtern

In Abbildung 2.12 b) ist ein vollständig selbstprüfender Checker für die Paritätscodewörter aus Tabelle 2.4 dargestellt. Einen Überblick über verschiedene Standardverfahren für die Synthese des Checkers für Paritätscodes findet man beispielweise in [38] oder [39].

2.6.3 Entwurf vollständig selbstprüfender Checker für Berger-Codes

In diesem Abschnitt werden grundlegende Verfahren und Algorithmen zur Konstruktion vollständig selbstprüfender Checker für die nächste Klasse fehlererkennender Codes, die in dieser Arbeit verwendet werden, namentlich Berger-Codes, beschrieben.

In Abbildung 2.13 ist das traditionelle Schema für den Entwurf vollständig selbstprüfender Checker für separierbare Codes gezeigt [31]. Die prinzipielle Struktur besteht aus zwei funktionalen Teilen: dem Kontrollbitgenerator N_1 und dem Comparator N_2.

Die Anzahl k der Ausgänge des Kontrollbitgenerators entspricht der Anzahl der Kontrollbits. Die Anzahl der Eingänge des Comparators entspricht der doppelten Anzahl der Kontrollbits. Der Comparator im Prüfteil des Systems führt den Vergleich $\{c_1, \cdots, c_k\}$ und $\{c'_1, \cdots, c'_k\}$ durch. Ein nach der aufgezeigten Struktur konstruierter Checker wird oft als *normaler Checker* bezeichnet und verfügt über die erwähnten Anforderungen an selbstprüfbare Checker.

In dieser Arbeit werden zwei Arten des Checkersdesigns für Berger-Codes auf mögliche und effektive Verwendung analysiert. Die Ursache hierfür liegt in der Tatsache, dass zwei Arten von Kontrollbit-Generatoren für Berger-Codes existieren. Diese beiden möglichen Designs sollen hier untersucht werden. Ein selbstprüfender Berger-Code Checker wird als kombinatorische Schaltung aufgebaut.

2.6.4 Synthese des Kontrollbitgenerators für Berger-Codes

Der Generator ist eine kombinatorische Schaltung, die die Anzahl der Einsen bzw. der Nullen an ihren Eingängen zählt. An den Ausgängen des Generators wird das binäre Äquivalent dieser Anzahl ausgegeben. Aus r Informationsbits erzeugt der Generator k Kontrollbits.

An den Ausgängen des Generators ist die direkte Realisierung der Kontrollbits des Codes möglich [40], oder aber auch die Realisation von Hilfskontrollbits, die jedoch anschließend invertiert werden [21]. Der vorliegende Umstand bestimmt das Vorhandensein zweier Hauptarten der Struktur und des Algorithmus der Generatorkonstruktion. Die zweite Art der Synthese des Generatoren ist zweckmäßi-

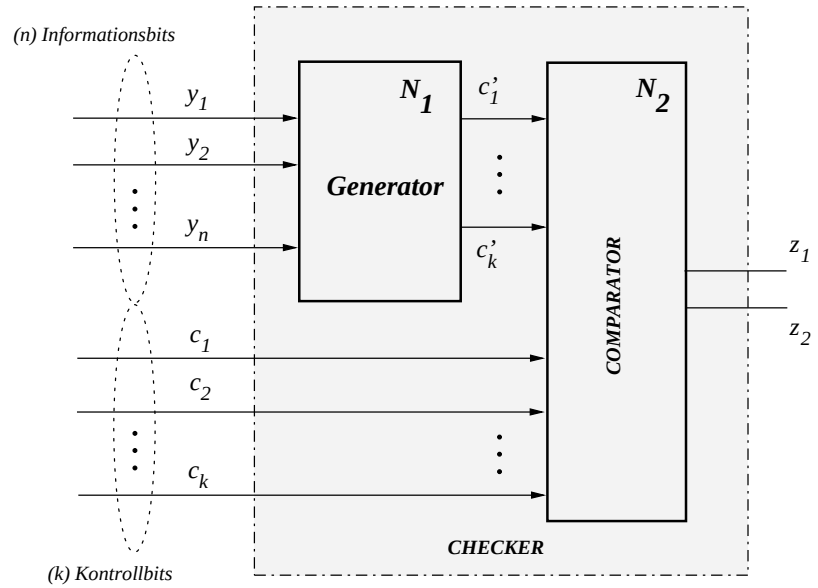

Abbildung 2.13: Vollständig selbstprüfender Checker für separierbare Codes

ger, da die Berechnung der Hilfskontrollbits direkt durch die Benutzung vollständiger Addierer (*full adders*) und Halbaddierer (*half adders*) erfolgen kann.

Ebenso ist auch die getrennte Realisierung des Kontrollbitgeneratoren für den Berger-Code bekannt [41], [42], [43]. In diesem Fall wird ein abgeänderter Berger-Code verwendet, in dem die Anzahl der Kontrollbits verringert ist (Abschnitt 2.4.2). Die getrennte Realisierung führt jedoch zu einer komplizierteren Struktur und erfordert mehr Flächenaufwand.

In dieser Arbeit ist die effektivere Methode zur Konstruktion des Generators praktisch realisiert [35]. Im Vergleich zu anderen Methoden erfordert die Generatorrealisierung mit dieser Methode einen geringeren Hardware-Aufwand. Weiter werden in der Arbeit zwei Algorithmen zur Konstruktion von Kontrollbitgeneratoren für den Berger-Code vorgestellt. Die Konstruktionsalgorithmen und die Strukturen der Kontrollbitgeneratoren hängen von der Länge des verwendeten Berger-Codes ab. Berger-Codes werden nach der Anzahl der Kontrollbits in „voll" (maximale Länge: $n = 2^k - 1$) und „modifiziert" (nicht maximale Länge: $n \neq 2^k - 1$) eingeteilt. In dieser Arbeit sind die Kontrollbitgeneratoren nach der zweiten Konstruktionsvariante der Konstruktion mit vollständigen Addierern und Halbaddierern verwirklicht. Ein Halbaddierer ist eine kombinatorische Schaltung und ebenfalls ein paralleler Zähler (Abbildung 2.14 a)), dessen Eingänge zwei einstellige binäre Zahlen x_1 und x_2 sind und der als Ergebnis die Summe Sum und den Übertrag c_{out} liefert. Ein Halbaddierer hat zwei Eingänge x_1 und x_2 und zwei Ausgänge Sum, c_{out} die die folgenden Funktionen realisieren:

$$Sum = x_1\bar{x}_2 \vee \bar{x}_1x_2 = x_1 \oplus x_2,$$
$$c_{out} = x_1x_2. \tag{2.7}$$

Ein vollständiger Addierer (Abbildung 2.14 b)), ist ein paralleler Zähler, der die logische Addition dreier einstelliger binärer Zahlen (d.h. drei Bits) gewährleistet. Der vollständige Addierer hat drei Eingänge

x_1, x_2, c_{in} und zwei Ausgänge *Sum*, c_{out}:

$$Sum = x_1 \oplus x_2 \oplus c_{in},$$
$$c_{out} = (x_1 x_2) \vee ((x_1 \oplus x_2) c_{in}). \tag{2.8}$$

In Tabelle 2.6 ist eine Wertetabelle (Wahrheitstafel) eines vollständigen Addierer und eines Halbaddierers gegeben.

Die Klassifikation des Berger-Codes bestimmt die Existenz zweier Algorithmen für die Generatorkonstruktion. Im folgenden wird ein Algorithmus für die Konstruktion eines Generators, der die Kontrollstellen eines Berger-Codes maximaler Länge erzeugt, gegeben.

Algorithmus 2.1:

1. Die Menge der Eingangskombinationen $B = \{x_1, x_2, \cdots, x_{I_a}\}$ ($I_a = 2^{P_a} - 1$) wird in drei Untermengen: $B_1 = \{x_1, x_2, \cdots, x_z\}$, $B_2 = \{x_{z+1}, \cdots, x_{I_a - 1}\}$ und $B_3 = \{x_{I_a}\}$ ($z = 2^{P_a - 1} - 1$) aufgeteilt.

2. Die Elemente der Mengen B_1 und B_2 sind die Eingänge der zwei entsprechenden Schaltungen A_1 und A_2. Diese Schaltungen werden als Kontrollbitgeneratoren für einen Code $(z + y)\ Sz$ verwendet, wobei $y = [\log_2(z + 1)]$ die Anzahl der Ausgänge einer Schaltung bestimmt (An den Ausgängen y_1, y_2, y_3 der Schaltungen A_1, A_2 wird die binäre Darstellung der Einsenanzahl erzeugt);

3. Die r-Bitkombinationen an den Ausgängen der Schaltungen A_1 bzw. A_2 werden mit ω_1 und ω_2 bezeichnet, die Einbitkombination am Eingang x_{I_a} mit ω_3;

4. Mit Hilfe der y vollständigen Addierer (FA) wird eine Schaltung für die Summierung der binären Zahlen ω_1, ω_2 und ω_3 gebildet;

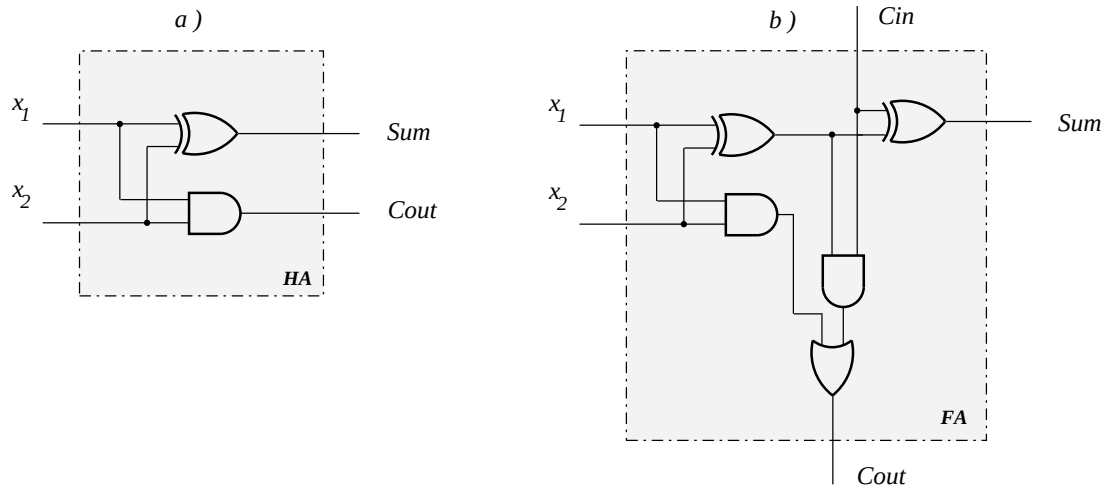

Abbildung 2.14: a) vollständiger Addierer, b) Halbaddierer

5. Wenn $P_a = 3$, dann werden die Schaltungen A_1 und A_2 als FA-Module aufgestellt. Wenn $P_a > 3$, dann werden die Schaltungen A_1 und A_2 mittels Wiederholung der Punkte 1-4 verwirklicht.

x_3	x_2	x_1	Sum	c_{out}
0	**0**	**0**	**0**	**0**
0	**0**	**1**	**1**	**0**
0	**1**	**0**	**1**	**0**
0	**1**	**1**	**0**	**1**
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Tabelle 2.5: Wertetabelle eines vollständigen Addierers und eines Halbaddierers (fett)

Beispiel:
In Abbildung 2.15 ist ein Kontrollbitgenerator für einen Berger-Code mit maximaler Länge 19S15 ($P_a = 4$) zu sehen.

- Da $z = 2^{4-1} - 1 = 7$, $y = [\log_2(7+1)] = 3$ gilt, sind die Schaltungen A_1 und A_2 vollständig identisch und jede Schaltung hat drei Ausgänge y_1, y_2, y_3;
- Die Menge der Eingangsvariablen $B = x_1, \cdots, x_{15}$ ist in drei Untermengen geteilt: $B_1 = \{x_1, \cdots, x_7\}$, $B_2 = \{x_8, \cdots, x_{14}\}$, $B_3 = \{x_{15}\}$;
- An den Ausgängen y'_1, y'_2, y'_3 des Blockes A_1 wird die binäre Zahl ω_1 gebildet, die der binären Darstellung der Einsenanzahl der Eingänge $\{x_1, \cdots, x_7\}$ entspricht;
- An den Ausgängen y''_1, y''_2, y''_3 des Blockes A_2 wird die binäre Zahl ω_2 gebildet (Einsenanzahl der Eingänge $\{x_7, \cdots, x_{14}\}$), und an dem Ausgang x_{15} wird die Ein-Bit Zahl ω_3 gebildet;
- Mit Hilfe dieser drei FA-Module ($y = 3$) wird eine Schaltung für die Addition der binären Zahlen ω_1, ω_2 und ω_3 aufgebaut.

Für die Prüfung des Generators sind acht Eingangskombinationen notwendig und hinreichend. Dabei wird jeder einzelne stuck-at-1/0 Fehler an den Eingängen und Ausgängen der FA- und HA-Module gefunden. Es werden ebenso beliebige Kombinationen einzelner interner stuck-at-1/0 Fehler innerhalb dieser Module entdeckt [35].

Für die unvollständigen Berger-Codes (nicht mit maximaler Länge) verändert sich der Algorithmus 2.1. für die Konstruktion des Generators auf folgende Weise.

Algorithmus 2.2:

1. Die Menge der Eingangskombinationen $B = (x_1, \cdots, x_{I_a})$ ist in q ($q \leq P_a - 1$) Untermengen $B_1 = \{x_1, \cdots, x_{t_1}\}$, $B_2 = \{x_{t_1+1}, \cdots, x_{t_2}\}$, $\cdots$, $B_{q-1} = \{x_{t_{q-2}+1}, \cdots, x_{t_{q-1}}\}$, $B_q = \{x_{t_q-1}, x_{t_q}\}$ so aufgeteilt, dass jede Menge B_i ($i \in \{1, 2, \cdots, q-1\}$) Elemente $2^{P_a - i} - 1$ enthält. Die Untermenge B_i besteht aus $b_q = I_A - \sum_{i=1}^{q-1} b_i$ Elementen, wobei $b_q \in \{0, 1, 2\}$ ist;

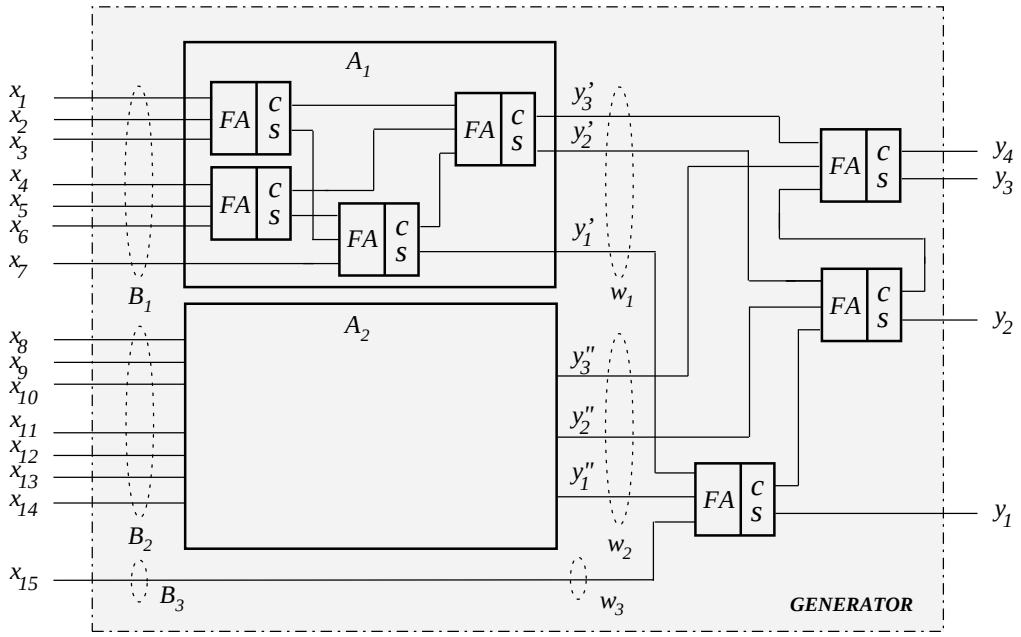

Abbildung 2.15: Kontrollbitgenerator für den Berger-Code 19S15 (mit maximaler Länge)

2. Die Elemente jeder Untermenge B_i $(i \in \{1, 2, \cdots, q-1\})$ entsprechen den Eingängen der Schaltung A_1, die mit Hilfe des Algorithmus 2.1. konstruiert wird;

3. Mit ω_i werden die Binärvektoren an den Ausgängen der Schaltung A_i bezeichnet; $\omega\,'$, $\omega\,''$ bezeichnen die einstelligen Binärvektoren an den Ausgängen der Schaltung A_2, die den $x \in B_q$ entsprechen;

4. Mit Hilfe der Module FA und HA wird eine Schaltung für die Summierung der binären Zahlen $\omega_1, \omega_2, \cdots, \omega_{q-1}, \omega\,'$ und $\omega\,''$ konstruiert.

Beispiel:
In Abbildung 2.16 ist ein Bespiel eines Kontrollbitgenerators für den Berger-Code mit nicht maximaler Länge 16S12 dargestellt.

1. Die Menge der Eingangsvariablen $B = (x_1, \cdots, x_{12})$ ist in drei Untermengen eingeteilt: $B_1 = \{x_1, \cdots, x_7\}$, $B_2 = \{x_8, x_9, x_{10}\}$, $B_3 = \{x_{11}, x_{12}\}$;

2. Die Schaltung A_1 wird mit Hilfe des Algorithmus 2.1. aufgebaut;

3. Für diesen Code hat die Untermenge B_2 drei Variablen, aus diesem Grund stellt die Schaltung A_2 einen vollständigen Addierer dar;

4. An den Ausgängen $y\,'_1, y\,'_2, y\,'_3$ der Schaltung A_1 wird der Drei-Bit-Vektor ω_1 gebildet;

5. An den Ausgängen $y\,''_1 = S$ und $y\,''_2 = C$ der Schaltung A_2 wird der Zwei-Bit-Vektor ω_2 gebildet;

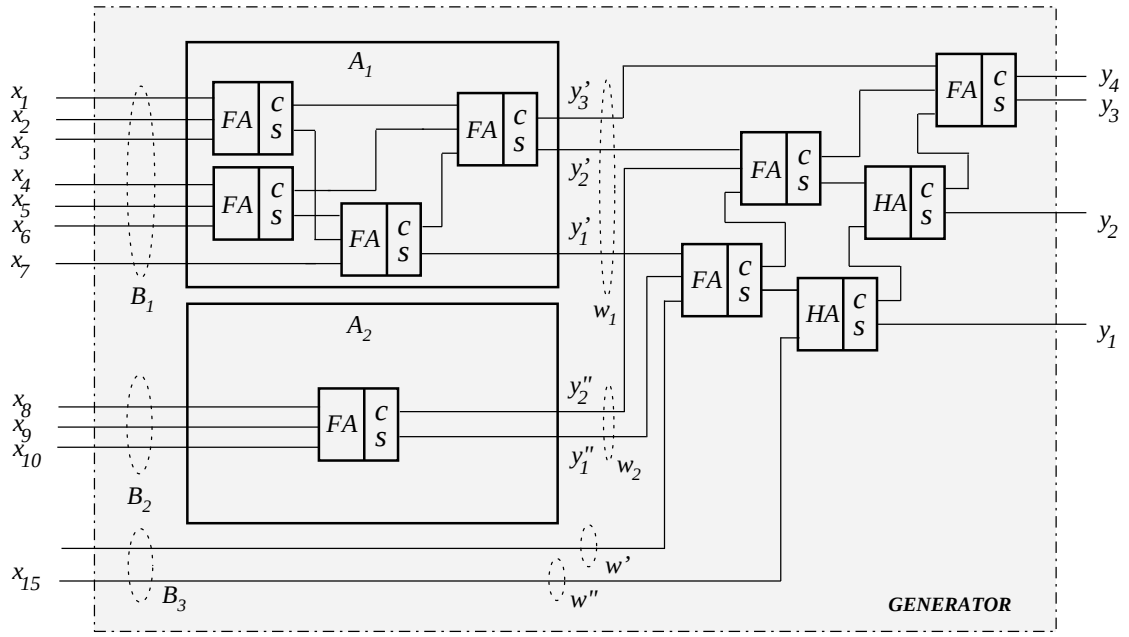

Abbildung 2.16: Kontrollbitgenerator für den Berger-Code 16S12 (mit nicht maximaler Länge)

6. An den Ausgängen x_{11} und x_{12} werden die Ein-Bit-Vektoren ω' und ω'' gebildet;
7. Mit Hilfe der drei FA-Module und der zwei HA-Module wird eine Schaltung für die Summierung der binären Zahlen $\omega_1, \omega_2, \omega'$ und ω'' aufgebaut;
8. Für die Prüfung des Generators sind acht Eingangskombinationen notwendig und hinreichend.

In [21] und [35] wurde bewiesen, dass die Generatoren, welche mit Hilfe der Algorithmus 2.1. und Algorithmus 2.2. aufgebaut werden, vollständig selbstprüfende Schaltungen sind. Es existiert ein zweiter wenig bekannter Algorithmus zur Konstruktion des Generators. Der folgende Algorithmus basiert auf den Arbeiten [35], [44] und wird in dem vorliegenden Abschnitt kurz beschrieben.

Algorithmus 2.3:

1. Die Funktionen, die der Generator an den Ausgängen verwirklicht, werden mit $S_0, S_1, \cdots, S_{P_a-1}$ bezeichnet. Dabei entsprechen die Funktionen S_0 und S_{P_a-1} dem niederwertigsten und dem hochwertigsten Bit eines Kontrollwortes;
2. Berechne die Ausgangsfunktionen S_i ($i \in \{0, 1, \cdots, r-1\}$) des Generators nach der Formel:

$$S_i = f^{m_1}\overline{f^{m_1+2^i}} \vee f^{m_2}\overline{f^{m_2+2^i}} \vee \ldots \vee f^{a}\overline{f^{b}}, \tag{2.9}$$

wobei f^m eine einfache symmetrische Funktion $f^m(x_1, x_2, \cdots, x_n)$ ist

$f^m(x_1, x_2, \cdots, x_n) = f^m(x_1, \cdots, x_n) = \bigvee_{i_1,i_2,\cdots,i_m \in \{1,2,\cdots,n\}} x_{i_1}, x_{i_2}, \cdots, x_{i_m}$[11], wenn

[11] Beispiel: Für den 4S2 Code lautet die Funktion $f^m = f^2(x_1, x_2, x_3, x_4) = x_1x_2 \vee x_1x_3 \vee x_1x_4 \vee x_2x_3 \vee x_2x_4 \vee x_3x_4$.

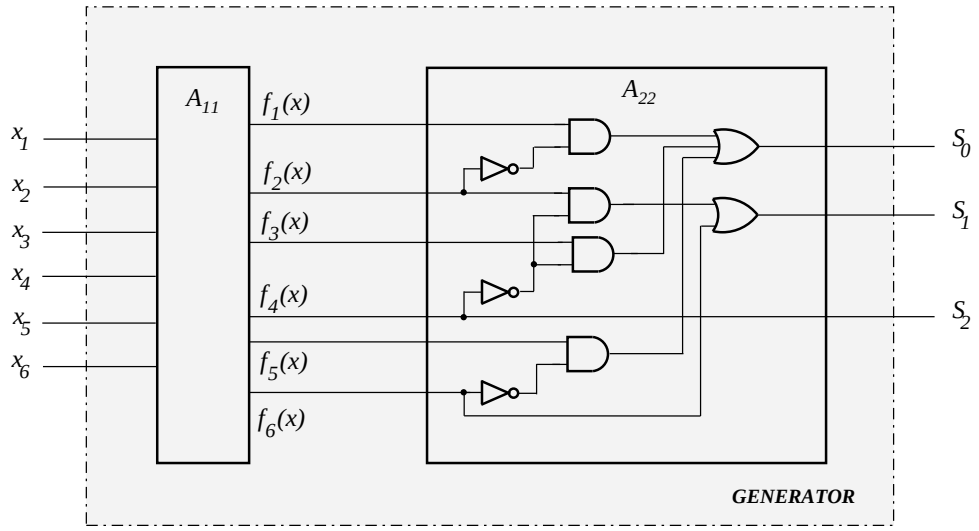

Abbildung 2.17: Kontrollbitgenerator für den Berger-Code $S6$

$m_1 = 2^i, m_{j+1} = m_j + 2^{i+1} (j \in \{1, 2, 3, \cdots\})$, $a = p' \cdot 2^{i+1} + 2^i$ und $p' =]p/2^{i+1}[$ ist.

Aus der Formel 2.9 folgt, dass der Generator als eine kombinatorische Schaltung, die aus zwei miteinander verbundenen Teilschaltungen besteht, verwirklicht werden kann. Die erste Teilschaltung wird mit A_1 bezeichnet, realisiert das System der einfachen symmetrischen Funktionen $f^m(x_1, x_2, \cdots, x_k)$ $(m \in \{1, 2, \cdots, k\})$ und stellt den logischen Transformator dar. Die zweite Teilschaltung A_2 realisiert das Funktionssystem aus Formel 2.9. Im Unterschied zu Schaltung A_2, die mit Hilfe der zweistufigen Implementierung realisiert wird, kann die Schaltung drei Realisationsvarianten haben: zweistufige Implementierung, iterative Struktur oder Verwendung der minimalen Funktionsdarstellung [35].

Beispiel:
In Abbildung 2.17 ist ein Kontrollbitgenerator für den Berger-Code 9S6 dargestellt. Die Schaltung A_1 ist eine zweistufige Schaltung, welche die folgenden einfachen symmetrischen Funktionen realisiert:
$f^1(x_1 \div x_6) = x_1 \vee x_2 \vee x_3 \vee x_4 \vee x_5 \vee x_6$,
$f^2(x_1 \div x_6) = x_1x_2 \vee x_1x_3 \vee x_1x_4 \vee x_1x_5 \vee \cdots \vee x_5x_6$,
$f^3(x_1 \div x_6) = x_1x_2x_3 \vee x_1x_2x_4 \vee x_1x_2x_5 \vee \cdots \vee x_4x_5x_6$,
$f^4(x_1 \div x_6) = x_1x_2x_3x_4 \vee \cdots \vee x_3x_4x_5x_6$,
$f^5(x_1 \div x_6) = x_1x_2x_3x_4x_5 \vee \cdots \vee x_2x_3x_4x_5x_6$,
$f^6(x_1 \div x_6) = x_1x_2x_3x_4x_5x_6$.

In der Generatorstruktur hat die zweite Schaltung A_2 drei Ausgänge, die die Funktionen aus der Formel 2.9 verwirklichen. An jedem zweiten Eingang der Schaltung A_2 ist ein Negator installiert. In [35] ist bewiesen, dass die Ausgangskombinationen, die an den Ausgängen der Schaltung A_1 entstehen, eine Testfolge für die Überwachung der Schaltung A_2 bilden. Dieser Kontrollbitgenerator ist vollständig selbstprüfend. Neben der vorgestellten Methode der Konstruktion des Generatoren existieren noch andere Konstruktionsmethoden. In [43] wird eine Methode auf Grundlage der modularen Benutzung

der Checker für m-aus-n Codes dargestellt. Es ist ebenso möglich, eine Methode zu wählen, die auf geteilter Generierung des Berger-Codes basiert [41].

2.6.5 Entwurf eines vollständig selbstprüfenden Checkers für nicht-separierbare Codes

M-aus-n Codes sind nicht-separierbare Codes, und diese m-aus-n Codes werden nicht nur dazu verwendet, um in Leitungen einzelne Fehler sondern auch alle unidirektionalen Fehler zu entdecken. Ein Code mit der Wortlänge m, dessen Codewörter alle n Einsen besitzen, heißt m-aus-n Code.

Der Checker für m-aus-n Codes erfüllt alle jenen Funktionen, die schon früher für andere Codes beschrieben wurden. Wenn m-aus-n Codewörter an den Eingängen der Checker anliegen, werden an seinen Ausgängen die Ausgangskombination $\{01\}$ oder $\{10\}$ erzeugt.

Unter den ersten Methoden, die für den Entwurf selbstprüfender Checker für m-aus-n Codes entwickelt wurden, war unter anderem die modulare Verbindung einiger Codeübersetzer (*code translator*). Diese Methode ist in [31] beschrieben. Zu den Vorteilen der modularen Methode, die auf der Code-Translation gegründet ist, gehört die Einfachheit der Realisierung dieser Methode. Der vollständig selbstprüfende Checker, vorgestellt in [31] besteht aus drei Codeübersetzern: m/n in 1/($\frac{n}{m}$); 1/($\frac{n}{m}$) in k/2k und k/2k in 1/2.

Als alternative Designmethode eines vollständig selbstprüfenden Checkers für m-aus-n Codes, soll nun die in [21] beschriebene Methode betrachtet werden. Das Konstruktionsprinzip und die Anzahl der Teilschaltungen ändert sich nicht, es ändert sich nur die Art der notwendigen Codeübersetzer: m/n in 1/s (s=4,5,6); 1/s in 2/4; 2/4 in 1/2, wobei $s = 4$ für m/2m+1 Codes, $s = 5$ für 2/n Codes, $s = 6$ für andere m-aus-n Codes.

Es existieren Methoden, die eine verringerte Menge an Codeübersetzern (Niveau) für die Transformation eines m-aus-n Codes in einen 1-aus-2 Code verwenden. So ist zum Beispiel in [45] eine Designmethode vorgestellt, in der der konstruierte vollständig selbstprüfende Checker nur zwei Teilschaltungen besitzt: zum einen findet eine Übersetzung von m-aus-n Codewörter in 1/s-Codewörter und zum anderen 1/s-Codewörter in 1-aus-2 Codewörter statt. In [46] wird ein Checker vorgestellt, der aus nur einem Übersetzer besteht, welcher m-aus-n Codewörter in 2-aus-4 Codewörter umwandelt. Dieser Checker ist für den 2-aus-4 Code vollständig selbstprüfend.

Die aufgeführten Methoden verwenden die bekannten „*majority*"-Schaltungen (siehe Formel 2.8) für die Einsenzählung an ihren Eingängen. Sie erfordern bedeutende Hardwarekosten, die proportional zu den Parametern ($\frac{m}{n}$) des Codes wachsen. Eine Verringerung der Hardwarekosten ist bei Benutzung untereinander verbundener Addierer, Halbaddierer und selbstprüfender Dual-Rail Checker möglich. In [47], [48], [49] werden Checker für m-aus-n Codes vorgestellt. Die Modifikation dieses Algorithmus für die Fälle $n \neq 2m$ ist in [50] ausführlich beschrieben. Leider ist eine vollständige Betrachtung aller Designmethoden für m-aus-n Codes schwer möglich, da eine große Anzahl dieser Methoden existiert, was über das Thema dieser Arbeit hinausgeht. Deshalb werden in diesem Teil der Dissertation nur die Hauptmethoden analysiert, die in der Praxis effektiv verwendet werden können.

Die Forschungen in dieser Arbeit sind auf die Konstruktion eines effektiven Algorithmus für die Synthese selbstprüfender Checker für eine spezielle Unterklasse der m-aus-n Codes, die 1-aus-n Codes gerichtet. Der Hauptgrund ist die breite praktische Benutzung dieser Codes. Ein bekanntes Anwendungsbeispiel der 1-aus-n Codes ist ihre Verwendung an den Ausgangsleitungen der Adressdecoder des RAM & ROM (*adress decoders outputs*) [51], für die Konstruktion der Checkers für m-aus-n Codes. In [31] wird aufgezeigt, dass der Checker für 1-aus-n Codes wie der selbstprüfende Translator, der

1-aus-n Codes in k-aus-2k Codes übersetzt, aufgebaut werden kann. Jedoch verfügt ein auf diese Weise konstruierter Checker für 1-aus-3 und 1-aus-7 Codes nicht über die Selbstprüfungseigenschaft, da ein solcher Checker nicht die notwendige Anzahl an Eingangswortkombinationen besitzt (weniger als 2^n).

In dieser Arbeit wird ein neues Verfahren beschrieben, das durch den Entwurf eines neuen Checkers für eine besondere Unterklasse der m-aus-n der Codes (mit $m = 1$ und $n = 3$), die Nachteile der beschriebenen Methoden vermeidet. Der vollständig selbstprüfende Checker für den 1-aus-3 Code wird in dem folgenden Abschnitt vorgestellt.

2.6.6 Neuer selbstprüfender Checker für den 1-aus-3 Code

Im Gegensatz zu einer großen Anzahl an Arbeiten, die sich mit der Konstruktion von Checkern für den m-aus-n Codes beschäftigen, sind die Forschungen auf dem Gebiet der Checkerkonstruktion für den 1-aus-3 Code nicht so zahlreich. In [52] wurde bewiesen, dass ein selbstprüfender Checker für den 1-aus-3 Code, der auf der Basis der Elemente AND/OR/NOT aufgebaut ist, nicht existiert. Dieses Problem wird damit erklärt, dass ein vollständiger Test für diesen Checker nicht existiert. Die Anzahl der 1-aus-3 Eingabewörter ist nicht groß genug für einen vollständigen Test des Checkers. In [35] ist eine der notwendigen Bedingungen für die Existenz eines Checker für einen m-aus-n Code ausführlich beschrieben.

Theorem 1 *Für den Test eines beliebigen selbstprüfenden m-aus-n Checkers, der als eine kombinatorische Schaltung realisiert ist, wird ein Test der Länge $t \geq 4$ benötigt.*

Als alternative Methodik der Checkerkonstruktion wird die Transformation des 1-aus-3 in einen anderen Code verwendet, für den ein selbstprüfender Checker existiert. Dieses Prinzip ist die Grundlage der sogenannten Methoden der „indirekten Synthese" für die Konstruktion von 1-aus-3 Checkern [53], [54]. Die Prüfung des 1-aus-3 Codes ist beispielsweise mit Hilfe einer kombinatorischen Schaltung in der Verbindung mit der Prüfung einiger anderen Codes mit konstantem Gewicht (m-aus-n) möglich. In diesen zwei Arbeiten wird der 1-aus-3 Code in m-aus-n Codes transformiert. In [54] wird eine Lösung für die Konvertierung eines 1-aus-3 Code in einen 1-aus-4 Code mit Hilfe einer sequentiellen Struktur vorgestellt.

Eine zweite mögliche Lösung der Aufgabe der Checkerkonstruktion für den betrachteten Code ist die Implementierung des Checker auf dem Transistorniveau. Beispiele solcher Checker auf Transistorbasis sind in den Arbeiten [55], [56] aufgeführt. Der Checker aus [55] stellt einen 1-aus-3 Checker vor, der aus 17 Transistoren besteht. Dieser Checker ist vollständig selbstprüfend für alle modellierten Fehler. Die Charakteristik eines beliebigen Checkers kann durch Parameter (Abschnitt 2.6) wie Schnelligkeit, Hardware-Bedarf und Anzahl der Niveaus, u.s.w. beschrieben werden. So hat zum Beispiel der Checker für 1-aus-3 Codes aus [56] 11 Transistoren, was sich natürlich auf alle Parameter auswirkt. Die dritte mögliche Variante für das Checker-Design wird in einer großen Anzahl von Arbeiten beschrieben. Hier wird der Checker für 1-aus-3 Codes auf Grundlage der NMOS/CMOS Technologie konstruiert [55], [57], [58], [59], [60].

In der Praxis ist es schwer, die Selbstprüfungsfähigkeit eines beliebigen Gerätes zu gewährleisten ohne eine bedeutende Vergrößerung der Fläche des Gerätes, ohne Erhöhung der Anzahl der zusätzlichen Elemente und ohne Erhöhung der Funktionsausführungszeit. In [56] wird der Eigenschaftsbegriff der „partionally code-disjoint" eingeführt. Diese Eigenschaft wird unter anderem benutzt für die Checkerkonstruktion, was eine kostengünstige Realisation ermöglicht. Von besonderem Interesse ist hierbei die

Möglichkeit der Realisation eines *partiell* selbstprüfenden Checkers für 1-aus-3 Codes. Diese Eigenschaft basiert auf der folgenden Idee: Wenn bei Auftreten eines inkorrekten Code-Worts des Gewichtes k der Checker nicht codetrennend, jedoch fehlersicher für ein inkorrektes Codewort des Gewichtes m (mit $m < k$) ist, so ist es möglich, den *k-fault tolerant partionally strongly fault secure* Checker zu konstruieren. Diese Abschwächung an die Anforderungen wird verwendet für den 1-aus-3 Checker, welcher *strongly fault-secure* und *partionally strongly code-disjoint* ist [61]. Eine umfassende Beschreibung der Eigenschaften *partionally strongly code-disjoint* und *partionally strongly fault-secure* findet sich in [62].

In dieser Arbeit wird ein neuer kombinatorischer Checker für den 1-aus-3 Code, der in [63] entwickelt wurde, vorgestellt. Der Checker hat alle notwendigen Parameter, die ein selbstprüfendes Gerät benötigt, um alle möglichen Stuck-at-Fehler zu erkennen. Für die Synthese des Checkers wird das modulare Prinzip verwendet. Hauptvorteil der Benutzung und Grund für die Wahl der modularen Checkerkonstruktion ist die Universalität mit der Möglichkeit der gleichzeitigen Benutzung mehrerer Codes.

Wie bereits in Abschnitt 2.6 beschrieben, können Checker untereinander nach drei Hauptcharakteristiken verglichen werden. Im Rahmen der Dissertation wurde nach einem bekannten Checker mit sehr guten Parametern gesucht, der als Vergleichsobjekt für den neuen Checker dienen soll. In der Arbeit von Paschalis et al. [64] wird ein Checker für den 1-aus-3 Code vorgeschlagen, der nach dem modularen Prinzip aufgebaut wird. Die Anzahl der funktionalen Blöcke für die Konstruktion des Checkers ist gleich zwei. Dieser Checker von [64] verfügt über eine ausgezeichnete Charakteristik. So ist bei-

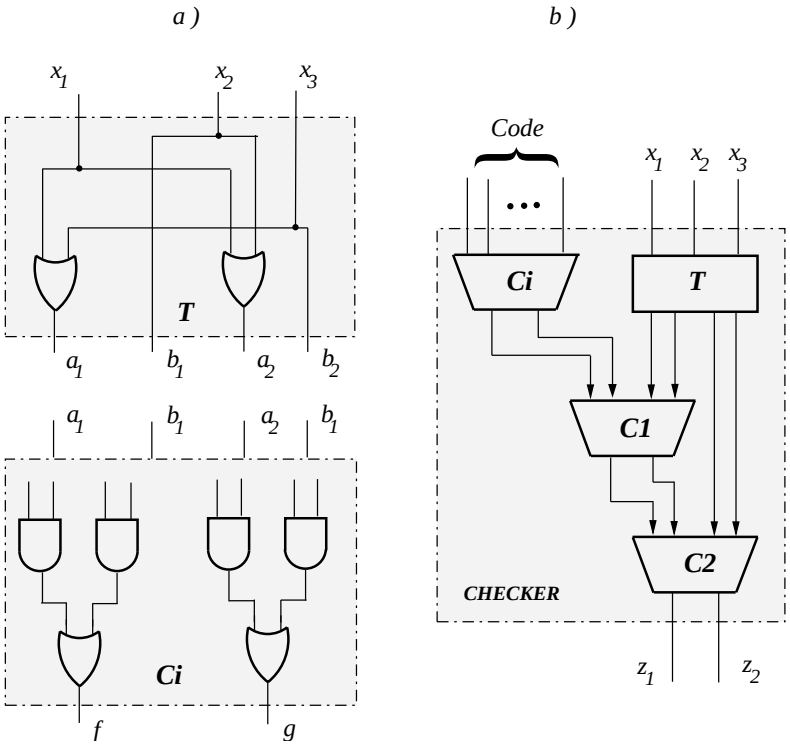

Abbildung 2.18: Funktionale Teilschaltungen für den 1-aus-3 Code Checker von [64]

spielsweise die Fläche des 1-aus-3 Code Checkers um 47% geringer als die Checkerfläche von [53].

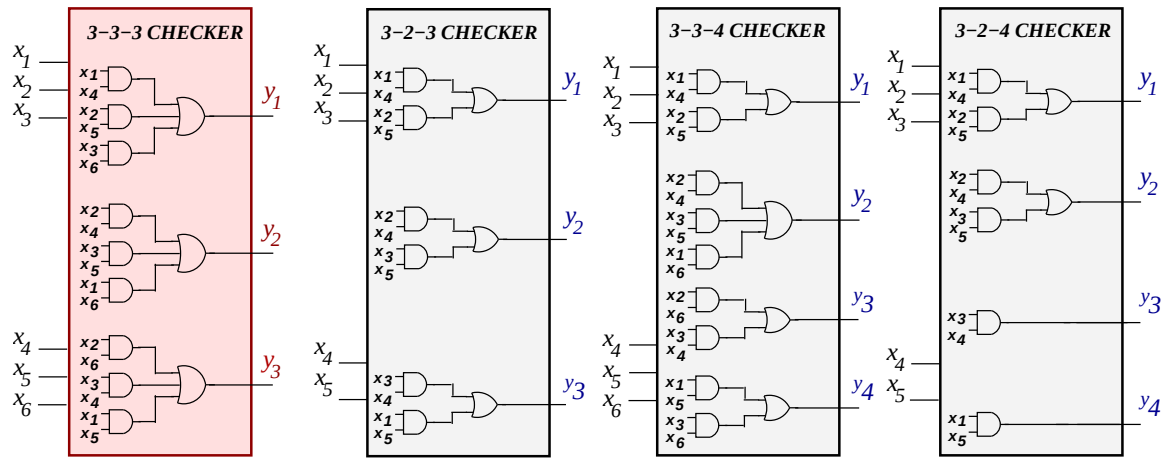

Abbildung 2.19: Codeübersetzer für den neuen vollständig selbstprüfenden 1-aus-3 Checker

Die Abbildung 2.18 a) illustriert zwei notwendige Blöcke für das Design des vorgestellten Checkers, und Abbildung 2.18 b) zeigt ein Beispiel für die mögliche Vereinigung der Blöcke.

Für die Konstruktion des neuen selbstprüfenden Checkers werden vier verschiedene Codeübersetzer-Blöcke benötigt [65], [63], [66]. In Abbildung 2.19 sind diese Codeübersetzern gezeigt.

Die Abbildung 2.19 a) stellt einen Checker vor, der sechs Eingänge und drei Ausgänge hat. Er realisiert die Transformation zweier Wörter (x_1, x_2, x_3) und (x_4, x_5, x_6) des 1-aus-3 Codes in ein Wort (y_1, y_2, y_3) des 1-aus-3 Codes. Diese Schaltung wird im folgenden die Bezeichnung 3-3-3 Checker tragen. Falls $x_6 = 0$, besitzt die Checkerstruktur fünf Eingänge und drei Ausgänge. Dieser neue 3-2-3 Checker wandelt ein Wort des 1-aus-3 Codes (x_1, x_2, x_3) und ein Wort des 1-aus-2 Codes (x_4, x_5) in das Wort (y_1, y_2, y_3) des 1-aus-3 Codes um. Der 3-2-3 Checker ist in der Abbildung 2.19 b) dargestellt. In der nächsten Abbildung 2.19 c) ist ein Checker mit sechs Eingängen und vier Ausgängen abgebildet. Dieser 3-3-4 Checker wird verwendet bei der Notwendigkeit der Kontrolle und Transformierung zweier Wörter (x_1, x_2, x_3) und (x_4, x_5, x_6) des 1-aus-3 Code in ein Wort (y_1, y_2, y_3, y_4) des 1-aus-4 Codes. Die Aufgabe der Transformation zweier Wörter (x_1, x_2, x_3) und (x_4, x_5) des 1-aus-3 Codes und des 1-aus-2 Codes an vier Ausgängen in Wörter (y_1, y_2, y_3, y_4) des 1-aus-4 Codes erledigt der 3-2-4 Checker (siehe Abbildung 2.19 d). Wie auf den Abbildungen zu sehen ist, haben die Bauelemente der Checker nicht mehr als zwei Niveaus. In den Strukturen werden nur Standardgatter mit zwei Eingängen verwendet, was die Checker-Strukturen unifiziert und die Realisierung vereinfacht. Die vorgestellten Schaltungen sind vollständig selbstprüfend, d.h. sie besitzen die beiden Eigenschaften „fehlersicher" und „selbsttestend".

Für den 3-3-3 Checker (Abbildung 2.19 a)) bedeutet die erste Eigenschaft (fehlersicher) das Folgende: Das Ausgabewort (y_1, y_2, y_3) enthält genau eine Eins nur in dem Fall, wenn beide Eingangswörter (x_1, x_2, x_3) und (x_4, x_5, x_6) ebenfalls jeweils nur eine Eins enthalten (z.B. falls $(x_1, x_2, x_3) = (100)$ und $(x_4, x_5, x_6) = (100)$, so ist $(y_1, y_2, y_3) = 100$). Das Vorhandensein zweier Einsen an den Ausgängen der Schaltung zeigt an, dass sich das Gewicht einer ihrer Eingangswörter ($\{y_1, \cdots, y_i\} \notin Y_{code}$) verändert hat (z.B. wenn $(y_1, y_2, y_3) = (101)$).

Die zweite Eigenschaft (selbsttestend) des Checkers bedeutet das Folgende: Für einen beliebigen einzelnen Defekt der Schaltung existiert eine solche Kombination zweier Eingangscodewörter, für die

an den Checkerausgängen ein Wort entsteht, das nicht dem Code zugehörig ist. Diese Eigenschaft wird durch die UND-Gatter des ersten Niveaus der Schaltung, welche mit den ODER-Gattern verbunden sind, garantiert.

Fehler in den UND-Elementen können zu zwei Fällen führen:

- Im ersten Fall wird ein Signal von 1 auf 0 gesetzt. Das geschieht bei der erwähnten Verknüpfung zweier Codewörter (x_1, x_2, x_3) und (x_4, x_5, x_6). Dann entsteht an den Ausgängen des Checker das Nicht-Codewort $(x_1, x_2, x_3) = (000)$.
- Im zweiten Fall wird an dem Ausgang eines UND-Gatters immer das Signal 1 gebildet, unabhängig von den Codewörtern, welche an den Eingängen anliegen. Dabei entsteht an den Ausgängen des Checker ein Nicht-Codewort, welches zwei Einsen enthält.

Codewörter	Fehler/Out	Fehler/Out	Fehler/Out	Fehler/Out
In: 010-100 Out: 010	s-a-1(x_3) 011	s-a-0(x_4) 000	s-a-0($OD2_1$) 000	s-a-1($OD1_3$) 110
In:010-01 Out: 100	s-a-1(x_3) 110	s-a-0(x_5) 000	s-a-0($OD1_2$) 000	s-a-1($OD3_1$) 101
In: 010-01 Out: 1000	s-a-1(x_3) 0110	s-a-0(x_4) 0000	s-a-0($OD2_1$) 0000	s-a-1($OD3_1$) 0110
In:010-1000 Out: 01	s-a-1(x_3) 1100	s-a-0(x_5) 0000	s-a-0($OD1_2$) 0000	s-a-1($OD2_1$) 1100

Tabelle 2.6: Stuck-at-0/1 Fehler in der Struktur des neuen 1-aus-3 Checker

Als Beispiel wird nun ein stuck-at-1 Fehler an dem Ausgang eines UND-Gatters untersucht. Der Fehler wird erkannt, wenn an den Eingängen die beiden Codewörter $(x_1, x_2, x_3) = (010)$ und $(x_4, x_5, x_6) = (100)$ anliegen. In diesem Fall ist $(y_1, y_2, y_3) = (110)$. Einige möglichen Fälle von Haftfehlern (stuck-at-1 Fehler, stuck-at-0 Fehler) sind in der Tabelle 2.6 aufgeführt.

Jeder der selbstprüfenden Module (*Self-Checking Checker/SCC*) kann für die Konstruktion selbstprüfender 1-aus-3 Checker einzeln oder in Verbindung mit anderen Modulen verwendet werden. In der Abbildung 2.20 sind zwei Beispiele vollständig selbstprüfender Checker für den 1-aus-3 Code mit 9

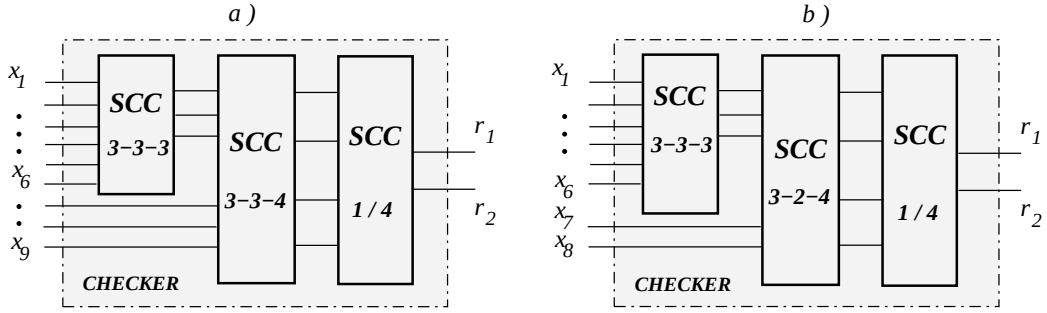

Abbildung 2.20: Selbstprüfender Checker für den 1-aus-3 Code mit a) 9 und b) 8 Eingängen

Inputs	Fläche / % von Verdopplung & Vergleich			Niveauanzahl / % von Verdopplung & Vergleich		
	neuer Checker	Checker von [64]	Verdoppl. TRC	neuer Checker	Checker von [64]	Verdoppl. TRC
1	2	3	4	5	6	7
6	43/65,1	44/66,7	66/100	5/71,4	5/71,4	7/100
9	70/66,7	72/68,5	105/100	7/77,8	9/100	9/100
12	97/67,3	100/69,4	144/100	7/77,8	7/77,8	9/100
15	124/67,7	128/69,9	183/100	9/100	11/122,2	9/100
18	151/68,2	156/70,2	222/100	9/81,8	9/81,8	11/100
21	178/68,2	184/70,4	261/100	9/81,8	11/100	11/100
24	205/68,3	212/70,7	300/100	9/81,8	9/81,8	11/100

Tabelle 2.7: Vergleich der Checker nach der notwendigen Fläche und der Niveauanzahl

und 8 Eingängen dargestellt. Um die Kosten der neuen Checkerkonstruktionsmethode bewerten zu können, wird die für den Checker erforderliche Fläche und die Niveauanzahl berechnet. Weiter wird in der Tabelle 2.7 der neue Checker für 1-aus-3 Codes mit dem selbstprüfenden Checker von [64] und dem Dual-Rail Checker von [28], der für die Konstruktion fehlererkennender Schaltungen mit der Methode „Verdopplung und Vergleich" notwendig ist, verglichen.

Tabelle 2.7 beschreibt die Ergebnisse der strukturellen Analyse der Checker. In der ersten Spalte der Tabelle ist die Anzahl der Checkereingänge angegeben. In den Spalten 2, 3 und 4 kann man den Flächebedarf jedes Checkers für verschiedene Eingangsanzahlen ablesen. Die Spalten 5, 6 und 7 enthalten für jede Schaltung die Niveauanzahl. Im Zähler steht der Flächebedarf (als Literalanzahl), der für die Realisation der Schaltungen der Checker notwendig ist bzw. die Niveauanzahl der Checkerstrukturen. Im Nenner steht der Prozentsatz der genannten Parameter im Vergleich zu „Verdopplung und Vergleich".

Für Realisierung des neuen Checkers für den 1-aus-3 Code ist 1,6% bis 2,4% weniger Fläche erforderlich als bei für die Realisierung des Checkers von [64]. Für einige Eingangsanzahlen ist die Niveauanzahl des neuen Checkers um 28,8% geringer als die Niveauanzahl des Checkers von [64]. Durchschnittlich ist die Niveauanzahl um 22% kleiner [63].

2.6.7 Der selbstprüfende Checker mit einem Ausgang

Alle Schaltungen selbstprüfender Checker, die in diesem Kapitel der Arbeit untersucht werden, sind kombinatorische Schaltungen mit zwei funktionalen Ausgängen, welche durch den Dual-Rail Code kodiert sind. Vom Gesichtspunkt der Vereinfachung der Aufgabe der Überwachung diskreter Geräte ist die Vereinigung der beiden Ausgangssignale des Kontrollteiles des Systems zu einem Signal manchmal notwendig (siehe Abbildung 2.21 a)) [67].

Das Schema des Checkers in der Abbildung 2.21 a) ist nicht selbsttestend. Der Grund hierfür ist der Umstand, dass sich ein beliebiger konstanter Fehler an dem Ausgang des Checkers nicht erkennen lässt. Für die Prüfung der Checkerstruktur kann ein spezielles Testsignal zur Erkennung konstanter Fehler eingefügt werden (siehe Abbildung 2.21 b). Das Prinzip des Einfügens eines zusätzlichen Testsignal-Eingangs ist die Grundlage für den *Built-In Self-Test* (BIST), welcher die zweite mögliche Fehlererkennungstechnik ist [68]. Die Idee der Vereinigung der Methode der Online-Fehlererkennungsdiagnostik

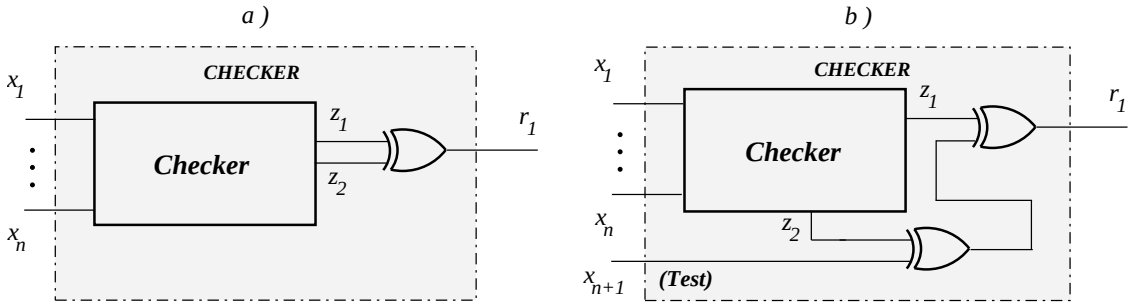

Abbildung 2.21: a) Kontrollschaltung mit einem Ausgang r_1; b) Kontrollschaltung mit einem Ausgang r_1 und einem speziellen Testeingang *Test*

und der *Built-In Self-Test*-Technik wurde erstmals in [69] beschrieben.

Auf Grundlage dieser Idee in [70] wird der vollständig selbstprüfende Checker mit einem Ausgang konstruiert. Als spezieller Fall der Konstruktion von Checkern mit bestimmten Charakteristiken wird in diesem Abschnitt der selbsttestende Checker von Sogomonyan [70] mit einem Ausgang untersucht. Der selbstprüfende Checker, in dem jedes logische Element des ersten Niveaus mit dem sich periodisch ändernden Kontrollsignal ergänzt wird, ist in der Abbildung 2.22 dargestellt. Die Idee der Verwendung eines sich periodisch ändernden Kontrollsignals ist auch für die Konstruktion vollständig selbstprüfender Checker mit zwei Ausgängen eine effektive Lösung [67].

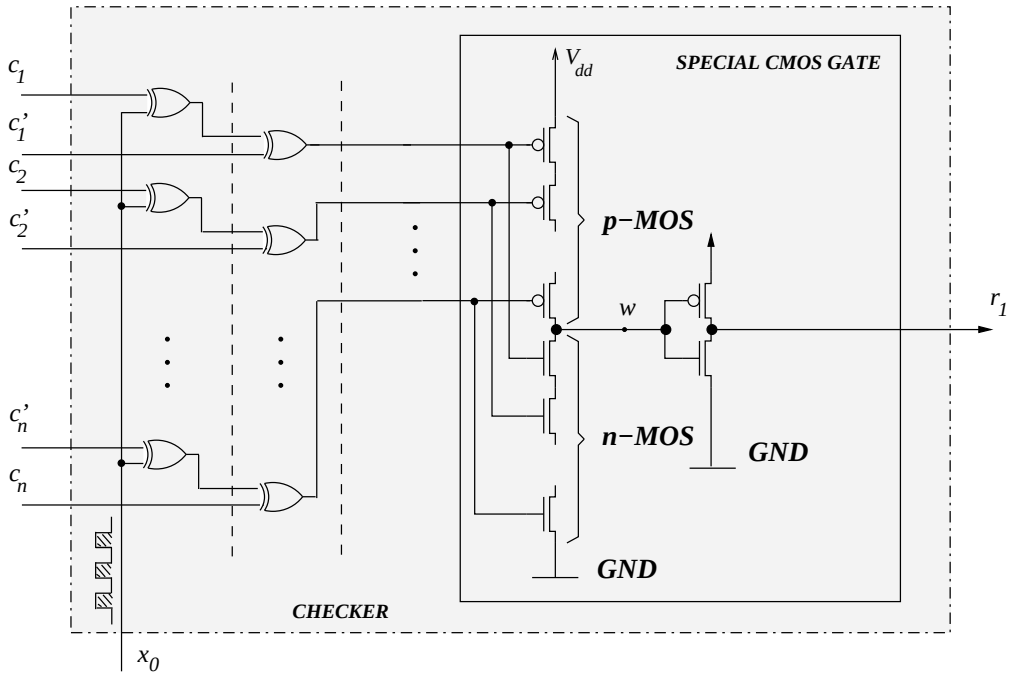

Abbildung 2.22: Vollständig selbstprüfender Checker mit einem Ausgang r_1

Der Checker funktioniert folgendermaßen. Bevor in der Schaltung ein Fehler auftritt und solange an den Eingängen $c'_i = \overline{c}_i$ gilt ($c'_i = c_i$) (wobei $i = 1, \cdots, n$), sind die Ausgänge der XOR-Elemente des zweiten Niveaus gleich $h_i = \overline{x}_0(x_0)$ und an den Eingängen des speziellen CMOS-Elements liegt entweder $\{11, \cdots, 1\}$ oder $\{00, \cdots, 0\}$ an. Für den Fall $h_1 = \cdots = h_n = \overline{x}_0$ sind die Werte am Ausgang des Checkers $\omega = x_0(\overline{x}_0)$ und $r_1 = \overline{x}_0(x_0)$ gleich. Auf diese Weise ändert sich im fehlerfreien Betrieb am Ausgang z_1 das Signal periodisch $\overline{x}_0(x_0)$. Bei Vorhandensein eines fehlerhaften Signals in der Checkerstruktur treten am Ausgangsignal r_1 Unperiodizitäten auf.

Ein wesentlicher Vorteil des Checkers mit einen Ausgang ist das Ausreichen nur zweier Testvektoren für die Prüfung anstatt 2^n Testvektoren, und ebenso das Benötigen von nur drei Niveaus bei einer beliebigen Eingangsanzahl. Es ist zu bemerken, dass der Checker mit einem periodischem Ausgang fähig ist, sowohl mit einem bestimmten Ausgangsignal als auch mit dem inversen Ausgangssignal zu arbeiten, im Gegensatz zu dem Checker von [28].

3 Die Logische Ergänzung als neue Fehlererkennungsmethode

Eine wesentliche Frage in diesem Kapitel ist, wie in den strukturellen Teil des Systems am wirksamsten Redundanz eingefügt wird, um eine möglichst hohe Fehlerüberdeckung zu erreichen.

In der Arbeit wird eine neue Methode zur Fehlererkennung für Berger-Codes und 1-aus-3 Codes unter Benutzung zusätzlicher Logik ausgearbeitet. Ein Ergebnis dieser Methode sind vollständig selbstprüfende Schaltungen, deren Parameter (Fehlererkennungswahrscheinlichkeit und benötigter Flächebedarf) mit den bis zum heutigen Tage bekannten Methoden verglichen werden.

Eine beliebige Struktur mit der Möglichkeit der Fehlererkennung besteht aus zwei Teilen: der funktionalen Logik und der Kontrollstruktur für diese Logik. Im vorhergehenden Kapitel wurde aufgezeigt, wie der Kontrollteil des Systems am wirksamsten aufgebaut wird. In diesem Kapitel wird untersucht, wie die Eigenschaften der funktionalen Logik vollständig genutzt werden können und wie die zusätzliche Logik am effektivsten konstruiert werden kann.

3.1 Prinzipielle Struktur der Logischen Ergänzung für kombinatorische Schaltungen

Für die Bestimmung der Effektivität der neuen Konstruktionsmethode für selbstprüfende Schaltungen ist es erforderlich, mit der Suche und der Beschreibung der Nachteile der traditionellen Konstruktion von Fehlererkennungsschaltungen zu beginnen.

Für eine ausführliche Analyse ist in der Abbildung 3.1 die traditionelle Struktur der Schaltungskontrolle unter Verwendung von systematischen und nicht-systematischen Codes (mit „*" gekennzeichnet) aufgeführt. Die vorgestellten Fehlererkennungsschaltungen bestehen aus drei Blöcken. Dabei besitzt die funktionale Schaltung $f(x)$ m Eingänge $x = (x_1, \cdots, x_m)$ sowie n funktionale Ausgänge $y_1 = f_1(x), \cdots, y_n = f_n(x)$, die die Informationsstellen der Codewörter darstellen. Bei der Verwendung von nicht-systematischen Codes für die Kontrolle wird anstatt des Coders $C(x)$ in der Fehlererkennungsstruktur der zusätzliche Block $B(x)$ verwendet. Der zusätzliche Block $B(x)$ erzeugt so zusätzliche Codewortstellen $b_1(x), \cdots, b_p(x)$, dass an den Ausgängen der vorgestellten Schaltung $f(x)$ und des Blockes $B(x)$ Wörter $(f_1(x), \cdots, f_n(x), b_1(x), \cdots, b_p(x))$ eines nicht-systematischen Codes gebildet werden. Die Zugehörigkeit der erzeugten Wörter zum verwendeten Code wird durch den Code Checker geprüft.

Die Abbildung 3.2 zeigt die Struktur der neuen Fehlererkennungsmethode unter Verwendung der Logischen Ergänzung und beliebiger fehlererkennender Codes. Die Anzahl der Blöcke ist in der neuen Methode ebenso gleich drei. Der zusätzliche Block g berechnet die Ergänzungsfunktionen $g_1(x)$, $g_2(x)$, $\cdots$, $g_k(x)$ von $x = (x_1, \cdots, x_m)$, die die Ausgabefunktionen $f_1(x), f_2(x), \cdots, f_n(x), n > k$ des funktionalen Blockes f so erweitern, dass an den Leitungen $h_1(x), h_2(x), \cdots, h_n(x)$ Wörter des systematischen oder nicht-systematischen Codes erzeugt werden. Die Wörter eines beliebigen Codes werden an den Ausgängen der „XOR"-Gatter erzeugt, deren Eingänge die Ausgangsleitungen der funk-

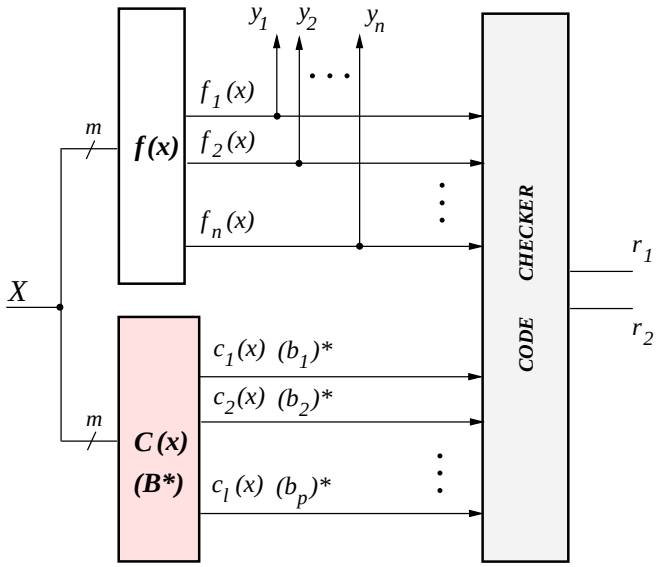

Abbildung 3.1: Die traditionelle Fehlererkennungsstruktur für systematische/nicht-systematische Codes*

tionalen Schaltung f und des zusätzlichen Blockes g sind. Funktional unterscheidet sich der Code Checker in der Struktur der neuen Methode nicht vom Code Checker (CC) in der Abbildung 3.1.

Das traditionelle Schema der Fehlererkennungsschaltung verfügt über die folgenden Hauptmängel:

1. Die Anzahl der Checkereingänge ist hoch. Die Anzahl der Checkereingänge ist immer gleich der Stellenanzahl des verwendeten Codes;

2. Es existiert (bis auf Optimierung) nur eine einzige mögliche Realisation des Coders $C(x)$, welcher die Kontrollstellen generiert, bzw. es existiert nur eine Realisation des zusätzlichen Blockes $B(x)$ im Falle der Verwendung von nicht-systematischen Codes;

3. Es ist möglich, dass an den Eingängen des Checkers nicht alle für den Test benötigten Codekombinationen erzeugt werden.

Die neue Konstruktionsmethode für Kontrollschaltungen soll die Verbesserung jedes dieser Kriterien, die für die traditionelle Kontrollstruktur bestimmt wurden, gewährleisten. Ein Kontrollschaltung, das die Logische Ergänzung benutzt, verfügt über die folgenden Charakteristiken [71]:

1. In der neuen Fehlererkennungsstruktur wird ein einfacheres Checkerschema verwendet. Die Anzahl der Checkereingänge ist um k kleiner, als die Zahl der Checkereingänge in der traditionellen Struktur;

2. Es existiert die Möglichkeit der Implementierung einer großen Auswahl verschiedener zusätzlicher Blöcke g Logischer Ergänzungen. Die komponentweise XOR-Summe $f_1(x) \oplus g_1(x)$, $\cdots, f_n(x) \oplus g_n(x)$ von $f_1(x), \cdots, f_n(x)$ und $g_1(x), \cdots, g_n(x)$ muss ein beliebiges Codewort, aber kein bestimmtes Codewort ergeben;

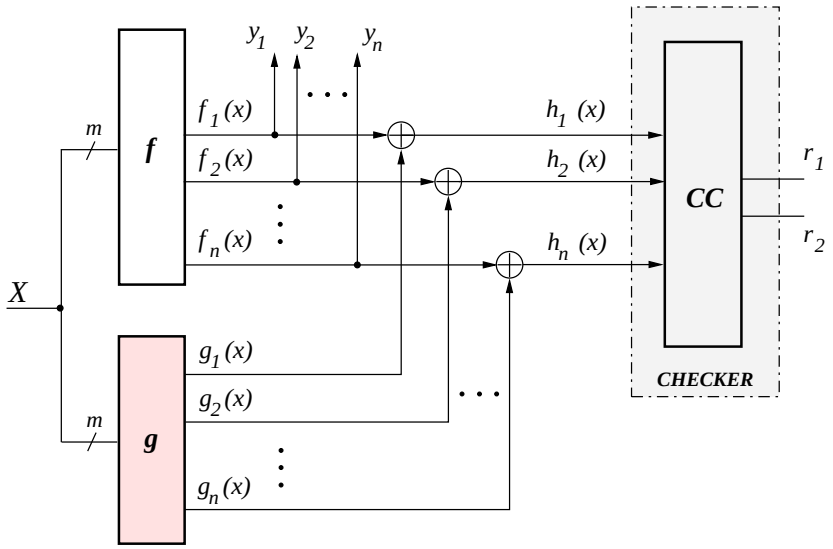

Abbildung 3.2: Das prinzipielle Schema der Fehlerüberwachung nach der neuen Methode der Logischen Ergänzung

3. Unter der Menge der Realisationsvarianten des Blockes g ist es möglich, eine solche zu finden, die die kleinste Fläche erfordert;

4. Die neue Methode der Fehlererkennung gewährleistet die Konstruktion selbstprüfender Schaltungen auch für jene Fälle, in denen die traditionelle Konstruktionsmethode versagt.

Als Beispiel soll nun eine kombinatorische Schaltung $f(x)$ mit sechs funktionellen Ausgängen betrachtet werden, die das folgende Funktionssystem realisiert:

$$f_1(x) = \overline{x}_1\overline{x}_2x_3,\ f_4(x) = x_1x_2\overline{x}_3,$$
$$f_2(x) = \overline{x}_1x_2\overline{x}_3,\ f_5(x) = x_1x_3,$$
$$f_3(x) = \overline{x}_1x_2x_3,\ f_6(x) = x_1x_3 \vee x_1\overline{x}_2.$$

In der Tabelle 3.1 sind für acht Eingangskombinationen die Ausgangswerte der sechs Funktionen aufgeführt. Die Analyse der Tabelle 3.1 zeigt, dass die Überwachung der Schaltung f mit der klassischen Fehlererkennungsmethode auf Grundlage des 2-aus-8 Codes möglich ist (Abbildung 3.3 a)). Zu diesem Zweck wird der zusätzliche Block $B(x)$ mit zwei Ausgängen $b_1(x), b_2(x)$ realisiert. Die Funktionswerte von $g_1(x), g_2(x)$ werden so bestimmt, dass jeder Checkereingangskombination ein Vektor des 2-aus-8 Codes entspricht. Aus den Funktionswerten von $b_1(x), b_2(x)$ folgt, dass die Funktionen bestimmt sind durch die folgenden Gleichungen: $b_1(x) = \overline{x}_1$, $b_2(x) = \overline{x}_2\overline{x}_3 \vee x_1\overline{x}_3$. In Abbildung 3.3 a) ist die Struktur der gesamten nach der traditionellen Methode konstruierten Fehlererkennungsschaltung aufgezeigt.

Die Struktur der Kontrollschaltung, die nach der neuen Methode der Logischen Ergänzung aufgebaut ist, wird in der Abbildung 3.3 b) vorgestellt. Diese Schaltung ist vollständig selbstprüfend. In ihr wird die Funktion der funktionalen Schaltung $f_6(x) = x_1x_3 \vee x_1\overline{x}_2$ durch die Funktion $g_6(x)$ der komplementären Schaltung zu der folgenden Funktion $h_6(x) = \overline{x}_2\overline{x}_3$ ergänzt. Dabei ist die zusätzliche Funktion $g_6(x) = x_1x_3 \vee \overline{x}_1\overline{x}_2\overline{x}_3$.

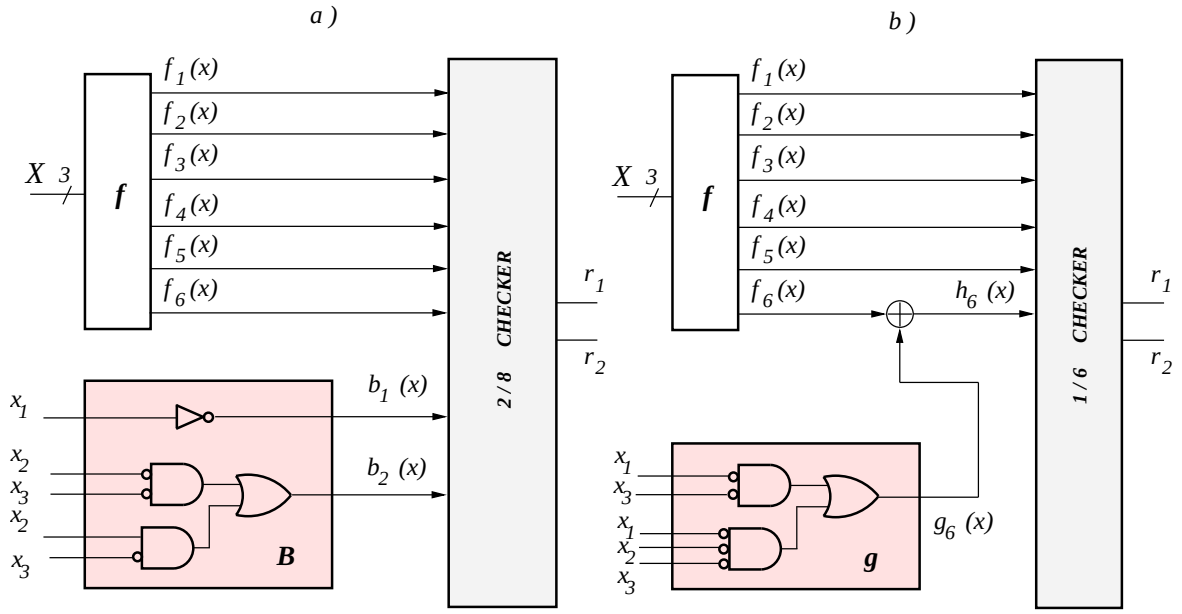

Abbildung 3.3: Beispiel einer Kontrollschaltung unter Benutzung a) klassische Online-Fehlererkennungsmethode, b) der neue Methode der Logischen Ergänzung

In der Tabelle 3.2 sind für acht Eingangskombinationen die Werte an den Ausgangsleitungen $f_1(x)$, $\cdots$, $f_6(x)$ aufgeführt. An den Eingängen eines XOR-Elementes wird das Auftreten aller Testkombinationen und an den Checkereingängen das Auftreten aller Codewörter des Testsatzes gewährleistet. Der 1-aus-6-Code Checker überwacht in der vorgestellten Struktur die Zugehörigkeit der auftretenden Codewörter zu dem 1-aus-6 Code. Am Beispiel der betrachteten Schaltung sind die zuvor beschriebenen Vorteile der neuen Methode zur Konstruktion von Fehlererkennungsschaltungen im Vergleich zu den traditionellen Methoden gut zu sehen.

Weiter wird in der Arbeit ein Effektivitätsbeweis der neuen Methode gegeben, der sich nicht nur auf

x_1	x_2	x_3	f_1	f_2	f_3	f_4	f_5	f_6	b_1	b_2
0	0	0	0	0	0	0	0	0	1	1
0	0	1	1	0	0	0	0	0	1	0
0	1	0	0	1	0	0	0	0	1	0
0	1	1	0	0	1	0	0	0	1	0
1	0	0	0	0	0	0	0	1	0	1
1	0	1	0	0	0	0	1	1	0	0
1	1	0	0	0	0	1	0	0	0	1
1	1	1	0	0	0	0	1	1	0	0

Tabelle 3.1: Wertetabelle einer kombinatorischen Schaltung f mit sechs Ausgängen (klassische Online-Fehlererkennungsmethode)

x_1	x_2	x_3	f_1	f_2	f_3	f_4	f_5	f_6	g_6	h_6
0	0	0	0	0	0	0	0	0	1	1
0	0	1	1	0	0	0	0	0	0	0
0	1	0	0	1	0	0	0	0	0	0
0	1	1	0	0	1	0	0	0	0	0
0	0	0	0	0	0	0	0	1	0	1
1	0	1	0	0	0	0	1	1	1	0
1	1	0	0	0	0	1	0	0	0	0
1	1	1	0	0	0	0	1	1	1	0

Tabelle 3.2: Wertetabelle einer kombinatorischen Schaltung f mit sechs Ausgängen (neue Methode der Logischen Ergänzung)

den geringeren Flächeverbrauch bezieht, sondern auch auf die Fehlerüberdeckung.

3.2 Konstruktionsprinzipien der komplementären Schaltungen

Die komplementäre Schaltung, die durch die komplementären Funktionen $g_1(x), \cdots, g_n(x), \ x = (x_1, \cdots, x_m)$ beschrieben wird, kann mit zwei verschiedenen Methoden konstruiert werden.

1. Die *analytische* Methode sieht die Aufstellung einer speziellen Tabelle mit den Funktionswerten der komplementären Schaltung vor. Ein Beispiel einer solchen Tabelle ist weiter unten aufgeführt.

   Die erste Spalte der Tabelle 3.3 enthält die drei Eingangskombinationen x_1, x_2, x_3, in der zweiten Spalte sind die vier Ausgangsfunktionen $f_1(x), f_2(x), f_3(x), f_4(x)$ der überwachten funktionalen Schaltung f aufgezeigt. Für jede Eingangskombination werden die Funktionswerte der funktionalen Schaltung in Codewörter, welche für die Kontrollorganisation (Spalte 3) verwendet werden, umgewandelt.

   Der fehlererkennende Code, welcher in der Tabelle 3.3 verwendet wurde und an den Leitungen $h_1(x), h_2(x), h_3(x), h_4(x)$ generiert wird, ist ein 1-aus-4 Code. In der nächsten Spalte sind

x_1	x_2	x_3	f_1	f_2	f_3	f_4	g_1	g_2	g_3	g_4	h_1	h_2	h_3	h_4
0	0	0	0	1	0	1	0	1	0	0	0	0	0	1
0	0	1	0	1	1	0	0	0	1	0	0	1	0	0
0	1	0	0	0	0	0	0	1	0	0	0	1	0	0
0	1	1	0	0	1	1	0	0	1	0	0	0	0	1
1	0	0	1	0	0	1	1	0	0	0	0	0	0	1
1	0	1	1	0	0	0	1	0	1	0	0	0	1	0
1	1	0	0	0	0	0	1	0	0	0	1	0	0	0
1	1	1	0	0	1	0	1	0	1	0	1	0	0	0

Tabelle 3.3: Wertetabelle einer kombinatorischen Schaltung f mit vier Ausgängen

die Werte der komplementären Funktionen $g_1(x), g_2(x), g_3(x), g_4(x)$ dargestellt, die nach der Formel $g_i(x) = f_i(x) \oplus h_i(x)$ berechnet wurden. Für das betrachtete Beispiel sind die komplementären Funktionen: $g_1(x) = x_1, g_2(x) = x_3, g_3(x) = \overline{x}_1\ \overline{x}_3, g_4(x) = 0$.

Mit diesem vorgestellten Funktionssatz wird die komplementäre Schaltung $g(x)$ realisiert. Falls die Funktion $g_i(x) = 0$ ist, so ist eine Gleichheit zwischen den Funktionen $h_i(x)$ und $f_i(x)$ gegeben: $f_i(x) = h_i(x)$. Das bedeutet, dass auf der Leitung $f_i(x)$ $(h_i(x))$ kein XOR-Gatter benötigt und dementsprechend nicht aufgestellt wird. Die konstruierte Komplementärschaltung $g(x)$ für 1-aus-4 Codes ist in Abbildung 3.4 a) dargestellt.

2. Die zweite Realisationsmethode der komplementären Schaltung ist die *strukturelle* Methode. Diese Methode gründet sich auf die Modifikation der funktionalen Schaltung $f(x)$ und deren Optimierung in die Schaltung $g(x)$. Zum Beispiel können bei der Organisation der Kontrolle der Schaltung $f(x)$ mit vier Ausgängen $f_1(x), f_2(x), f_3(x), f_4(x)$ die komplementären Funktionen $g_1(x), g_2(x), g_3(x), g_4(x)$ nach den folgenden Gleichungen berechnet werden:

$$g_1(x) = 0,\ g_2(x) = f_1(x)f_2(x), g_3(x) = (f_1(x)\ f_2(x)) \vee f_3(x),$$
$$g_4(x) = (f_1(x)\ f_2(x)\ f_3(x)) \vee f_3(x) \vee f_4(x) \vee (f_1(x)\ f_2(x)\ f_3(x)\ f_4(x)).$$

In Abbildung 3.4 b) ist die komplementäre Schaltung $g(x)$ dargestellt, welche nach den Ausgängen $g_2(x), g_3(x), g_4(x)$ optimiert wurde.

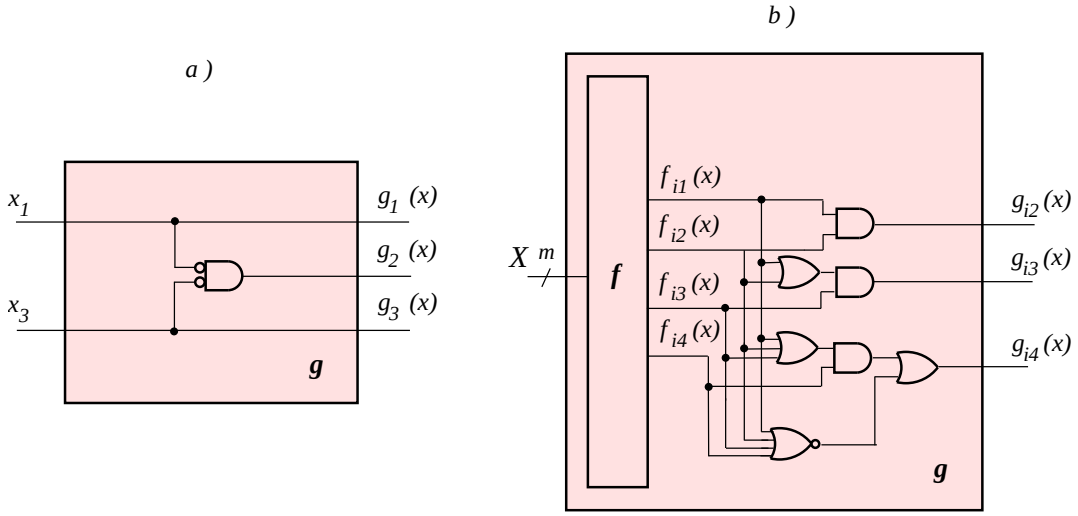

Abbildung 3.4: a) Komplementäre Schaltung $g(x)$ (*analytische* Methode), b) Komplementäre Schaltung $g(x)$ (*strukturelle* Methode)

Ein wichtiger Parameter bei der Konstruktion der komplementären Schaltung - der Flächebedarf der komplementären Schaltung - ist dabei abhängig von der Struktur der funktionalen Schaltung sowie der verwendeten Reihenfolge ihrer Ausgänge $f_1(x), f_2(x), f_3(x), f_4(x)$ (in späteren Abschnitten wird auf dieses Problem besondere Aufmerksamkeit gelenkt).

Im folgenden Abschnitt wird der Aufbau selbstprüfender Strukturen nach der neuen Methode der Logischen Ergänzung unter Verwendung des Berger-Codes vorgestellt.

3.3 Die Verwendung des Berger-Codes in der neuen Methode der Logischen Ergänzung

Die erste Arbeit, in der separierbare Codes für die funktionale Diagnostik verwendet wurden, ist die Arbeit von Ashijire und Reddy [40]. Bekannte Beispiele für die Anwendung struktureller Methoden der funktionalen Diagnostik unter Verwendung von Berger-Codes sind [72], [42], [41], [43], [27].

Dabei wird einige Male die funktionale Logik modifiziert, um eine Verbesserung der Parameter zu erhalten. So wird in der Arbeit von [73] die funktionale Logik auf solche Weise abgeändert, so dass als Ergebnis des stuck-at-1/0 Fehlers an den Schaltungsausgängen unidirektionale Fehler angezeigt werden. Zum Beispiel in [74] wird zur Erreichung dieses Ziels die originale funktionale Schaltung als Zwei-Level-Realisierung implementiert. In der neuen Methode der Logischen Ergänzung ändert sich nicht die Struktur der funktionalen Schaltung. Im Vergleich zu den Methoden der Strukturmodifikation bedeutet dies Vorteile in den Bereichen Zeit (Schnelligkeit), Flächebedarf, Performance und in weiteren Bereichen.

Die neue Methode modifiziert nur die Menge der gewählten Ausgänge der funktionalen Schaltung durch die Ausgänge der speziellen komplementären Logik. Im Sonderfall der Verwendung von Berger-Codes werden, vorausgesetzt, dass keine Fehler auftraten, an den Ausgängen der funktionalen Logik nach der teilweisen Ergänzung Wörter des Berger-Codes bereit gestellt.

Im folgenden wird ein Beispiel zur Konstruktion vollständig selbstprüfender Schaltungen auf Grundlage der angebotenen Methode der Logischen Ergänzung untersucht. Nach der Erläuterung dieser Methode wird experimentell ein Vergleich zwischen der erhaltenen Struktur mit früher bekannten Strukturen vorgenommen werden. Für die Bestimmung der Effektivität der angebotenen Methode wird der 5S2 Berger-Code mit drei Informationsstellen und zwei Kontrollstellen verwendet (Tabelle 3.4).

y_1	y_2	y_3	y_4	y_5
0	0	0	1	1
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	0	0

Tabelle 3.4: 5S2 Berger-Code

Der 5S2 Berger-Code ist ein vollständig separierbarer Code (Abschnitt 2.4.2). Aus diesem Grunde beschreiben die inversen Werte der Kontrollstellen y_4, y_5 die Anzahl der Einsen der Informationsstellen y_1, y_2, y_3, bzw. die tatsächlichen Werten der Kontrollstellen y_4, y_5 sind die binäre Darstellung der Anzahl der Nullen in den Informationsstellen y_1, y_2, y_3. Die Berechnung der Kontrollstellen für den 5S2 Berger-Code ist durch Benutzung der folgenden Formel möglich.

$$\begin{aligned} y_4 &= \overline{y_1y_2 \vee y_1y_3 \vee y_2y_3} = \overline{V}(y_1, y_2, y_3), \\ y_5 &= \overline{y_1 \oplus y_2 \oplus y_3} = \overline{P}(y_1, y_2, y_3), \end{aligned} \tag{3.1}$$

wobei $V(y)$ die Voter-Funktion und $P(y)$ die Paritätsfunktion ist.

Falls der 5S2 Berger-Code beispielsweise an den Schaltungsausgängen vollständig ohne Rest implementiert ist, wird eine kombinatorische Schaltung f mit drei Eingängen und fünf Ausgängen verwendet. An den Schaltungsausgängen werden die folgenden Booleschen Funktionen verwirklicht:

$$f_1(x) = \overline{x}_1\overline{x}_3 \vee x_2\overline{x}_3 \vee x_1\overline{x}_2x_3, f_2(x) = \overline{x}_1\overline{x}_2 \vee \overline{x}_1x_3, f_3(x) = \overline{x}_1x_2 \vee x_3,$$
$$f_4(x) = x_1x_2 \vee x_1\overline{x}_2\overline{x}_3, f_5(x) = \overline{x}_1 \vee x_1\overline{x}_2x_3. \quad (3.2)$$

In der Tabelle 3.5 sind für drei Eingangsvariablen die Werte der Ausgangsfunktionen $f_1(x), \cdots, f_5(x)$ aufgeführt. Die neue Methode sieht die Ergänzung der Funktionen der funktionalen Schaltung f mit dem Ziel der Erzeugung von Codewörtern an allen (abgeänderten und unabgeänderten) Ausgängen vor.

x_1	x_2	x_3	f_1	f_2	f_3	f_4	f_5	h_4	h_5	g_4	g_5
0	0	0	1	1	0	0	1	0	1	0	0
0	0	1	0	1	1	0	1	0	1	0	0
0	1	0	1	0	1	0	1	0	1	0	0
0	1	1	0	1	1	0	1	0	1	0	0
1	0	0	0	0	0	1	0	1	1	0	1
1	0	1	1	1	1	0	1	0	0	0	1
1	1	0	1	0	0	1	0	1	0	0	0
1	1	1	0	0	1	1	0	1	0	0	0

Tabelle 3.5: Wertetabelle für eine kombinatorische Schaltung f mit fünf Ausgängen unter Verwendung des 5S2 Berger-Codes

Aus den Parametern des verwendeten Berger-Codes folgt, dass die Ergänzung zweier Ausgänge der funktionalen Schaltung f erforderlich ist. Die Folge der Ergänzung ist die Modifikation der Schaltungsausgänge in Kontrollstellen des 5S2 Berger-Codes. Dabei werden die komplementären Funktionen $g_4(x), g_5(x)$ der Logischen Ergänzung $g(x)$ mit dem folgenden Ausdruck ausgerechnet:

$$h_i(x) = f_i(x) \oplus g_i(x) \quad (3.3)$$

Die Funktionen $h_4(x)$, $h_5(x)$ sind die Kontrollstellen der Codewörter $\{h_1(x)$, $h_2(x)$, $h_3(x)$, $h_4(x)$, $h_5(x)\}$ ($h_4(x) = f_4(x) \oplus g_4(x)$, $h_5(x) = f_5(x) \oplus g_5(x)$) siehe Tabelle 3.5) für alle $x \in X$. Zusätzlich sind die Werte an den Ausgängen der komplementären Schaltung $g_4(x), g_5(x)$ für alle $x \in X$ in Tabelle 3.5 aufgeführt. Die Analyse der Ergebnisse der komplementären Funktionen für die funktionalen Schaltungsausgänge zeigt auf, dass diese Funktionen für das vorliegende Beispiel nach den folgenden Formeln berechnet werden:

$$g_4(x) = 0,\ g_5(x) = x_1\overline{x}_2. \quad (3.4)$$

Für die Gewährleistung des Auftretens von Codewörtern des 5S2 Berger-Codes an den Checkereingängen ist nur die Ergänzung eines Ausganges der Schaltung f notwendig. In Abbildung 3.5 ist eine komplementäre Schaltung $g(x)$ für die Ergänzung der funktionalen Schaltung $f(x)$ mit fünf Ausgängen und der 5S2 Berger-Code Checker dargestellt.

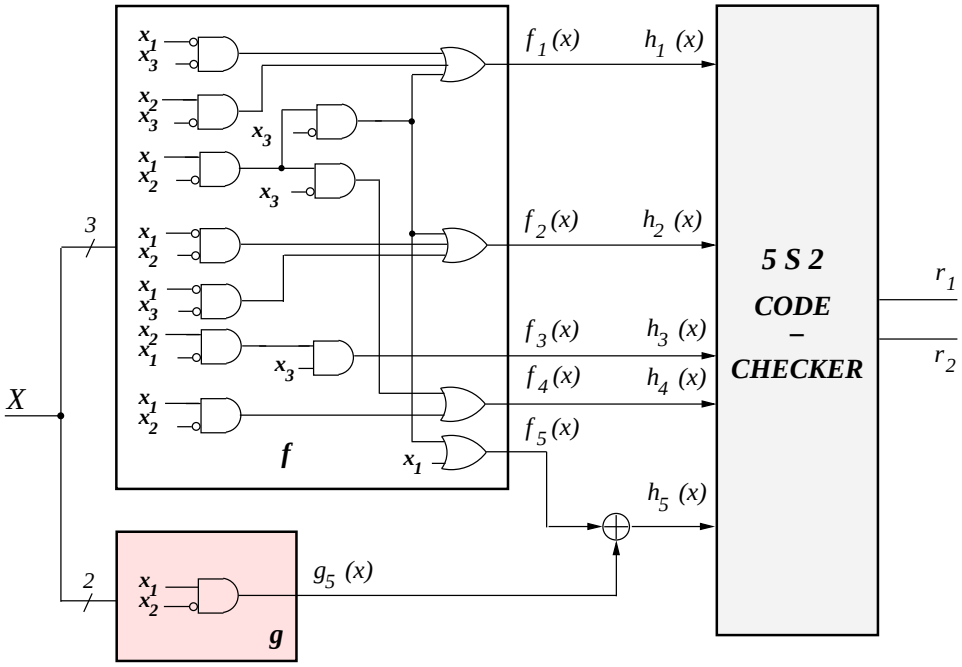

Abbildung 3.5: Die neue Methode der funktionalen Diagnostik unter Verwendung des Berger-Codes und der komplementären Schaltung

Die Verwendung des Berger-Codes in der traditionellen Methode der Schaltungskontrolle mit fünf Ausgängen erfordert das Vorhandensein dreier Kontrollausgänge. In diesem Falle stellen die Ausgänge der funktionalen Schaltung die Informationsstellen des 8S3 Berger-Codes dar.

In der Tabelle 3.6 sind für acht Eingangskombinationen die Werte der Informationsstellen $f_1(x)$, $\cdots, f_5(x)$ und der Kontrollstellen $c_1(x), c_2(x), c_3(x)$ des betrachteten Berger-Codes aufgeführt. Aus der Tabelle folgt, dass die Kontrollstellen vom Coder $C(x)$ durch die folgenden Ausdrücke beschrieben werden:

$$f_6(x) = \overline{x}_1 \vee x_1x_2 \vee x_1\overline{x}_3, f_7(x) = x_1\overline{x}_2, f_8(x) = x_1x_2 \vee x_1x_3. \tag{3.5}$$

Die kombinatorische Schaltung $f(x)$ mit fünf Ausgängen, die zusätzliche Schaltung (Coder $C(x)$) mit drei Ausgängen und der Checker für den 8S3 Berger-Code sind in der Abbildung 3.6 aufgezeigt. In diesem Beispiel einer elementaren Schaltung wurde der prinzipielle Unterschied beider Methoden der funktionalen Diagnostik kombinatorischer Schaltungen aufgezeigt.

Es ist leicht zu sehen, dass der Code Checker in der neuen Methode der Logischen Ergänzung eine kleinere Realisationsfläche erfordert, dass in der Schaltung ein einfacherer Code überwacht wird, und dass die zusätzliche Logik in dieser Struktur ebenso eine geringere Fläche besitzt. Die notwendige Realisationsfläche der selbstprüfenden Schaltung konstruiert nach der neuen Methode (Abbildung 3.5) beträgt 62.2% der Fläche der Schaltung, welche nach der traditionellen Methode implementiert wurde (Abbildung 3.6).

Die Prozentzahl der in der Praxis verwendeten Schaltungen mit fünf Ausgängen im Verhältnis zur Anzahl der Schaltungen mit vielen Ausgängen ist relativ klein. Deshalb wird im folgenden ein Algorithmus zur Konstruktion selbstprüfender Schaltungen nach der neuen Methode mit n Ausgängen

x_1	x_2	x_3	f_1	f_2	f_3	f_4	f_5	c_1	c_2	c_3
0	0	0	1	1	0	0	1	1	0	0
0	0	1	0	1	1	0	1	1	0	0
0	1	0	1	0	1	0	1	1	0	0
0	1	1	0	1	1	0	1	1	0	0
1	0	0	0	0	0	1	0	1	1	0
1	0	1	1	1	1	0	1	0	1	1
1	1	0	1	0	0	1	0	1	0	1
1	1	1	0	0	1	1	0	1	0	1

Tabelle 3.6: Wertetabelle für eine kombinatorische Schaltung f mit fünf Ausgängen unter Verwendung des 5S2 Berger-Codes

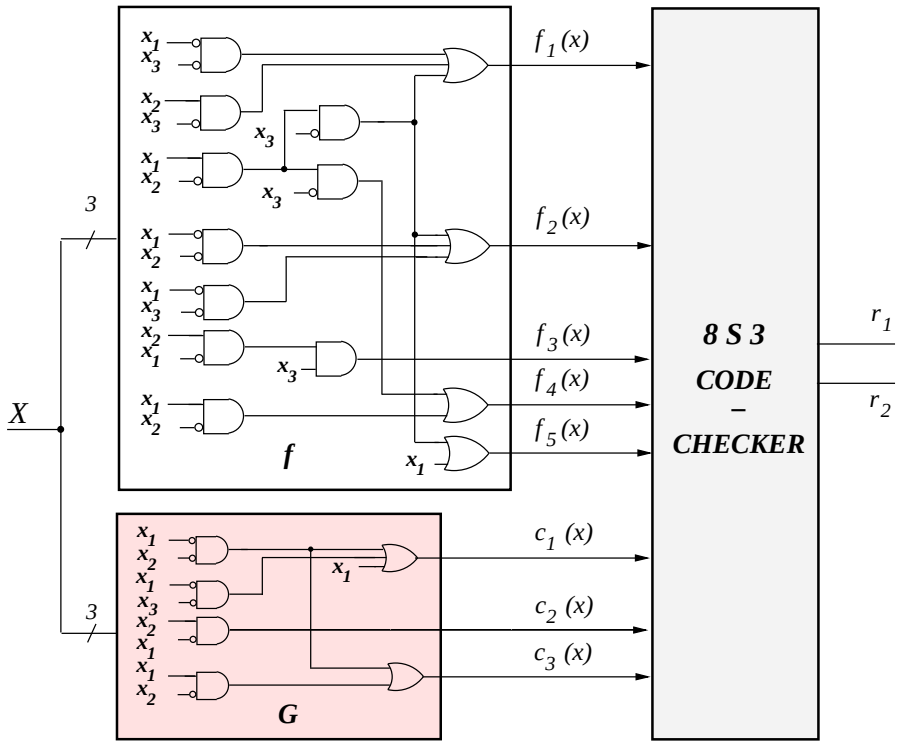

Abbildung 3.6: Traditionelle Methode der funktionalen Diagnostik unter Verwendung des 8S3 Berger-Codes und eines Coders $C(x)$

beschrieben werden, wobei $n > 5$. Zunächst werden die Ausgänge in i Gruppen von Ausgängen X_i^s aufgeteilt. Die Parameter dieses Codes bestimmen die Anzahl der Ausgänge der Gruppen. Für den 5S2 Berger-Code sieht diese Ausgängeverteilung so aus: Die Ausgänge werden in Gruppen von je fünf Ausgängen aufgeteilt, wobei die letzte Gruppe X_{rest} aus dem Rest der Ausgänge besteht $X_{rest} < 5$.

Am Beispiel der Kontrollstruktur einer kombinatorischen Schaltung mit 12 funktionalen Ausgängen, welche in der Abbildung 3.7 aufgeführt ist, wird nun ein Verteilungsalgorithmus beschrieben. Die Aus-

gänge $f_1(x), \cdots, f_{12}(x)$ der Schaltung f werden in zwei gleich große Gruppen $X_1^5 = f_1(x), \cdots, f_5(x)$ und $X_2^5 = f_6(x), \cdots, f_{10}(x)$ sowie eine Restgruppe X_{rest} aufgeteilt. Dabei enthält die Restgruppe zwei Ausgänge $X_{rest} \in f_{11}(x), f_{12}(x)$. Die Ausgänge der Restgruppe werden in der Kontrollschaltung invertiert. Die Funktionen $f_4(x)$, $f_5(x)$ der ersten Gruppe von Ausgängen X_1^5 werden durch zwei Antivalenzfunktionen (XOR-Funktionen) mit den Ausgängen $g_4(x)$, $g_5(x)$ der komplementären Schaltung $g(x)$ ergänzt. Analog werden die Ausgänge $f_9(x)$, $f_{10}(x)$ der zweiten Gruppe X_2^5 durch die Ausgänge $g_9(x)$, $g_{10}(x)$ ergänzt. Daraufhin werden an den Leitungen für alle Eingangskombinationen Wörter des 5S2 Berger-Codes erzeugt. Die komplementären Funktionen, die die Struktur der komplementären Schaltung $g(x)$ bestimmen, werden durch die folgenden Gleichungen beschrieben:

$$g_4(x) = \overline{V}(f_1, f_2, f_3) \oplus f_4,\ g_5(x) = \overline{P}(f_1, f_2, f_3) \oplus f_5,$$
$$g_9(x) = \overline{V}(f_6, f_7, f_8) \oplus f_9,\ g_{10}(x) = \overline{P}(f_6, f_7, f_8) \oplus f_{10}, g_{11}(x) = \overline{f}_{11},\ g_{12}(x) = \overline{f}_{12}. \quad (3.6)$$

Nun wird ein Beispiel für die Konstruktion einer Fehlererkennungsschaltung nach der traditionellen Methode mit der gleichen funktionalen Schaltung mit 12 Ausgängen betrachtet werden. Die Implementierung der funktionalen Diagnostik dieses Beispiels ist in der Abbildung 3.8 zu sehen. Die Ausgänge der funktionalen Schaltung $f(x)$ werden auch hier in zwei gleich große Gruppen von Ausgängen

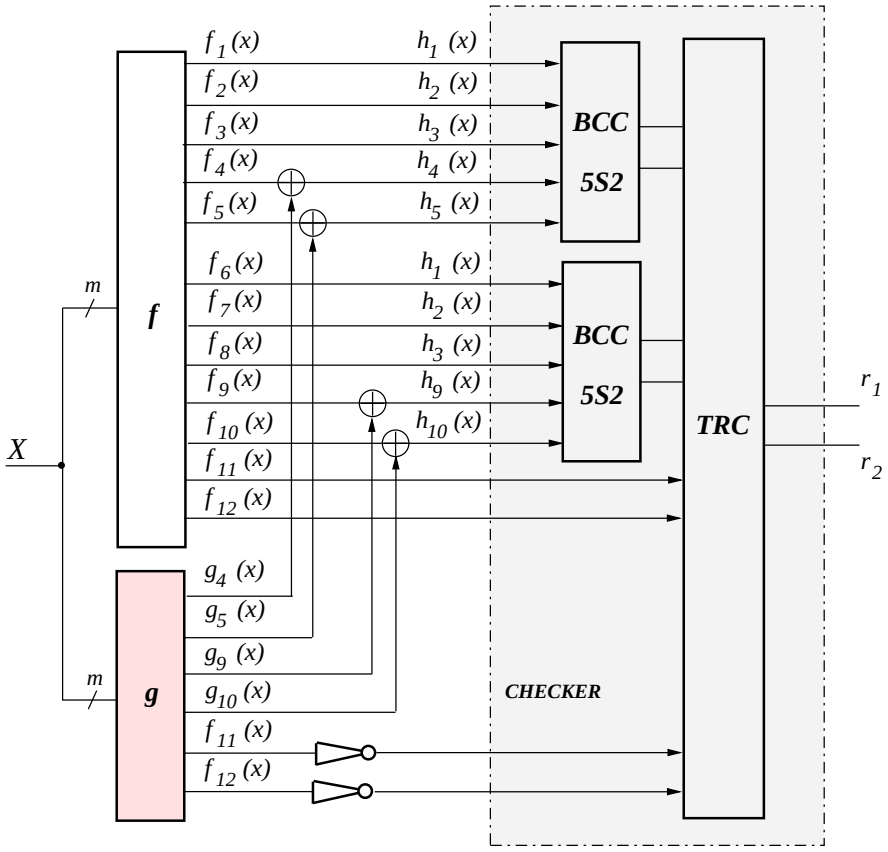

Abbildung 3.7: Neue Methode der Kontrolle für eine kombinatorische Schaltung mit 12 Ausgängen

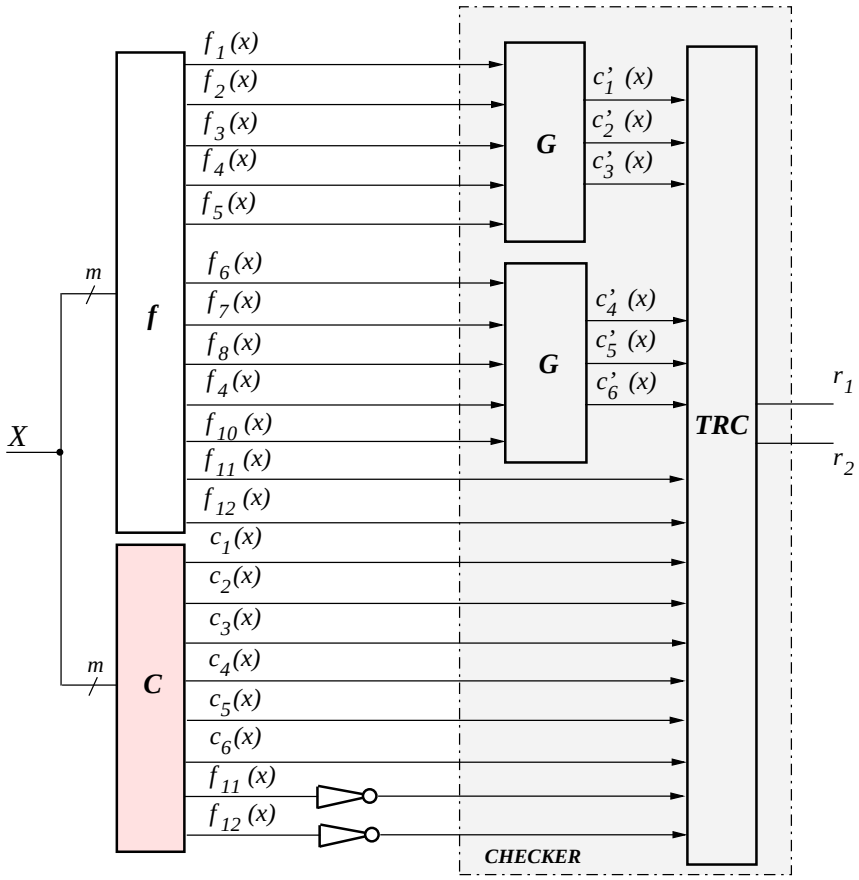

Abbildung 3.8: Funktionale Diagnostik einer kombinatorischen Schaltung mit 12 Ausgängen nach der traditionellen Methode

$X_1^5 = f_1(x), \cdots, f_5(x)$ und $X_2^5 = f_6(x), \cdots, f_{10}(x)$ aufteilt. Für jede Gruppe berechnet der Coder $C(x)$ Kontrollstellen $\{c_1(x), \cdots, c_3(x)\}, \{c_4(x), \cdots, c_6(x)\}$ des 8S3 Berger-Codes. Die Ausgänge $c_7(x)$, $c_8(x)$ des zusätzlichen Blockes $C(x)$ erhalten die invertierten Werte der beiden verbleibenden Ausgänge $f_{11}(x), f_{12}(x)$ der funktionalen Logik. Der Dual-Rail-Checker (*TRC*) vergleicht anschließend die Werte der Generatorausgänge $c'_1(x), \cdots, c'_6(x)$ mit den invertierten Werten $c_1(x), \cdots, c_6(x)$.

In beiden Strukturen der Abbildungen 3.7 und 3.8 besteht der Checker aus zwei Teilen: einem Kontrollbitgenerator und einem Dual-Rail-Checkers. Der Unterschied zwischen den beiden Strukturen ist der Fakt, dass in der traditionellen Methode ein komplizierterer Code verwendet werden muss.

3.4 Suche nach der optimalen Zusammensetzung der Gruppen von Ausgängen. Struktur der komplementären Logik für den Berger-Code

Einer der Nachteile der traditionellen Diagnostik-Methode ist die starke Abhängigkeit der zusätzlichen Schaltung $C(x)$ von der funktionalen Schaltung $f(x)$. Die Struktur des Coders $C(x)$ ist streng

bestimmt, und manchmal existiert nur eine Realisierung. So stimmt bei der Methode „Verdopplung und Vergleich" der zusätzliche Block vollständig mit dem Block der funktionalen Schaltung überein. Bei der Paritätsbitprüfung wird der *Predictor*-Block $P(y)$ nach der Formel 2.2 (aus Abschnitt 2.4.1) gebildet und ist ebenfalls vollständig bestimmt.

Im Gegensatz hierzu kann in der neuen Methode der Logischen Ergänzung die komplementäre Schaltung eine große Anzahl von Realisationsvarianten haben. Jede Funktion $g_1(x), \cdots, g_n(x)$ wird nicht direkt von den Werten der Funktionen $f_1(x)$, $\cdots$, $f_n(x)$ abgeleitet, da der Ausgangsvektor $f_1(x)$, $\cdots$, $f_n(x)$ in einen beliebigen Codevektoren $h_1(x), \cdots, h_n(x)$ umgewandelt werden kann. Wegen der großen Auswahl an Realisationsvarianten findet sich voraussichtlich auch eine optimale Variante. Als Kriterium für die Wahl der Realisationsvariante der komplementären Schaltung wird wieder der minimale Flächebedarf verwendet. Der Flächebedarf einer Schaltung wurde durch die Darstellung der Schaltung mit Hilfe der Standardzellenbibliothek[1] *mcnc.genlib* von *SIS* [75] bestimmt.

Der heuristische Algorithmus zur Bestimmung der Zusammensetzung der Gruppen von Ausgängen X_i^5 für den betrachteten Berger-Code wird im folgenden dargestellt. Die kombinatorische Schaltung wird als Liste logischer Elemente (Gatternetzliste) beschreiben und für jeden Ausgang wird die Komplexität bestimmt.

Algorithmus 3.1:

1. Für die ersten drei Ausgänge $f_{i1}(x)$, $f_{i2}(x)$, $f_{i3}(x)$ jeder Gruppe X_i^5 werden diejenigen drei Ausgänge der funktionalen Schaltung mit maximaler Komplexität gewählt;

2. Für die Auswahl der weiteren Ausgängen jeder Gruppe wird nun die Schaltung $g(x)$ der Abbildung 3.9 konstruiert.

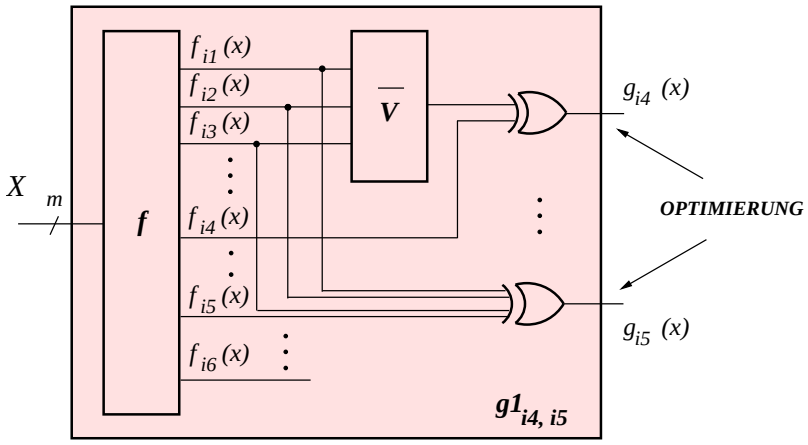

Abbildung 3.9: Die Konstruktion der komplementären Schaltung

Es existieren C_{n-3}^2 Varianten, drei bereits fixierte Ausgänge mit jeweils zwei aus den $(n-3)$ verbliebenen Ausgängen zu kombinieren, um die Schaltung $g(x)$ zu konstruieren. Für jede Variante werden die Funktionen $g_{i4}(x)$ und $g_{i5}(x)$ berechnet, anschließend wird die Schaltung $g(x)$ optimiert nach dem Flächebedarf.

[1] In einer Standardzellenbibliothek ist das funktionale, elektrische und zeitliche Verhalten von Grundelementen gespeichert.

3. Nach der Analyse aller Varianten der Schaltung $g(x)$ wird die Schaltung $g1_{i4,i5}(x)$ mit minimalem Flächebedarf gewählt. Es wird nun eine Schaltung nach der neuen Methode konstruiert, in welcher $h_1(x) = f_1(x), h_2(x) = f_2(x), h_3(x) = f_3(x)$, $h_4(x) = f_4(x) \oplus g_4(x)$, $h_5(x) = f_5(x) \oplus g_5(x)$, die verbliebenen Ausgänge $f_6(x), \cdots, f_n(x)$ sowie ihre invertierten Werte $\overline{f}_6(x)$, $\cdots$, $\overline{f}_n(x)$ die Checkereingänge sind. Die Fläche $A_{vergleich}(g_4, g_5)$ dieser Schaltung wird berechnet ([75]) und mit der benötigten Fläche A_{dc} für die gleiche Schaltung mit n Ausgängen, welche mit der Fehlererkennungsmethode „Verdopplung und Vergleich" realisiert wurde, verglichen. Falls $A_{vergleich}(g_4, g_5) < A_{dc}$, so wird die Gruppe X_i^5 von Ausgängen endgültig fixiert und der Algorithmus für die Suche der nächste Gruppe von Ausgängen X_{i+1}^5 wiederholt.

4. Aus der Menge der verbliebenen Ausgänge $f_6(x), \cdots, f_n(x)$ werden die nachfolgenden drei Ausgänge $f_{i6}(x), f_{i7}(x), f_{i8}(x)$ mit maximaler Komplexität gewählt. Für jedes Paar $f_{i9}(x)$, $f_{i10}(x)$ aus der Menge der ungenutzten Ausgänge $f_9(x), \cdots, f_n(x)$ wird eine Schaltung $g2_{i9,i10}$ mit vier Ausgängen zusammengesetzt, welche in der Abbildung 3.10 zu sehen ist. Die Optimierung dieser Schaltung bezüglich der Ausgänge $g_5(x), g_6(x), g_{i9}(x), g_{i10}(x)$.

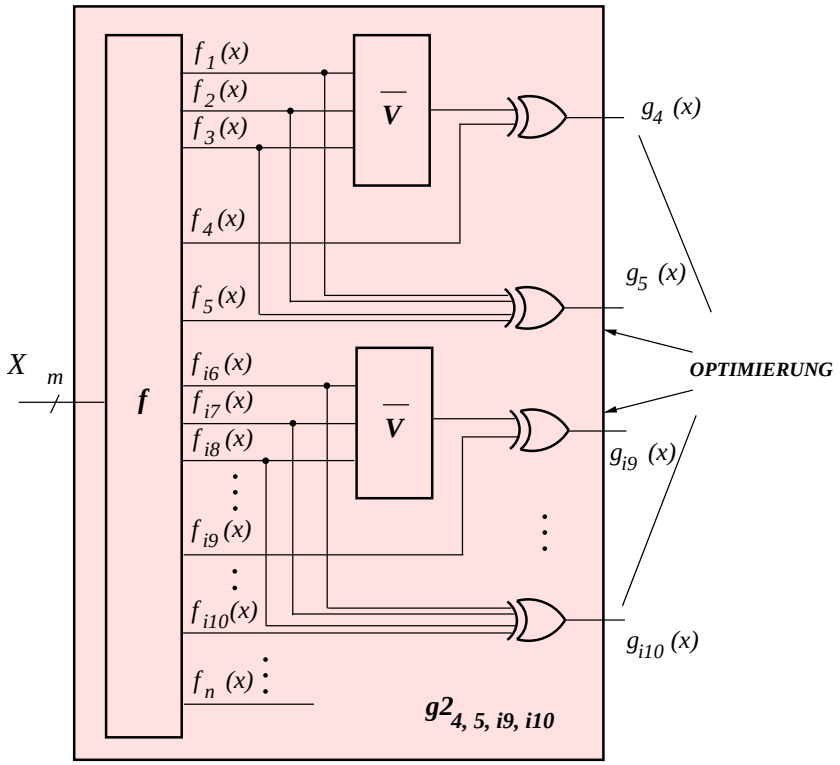

Abbildung 3.10: Die Konstruktion der komplementären Schaltung

5. Als vorläufige Fixierung wird die Schaltung $g2_{5,6,i9,i10}$ mit jenem Paar $f_{i9}(x), f_{i10}(x)$ der Ausgänge gewählt, bei welchem die Fläche $A_{vergleich}(g_4, g_5, g_{i9}, g_{i10})$ minimal ist. Die Fläche $A_{vergleich}(g_4, g_5, g_{i9}, g_{i10})$ dieser Schaltung zuzüglich der invertierten Restausgänge $\overline{f}_{11}(x), \cdots$, $\overline{f}_n(x)$ wird mit der Fläche A_{dc} der Struktur, die für die Realisierung mit der Fehlererkennungs-

methode „Verdopplung und Vergleich" notwendig ist, verglichen. Zusätzlich wird die Fläche $A_{vergleich}(g_4, g_5, g_{i9}, g_{i10})$ mit der Fläche $A_{vergleich}(g_4, g_5)$ verglichen.

Falls $A_{vergleich}(g_4, g_5, g_{i9}, g_{i10}) < A_{dc}$ und $A_{vergleich}(g_4, g_5, g_{i9}, g_{i10}) < A_{vergleich}(g_4, g_5)$, so wird die Gruppe der Ausgänge X^5_{i+1} endgültig fixiert und der Algorithmus für die Suche der nächste Gruppe wiederholt solange die Zahl der verbleibenden Ausgänge mindestens fünf beträgt.

3.5 Experimentelle Ergebnisse

In diesem Abschnitt werden die experimentellen Ergebnisse dargestellt. Alle Ergebnisse in diesem Abschnitt wurden unter Verwendung der MCNC-Benchmark-Schaltungen erhalten [76]. Die Schaltungen wurden vor Beginn der Effektivitätsanalyse der neuen Methode durch *Technology Mapping*[2] in einer Standardzellenbibliothek (*mcnc.genlib* [75]) dargestellt . Dabei wurden keine komplexen Gatter, sondern nur einfache Gattertypen verwendet. Die Effektivität der neuen Methode wird nach zwei Kriterien bewertet:

- nach dem Flächeverbrauch, die für die Realisation der Struktur notwendig ist;
- nach der Wahrscheinlichkeit der Fehlererkennung, die die konstruierte Struktur gewährleistet.

Die Werte dieser Parameter der neuen Methode der Logischen Ergänzung werden mit den entsprechenden Parametern der Fehlererkennungsstrukturen, die nach der traditionellen Methode der Fehlererkennung unter Verwendung des Berger-Codes aufgebaut sind, verglichen. Zusätzlich werden die Schaltungen der neuen Methode bzw. der traditionellen Methode mit Schaltungen, welche nach der Fehlererkennungsmethode „Verdopplung und Vergleich" aufgebaut wurden, bezüglich ihres Flächebedarfs miteinander verglichen.

Mit jeder MCNC-Benchmark-Schaltung wurden zwei Experimente durchgeführt:

Experiment 1 Im ersten Experiment wurde eine Fehlererkennungsschaltung nach der neuen Methode implementiert. In diesem Experiment wurden mit dem Algorithmus 3.1. die Gruppen von Ausgängen der funktionalen Schaltung bestimmt, welche für die Organisation der Kontrolle unter Verwendung des 5S2 Berger-Codes benutzt werden. Dabei wurden für jede MCNC-Benchmark-Schaltung die komplementäre Schaltung $g(x)$ synthesiert, und es wurden die Gruppen von Ausgängen bestimmt, welche die kleinste Komplexität der komplementären Logik $g(x)$ besitzen (Abbildung 3.7).

Experiment 2 Zum Vergleich wurde im zweiten Experiment eine Fehlererkennungsschaltung nach der traditionellen Methode realisiert. Dafür wurden für jede im ersten Experiment gefundene Gruppe von Ausgängen drei Kontrollstellen mit dem 8S3 Berger-Code bestimmt. Der nächste Schritt war die Konstruktionen der Coder-Schaltung $C(x)$, des Kontrollbitgenerators und des Checkers sowie das Zusammenführen aller genannten funktionalen Teile (Abbildung 3.8).

Der Flächebedarf und die Fehlerüberdeckung der Fehlererkennungsschaltungen aus den beiden Experimenten sind in den Tabellen 3.7 und 3.8 dargestellt.

[2] Optimierte Gatternetzliste (Netzwerk von Gattern) (NAND, NOR, MUX, FFs usw.) besteht aus Gattern einer gegebenen Standardzellenbibliothek.

$Circuit$	in/out	A_{or}	X^5	Fläche			$\frac{A_{ges}^{LE}}{A_{ges}^{VV}}, \%$	$\frac{A_{ges}^{LE}}{A_{ges}^{TM}}, \%$	$\overline{\mu}$
				A_g	A_{Ch}	A_{ges}^{LE}			
1	2	3	4	5	6	7	8	9	10
cm42a	4/10	43	2	20	90	153	75,0	67,4	1,0000
cu	14/11	84	2	79	103	266	88,9	86,4	0,9963
pm1	16/13	69	2	75	129	273	92,5	79,6	1,0000
unreg	36/16	152	1	166	182	500	99,8	91,6	0,9997
ldd	9/19	116	3	115	193	424	90,8	86,7	0,8879
ttt2	24/21	270	1	282	247	799	99,7	87,3	0,9997
lal	26/19	147	1	157	221	525	99,2	97,0	0,8692
cm138a	6/8	41	1	25	78	144	81,3	79,1	0,9992
alu2	10/6	485	1	473	52	1010	97,5	95,2	0,7862
count	35/16	225	1	250	182	657	101,7	93,8	1,0000
my_adder	33/17	285	1	309	195	789	101,3	94,7	0,9999
c8	28/18	176	1	197	208	581	101,2	94,6	0,9164
cc	21/20	80	1	99	234	413	101,2	89,9	0,9243
pcler8	21/17	140	1	156	195	491	100,4	94,0	0,9199
tcon	17/16	41	1	62	182	285	102,5	86,3	0,9918
Mittelwert							95,5	88,2	**0,953**

Tabelle 3.7: Experimentelle Ergebnisse. Neue Methode der Fehlererkennung unter Verwendung des Berger-Codes

Die Namen der MCNC-Benchmark-Schaltungen sind in der Spalte 1 aufgelistet. Die Spalten 2 und 3 beschreiben die Eingangs-/ Ausgangs-Anzahl sowie den Flächebedarf A_{orig} der optimierten Schaltungen. Die Optimierung wurde mit Hilfe von *script.rugged (SIS)* durchgeführt. Spalte 4 gibt die Anzahl der Gruppen von Ausgängen $\mid X^5 \mid$ an. In den Spalten 5, 6 und 7 sind der Flächeverbrauch A_g für die komplementäre Schaltung $g(x)$, der Flächeverbrauch für den Checker A_{CH} und der Flächeverbrauch für die Gesamtstruktur A_{ges}^{LE} dargestellt. Die Spalte 8 zeigt das Verhältnis zwischen der Gesamtfläche A_{ges}^{LE} und jener Fläche A_{ges}^{VV}, die für eine Schaltung benötigt wird, welche nach der Methode „Verdopplung und Vergleich" gebaut wird. Für die betrachteten MCNC-Benchmark-Schaltungen ist der durchschnittliche Wert 95,5%. Die Gesamtfläche A_{ges}^{LE} der neuen Fehlererkennungsschaltung ist prozentual bezüglich der Gesamtfläche A_{ges}^{TM} der traditionellen Fehlererkennungsschaltung (Experiment 2) in der nächsten Spalte 9 gegeben. Der mittlere Wert liegt bei 88.2%.

Für die Bestimmung der Fehlererkennungsfähigkeit der neuen Fehlererkennungsschaltungen wird in diesem experimentellen Teil die Wahrscheinlichkeit der Erkennung eines stuck-at-1/0 Fehlers im normalem Modus berechnet. Für die Modellierung dieser Fehlererkennungsfähigkeit wurden pseudozufällige Eingangsvektoren benutzt, und es wurde der folgende Algorithmus durchgeführt.

Für jede untersuchte Schaltung wird die Menge Φ aller möglichen einzelnen stuck-at-1/0 Fehler $\Phi \in \{\varphi_1, \cdots, \varphi_k\}$ bestimmt. Für jeden möglichen Fehler $\varphi_i \in \Phi$ wurden 10 000 pseudozufällige

Eingangsvektoren $X_1(x_1, \cdots, x_m), \cdots, X_{10000}(x_1, \cdots, x_m)$ an die Eingänge $x_1, \cdots, x_m$ der MCNC-Benchmark-Schaltung angelegt.

Mittels Analyse der Ausgabewerte des Berger-Code Checkers wurde für jeden Fehler bestimmt, ob er mit dem gegebenen Eingangsvektor $X(x_1, \cdots, x_m)$ der Gesamtschaltung erkannt wird. Folgende Gleichung definiert die Wahrscheinlichkeit $\mu(\varphi)$ (in Prozent), dass ein Fehler $\varphi_i \in \Phi$ an den Ausgängen der Schaltung nicht erkannt wird:

$$\mu(\varphi_i) = \frac{N(\varphi_i) - n(\varphi_i)}{N(\varphi_i)} \bullet 100\%. \tag{3.7}$$

Hierbei ist $N(\varphi_i)$ die Anzahl der Eingangsvektoren $X(x_1, \cdots, x_m)$, welche mindestens an einem der Ausgänge $f_1(x), \cdots, f_n(x)$ der funktionalen Schaltung f bei Vorhandensein des Fehlers φ_i einen inkorrekten Wert liefern. $n(\varphi_i)$ ist die Anzahl der Eingangsvektoren, für die der Fehler durch einen Berger-Code Checker erkannt wird.

Der Mittelwert $\overline{\mu}$ der Wahrscheinlichkeit $\mu(\varphi_i)$ für alle möglichen stuck-at-1/0 Fehler in den untersuchten MCNC-Benchmark-Schaltungen wird folgendermaßen bestimmt:

$$\overline{\mu} = \frac{1}{R} \sum_{i=1}^{R} \mu(\varphi_i). \tag{3.8}$$

Die mittlere Wahrscheinlichkeit der Fehlererkennung $\overline{\mu}$ beträgt in der neuen Fehlererkennungsschaltung unter Verwendung des Berger-Codes 0,953 (Spalte 10).

Neben der Bestimmung der Wahrscheinlichkeit zeigt die Analyse der Experimente, dass der Gesamtflächeverbrauch der neuen Fehlererkennungsschaltungen zur Anzahl der bestimmten Gruppen von Ausgängen proportional ist. Die besten Ergebnisse liefern die Fälle, in denen die Anzahl der Gruppen von Ausgängen (fünf Ausgänge pro Gruppe) maximal ist (*cm42a, cu, pm1, cm138a, ldd*). Für die verwendeten MCNC-Benchmark-Schaltungen liegt der mittlere Flächebedarf um 4,5% unter dem Flächebedarf für die entsprechenden Fehlererkennungsschaltungen unter Verwendung der Methode „Verdopplung und Vergleich" und um 11,8% unter dem Flächebedarf der traditionellen Fehlererkennungsschaltungen.

Tabelle 3.8 zeigt die experimentellen Ergebnisse für das zweite Experiment. In dieser Tabelle sind die untersuchten Parameter für die traditionelle Kontroll-Methode unter Verwendung des Berger-Codes bestimmt. Die Spalten 1, 2, 3 und 4 der Tabelle 3.8 entsprechen den Spalten 1, 4, 6 und 7 der Tabelle 3.7. Die Spalte 5 stellt das Verhältnis des Flächebedarfs der jeweiligen Gesamtschaltung A_{ges}^{TM} zum Flächebedarf A_{ges}^{VV} einer Fehlererkennungsschaltung nach der Methode der Schaltungsverdopplung und des Vergleichs der Ausgänge in Prozent dar. Der durchschnittliche Wert liegt bei 107.5%. Die Fläche A_{ges}^{TM} der traditionellen Gesamtschaltung prozentual zur notwendigen Fläche A_{ges}^{LE} der neuen Fehlererkennungsschaltung und die Fehlererkennungswahrscheinlichkeit $\overline{\mu}$ für die traditionellen Schaltungen sind in den Spalten 6, 7 dargestellt. Für die traditionelle Kontrolle der Schaltungen ist 12,9% mehr Flächeaufwand erforderlich, die durchschnittliche Fehlererkennungswahrscheinlichkeit liegt bei $\overline{\mu} = 0.900$.

Die experimentellen Untersuchungen zeigen, dass die Fehlererkennungswahrscheinlichkeit der neuen Methode der Logischen Ergänzung um 5% höher liegt als bei der traditionellen Methode. Zusätzlich verringern sich die Implementierungskosten bei der Realisation der neuen Methode um 12%.

$Circuit$	Fläche			$\frac{A_{ges}^{TM}}{A_{ges}^{VV}}\%$	$\frac{A_{ges}^{TM}}{A_{ges}^{LE}}\%$	$\overline{\mu}$
	A_{coder}	A_{Ch}	A_{ges}^{LE}			
1	2	3	4	5	6	7
cm42a	22	162	227	111,3	148,4	1,000
cu	49	175	308	103,0	115,8	0,6561
pm1	73	201	343	116,3	118,7	0,9623
unreg	176	218	546	109,0	109,2	0,9284
ldd	72	301	489	104,7	115,3	0,8724
ttt2	262	283	815	101,7	102,0	0,9483
lal	137	257	541	102,3	103,1	1,0000
cm138a	27	114	182	102,8	126,4	0,9983
alu2	477	88	1050	101,3	104,0	0,7265
count	257	218	700	108,4	106,5	0,9126
my_adder	317	231	833	106,9	105,6	0,8074
c8	194	244	614	107,0	105,7	1,0000
cc	109	270	459	112,5	111,1	0,8164
pcler8	151	231	522	106,7	106,3	0,8725
tcon	71	218	330	118,7	115,8	0,9941
Mittelwert				107,5	112,9	**0,900**

Tabelle 3.8: Experimentelle Ergebnisse. Traditionelle Methode der Fehlererkennung

3.6 Die Benutzung des 1-aus-3 Codes in der neuen Fehlererkennungsmethode der Logischen Ergänzung

In dem Schema der Logischen Ergänzung unter Verwendung separierbarer Codes werden nur die Leitungen, welche Kontrollstellen realisieren, ergänzt. Im Falle der Benutzung systematischer nicht-separierbarer Codes benötigt die Implementierung der neuen Fehlererkennungsmethode eine Modifikation der Struktur der komplementären Schaltung und eine Veränderung des Algorithmus zur Bestimmung der Gruppen von Ausgängen. Da es nicht möglich ist, Codewörter nicht-separierbarer Codes in Kontroll- und Informationsstellen aufzuteilen, kann bei der Ergänzung der originalen Schaltungen ein beliebiger Ausgang oder eine beliebige Menge von Ausgängen ergänzt werden, um Codewörter des gewählten Codes zu erhalten. Im folgenden Abschnitt der Arbeit wird aufgezeigt, wie das Prinzip der Logischen Ergänzung für die Schaltungskontrolle unter Verwendung des 1-aus-3 Codes wirksam verwirklicht wird. Zusätzlich wurde mit dem Ziel der Verringerung des Flächeaufwands ein Experiment durchgeführt, welches die Effektivität der Verwendung von Invertoren an den Schaltungsausgängen bestimmt.

In Abbildung 3.11 ist das Prinzip der Überwachung einer Schaltung f im laufenden Betrieb durch eine komplementäre Schaltung g unter Verwendung von Invertoren an den Schaltungsausgängen dargestellt [77]. In dieser Struktur hat die funktionale Schaltung m Eingänge $x_1, \cdots, x_m$ und n Ausgänge

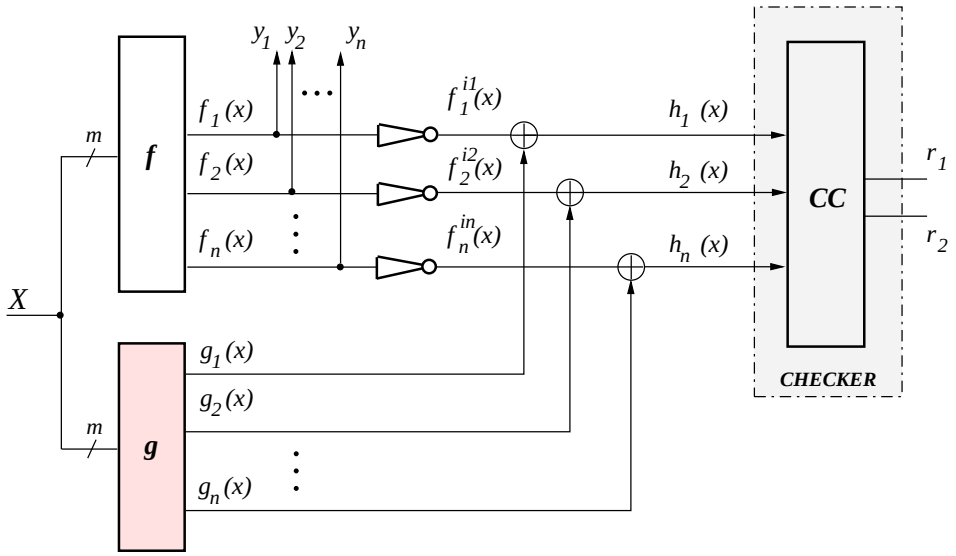

Abbildung 3.11: Die prinzipielle Struktur der Methode der Logischen Ergänzung

$y_1, \cdots, y_n$. Die funktionellen Ausgänge $y_1, \cdots, y_n$ realisieren Boolesche Funktionen $f_1(x)$, $\cdots$, $f_n(x)$. Die komplementäre Schaltung g mit m Eingängen $x_1, \cdots, x_m$ realisiert an ihren n Ausgängen die komplementären Funktionen $g_1(x), \cdots, g_n(x)$. Die Ausgangsleitungen der Schaltungen f und g werden auf solche Weise verbunden, dass die komponentenweise XOR-Summe

$$f_1(x) \oplus g_1(x), \cdots, f_n(x) \oplus g_n(x)$$

die Elemente des betrachteten 1-aus-3 Codes ergeben. Die Fragezeichen über den NOT-Gattern in Abbildung 3.11 bedeuten, dass nicht jeder Ausgang invertiert wird. Das heißt, in der Realität werden nur einige Ausgänge durch NOT-Gatter erweitert. Für die Bestimmung dieser Menge von Ausgängen wurde ein spezieller Algorithmus ausgearbeitet.

Es wird nun ein Beispiel einer kombinatorischen Schaltung $B_f(x)$ mit drei Ausgängen untersucht, die die folgenden Booleschen Funktionen realisieren:

$$\begin{aligned} y_1 = f_1(x) = \overline{x}_1 \vee x_2, y_2 = f_2(x) = \overline{x}_1 x_2 \vee \overline{x}_2 \overline{x}_3, \\ y_3 = f_3(x) = \overline{x}_1 \overline{x}_2 \overline{x}_3 \vee x_1 x_2 \vee x_1 \overline{x}_3. \end{aligned} \tag{3.9}$$

An diesem elementaren Beispiel wird der Design-Algorithmus zur Konstruktion einer vollständig selbstprüfenden Schaltung unter Verwendung des nicht-separierbaren 1-aus-3 Codes erläutert. Zunächst wird die Struktur der komplementären Schaltung $g(x)$ untersucht. Parallel werden die Ausgänge bestimmt, an welchen zur Verringerung des Flächebedarfs Invertoren installiert werden müssen.

Die Abbildung 3.12 zeigt, dass die komplementäre Schaltung $g(x)$ mit zwei Ausgängen durch Modifikation und Optimierung der ursprünglichen Schaltung $f(x)$ entsteht. Die Ausgänge $g_{i2}(x), g_{i3}(x)$ der Schaltung $g(x)$ implementieren die folgenden Funktionen:

$$\begin{aligned} g_{i2}(x) = f_{i1}(x) f_{i2}(x), \\ g_{i3}(x) = f_{i3}(x)\Big(f_{i1}(x) \vee f_{i2}(x)\Big) \vee \overline{f}_{i1}(x) \overline{f}_{i2}(x) \overline{f}_{i3}(x). \end{aligned} \tag{3.10}$$

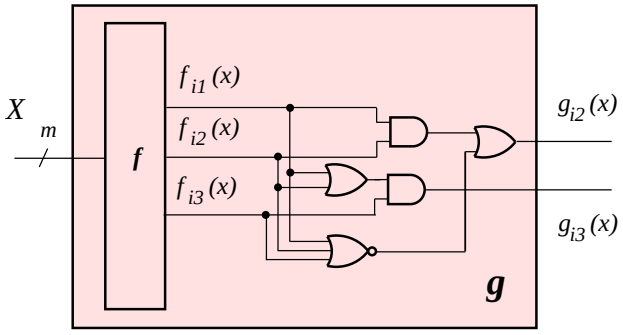

Abbildung 3.12: Das prinzipielle Schema der Logischen Ergänzung für den 1-aus-3 Code

Eine Besonderheit der vorgestellten komplemenären Schaltung ist die folgende Eigenschaft. Der Ausgang des NOR-Gatters, welches die Schaltungsausgänge $\overline{f}_1(x)\overline{f}_2(x)\overline{f}_3(x)$ miteinander verbindet, kann durch das OR-Gatter mit einem beliebigen der beiden AND-Gatter verbunden werden. In der Abbildung 3.12 ist das NOR-Gatter mit dem Ausgang $g_{i2}(x)$ verbunden, hätte aber ebenso mit dem Ausgang $g_{i3}(x)$ verbunden sein können. In jedem Fall entstehen 1-aus-3 Codewörter an den Leitungen $h_1(x), h_2(x)$ und $h_3(x)$.

Für das Beispiel der betrachteten Schaltung $B_f(x)$ sind die Ausgänge der komplementäre Schaltung $g(x)$:

$$g_2(x) = \overline{x}_1(x_2 \vee \overline{x}_3),\ g_3(x) = \overline{x}_2 x_3 \vee x_1 x_2 \vee x_1 \overline{x}_3. \tag{3.11}$$

Die Realisation der neuen Methode der Logischen Ergänzung für die betrachtete Schaltung mit drei Ausgängen ist in Abbildung 3.13 zu sehen. Aus der Abbildung 3.13 folgt, dass der erste Ausgang $f_1(x)$ der Schaltung unverändert bleibt bzw. bei Verwendung eines Negators invertiert wird. Der zweite und dritte funktionale Ausgang $f_2(x), f_3(x)$ der Schaltung $B_f(x)$ wird durch XOR-Gatter mit Ausgängen $g_2(x), g_3(x)$ der Schaltung $g(x)$ auf solche Weise verbunden, dass auf den $h_1(x), h_2(x)$ und $h_3(x)$-Leitungen Wörter des 1-aus-3 Codes erzeugt werden. In der Tabelle 3.9 wird deutlich, dass unter Anwendung der Formeln 3.10 an den Checkereingängen nur Wörter des betrachtetes 1-aus-3 Codes erscheinen.

x_1	x_2	x_3	$f_1(x)$	$f_2(x)$	$f_3(x)$	$g_1(x)$	$g_2(x)$	$g_3(x)$	$h_1(x)$	$h_2(x)$	$h_3(x)$
0	0	0	1	1	0	0	1	0	1	0	0
0	0	1	1	0	1	0	0	1	1	0	0
0	1	0	1	1	0	0	1	0	1	0	0
0	1	1	1	1	0	0	1	0	1	0	0
1	0	0	0	1	1	0	0	1	0	1	0
1	0	1	0	0	0	0	0	1	0	0	1
1	1	0	1	0	1	0	0	1	1	0	0
1	1	1	1	0	1	0	0	1	1	0	0

Tabelle 3.9: Wertetabelle für die fehlererkennende Schaltung aus Abbildung 3.13

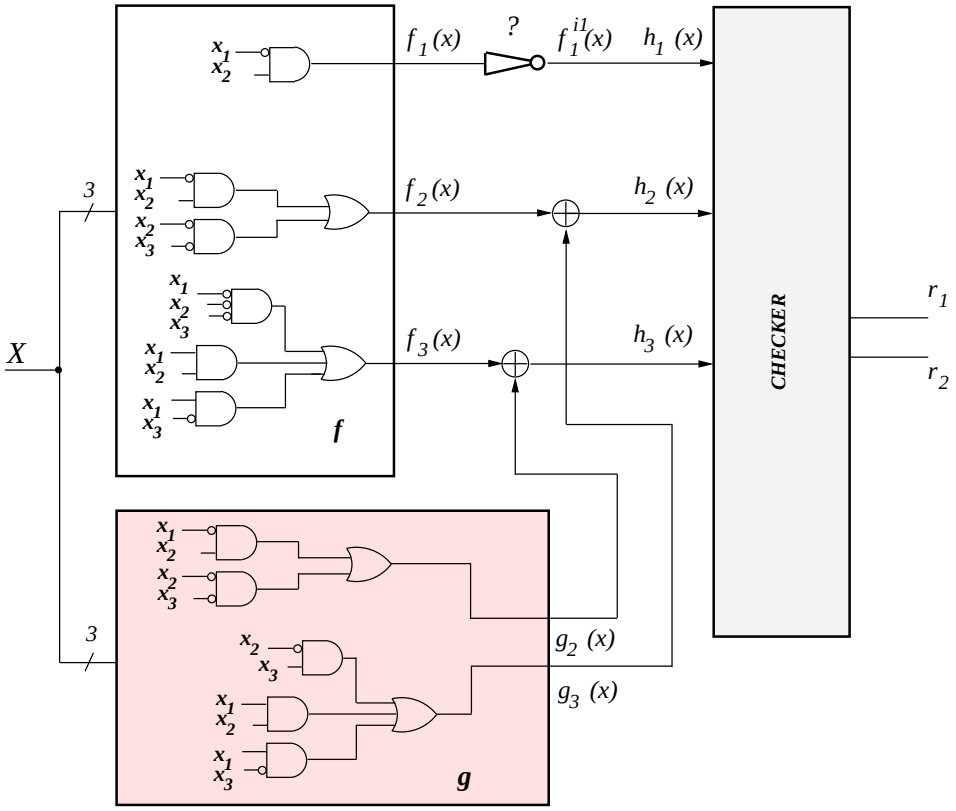

Abbildung 3.13: Das prinzipielle Schema der Methode der Logischen Ergänzung für den 1-aus-3 Code

In diesem Abschnitt wird die Grundlage für das Erhalten der minimale Realisationsfläche einer Fehlererkennungsschaltung unter Verwendung des 1-aus-3 Codes aufgestellt. Im experimentellen Teil dieses Dissertation wird gezeigt, dass die Realisationsfläche um so kleiner ausfällt, je mehr Nullen die Funktion $f_i(x)$ enthält.

Annahme:
Die Komplexität der Booleschen Funktion ist umgekehrt proportional zur Einsenanzahl N_i^1 in der Wertetabelle der Funktion $f_i(x)$ mit m pseudozufälligen Eingangsvektoren $x_1, \cdots, x_m$.

Zur Vereinfachung einer kombinatorischen Schaltung muss bei großer Einsenanzahl einer Funktion in der Wertetabelle der dazugehörige Ausgang y_i invertiert werden. Um den Flächebedarf für die Realisation der komplementären Funktion $g_2(x) = f_1(x) \wedge f_2(x)$ zu minimieren, wird die Einsenanzahl $N_{g_2}^1$ dieser Funktion ausgerechnet. Für den Fall, dass die Einsenanzahl größer ist als die Nullenanzahl $N_{g_2}^1 > N_{g_2}^0$, werden die Ausgänge $f_{i1}(x), f_{i2}(x)$ invertiert. Die Einsen-/Nullenanzahlen der Funktionen knnen bei kleiner Funktionsanzahl in der kombinatorischen Schaltung mit Hilfe der Wertetabelle leicht bestimmt werden. Bei großer Funktionsanzahl kann für diese Aufgabe eine Schaltung mit n Funktionen mit pseudozufälligen Eingangsvektoren simuliert werden.

Die Effektivität des Prinzips der Zählung der Einsen und der Invertierung der Ausgänge im Falle von

$N^1_{g_2} > N^0_{g_2}$, wird an der obigen Beispielschaltung $B_f(x)$ mit drei Ausgängen vorgeführt. Dazu werden für einen Ausgang $y_i^{k_i} = f_i^{k_i}(x), k_i = \{0, 1\}$ die folgenden Bezeichnungen verwendet:

- $y_i^0 = \overline{y}_i = \overline{f}_i(x)$ - invertierter Ausgang;
- $y_i^1 = y_i = f_i(x)$ - nicht invertierter Ausgang.

Für den ersten Fall wird die Boolesche Funktion $f_i^0 = \overline{f}_i(x)$ realisiert. Analog wird im zweiten Fall an einem nicht invertierten Ausgang die Boolesche Funktion ohne Invertierung $f_i^1 = f_i(x)$ verwirklicht. Die Einsenanzahl für $y_1 = f_1(x)$ in der Tabelle 3.10 ist gleich sechs.

x_1	x_2	x_3	$f_1(x)$	$f_2(x)$	$f_3(x)$	$g_2(x)$	$g_3(x)$	$\overline{f}_1(x)$	$\hat{g}_2(x)$	$\hat{g}_3(x)$
0	0	0	1	1	0	1	0	0	0	0
0	0	1	1	0	1	0	1	0	0	0
0	1	0	1	1	0	1	0	0	0	0
0	1	1	1	1	0	1	0	0	0	0
1	0	0	0	1	1	0	1	1	1	1
1	0	1	0	0	0	0	0	1	0	0
1	1	0	1	0	1	0	1	0	0	0
1	1	1	1	0	1	0	1	0	0	0

Tabelle 3.10: Wertetabelle für die fehlererkennende Schaltung aus Abbildung 3.13 b) mit einem invertiertem Ausgang

Bei Invertierung des ersten Ausganges $\overline{y}_1 = \overline{f}_1(x)$ (Abbildung 3.13 b)) verringert sich die Einsenanzahl von sechs auf zwei. Für die verbleibenden Ausgänge $f_2(x)$ und $f_3(x)$ ist die Einsenanzahl gleich der Nullenanzahl und folglich werden diese Schaltungsausgänge nicht invertiert. Nach Invertieren des Ausganges $f_1(x)$ wird die komplementäre Schaltung mit $\hat{g}(x)$ bezeichnet, ihre zwei Ausgänge mit $\hat{g}_2(x), \hat{g}_3(x)$. Sie verwirklichen die folgenden Funktionen:

$$\hat{g}_2(x) = \overline{f}_1(x) \wedge f_2(x) = x_1\overline{x}_2\overline{x}_3,$$
$$\hat{g}_3(x) = (\overline{f}_1(x) \vee f_2(x))f_3(x) \vee f_1(x)\overline{f}_2(x)\overline{f}_3(x) = x_1\overline{x}_2\overline{x}_3 = \hat{g}_2(x). \quad (3.12)$$

Aus der gegebenen Formeln folgt, dass bei Invertierung des Ausgang $f_1(x)$ die Funktionen $\hat{g}_2(x)$ und $\hat{g}_3(x)$ überein stimmen. Die komplementäre Schaltung muss nur $\hat{g}_2(x)$ implementieren und folglich ist der Flächebedarf der erhaltenen Schaltung $\hat{g}(x)$ bedeutend geringer als der Flächebedarf der Schaltung $g(x)$.

3.7 Bestimmung der Zusammensetzung der Gruppen von Ausgängen

Im Abschnitt 3.3 wurde festgestellt, dass bei Verwendung des Berger-Code eine Abhängigkeit zwischen dem Flächebedarf und der Zusammensetzung der Gruppen von Ausgängen besteht. Eine allgemeine Idee zur Erreichung der minimalen Fläche der komplementären Schaltung $g(x)$ ist daher das Streben nach Verbrauch einer kleineren Fläche für die Ausgänge als für die Ausgänge der originalen Schaltung $f(x)$.

Bei der Arbeit mit dem 1-aus-3 Code wird die Suche nach den optimalen Ausgängen und die Fixierung der Gruppen von Ausgängen durch die Simulierungsergebnisse bestimmt. Wie im Abschnitt 3.4 erläutert, ist die Anzahl der Ausgänge jeder Gruppe X_i^s bereits festgelegt, und zwar soll die Ausgangsanzahl gleich der Codewortlänge des gewählten Codes sein. So ist bei dem 1-aus-3 Code die Anzahl der funktionalen Ausgänge je Gruppe X_i^3 gleich drei. Dafür werden die n Ausgänge der Schaltung $f(x)$ in $i = n/3$ unabhängige Gruppen aufgeteilt, wobei eine Gruppe X_{rest} restlicher Ausgänge verbleibt. Die Ausgänge dieser Restmenge werden invertiert.

Also sind für jede Gruppe X_i^3 von Ausgängen der funktionalen Logik durch die komplementäre Logik zwei komplementäre Ausgänge gegeben:

$$g_{k_2}(x) = f_{k_1}^{j_{k_1}}(x) f_{k_2}^{j_{k_2}}(x) \vee \overline{f}_{k_1}^{j_{k_1}}(x) \overline{f}_{k_2}^{j_{k_2}}(x) \overline{f}_{k_3}^{j_{k_3}}(x)$$

und

$$g_{k_3}(x) = \left(f_{k_1}^{j_{k_1}}(x) \vee f_{k_2}^{j_{k_2}}(x)\right) f_{k_3}^{j_{k_3}}(x)$$

und jedes Tripel

$$h_{k_1}(x) = f_{k_1}^{j_{k_1}}(x),\ h_{k_2}(x) = f_{k_2}^{j_{k_2}}(x) \oplus g_{k_2}(x),\ h_{k_3}(x) = f_{k_3}^{j_{k_3}}(x) \oplus g_{k_3}(x)$$

ist Element des 1-aus-3 Codes bei Abwesenheit von Fehlern.

Zuerst werden diejenigen Ausgänge bestimmt, welche invertiert werden müssen. Dafür wird vor der Durchführung des Algorithmus zur Bestimmung der Gruppen von Ausgängen die Schaltung $f(x)$ mit 10 000 pseudozufälligen Eingangsvektoren simuliert. Für den Fall, dass für diese pseudozufälligen Eingangsvektoren am Ausgang $y_i^{k_i} = f_i^{k_i}(x)$ häufiger Einsen anliegen als Nullen, so wird dieser Ausgang invertiert.

Im folgenden wird der Algorithmus 3.2 zur Bestimmung der Gruppen von Ausgängen für den 1-aus-3 Code beschrieben.

Algorithmus 3.2:

1. Die Schaltung $f(x)$ wird als Gatternetzliste dargestellt;
2. Für jeden Ausgang $f_j(x)$ der betrachteten Schaltung wird seine Komplexität bestimmt;
3. Als erster Ausgang $f_{k_1}^{j_{k_1}}(x)$ der Gruppe von Ausgängen X_1^3 wird derjenige Ausgang der Ausgangsmenge mit maximaler Komplexität gewählt. Dieser Ausgang wird nicht durch die komplementäre Funktion $g_{k_1}(x)$ der Schaltung $g(x)$ ergänzt;
4. Als zweiter Ausgang $f_{k_2}^{j_{k_2}}(x)$ der Gruppe von Ausgängen X_1^3 wird derjenige Ausgang der noch ungenutzten Ausgänge genommen, für den nach der Simulierung der Schaltung mit 10 000 pseudozufälligen Eingangsvektoren die Eisenanzahl für die Funktion $f_{k_1}^{j_{k_1}}(x) \wedge f_{k_2}^{j_{k_2}}(x)$ minimal ist. Für diesen Ausgang ist die Nullenanzahl der komplementären Funktion $g_{k_2}(x)$ maximal, und so hat die Realisierung von $g_{k_2}(x)$ mit großer Wahrscheinlichkeit eine kleine Fläche;
5. Die Suche nach dem dritten Ausgang $f_{k_3}^{j_{k_3}}(x)$ ist in jeder Gruppe von Ausgängen X_i^3 identisch. Aus der verbliebenen Menge der Ausgänge wird derjenige Ausgang bestimmt, für den nach der Simulierung der Schaltung mit 10 000 pseudozufälligen Eingangsvektoren die Einsenanzahl für die Funktion $\left(f_{k_1}^{j_{k_1}}(x) \vee f_{k_2}^{j_{k_2}}(x)\right) f_{k_3}^{j_{k_3}}(x)$ minimal ist. Auf diese Weise ist der Flächebedarf von $g_{k_3}(x)$ wegen der maximalen Nullenanzahl vermutlich minimal;

6. Dieser Algorithmus wird für jede Gruppe von Ausgängen X_i^3 solange wiederholt, bis die Menge der ungenutzten Ausgänge weniger als drei Elementen enthält.

3.8 Experimentelle Ergebnisse

Experimentell wurden MCNC-Benchmark-Schaltungen untersucht, die durch die neue Methode der Logischen Ergänzung in selbstprüfende Schaltungen transformiert wurden. Die Ergebnisse der strukturellen und funktionellen Analyse dieser kombinatorischen MCNC-Benchmark-Schaltungen werden im folgenden vorgestellt. Durch Bewertung der erforderlichen Fläche werden die Kosten für den Entwurf einer selbstprüfenden Schaltung unter Verwendung des 1-aus-3 Codes eingeschätzt. Zusätzlich wird die Fehlererkennungsfähigkeit der untersuchten Strukturen berechnet.

In Tabelle 3.8 wird für 13 MCNC-Benchmark-Schaltungen die Fehlererkennungswahrscheinlichkeit $\mu(\varphi)$ sowie die benötigte Fläche A für die gesamte Kontrollschaltung dargestellt. In den Spalten 1 und 2 sind Name und Anzahl der Ein-/Ausgänge der betrachteten Schaltung zu finden. In den Spalten 3 und 4 stehen der Flächebedarf der nicht optimierten A und der optimierten A_{optim} funktionalen Schaltungen. Die folgenden sechs Spalten 5-10 stellen die Parameter der neuen selbstprüfenden Schaltungen dar. Die Spalte 5 beschreibt die Anzahl der Gruppen von Ausgängen $\mid X^3 \mid$, die unter Verwendung des Algorithmus 3.2 gefunden wurden, Spalte 6 den Flächebedarf A_g für die komplementäre Schaltung $g(x)$ und Spalte 7 die Anzahl der eingerichteten NOT-Elemente an den Ausgängen der Schaltung. Die

				LOGISCHE ERGÄNZUNG						VERDOPPL. & VERGL.				
$Circuit$	in/out	A	A_{optim}	$\mid X^3 \mid$	A_g	$\mid N_i \mid$	A_{CH}	$A_{x/i}$	A_{ges}^{LE}	A_{CH}	A_i	A_{ges}^{VV}	$\overline{\mu}$ (%)	$\frac{A_{des}^{LE}}{A_{ges}^{VV}}$
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
cht	47/36	419	262	12	257	0	322	48	1046	525	36	1399	97,83	0,7476
unreg	36/16	209	171	5	115	0	154	41	519	255	16	689	98,60	0,7532
cu	14/11	105	86	3	52	1	100	27	284	150	11	371	99,89	0,7654
sct	19/15	205	117	5	87	7	133	47	472	210	15	635	99,95	0,7433
ttt2	24/21	442	300	7	235	0	187	56	920	300	21	1205	96,51	0,7634
tcon	17/16	73	49	5	40	5	154	46	313	225	16	387	99,90	0,8087
term1	34/10	797	230	3	191	2	100	27	1115	135	10	1739	93,10	0,6411
count	35/16	239	210	5	117	16	154	57	567	225	16	719	96,30	0,7885
c8	28/18	328	214	6	188	5	133	53	702	255	18	929	99,74	0,7556
lal	26/19	249	139	3	114	9	148	43	554	270	19	787	99,32	0,7039
pm1	16/13	85	66	4	34	7	121	40	280	180	13	363	94,21	0,7124
vda	17/39	1593	810	13	695	1	349	105	2742	570	39	3795	98,46	0,7225
x1	51/35	661	456	11	392	12	331	102	1486	510	35	1867	97,00	0,7950
Mittelwert													96,78	0,7462

Tabelle 3.11: Experimentelle Ergebnisse. Neue Methode der Fehlererkennung unter Verwendung des 1-aus-3 Codes

Spalten 8, 9 und 10 enthalten die folgenden Werte: die Fläche A_{CH}, die für die Implementierung des 1-aus-3 Code Checkers benötigt wird, die Fläche der installierten XOR/NOT-Gatter $A_{x/i}$ und die Fläche A_{ges}^{LE} der gesamten selbstprüfenden Schaltung. Die Spalten 11 und 13 stellen die gleichen Werte wie 8 und 10 dar, jedoch bei Realisation mit der Fehlererkennungsmethode „Verdopplung und Vergleich". Für diese Fehlererkennungsmethode ist in Spalte 12 der Flächeverbrauch der NOT-Elemente zu finden. In Spalte 14 sind für alle untersuchten MCNC-Benchmark-Schaltungen die Wahrscheinlichkeit $\overline{\mu}$ der Erkennung von stuck-at-1/0 Fehlern im normalen Modus unter Verwendung komplementärer Schaltungen und 1-aus-3 Codes gegeben. Bei den betrachteten Schaltungen liegt dieser Wert durchschnittlich bei 96,78%. Für sechs dieser Schaltungen beträgt dieser Wert mehr als 99%, das sind 46% der unter-

suchten Schaltungen. Die Spalte 15 beschreibt den relativen Flächebedarf der neuen selbstprüfenden Schaltungen A_{ges}^{LE} bezüglich der erforderlichen Fläche A_{ges}^{VV} bei „Verdopplung und Vergleich". Die benötigte Fläche bei Verwendung der neuen Methode der Logischen Ergänzung ist bis zu 35% geringer (bei *term1*) als bei „Verdopplung und Vergleich". Durchschnittlich beträgt der Flächeverbrauch der neuen selbstprüfenden Schaltung 74,62% des Flächebedarfs einer verdoppelten Schaltung.

3.9 Zusammenfassung

In diesem Kapitel wurde die neue Methode für die Kontrolle kombinatorischer Schaltungen vorgestellt. Die neue Methode der Logische Ergänzung gehört zu den Methoden der funktionalen Diagnostik, ebenso wie die Methode der Paritätsbitprüfung, die Methode „Verdopplung und Vergleich" oder die traditionelle Methode der Schaltungskontrolle unter Verwendung systematischer (nicht-systematischer) Codes. In der Gegenwart basieren alle Methoden der funktionalen Diagnostik auf der Verwendung zusätzlicher Funktionen. Deshalb ist es zweckmäßig, solche Methoden von dem Gesichtspunkt aus zu entwerfen, dass die zusätzliche Logik den kleinstmöglichen Flächebedarf erfordert.

Auch die neue Methode ist auf der Synthese zusätzlicher Logik gegründet, dank derer die Kontrolle der funktionalen Logik gewährleistet. In diesem Kapitel wurde am Beispiel des Berger-Codes und des 1-aus-3 Codes die Effektivität der neuen Methode der logischen Ergänzung bewiesen. Im Vergleich zu den anderen Methoden erfordert die neue Methode einen kleineren Flächebedarf in der Realisation der komplementären Logik. Das wesentliche Prinzip der Methode der Logischen Ergänzung ist die Ergänzung der Gruppen der funktionalen Schaltungsausgänge, so dass an den ergänzten Ausgängen ein bestimmter beliebiger fehlererkennender Code entsteht. Eine große Teil dieses Kapitels war die Suche nach einem optimalen Algorithmus zur Bildung der Gruppen von Ausgängen. Es wurde aufgezeigt, dass die Gruppenzusammenstellung den Optimierungsgrad der zusätzlichen Logik entscheidend beeinflusst. Aus diesem Grunde wurden für die Fixierung der Gruppenausgänge verschiedene Kriterien untersucht.

Im Folgenden werden heuristische Algorithmen genannt, welche in der Arbeit nicht untersucht wurden, jedoch den Optimierungsgrad der komplementären Logik bei Verwendung der neuen Methode weiter steigern können:

- Das Auffinden solcher Gruppen von Ausgängen X_i^s, so dass die Zahl der nicht ergänzten Funktionen maximiert wird. Beispielsweise besteht die Aufgabe für den 1-aus-3 Code in der Suche solcher Gruppen von Ausgängen, für die mehr als eine der Ergänzungsfunktionen $g_{i1}(x)$ gleich Null ist;

- Die Benutzung einer Funktion $g_{ij}(x)$ der Schaltung $g(x)$ für die gleichzeitige Logische Ergänzung von Ausgängen verschiedener Gruppen;

- Ausnutzung von Abhängigkeiten der funktionalen Ausgänge von den Eingangskombinationen $X = x_1, \cdots, x_m$ für die Konstruktion der komplementären Logik.

4 Optimierung der komplementären Schaltungen unter Verwendung der Eigenschaften von *Don't-Cares*

In diesem Dissertationsabschnitt wird ein Algorithmus zur Synthese der komplementären Logik unter Benutzung partiell unbestimmter Werte von Booleschen Funktionen entwickelt. Es wird eine Antwort auf die Frage gefunden, wie eine Verbesserung des Flächebedarfs der Komplementärschaltung unter Verwendung der Eigenschaften von partiell unbestimmten Werte möglich ist. Ebenso werden effektive Algorithmen zur Bestimmung der optimalen Gruppen von Ausgängen und zur endgültigen Fixierung der Gruppen von Ausgängen für die Implementierung der komplementären Logik mit minimalem Flächeverbrauch entwickelt. Inbesondere wird der 1-aus-3 Code ausführlich untersucht.

4.1 Traditionelle *Don't-Care* Bedingungen in digitalen Systemen

Bei vorliegenden praktischen Problemstellungen für kombinatorische Schaltkreise sind nicht für jede Eingangskombination alle Werte der zugehörigen Ausgangskombination eindeutig vorgegeben[1]. Man spricht in diesem Fall von *Don't-Care* Bedingungen, partiell definierten Funktionen oder unvollständig bestimmten Booleschen Funktionen [8]. *Don't-Care* Zustände können ihre Ursache entweder darin haben, dass es aus irgendwelchen Gründen gleichgültig ist, welchen Wert ein Ausgang bei einem bestimmten Eingangsvektor annimmt, oder dass die betreffende Eingangskombination aufgrund von Beschränkungen überhaupt nicht auftreten kann [15]. Auf folgende Weise werden *On-Set, Off-Set* und *Don't-Care* Set definiert:

$$\begin{aligned}\text{On-Set } (f) = F &= \{\underline{\widehat{x}} \in X \mid f(\underline{\widehat{x}}) = 1\},\\ \text{Don't-Care Set } (f) = D &= \{\underline{\widehat{x}} \in X \mid f(\underline{\widehat{x}}) = *\},\\ \text{Off-Set } (f) = \overline{F \cup D} &= \{\underline{\widehat{x}} \in X \mid f(\underline{\widehat{x}}) = 0\},\end{aligned}$$

wobei X die Menge aller möglichen Inputs von f ist.

Die Verwendung von *Don't-Care* Bedingungen in der Digitaltechnik hat verschiedene Vorteile. Es existiert eine Vielzahl an Meinungen [78], [79], [80] über die Einsatzvarianten der *Don't-Care* Implementierung, und im Bereich der technischen Informatik und Mathematik ist die Optimierung/Minimierung Boolescher Funktionen insbesondere von Funktionensbündeln unter Verwendung von *Don't-Cares* nicht vollständig gelöst. Diese Tatsache ist durch die folgenden Hauptfaktoren bedingt [81]:

- Kompliziertheit der Darstellung der *Don't-Cares* für die Gewährleistung eines effektiven Entwurfs;
- Die Kompliziertheit bei gegebener *Don't-Care* Darstellung, Vorteile bei der logischen Synthese zu erhalten

[1] Don't-Care - Für diese Eingangskombinationen ist der Funktionswert völlig beliebig.

- Die Kompliziertheit der Verifizierung des Design (*design verification*) bei Verwendung von *Don't-Cares*.

Sieben kombinatorische Schaltungen werden jewels für 20%, 30%, ... , 90% *Don't-Care* Werte optimiert und die Abhängigkeit der erforderlichen Fläche von der prozentualen Anzahl der *Don't-Care* Bedingungen werden in Abbildung 4.1 dargestellt.

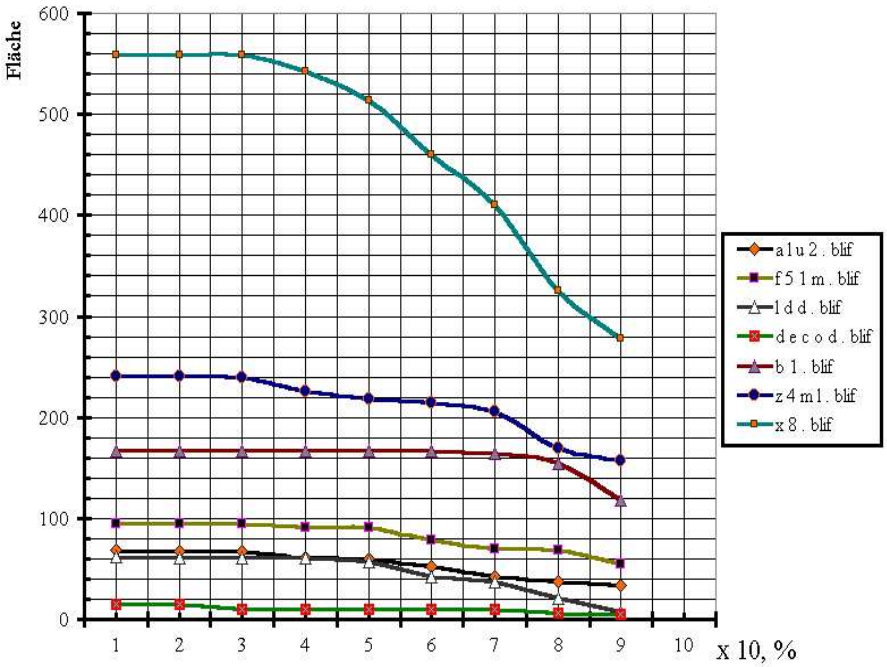

Abbildung 4.1: Abhängigkeit der Fläche der optimierten kombinatorischen Schaltungen von der prozentualen Anzahl der *Don't-Cares* (Vereinfachung einer unvollständig definierten Funktion

Auf der vertikalen Achse der gegebenen Grafik ist der Flächeverbrauch der optimierten Schaltungen abgetragen. Aus den erhaltenen Werten dieser sieben Schaltungen kann man schließen, dass die Abhängigkeit zwischen den beiden erwähnten Parametern proportionalen Charakter trägt. Ebenso ist offensichtlich, dass der Flächeverbrauch der Schaltung bei Vergrößerung der Anzahl der *Don't-Cares* in der Schaltung gegen Null strebt. Dies ist leicht an den Abhängigkeitslinien zu sehen. Die Analyse der experimentelle Ergebnisse zeigt, dass für einen Prozentsatz ab 45% *Don't-Cares* in kombinatorischen Schaltungen eine substantielle Reduktion der Flächenbedarfs erwartet werden kann. Nach dieser 45%-Schwelle sinkt der Flächeverbrauch der kombinatorischen Schaltungen radikal.

4.2 Mögliche Verwendung der partiellen *Don't-Cares* in der neuen Methode der Logische Ergänzung

In diesem Kapitel beschreiben wir, wie man komplementäre Fehlererkennungsschaltungen unter Verwendung der partiellen *Don't-Cares*, die beim Entwurf der komplementären Schaltung entstehen, entwerfen kann. Dabei soll der Flächenbedarf der komplementären Schaltung optimiert werden.

Die Methode des Entwurfes von Fehlererkennungsschaltungen durch komplementäre Schaltung ist in Abbilldung 4.2) dargestellt. Die zu überwachende Schaltung f wird dabei durch die komplementäre Schaltung g ergänzt, so dass die komponentenweise XOR-Summe $h = h_1, \cdots, h_n = f_1 \oplus g_1, \cdots, f_n \oplus g_n$ der Ausgänge von f und g Element eines gewählten Codes ist.

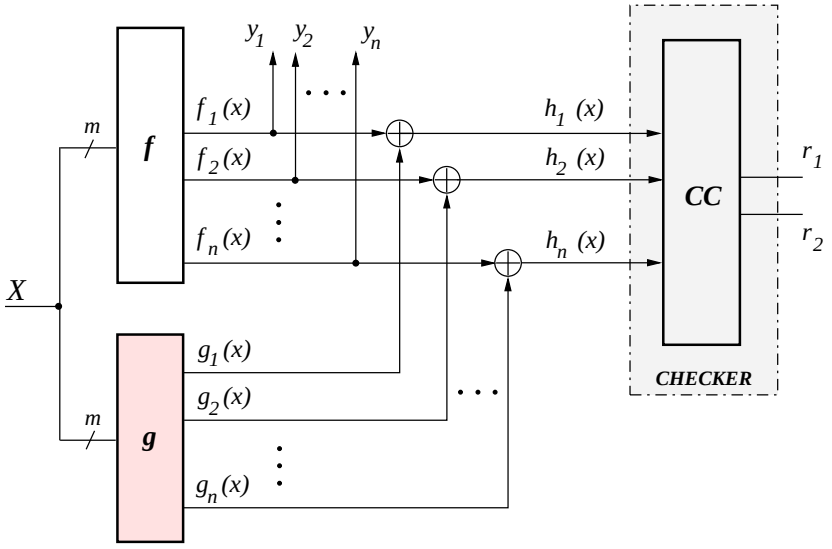

Abbildung 4.2: Neue Fehlererkennungsmethode

Die Ausgänge von g sind nur durch die Bedingung eingeschränkt, dass die komponentenweise XOR-Summe mit den Ausgängen von f der funktionalen Schaltung eines der möglichen Codewörter des betrachteten Codes ergibt. Darüber hinaus kann bei gleichen Funktionswerten $f_1(x), \cdots, f_n(x)$ die Funktion $g_i(x), i \neq n$ verschiedene Werte annehmen. Deshalb sind die Ausgänge $g_i(x)$ der komplementären Logik nicht vollständig bestimmt und somit *Don't-Cares*. Wir bezeichnen sie als partielle *Don't-Cares*. Bei der Synthese der komplementären Schaltung müssen für jeden Eingangsvektor X die Ausgänge der komplementären Schaltung Untermenge aus einer des n-dimensionalen binären Vektoren sein. Diese Untermenge von Vektoren besteht aus allen Vektoren, welche komponentenweise mit dem Ausgangsvektor $f(x)$ von f XOR-addiert einen Code-Vektor ergeben. Mathematisch gesehen müssen die Ausgänge $g(x)$ Elemente des Cosets bezüglich des Codes des Ausgabevektors $f(x)$ sein.

In diesem Kapitel wird die Untersuchung der Implementierung von partiellen *Don't-Cares* in kombinatorische Schaltungen unter Verwendung der neuen Methode vorgestellt. Dabei wird die Erreichung eines hohen Niveaus in der Schaltungsoptimierung und entsprechend eine Verkleinerung der Gesamtfläche der Fehlererkennungsschaltung erwartet. Der Algorithmus zur Implementierung der partiellen *Don't-Cares* in die Schaltung mit der neuen Fehlererkennungsmethode wird am Beispiel des 1-aus-3

Codes entwickelt.

Wie auch im vorigen Kapitel werden die funktionalen Ausgänge von f in Gruppen gleicher Länge eingeteilt. Die Ausgänge der Schaltung f werden in Gruppen zu je drei Ausgängen und eventuell verbleibende ein oder zwei Ausgänge aufgeteilt. Der jewels erste Ausgang einer Gruppe von drei Ausgängen bleibt unverändert, der zweite und dritte Ausgang werden jeweils durch eine Funktion ergänzt, also mit ihr XOR-verknüpft.

Für die kombinatorische Schaltung mit drei Ausgängen wird also die komplementäre Schaltung g mit nur zwei Ausgängen $g_2(x), g_3(x)$ entworfen (Abbildung 4.3). Der erste Ausgang der funktionellen Schaltung bleibt unmodifiziert. Für eine kombinatorische Schaltung mit den drei Ausgängen f_1, f_2 und f_3 wird dann eine komplementäre Schaltung g mit den beiden Ausgängen g_2 und g_3 so bestimmt, dass für alle $x \in X$ der Werte $f_1(x), f_2(x) \oplus g_2(x)$ und $f_3(x) \oplus g_3(x)$ Element eines 1-aus-3 Codes sind. Der Ausgang $f_1(x)$ der Schaltung f muss dann nicht mit einer komplementären Funktion XOR-verknüpft werden. Ist nun $f_1(x) = 1$ für ein bestimmtes $x \in X$, dann ist $f_2(x) \oplus g_2(x) = f_3(x) \oplus g_3(x) = 0$, notwendig damit ein 1-aus-3 Codewort erzeugt wird, und $g_2(x)$ und $g_3(x)$ sind durch $f_2(x) = g_2(x)$ und $f_3(x) = g_3(x)$ bestimmt. Ein hoher Optimierungsgrad der komplementären Schaltung ergibt sich, wenn $f_1(x) = 0$ gilt. In diesen Fall kann entweder:

$$f_2(x) \oplus g_2(x) = 1 \text{ und } f_3(x) \oplus g_3(x) = 0 \text{ oder } f_2(x) \oplus g_2(x) = 0 \text{ und } f_3(x) \oplus g_3(x) = 1$$
$$\text{bzw.}$$
$$g_2(x) = \overline{f}_2(x) \text{ und } g_3(x) = f_3(x)$$
$$\text{oder}$$
$$g_2(x) = f_2(x) \text{ und } g_3(x) = \overline{f}_3(x)$$

gelten.

Solche *Don't-Cares* werden verwendet, um eine optimale Synthese der komplementären Schaltung zu erreichen. Aber die Benutzung dieser *Don't-Cares* unterscheidet sich von derjenigen im konventionellen Sinne. Deshalb wird in dieser Arbeit eine Klassifikation der *Don't-Cares* in zwei Gruppen vorgestellt:

1. Der Einsatz eines Elements der ersten Gruppe ist obligatorisch, aber ein solches wird nur in einer der beiden Funktionen $g_2(x), g_3(x)$ verwendet, welche an den Ausgängen der komplementären Schaltung $g(x)$ realisiert werden. In welcher Funktion ein solcher *Don't-Care* Wert gleich 1 sein soll (und gleichzeitig 0 in der anderen Funktion), diese Frage muss beantwortet werden mit dem Ziel, einen höheren Optimierungsgrad der komplementären Schaltung $g(x)$ zu erreichen.

2. Die Verwendung eines Elements der zweiten *Don't-Care* Gruppe ist optional, aber wenn es verwendet wird, so muss es in beiden an den Ausgängen $g_2(x), g_3(x)$ der komplementären Schaltung realisierten Funktionen eingesetzt werden. Die Entscheidung, eine Untermenge der zweiten Gruppe auf 1 zu setzen, hängt von der Frage ab, ob dies zu einem höheren Optimisierungsgrad führt.

Im nächsten Abschnitt wird ein heuristischer Algorithmus aufgeführt, welcher den ersten Schritt im Bereich der *Don't-Care*-Implementierung in die komplementäre Schaltung mit dem Ziel der Erreichung eines höheren Optimierungsgrades darstellt, was die oben gestellten Fragen zu beantworten vermag.

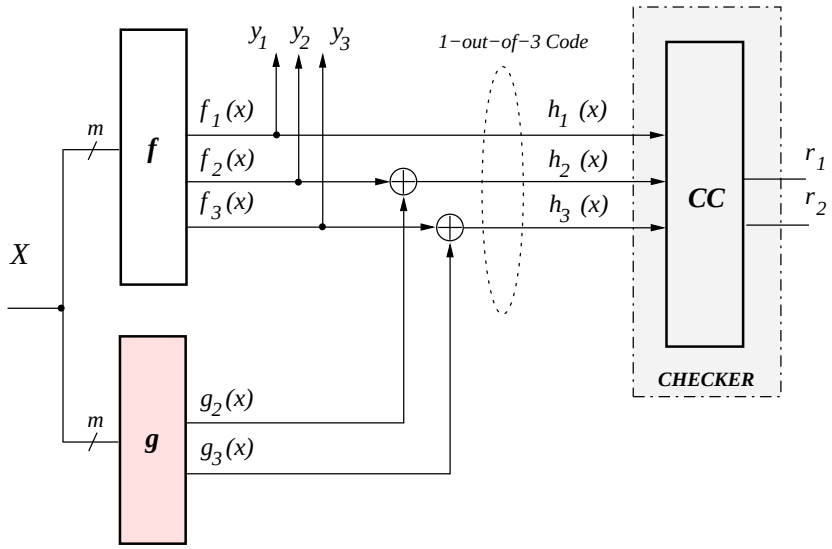

Abbildung 4.3: Neue Methode der Logische Ergänzung für eine Schaltung mit drei Ausgängen

4.3 Algorithmus für die *Don't-Care*-Implementierung

Der in diesem Abschnitt vorgestellte Algorithmus wird realisiert am Beispiel des 1-aus-3 Code. Zur Erläuterung der vorgeschlagenen Implementierung der *Don't-Care* Bedingungen wird eine kombinatorische Schaltung f mit drei funktionalen Ausgängen $y_1(x), y_2(x)$ und $y_3(x)$ betrachtet. Für die Menge der Eingangsvektoren $x_1, \cdots, x_m$ realisieren die Ausgänge der Schaltung die Funktionen $f_1(x)$, $f_2(x)$, $f_3(x)$[2].

Gemäß der vorgestellten Kontrollmethode der Logischen Ergänzung unter Verwendung des 1-aus-3 Codes werden nur zwei Ausgänge $f_2(x), f_3(x)$ der Schaltung f durch die Ausgänge $g_2(x), g_3(x)$ der komplementären Logik ergänzt (Abbildung 4.3). Der Ausgang $f_1(x)$ wird nicht ergänzt, d.h., für ihn ist keine Ergänzungsfunktion $g_1(x)$ nötig, somit besitzt die komplementäre Schaltung $g(x)$ nur zwei Ausgänge. Das Ziel des zur Implementierung der *Don't-Cares* aufgestellten Algorithmus besteht in der Suche nach einer Realisation der komplementären Funktionen $g_i(x)$ mit kleinstem Flächebedarf.

Die Analyse der Fehlererkennungsschaltung unter Verwendung von 1-aus-3 Codes für die Kontrolle zeigt auf, dass folgende Varianten der Realisation der komplementären Funktionen $g_2(x), g_3(x)$ existieren:

- **Variante 1 -** $f_1(x) = 1$ (am ersten Ausgang wird das Signal einer logischen Eins gebildet):

  Wegen Verwendung des 1-aus-3 Codes sind die Eingangsleitungen des Code Checkers fixiert auf das Codewort 100. Das bedeutet, dass unabhängig davon, welche Werte $f_2(x)$ und $f_3(x)$ annehmen, die Gleichungen $h_2(x) = f_2(x) \oplus g_2(x) = 0$ und $h_3(x) = f_3(x) \oplus g_3(x) = 0$ erfüllt sein. Das setzt voraus, dass die Implementierung der komplementären Funktionen $g_2(x), g_3(x)$ auf eine solche Weise geschehen muss, so dass $f_2(x) = g_2(x)$, $f_3(x) = g_3(x)$ ist. Die vier Fälle

[2]Bei n gegebenen funktionellen Ausgängen, $n > 3$, wird die gesamte Menge der Ausgänge so in Gruppen aufgeteilt, dass jede der Gruppen drei Ausgänge enthält. Mögliche Algorithmen für die Zusammenstellung der Gruppen von Ausgängen wurden bereits in den vorigen Abschnitten dieser Dissertation aufgestellt und untersucht.

der Belegung der komplementären Funktionen und der Funktionen der Schaltung f sind in der Tabelle 4.1 dargestellt.

Fall	Funktionen		Implementierung	
	$f_2(x)$	$f_3(x)$	$g_2(x)$	$g_3(x)$
1	0	0	0	0
2	1	1	1	1
3	0	1	0	1
4	1	0	1	0

Tabelle 4.1: Realisation der Funktionen $g_2(x), g_3(x)$ für $f_1(x) = 1$

- **Variante 2 -** $f_1(x) = 0$ (am ersten Ausgang wird das Signal der logischen Null gebildet):

  Im Unterschied zur ersten Variante können hier an den Leitungen $h_1(x), h_2(x), h_3(x)$ die zwei Wörter des 1-aus-3 Codes anliegen: $\{010\}$ oder $\{001\}$. Das 1-aus-3 Codewort ist also nicht fixiert, und so existieren für diese Variante zwei mögliche Realisierungen der komplementären Funktionen $g_2(x), g_3(x)$, siehe Tabelle 4.2.

Fall	**Funktionen**		**Implementierung $g_2(x)$, $g_3(x)$ mit**			
			Implementierung I: $(h_1, h_2, h_3) = (0, 1, 0)$		Implementierung II: $(h_1, h_2, h_3) = (0, 1, 0)$	
	$f_2(x)$	$f_3(x)$	$g_2(x)$	$g_3(x)$	$g_2(x)$	$g_3(x)$
1	0	0	1	0	0	1
2	1	1	0	1	1	0
3	0	1	1	1	0	0
4	1	0	0	0	1	1

Tabelle 4.2: Realisation der Funktionen $g_2(x), g_3(x)$ für $f_1(x) = 0$

Für diese Fälle sollen im folgenden einige Definitionen und Notationen beschrieben werden.

Definitionen und Notationen:
Für drei gegebene Funktionen $f_1(x), f_2(x), f_3(x)$ werden zwei Funktionen definiert, die als D-Funktion bezeichnete Funktion

$$D(x) = \overline{f}_1(x) \wedge (f_2(x) \oplus f_3(x))$$

und die als S-Funktion bezeichnete Funktion

$$S(x) = \overline{f}_1(x) \wedge (f_2(x) \overline{\oplus} f_3(x)).$$

Es ist offensichtlich, dass die beiden Realisierungen der Fälle 1 und 2 aus der Tabelle 4.2 verknüpft durch eine logische OR-Operation zur Funktion $D(x)$ führt. Analog erhält man $S(x)$ aus den Fällen 3 und 4. Für eine Funktion $f_i(x)$ sei die Anzahl der *True Minterms*[3] in der Funktion mit $T(f)$ gekennzeichnet, und der optimale Flächebedarf, um diese Funktion zu realisieren, sei $A(f)$.

Wenn die Anforderung an die Implementierung einer beliebigen komplementären Funktion $g_i(x)$ gleich Null ist, muss offensichtlich nichts getan werden. Beispielsweise muss im Falle 1 der Tabelle 4.1 nichts implementiert werden. Aber für die Fälle 3, 4 der Tabelle 4.1 ist der Einsatz einiger logischer Gatter für die Realisierung der Funktionen $g_2(x) = 1, g_3(x) = 1$ notwendig. Für $f_1(x) = 0$ müssen in den Fällen 1 und 2 aus Tabelle 4.2 jeweils nur eine der beiden Implementationen realisiert werden. In den Fällen 3 und 4 aus Tabelle 4.2 wird jeweils die Realisierung der Funktionen $g_2(x) = 0$ und $g_3(x) = 0$ gewählt, so dass die Implementierung der komplementären Logik g nicht erforderlich ist.

Zur Zusammenfassung sind im folgenden die Funktionen der komplementären Schaltung $g(x)$ genannt, die schließlich implementiert werden müssen, um die Kontrolle der Schaltung $f(x)$ unter Verwendung des 1-aus-3 Codes zu gewährleisten.

Implementation-1:

Falls für einen beliebigen Eingangsvektor X, $f_1(x) = 1$ und $f_2(x) = 1$ ist (Fall 2 und 4 in Tabelle 4.1), dann muss die komplementäre Funktion $g_2(x) = 1$ sein: $g_2(x) = f_1(x) \wedge f_2(x)$;

Implementation-2:

Falls für einen beliebigen Eingangsvektor X, $f_1(x) = 1$ und $f_3(x) = 1$ ist (Fall 1 und 4 in Tabelle 4.1), dann muss die komplementäre Funktion $g_3(x) = 1$ sein: $g_3(x) = f_1(x) \wedge f_3(x)$;

Implementation-3:

Falls für einen beliebigen Eingangsvektor X, $f_1(x) = 1$ und $f_2(x) = f_3(x)$ ist (Fall 1 und 2 in Tabelle 4.2), dann muss eine der Funktionen $g_2(x)$ und $g_3(x)$ gleich 1 sein (Jeder Minterm der Funktion $D(x)$ muss entweder in der Funktion $g_2(x)$ oder in $g_3(x)$, nicht aber in beiden sein);

Implementation-4 (Optional):

Falls für einen beliebigen Eingangsvektor X, $f_1(x) = 0$ und $f_1(x) \neq f_2(x)$ ist, (Fall 3 und 4 in Tabelle 4.2), dann können beide Funktionen $g_2(x) = 1$ und $g_3(x) = 1$ sein (d.h. eine Funktion $S_d(x) \subseteq S(x)$ kann in beiden Funktionen $g_2(x)$ und $g_3(x)$ enthalten sein).

Aus der gegebenen Klassifikation können folgende Punkte abgeleitet werden:

1. Die Implementationen 1 und 2 sind fest;
2. Für die Implementation-3 muss entschieden werden, welche der Funktionen $g_2(x)$ und $g_3(x)$ gleich 1 gesetzt werden soll. Die Entscheidung wird getroffen in Abhängigkeit davon, welche Realisation eine geringere Fläche erfordert;

[3] Konjunktionsterm, der einen einzelnen Punkt innerhalb eines Wertefeldes von Variablen definiert.

3. Die Implementation-4 ist optional.

Die Frage ist, wie (und sogar ob) eine Untermenge $S_d(x) \subseteq S(x)$ ausgewählt werden soll, die in beide Funktionen $g_2(x)$ und $g_3(x)$ aufgenommen wird. Die Antwort ist: Wenn eine solche Menge $S_d(x)$ existiert, welche - enthalten in beiden Funktionen $g_2(x)$ und $g_3(x)$ - den Hardwarebedarf verkleinert, so wird $S_d(x)$ gewählt.

4.4 Optimierung der komplementären Schaltungen

Nun wird am Beispiel einer einfachen kombinatorischen Schaltung der Versuch unternommen, das Optimum der komplementären Schaltung $g(x)$ unter Berücksichtigung der Erkenntnisse des letzten Abschnitts zu finden.

Als Beispiel soll hierfür eine kombinatorische Schaltung $f(x)$ mit vier Eingängen x_1, x_2, x_3, x_4 und drei funktionalen Ausgängen $f_1(x), f_2(x), f_3(x)$ dienen, die durch die folgenden *K-Maps* (Karnaugh-Diagramm) beschrieben ist. Der Flächeverbrauch wird zur Vereinfachung mit der Anzahl der Literale

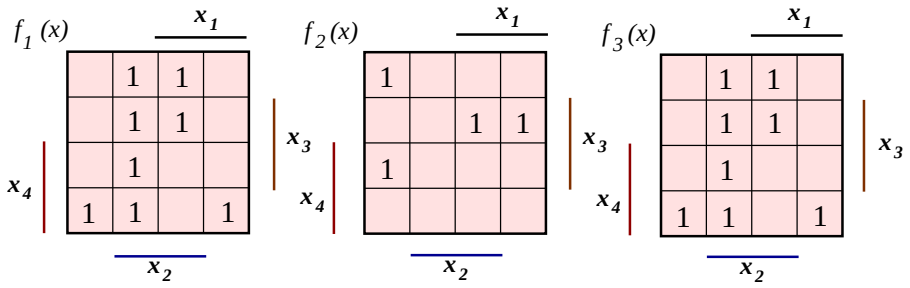

Abbildung 4.4: Kombinatorische Schaltung $f(x)$ mit drei Ausgängen (mit einer Fläche von 27)

in der *Two-Level* Realisierung angegeben und ist für die Funktionen in der Abbildung 4.4 jeweils 6, 14 bzw. 7. Somit ist die Gesamtfläche A_f dieser Schaltung $f(x)$ gleich 30.

Die Funktionen $D(x)$ und $S(x)$ zeigen die Abbildungen 4.5 a) und b). Gemäß Implementation-1

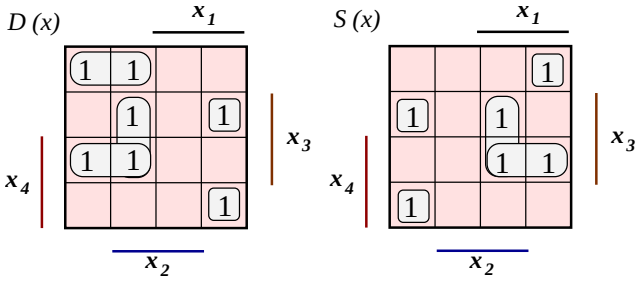

Abbildung 4.5: Die Funktionen $D(x)$ und $S(x)$ für die Schaltung $f(x)$ aus Abbildung 4.4

und Implementation-2 müssen die Funktionen $g_2(x)$ und $g_3(x)$ die Funktionen $g_{2int}(x) \subseteq g_2(x)$ und $g_{3int}(x) \subseteq g_3(x)$ implementieren (Abbildung 4.6).

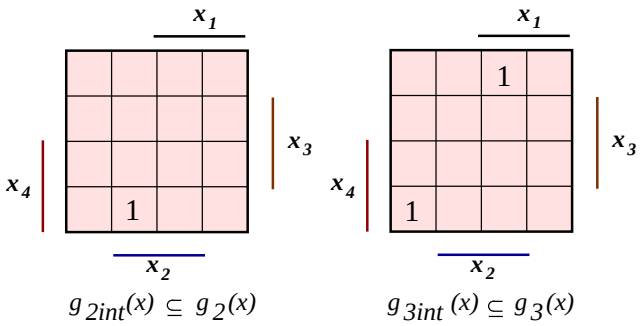

Abbildung 4.6: Ausführung der Implementationen 1 a) und 2 b) für die Schaltung $f(x)$ aus Abbildung 4.4

Wenn bei Ausführung der Implementation-3 die Funktion $D(x)$ nur in die Funktion $g_2(x)$ aufgenommen wird, so beträgt die Gesamtfläche A_g (A_g=24) der komplementären Logik $g(x)$ relativ zur funktionalen Schaltung $f(x)$ 89%. Die Minterme von $D(x)$ sind als D dargestellt, deren Werte sind 1. Wenn aber die Funktion $D(x)$ nur in die komplementäre Funktion aufgenommen wird, so verringert sich die Fläche (A_g=23) auf 85% von $f(x)$. Beide Fälle sind in den Abbildungen 4.7 a) und b) dargestellt.

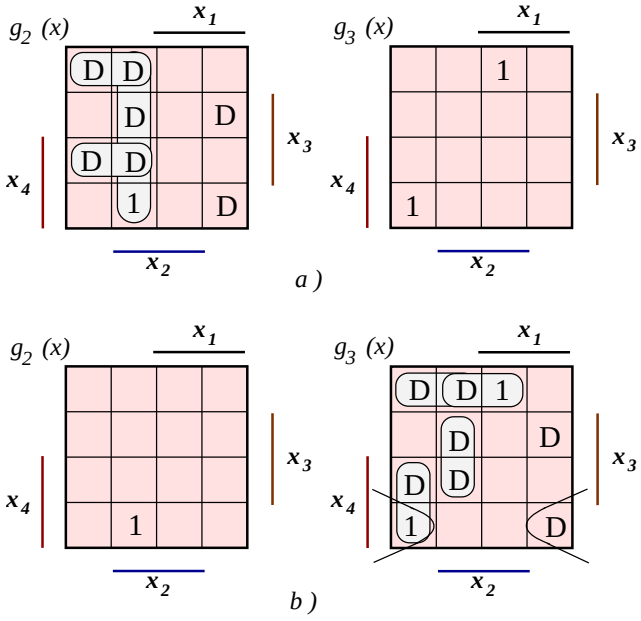

Abbildung 4.7: Implementation-3 für den Fall $D(x)$ (*included*) in $g_2(x)$ und $g_3(x)$ (a) mit einer Fläche von 24, b) mit einer Fläche von 23)

Wenn jedoch die Primimplikanten[4] der Funktion $D(x)$ genau zwischen den Funktionen $g_2(x)$ und $g_3(x)$ verteilt werden, so sinkt der Flächebedarf A_g auf 81% von $f(x)$ ($A_g = 22$). Weiterhin existiert

[4]Größtmöglicher Implikant, der nicht vollständig Bestandteil eines anderen Implikanten ist.

eine optimalere Realisierung von $g_2(x)$ und $g_3(x)$, wenn einige *True Minterms* von $S(x)$ in beide Funktionen aufgenommen werden. Wird dies an der Beispielschaltung durchgeführt (siehe Abbildung 4.8), entsteht ein Flächebedarf von A_g=18 (67% von $f(x)$).

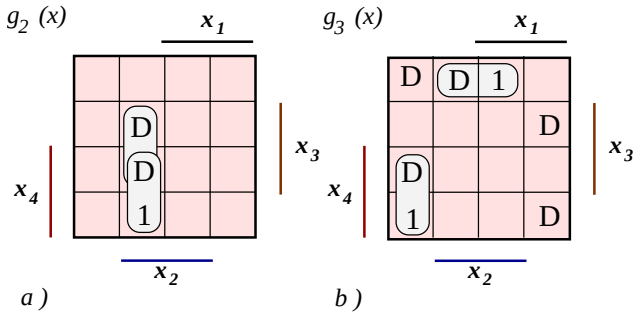

Abbildung 4.8: Implementation-3 für den Fall $D(x)$ (*distributed*) in $g_2(x)$ und $g_3(x)$ (mit einer Fläche von 22)

Die Minterme von $S(x)$, welche in beiden Funktionen verwendet werden, sind mit S gekennzeichnet, S steht für 1. Wird die Funktion $S(x)$ auf solche Weise gemäß Implementation-4 in die komplementären Funktionen eingefügt, kann dies zu einer erheblichen Reduzierung der Fläche der Schaltung $g(x)$ führen.

Das Finden der optimalen Realisierung der komplementären Schaltung $g(x)$ ist keine komplizierte Aufgabe und kann mit Hilfe von *K-Maps* geschehen (siehe Abbildung 4.9).

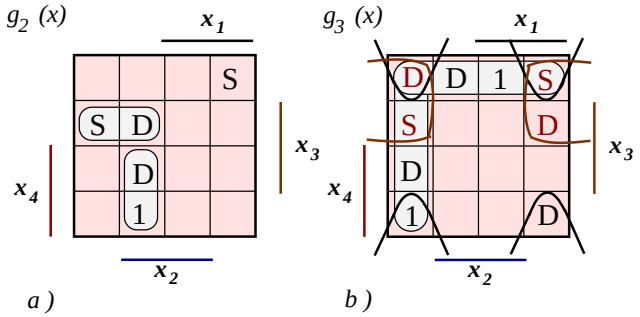

Abbildung 4.9: Optimale Realisierung der Schaltung $g(x)$ mit Implementation-4 (mit einer Fläche von 18)

Allerdings sieht die Situation bei einer kombinatorischen Schaltung mit einer großen Anzahl von Ausgängen schwierig aus. In dieser Arbeit werden solche Schaltungen als Netzliste untereinander verbundener logischer Elemente dargestellt, und nicht als Wertetabelle, und folglich erfordert die Ermittlung der Funktionen $D(x)$ und $S(x)$ einen anderen Algorithmus.

Für das Erhalten dieser Funktionen sind weiter unten heuristische Algorithmen für die Optimierung einer beliebigen Schaltung $g(x)$ unter Verwendung der Eigenschaften von *Don't-Cares* vorgestellt. Die Grundlage für diese Algorithmen ist die Ausführung notwendiger logischer Operationen durch Verbindung der entsprechenden Gatter an den Ausgangsleitungen der gegebenen Schaltung $g(x)$.

Algorithmus 4.1: *Optimiere_G*

Eingabe: Eine Schaltung mit drei Ausgängen,
die $f_1(x)$, $f_2(x)$ und $f_3(x)$ realisieren;
Ausgabe: Die komplementäre Schaltung G mit
den Ausängen $g_2(x)$ und $g_3(x)$;

1. $g_{2int}(x)$ = $f_1(x) \wedge f_2(x)$, $g_{3int}(x) = f_1(x) \wedge f_3(x)$;
2. $g_2(x) = g_{2int}(x)$, $g_3(x) = g_{3int}(x)$;
3. $D(x) = \bar{f}_1(x) \wedge (f_2(x) \bar{\oplus} f_3(x))$
4. Wenn $D(x)$ = Null, dann gehe zu Schritt 5
sonst formuliere $D(x)$
als Summe von Produkttermen[5] (SOP)

$D(x) = d_1(x) \vee d_2(x) \vee \cdots \vee d_i(x) \vee \cdots \vee d_p(x)$;
Wähle ein beliebiges $d_i(x) \subseteq D(x)$, und finde
Δa_2 = Fläche $(g_2(x) \vee d_i(x))$ – Fläche $(g_2(x))$;
Δa_3 = Fläche $(g_3(x) \vee d_i(x))$ – Fläche $(g_3(x))$;

Wenn $\Delta a_2 < \Delta a_3$, dann $g_2(x) = g_2(x) \vee d_i(x)$
sonst $g_3(x) = g_3(x) \vee d_i(x)$
$D(x) = D(x) \wedge \bar{d}_i(x)$;

5. Gehe zu Schritt 4.
6. **Ausgabe**: Die Implementation von $g_2(x)$ und $g_3(x)$ entspricht
der komplementären Schaltung G:
Benenne $g_{2D}(x) = g_2(x), g_{3D}(x) = g_3(x)$;
7. *Include_S.*

Algorithmus 4.2: *Include_S*

Eingabe: $f_1(x)$, $f_2(x)$, $f_3(x)$, $g_{2D}(x)$, $g_{3D}(x)$
Ausgabe: Neue komplementäre Schaltung G mit neuen $g_2(x)$ und $g_3(x)$

1. $g_2(x) = g_{2D}(x), g_3(x) = g_{3D}(x)$;
2. $S(x) = \bar{f}_1(x) \wedge (f_2(x) \oplus f_3(x))$;
3. Wenn $S(x)$ = Null, dann gehe zu Schritt 5
sonst formuliere $S(x)$ als Summe von
Produkttermen (SOP):

$S(x) = y_1(x) \vee y_2(x) \vee \cdots \vee y_i(x) \vee, \cdots, \vee y_k(x)$;
Für jedes $y_i(x) \subseteq Y(x)$,
finde A = Fläche $(g_2(x) y_i(x))$ + Fläche $(g_3(x) \vee y_i(x))$;
wenn $A <$ Fläche $(g_2(x))$+ Fläche $(g_3(x))$;
dann $g_2(x) = g_2(x) \vee y_i(x)$
$g_3(x) = g_3(x) \vee y_i(x)$
$S(x) = S(x) \wedge \bar{y}_i(x)$
formuliere $S(x)$ als Summe von
Produkten (SOP):

[5] AND/OR- Strukturen, Bestimmung durch der 1-Felder (NAND).

$S(x) = y_1(x) \vee y_2(x) \vee, \cdots, \vee y_i(x) \vee, \cdots, \vee y_k(x)$.

5. **Ausgabe**: Implementation von $g_2(x)$ und $g_3(x)$ ist entspricht der neuen komplementären Schaltung G.

4.5 Bestimmung der Gruppen von Ausgängen unter Verwendung von *Don't-Cares*

In diesem Abschnitt wird ein neues Verfahren für Einteilung der Schaltungsausgänge im Grupppen zu je drei Ausgängen vorgestellt, das auf der Anzahl der Implikanten von den jeweligen Ausgängen basiert. Die Anzahl der Implikanten wird auf der Basis einer Schaltungssimulation für pseudozufällige Inputs geschätzt. Die dabei von einzelnen Schaltungsausgängen ausgegebenen Einsen dienen der Schätzung der Implikantenanzahl.

Die Bestimmung der Komplexität der funktionalen Ausgänge wird in der Arbeit [82] detailiert betrachtet. Es ist ein System dreier Funktionen $f_1(x)$ $f_2(x)$ und $f_3(x)$ einer kombinatorischen Schaltung gegeben, welche in einer Gruppe von Ausgängen vereinigt sind, siehe Abbildung 4.3. Das folgende Lemma bestimmt die Gesamtzahl der *True Minterms*. Dabei wird für diese Gruppe von Ausgängen $f_1(x)$, $f_2(x)$, $f_3(x)$ die komplementäre Logik mit den Funktionen $g_2(x), g_3(x)$ realisiert. Falls ein beliebiger *True Minterm* in beiden Funktionen $g_2(x)$ und $g_3(x)$ auftritt, so wird er nur einmal in nur einer dieser Funktionen verwendet.

Lemma:
Bei der Realisierung der Schaltung von Abbildung 4.3 ist:
$T(g_2(X) \vee g_3(X)) \geq T(f_1(X) \wedge f_2(X) \vee f_1(X) \wedge f_3(X) \vee f_1(X) \wedge f_3(X) \vee \bar{f}_1(X) \wedge \bar{f}_3(X) \wedge \bar{f}_3(X))$.
Beweis:
Für den Beweis dieses Lemmas werden nur jene Fälle betrachet, die implementiert werden müssen, d.h. die Fälle 2, 3, 4 aus Tabelle 4.1 und die Fälle 1 und 2 aus Tabelle 4.2. Dabei wird mit $T(f)$ die Anzahl der *True Minterms* bezeichnet:
$T(f_1(X) \wedge f_2(X) + f_1(X) \wedge f_3(X) + \overline{f}_1(X) \wedge f_2(X)) \overline{\oplus} f_3(X))$,
wobei die Funktion von T in den Klammern zusätzlich vereinfacht werden kann zu:
$f_1(X) \wedge f_2(X) \vee f_1(X) \wedge f_3(X) \vee f_1(X) \wedge f_3(X) \vee \overline{f}_1(X) \vee \overline{f}_2(X) \vee \overline{f}_3(X)$.
Es ist offensichtlich, dass die Fälle 3 und 4 aus Tabelle 4.2 eine größere Anzahl *True Minterms* enthalten. Das Lemma ist somit bewiesen.

Abschließend muss gesagt werden, dass die Bedeutung dieses Lemmas wichtig ist, und zwar aus folgendem Grunde. Das Lemma beweist, dass bei gegebenen drei Funktionen $f_1(x), f_2(x), f_3(x)$ die Gesamtzahl der Minterme, welche in die komplementäre Schaltung $g(x)$ implementiert werden muss, nicht von der Wahl des ersten Ausganges $f_1(x)$ dieser drei Funktionen abhängt. Allerdings kann diese Auswahl den Freiheitsgrad der Optimierung beeinflussen, was im nächsten Abschnitt diskutiert wird.

4.6 Auswahl der $f_{i1}(x)$, $f_{i2}(x)$, $f_{i3}(x)$ - Funktionen in jeder Gruppe von Ausgängen

Trotz des Umstandes, dass die Wahl des Ausgangs $f_1(x)$ sich nicht auf die Anzahl der *True Minterms* auswirkt, kann die Wahl des ersten Ausgangs die Werte der drei Funktionen $D(x)$, $g_{2_{int}}(x)$ und $g_{3_{int}}(x)$

verändern, wobei $g_{2_{int}}(x) = f_1(x) \wedge f_2(x)$, $g_{3_{int}}(x) = f_1(x) \wedge f_3(x)$ ist.

Offensichtlich ist die Funktion $D(x) \vee g_{2_{int}}(x) \vee g_{3_{int}}(x)$ unabhängig von der Wahl von $f_{i1}(x)$ bestimmt. Allerdings kann die Wahl von $f_1(x)$ dazu beitragen, dass einige Minterme von $g_{2_{int}}(x)$ und $g_{2_{int}}(x)$ zu $D(x)$ hinübergehen und umgekehrt. Darum ist es möglich, die Abhängigkeit von der Wahl zu bestimmen. Eine beliebige Eingangskombination $X \subseteq D(x)$ kann entweder in der Funktion $g_2(x)$ oder in $g_3(x)$ eingeschlossen sein müssen. Mit anderen Worten, die Positionen der Funktionen $g_{2_{int}}(x)$ und $g_{3_{int}}(x)$ sind festgelegt, während einige *True Minterms* in der Funktion $D(x)$ nicht fixiert sind.

Offensichtlich soll die Wahl des ersten Ausganges $f_1(x)$ das Vorhandensein einer größeren Anzahl beweglicher Positionen begünstigen, so dass die Auswahl größer wird. In diesem Sinne, wählt man den ersten Ausgang $f_{i1}(x)$ aus einer Gruppe mit drei Ausgängen auf eine solche Weise, dass die Anzahl der Minterme in $g_{2_{int}}(x)$ und $g_{3_{int}}(x)$ minimal wird. Aus diesem Grunde muss die Wahl der Ausganges $f_1(x)$ darauf gegründet werden, dass nach Festlegung des Ausgänge innerhalb des Funktionstripels $f_1(x), f_2(x), f_3(x)$ die Anzahl der *True Minterms* in den Funktionen $g_{2_{int}}(x)$ und $g_{3_{int}}(x)$ minimal ist.

Als erster Ausgang aus der Gruppe der drei Ausgänge wird derjenige gewählt, bei welchem die Anzahl der Nullen groß ist, was zu einer kleinen Anzahl an Einsen in den Funktionen $g_{2_{int}}(x)$ und $g_{3_{int}}(x)$ führt.

Für jede der erhaltenen Funktionen $f_{i1}(x), f_{i2}(x), f_{i3}(x),\ \forall i,\ 1 \leq i \leq\ t$, mit t ist die Anzahl der Gruppen von Ausgängen (bei Verwendung des 1-aus-3 Code ist $t = n/3$), werden für die Wahl der Ausgänge $f_{i2}(x)$ und $f_{i3}(x)$ alle Kombinationen von zwei Funktionen aus der Menge der restlichen $(n - t)$ Funktionen gebildet.

Durch Simulierung der Schaltung mit 10 000 pseudozufälligen Eingangsvektoren wird für jede Kombination die Anzahl der *True Minterms* mithilfe der Formel aus dem Lemma ermittelt. Unter Verwendung dieses Lemmas ist es sehr leicht, einen Algorithmus zu realisieren, welche die Einsatz von Invertoren an den Ausgängen von $f_{i2}(x)$ und $f_{i3}(x)$ bestimmt. Die Werte N_i^1 werden für jede der vier Kombination, NOT-Gatter aufzustellen (siehe unten), bestimmt.

$$
\begin{array}{l}
f_1(x),\ \underline{f_{i2}}(x),\ f_{j3}(x) \\
f_1(x),\ \overline{f_{i2}}(x),\ \underline{f_{j3}}(x) \\
f_1(x),\ \underline{f_{i2}}(x),\ \overline{f_{j3}}(x) \\
f_1(x),\ \overline{f_{i2}}(x),\ \overline{f_{j3}}(x)
\end{array}
$$

Aus allen Kombinationen wird diejenige Variante gewählt, welche den kleinsten Wert N_i^1 besitzt. Bei Varianten mit Invertoren müssen die entsprechenden funktionalen Ausgänge $f_{i2}(x)$ und $f_{i3}(x)$ in der Komplementärschaltung anschließend erneut negiert werden.

Die experimentellen Ergebnisse, die bei Verwendung des entwickelten Algorithmus erhalten wurden, sind in der Tabelle 4.3 vorgestellt. Bei allen untersuchten MCNC-Benchmark-Schaltungen liegt die Größe der Fläche der komplementären Logik mit implementierten *Don't-Cares* unter 100% des Flächebedarfs der funktionalen Schaltung. Es ist offensichtlich, dass die erhaltenen Ergebnisse besser sind als bei Verwendung der Methode „Verdopplung und Vergleich".

Die durchgeführte Arbeit hat aber die Hoffnungen, den Flächebedarf der komplementären Schaltung bei Verwendung des 1-aus-3 Codes und partiellen *Don't-Care* Bedingungen und damit die Fläche der gesamten Fehlererkennungsschaltung der Logischen Ergänzung auf unter ca. 30% zu reduzieren nicht erfüllt. Dennoch stellt der in diesem Abschnitt vorgestellte Algorithmus eine gute Basis für nachfolgende Forschungen dar und ist ein wichtiger theoretischer Anfang für die Arbeit im Bereich der Benutzung der Eigenschaften von *Don't-Care* Zuständen in der neuen Methode der Logischen Ergänzung. Erstens

$Circuit$	inputs	outputs	Fläche $g(x)/f(x)$, [%]
b1	3	4	46,15
cm138a	6	8	58,54
cm42a	4	10	65,85
cu	14	11	50,6
pde	19	9	77,45
decod	6	16	62,9
ldd	9	19	58,62
pm1	16	13	88,06
x2	10	7	56,72
vda	17	39	97,61
z4ml	7	17	58,62
pcler8	27	17	73,57
cm82a	5	3	68,75
Mittelwert			66,42

Tabelle 4.3: Experimentelle Ergebnisse

kann diese Methode auf 1-aus-n Codes erweitert werden. Ferner wurde in dieser Methode, um die optimale Schaltung $g(x)$ zu erhalten, ein Algorithmus zur Verteilung aller Primimplikanten der D-Funktion auf zwei Gruppen und ein anderer Algorithmus zur Wahl einer Untermenge von Primimplikanten der S-Funktion verwendet. Der Autor vermutet, dass die Optimierung der komplementären Schaltung erfolgreicher wird, wenn die Primimplikanten gleichmäßig aufgeteilt werden.

4.7 Zusammensetzung der Gruppen von Ausgängen

Während der gesamten Zeit der Aufstellung und Begründung der Theorie der Methode der Logischen Ergänzung wurde eine große Anzahl von Experimenten durchgeführt zur Suche nach einem Algorithmus für die Zusammenstellung der Gruppen von Ausgängen mit der Absicht, die Fläche der komplementären Schaltung zu minimieren.

Die theoretische und experimentelle Arbeit, welche bei der Realisierung der *Don't-Cares* in die Struktur der neuen Methode durchgeführt wurde und welche bereits oben beschrieben wurde, kann für die Suche nach einem optimalen Algorithmus zur Auswahl der Ausgänge in die Gruppen verwendet werden. Nun werden die Ergebnisse der Experimente, deren Ziel es war, den Flächenaufwand der Komplementärschaltungen für den 1-aus-3 Code zu bestimmen, vorgestellt. Es wurden zwei Algorithmen realisiert. Eine zusätzliche Aufgabe dieser weiter unten beschriebenen Algorithmen ist das Feststellen des Bestehens einer Abhängigkeit der Realisationsfläche der Schaltung $g(x)$ von dem verwendeten Kriterium der Auswahl der Ausgänge einer Gruppe.

Im ersten vorgestellten Algorithmus wird für den Ausgang $f_{i1}(x)$ derjenige Ausgang gewählt, welcher bei Simulierung der Schaltung mit 10 000 pseudozufälligen Eingangsvektoren die größte Anzahl von Nullen aufweist. Die Wahl des ersten Ausganges $f_{i1}(x)$ im zweiten Algorithmus trifft auf

denjenigen Ausgang, dessen zugehörige Funktion die größte Anzahl an Literalen enthält. So wird in diesem Abschnitt der Dissertation auf experimentellem Wege nachgewiesen oder widerlegt, dass eine Abhängigkeit des Parameters der Schaltungsfläche vom für die Wahl der Ausgänge benutzten Kriterium besteht. Ergänzend wird die Möglichkeit untersucht, die Anzahl der Nullen als Kriterium für die Bestimmung der Komplexität der funktionalen Ausgänge der Schaltung zu verwenden.

Algorithmus 4.3:

Der erste Algorithmus besteht aus zwei Teilen- aus der Simulierung der Schaltungen mit 10 000 pseudozufälligen Eingangsvektoren und aus der Bestimmung der Gruppen von Ausgängen auf Grundlage der erhaltenen Daten. Als Generator der pseudozufälligen Kombinationen kann jeder beliebige Generator benutzt werden, welcher ein Unterprogramm des *CAD*-Werkzeugs ist.

Nun wird mit dem Ziel der Bestimmung der Ausgänge $f_{i1}(x)$ die $(n \times m)$ Matrix der Simulierungsergebnisse analysiert, wobei n die Zahl der generierten Eingangsvektoren und m die Zahl der funktionalen Ausgänge der Schaltung ist.

Für die Wahl des ersten Ausganges $f_{i1}(x)$ in jede Gruppe X_i^3 wird das obenangeführte Kriterium der maximalen Anzahl $N_i^{0(1)}$ der Nullen (Einsen) verwendet. Aus der Gesamtzahl der funktionalen Ausgänge werden diejenigen Ausgänge gewählt, welche eine maximale Anzahl an Nullen (oder Einsen) aufweisen. Offensichtlich können auf Grundlage der Simulierungsdaten auch andere Kriterien für die Fixierung der Ausgänge verwendet werden, beispielsweise kann die maximale Differenz zwischen den Mengen N_i^1 und N_i^0 als Auswahlkriterium dienen.

Für die Wahl des zweiten Ausganges $f_{i2}(x)$ in jeder Gruppe werden wieder die Simulierungsdaten verwendet. Dabei werden in der Matrix jene Zeilen fixiert, in welchen die Werte der bereits ausgewählten Ausgänge $f_{i1}(x)$ gleich Eins sind: $f_{i1}(x) = 1$. Die Anzahl dieser Zeilen $Z_{f_1=1}$ ist offensichtlich gleich der Anzahl der Einsen in den Funktionen $f_{i1}(x)$.

Der nächste Schritt des Algorithmus besteht in der Bestimmung der Anzahlen N_i^1 und N_i^0 der Funktionen

$$F_{AND/T}(x) = f_{i1}(x) \wedge f_T(x) \tag{4.1}$$

für die Menge jener Eingangskombinationen, für welche die Funktion des ersten Ausganges $f_{i1}(x) = 1$ ist. Dabei ist $f_T(x)$ jeweils ein funktionaler Ausgang der Schaltung aus der Menge der noch nicht verwendeten Ausgänge. Als zweiter Ausgang $f_{i2}(x)$ jeder Gruppe von Ausgängen wird derjenige Ausgang $f_T(x)$ mit der maximalen Anzahl von Nullen in der entsprechenden Funktion $F_{AND/T}(x)$ gewählt.

Der dritte Ausgang $f_{i3}(x)$ berechet sich für jede Gruppe von Ausgängen X_i^3 nach dem gleichen Prinzip. Der einzige Unterschied ist die Funktion $D(x)$, für welche (nur in den Zeilen, in denen $f_{i1}(x) = 1$ ist) die Anzahl der Nullen oder Einsen gezählt wird. Diese Funktion $D(x)$ wird hier vorgestellt:

$$D(x) = (f_1(x) \wedge f_2(x)) \vee f_3(x) \wedge \overline{f_1}(x) \vee \overline{f_2}(x) \vee \overline{f_3}(x), \tag{4.2}$$

$$D^\star(x) = (f_1(x) \wedge f_2(x)) \vee \overline{f_3}(x) \wedge \overline{f_1}(x) \vee \overline{f_2}(x) \vee f_3(x). \tag{4.3}$$

Der Unterschied zwischen den Formeln zur Berechnung der Funktionen $D(x)$ und $D^\star(x)$ liegt in der Invertierung der Funktion f_{i3}. Für jede Funktion wird die Anzahl der Nullen und Einsen gezählt. Als dritter und letzter Ausgang $f_{i3}(x)$ für jede Gruppe von Ausgängen wird der Ausgang mit der maximalen Nullenanzahl in der Wertetabelle der Funktion gewählt, zusätzlich wird dieser Wert mit der Anzahl der Einsen in der Funktion $D^\star(x)$, in welcher der Ausgang $f_{i3}(x)$ invertiert wird, verglichen.

Im Falle, dass die Anzahl der Einsen in der Funktion $D^{\star}(x)$ bei Verwendung eines beliebigen Ausganges $f_{i3}(x)$ größer ist als die Anzahl der Nullen in der Funktion $D(x)$, so ist es erforderlich, diesen Ausgang in der Struktur der gesamten Fehlererkennungsschaltung zu invertieren. Es ist möglich, dass nach der Ausführung des Algorithmus eine Restgruppe X_{rest} verbleibt, in welcher die Anzahl der Ausgänge kleiner ist als drei. Entspechend der Methode der Logischen Ergänzung erfordern die Ausgänge der Restgruppe jeweils ihre Invertierung in der komplementären Logik, siehe Abbildung 3.7.

Algorithmus 4.4:

Der zweite Algorithmus unterscheidet sich vom ersten nur in der Wahl des ersten Ausganges jeder Gruppe. Die Ausgänge $f_{i2}(x)$ und $f_{i3}(x)$ werden auf die gleiche Weise wie oben bestimmt. Wie bereits weiter oben erwähnt wurde, wird für den ersten Ausgang einer Gruppe derjenige mit der größten Anzahl an Literalen genommen. Dabei wird zusätzlich die Zahl der Nullen und Einsen in der Wertetabelle der gewählten Funktion ermittelt. Im Falle, dass $N_i^1 > N_i^0$, wird an dem Ausgang ein Invertor aufgestellt.

An jeder der 11 betrachteten Schaltungen wurden die beiden vorgestellten Algorithmen realisiert. Die Fläche der Komplementärschaltung A_g^1, welche als Ergebnis des Algorithmus 4.3. erhalten wurde, sowie die Fläche der Komplementärschaltung A_g^2, welche durch den Algorithmus 4.4. erhalten wurde, werden mit der Fläche der nicht optimierten funktionalen Schaltung A verglichen. Zusätzlich wurde in den Experimenten die Implementierung der folgenden Strukturbesonderheit der komplementären Logik für den 1-aus-3 Code realisiert:

> In der vorgestellten Struktur der komplementären Schaltung $g(x)$ aus Abschnitt 3.6 (Abbildung 3.12) ist die Möglichkeit gegeben, das NOR-Gatter ($\overline{f}_1(x), \overline{f}_2(x), \overline{f}_3(x)$) entweder mit dem Ausgang des AND1-Gatters ($f_1(x) \wedge f_2(x)$) (Fall 1, Tabelle 4.4) oder mit dem Ausgang des AND2-Gatters ($[f_1(x) \vee f_2(x)] \wedge f_3(x)$) (Fall 2, Tabelle 4.5) durch ein OR-Gatter zu verbinden. In beiden Fällen wird an den Leitungen $h_1(x), h_2(x)$ und $h_3(x)$ der gesamten fehlererkennenden Schaltung die Erzeugung von 1-aus-3 Codewörtern gewährleistet.

Die Antwort auf die Frage, ob zwischen der Bestimmung der Gruppen von Ausgängen mit Algorithmus 4.3. und Algorithmus 4.4. eine Differenz im Flächenverbrauch der komplementären Strukturen $g(x)$ besteht, wird in den Tabellen 4.4 und 4.5 gegeben. In der ersten, zweiten und dritten Spalte der Tabellen 4.4 und 4.5 sind die Namen der Schaltungen, die Fläche A der nicht optimierten funktionalen Schaltung und die Fläche A_{opt} der optimierten Schaltung gegeben. Die Flächen der Komplementärschaltung, deren Gruppen von Ausgängen mit Algorithmus 4.3. (A_g^1) und Algorithmus 4.4. (A_g^2) gebildet wurden, sind in den Spalten 4 und 5 dargestellt. Das prozentuale Verhältnis der Fläche A zu den Flächen A_g^1, A_g^2 für jede der 11 Schaltungen ist in den Spalten 6 und 7 angeführt.

Aus der Tabelle 4.4 folgt: der Mittelwert der Komplementärschaltungsfläche ist bei Verwendung des ersten Algorithmus zur Fixierung der Gruppen von Ausgängen unwesentlich besser (um 1,2%), bei welchem für den ersten Ausgang derjenige mit der maximalen Anzahl an Literalen gewählt wird. In einigen Fällen ist es angebracht, den zweiten Algorithmus einzusetzen (*pm1.blif, sct.blif*).

Die erhaltenen prozentualen Unterschiede sind unbedeutend. Auf diese Weise wurde auf experimentellem Wege bewiesen, dass für die Bestimmung der Komplexität eines funktionalen Ausganges sowohl die Anzahl der Nullen bei Simulierung der Schaltung mit pseudozufälligen Vektoren als auch die Anzahl der Literale der an diesem Ausgang realisierten Booleschen Funktion verwendet werden kann.

Die Verwendung des einen oder anderen der beiden Kriterien beeinflusst die Fläche der Schaltung $g(x)$ nicht wesentlich. Daher kann die Wahl des Algorithmus zur Zusammenstellung der Gruppen auf

$Circuit$	inputs	outputs	A	A_{opt}	A_g^1	A_g^2	A/A_g^1	A/A_g^2
ttt2	24	21	442	300	236	245	53,4	55,4
c8	28	18	328	214	187	195	57,0	59,4
sct	19	15	205	117	116	109	56,6	53,2
lal	26	19	249	139	145	146	58,2	58,6
cu	14	11	105	86	57	58	54,3	55,0
pm1	16	13	85	86	72	65	84,7	76,5
term1	34	10	797	230	195	186	24,5	23,3
vda	17	39	1593	810	723	717	45,4	45,0
x1	51	35	661	458	393	390	59,4	59,0
cht	47	36	419	262	282	283	67,3	67,5
unreg	36	16	209	171	142	142	68,0	68,0
Mittelwert							57,2%	56,4%

Tabelle 4.4: Algorithmus 4.3., Algorithmus 4.4 zur Bestimmung der Ausgänge, Verbindung des NOR-Elementes mit der ersten Leitung

der Einfachheit der Implementierung gegründet sein, oder sie hängt ab von der Softwareausstattung (*CAD Tools*). Die Ergebnisse, welche durch die Experimente zur Bestimmung der Fläche der Schaltung

$Circuit$	inputs	outputs	A	A_{opt}	A_g^1	A_g^2	A/A_g^1	A/A_g^2
ttt2	24	21	442	300	236	245	53,4	55,4
c8	28	18	328	214	183	189	55,8	57,6
sct	19	15	205	117	115	109	56,0	53,2
lal	26	19	249	139	144	145	58,0	58,2
cu	14	11	105	86	57	58	54,3	55,0
pm1	16	13	85	86	72	64	84,7	75,3
term1	34	10	797	230	195	186	24,5	23,3
vda	17	39	1593	810	726	727	45,6	45,6
x1	51	35	661	458	393	389	59,4	58,8
cht	47	36	419	262	282	282	67,3	67,3
unreg	36	16	209	171	142	142	68,0	68,0
Mittelwert							57%	56,2%

Tabelle 4.5: Beide Algorithmen zur Bestimmung der Ausgänge, Verbindung des NOR-Elementes mit der zweiten Leitung

$g(x)$ für den zweiten Fall erhalten wurden, sind in Tabelle 4.5 angegeben. Bei einer solchen Variante der internen Struktur der Schaltung $g(x)$ bei Verwendung des zweiten Algorithmus zur Bildung der Gruppen von Ausgängen ist die Fläche A_g^2 im Mittel um 0.8% geringer als die Fläche A_g^1.

Der Vergleich der Ergebnisse der Tabellen 4.4 und 4.5 führt zu der Erkenntnis, dass es praktisch keinen Unterschied zwischen der Verbindung des NOR-Elementes mit der ersten oder der Verbindung

des NOR-Elementes mit der zweiten Leitung gibt. Diese Tatsache bestätigt die weiter oben in diesem Abschnitt gezogene Schlussfolgerung über die Unabhängigkeit der Fläche der Schaltung $g(x)$ von der Art und Weise des Anschlusses des NOR-Elementes in der Komplementärschaltung.

4.8 Alternative Variante zur Konstruktion einer Komplementärschaltung für den 1-aus-3 Code

Zu Beginn der Arbeit wurde bei der Aufzählung der Vorteile der neuen Methode der Logischen Ergänzung der große Freiheitsgrad bzw. die hohe Anzahl der Realisationsvarianten der Struktur der komplementären Logik genannt.

Die Struktur der komplementären Schaltung $g(x)$ in Abbildung 3.12 aus Abschnitt 3.6 ist nicht die einzige Realisationsmöglichkeit. Am Beispiel des 1-aus-3 Codes wird dieser Fakt bewiesen, und es wird in diesem Abschnitt eine Logische Ergänzungsschaltung g_{alt}, dargestellt, welche „alternativ" genannt wird. Für diese Schaltung werden ebenfalls Experimente zur Bestimmung ihrer Charakteristik durchgeführt.

In der Tabelle 4.6 sind für acht Eingangsvektoren der ursprünglichen Schaltung $f(x)$ acht Ausgangswerte der Ausgänge $g_2(x), g_3(x)$ der Ergänzungsschaltung und die entsprechenden acht 1-aus-3 Codewörter an den Leitungen $h_1(x), h_2(x), h_3(x)$ dargestellt. Aus der Tabelle 4.6 folgt, dass die

f_1	f_2	f_3	g_2	g_3	h_1	h_2	h_3
0	0	0	0	1	0	0	1
0	0	1	0	0	0	0	1
0	1	0	0	0	0	1	0
0	1	1	0	1	0	1	0
1	0	0	0	0	1	0	0
1	0	1	0	1	1	0	0
1	1	0	1	0	1	0	0
1	1	1	1	1	1	0	0

Tabelle 4.6: Wertetabelle der alternativen komplementären Schaltung $g_{alt}(x)$

Ausgangsfunktionen $g_2(x), g_3(x)$ der komplementären Schaltung g_{alt} durch die folgenden Ausdrücke beschrieben werden:

$$\begin{aligned} g_2(x) &= f_1(x) \wedge f_2(x), \\ g_3(x) &= f_3(x) \overline{\oplus} (f_1(x) \vee f_2(x)). \end{aligned} \tag{4.4}$$

Das alternative Schema der Logischen Ergänzung $g_{alt}(x)$ für eine Fehlererkennungsschaltung unter Verwendung des 1-aus-3 Code zeigt Abbildung 4.11. Das Auftreten von ausschließlich Codewörtern an den Leitung $h_1(x), h_2(x), h_3(x)$ verifiziert, dass diese Schaltung als Logische Ergänzungsschaltung verwendet werden kann. Für jede Eingangskombination X generiert die alternative Ergänzungsschaltung Werte, die durch Addition mit den Werten von $f_1(x), f_2(x), f_3(x)$ durch das Modul 2 an den Leitungen $h_1(x), h_2(x), h_3(x)$ 1-aus-3 Codewörter bilden. Verglichen mit der Ergänzungsschaltung in Abbildung 3.12 aus dem Abschnitt 3.6 kann man annehmen, dass die Schaltung in Abbildung 4.11

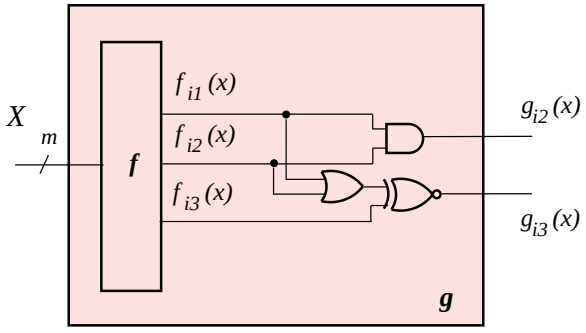

Abbildung 4.10: Alternatives Schema der Logischen Ergänzung für den 1-aus-3 Code

einen wesentlich geringeren Flächeverbrauch aufweist. Diese Vermutung wurde im Rahmen dieser Arbeit mit Hilfe von Experimenten überprüft.

In der Tabelle 4.7 sind die Ergebnisse des Experimentes zur Bestimmung der benötigten Realisationsfläche der alternativen Schaltung für 11 MCNC-Benchmark-Schaltungen gegeben. Für alle kom-

$Circuit$	A	A_{opt}	A_g^1	A_g^{wi}	A_g^{ti}	$\frac{A_g^1}{A}$	$\frac{A_g^1}{A_{opt}}$	$\frac{A_g^{wi}}{A}$	$\frac{A_g^{wi}}{A_{opt}}$	$\frac{A_g^{ti}}{A}$	$\frac{A_g^{ti}}{A_{opt}}$
ttt2	442	300	208	200	183	47,0	69,3	45,2	66,6	41,4	61,0
c8	328	214	153	151	158	46,6	71,5	46,0	70,5	48,2	73,8
sct	205	117	86	87	82	42,0	73,5	42,4	74,3	40,0	70,0
lal	249	139	119	114	103	47,8	85,6	45,8	82,0	41,4	74,1
cu	105	86	38	45	44	36,2	44,2	42,8	52,3	42,0	51,2
pm1	85	86	42	45	43	49,4	48,8	53,0	52,3	50,5	50,0
term1	797	230	148	164	157	18,5	64,3	20,5	71,3	19,6	68,2
vda	1593	810	650	676	588	40,8	80,2	42,4	83,4	37,0	72,5
x1	661	458	356	352	314	53,8	77,7	53,2	76,8	47,5	68,5
cht	419	262	217	222	220	51,7	82,8	53,0	84,7	52,5	84,0
unreg	209	171	122	127	122	58,3	71,3	60,7	74,2	58,3	71,3
Mittelwert						44,7%	70%	46%	71,7%	43,5%	67,7%

Tabelle 4.7: Experimentelle Ergebnisse

binatorischen Schaltungen wurde bei Konstruktion der Schaltung $g_{alt}(x)$ die Zusammenstellung der Gruppen auf Grundlage des Algorithmus 4.3., welcher im letzten Abschnitt beschrieben wurde, durchgeführt. Dabei wurden drei Varianten der Installation von Invertoren an den Ausgängen der Schaltung realisiert.

Die erste Variante ist die direkte Realisierung des Algorithmus 4.3., wobei die Invertoren an den Ausgängen der Schaltung in Abhängigkeit der Verhältnisses zwischen der Nullen- und Einsenanzahl aufgestellt werden. Die zweite Variante der Ergänzungsschaltung ist charakterisiert durch die Abwesenheit von Invertoren an den Ausgängen. Die dritte Variante ist die Realisierung des Algorithmus, bei welchem die Invertoren an jedem der Ausgänge installiert werden. Dieser Fall wird „totale Invertie-

rung" genannt.

In der ersten Spalte der Tabelle 4.7 sind die Namen der Schaltungen aufgeführt. Die Spalten 2 und 3 beschreiben die benötigte Realisationsfläche der ursprünglichen Schaltung $f(x)$ vor A und nach A_{opt} der Optimierung. Die Werte dieser Parameter werden verglichen mit den entsprechenden Werten, die bei Realisierung der alternativen Ergänzungsschaltung erhalten wurden.

Die benötige Fläche A_g^1 der Realisation der alternativen Ergänzungsschaltung ist in Spalte 4 dargestellt. Die erhaltenen Flächenwerte für die erste, zweite und dritte Realisationsvariante der Logischen Ergänzung werden entsprechend in den Spalten 4 (A_g^1), 5 (A_g^{wi}) und 6 (A_g^{ti}) vorgestellt. In den nächsten Spalten 7-12 sind für jede der 11 Schaltungen die Verhältnisse der erhaltenen Schaltungsflächen $g_{alt}(x)$ jeder Variante zur den Flächen der originalen nicht-optimierten und der optimierten Schaltung gegeben.

Zusätzlich wurden Experimente durchgeführt mit dem Ziel der Bestimmung der genannten Parameter für den Fall, dass die Invertoren nacheinander an den Ausgängen der Gruppen der Komplementärschaltung $g_{alt}(x)$ aufgestellt werden (drei Varianten). Anschließend wird die Realisationsfläche der Komplementärschaltung eingeschätzt. Die schematische Aufstellung der Invertoren auf $g_{i2}(x), g_{i3}(x)$ ist in Abbildung 4.11 dargestellt. Das Fragezeichen bedeutet, dass die Benutzung von Invertoren an den Ausgängen der originalen Schaltung $f_{i1}(x), f_{i2}(x), f_{i3}(x)$ durch den Algorithmus 4.3. berechnet wird.

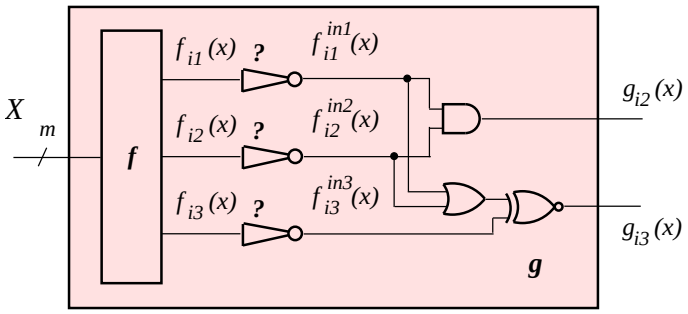

Abbildung 4.11: Alternatives Schema der Logischen Ergänzung mit Invertoren

In der Tabelle 4.8 sind die experimentelle Ergebnisse der Bestimmung des Flächebedarfs der komplementären Schaltungen für 11 MCNC-Benchmarks-Schaltungen dargestellt. In den Spalten stehen: A_g^{iA} - Fläche der Schaltung $g(x)$ bei Aufstellung eines Invertoren an die Ausgangsleitung $g_{i2}(x)$, A_g^{iB} - die Fläche der Schaltung $g(x)$ bei Aufstellung eines Invertoren an den Ausgang $g_{i3}(x)$ und dementsprechend A_g^{iAB} - die Fläche der Schaltung $g(x)$ bei gleichzeitiger Aufstellung zweier Invertoren an die beiden Ausgänge $g_{i2}(x), g_{i3}(x)$ der Schaltung $g(x)$.

Die experimentellen Ergebnisse aus Tabelle 4.7 und Tabelle 4.8 gestatten, die folgenden Schlüsse zu ziehen:

1. Die mittlere Fläche der alternativen Struktur der Logische Ergänzung $g(x)$ für den 1-aus-3 Code ist um 18,5% kleiner als die mittlere Fläche der Schaltungen $g(x)$ aus Abschnitt 3.6;

$Circuit$	A	A_{opt}	A_g^{iA}	A_g^{iB}	A_g^{iAB}	$\frac{A_g^{iA}}{A}$	$\frac{A_g^{iA}}{A_{opt}}$	$\frac{A_g^{iB}}{A}$	$\frac{A_g^{iB}}{A_{opt}}$	$\frac{A_g^{iAB}}{A}$	$\frac{A_g^{iAB}}{A_{opt}}$
ttt2	442	300	213	213	222	48,1	71,0	48,1	71,0	50,2	74,0
c8	328	214	151	144	144	46,0	70,5	44,0	67,2	44,0	67,2
sct	205	117	93	94	94	45,4	79,4	45,8	80,3	45,8	80,3
lal	249	139	117	119	121	47,0	84,2	47,8	85,6	48,6	87,0
cu	105	86	36	35	38	34,3	41,8	33,3	40,7	36,2	44,2
pm1	85	86	45	42	44	53,0	52,3	49,4	48,8	51,7	51,2
term1	797	230	175	176	177	22,0	76,0	22,0	76,5	22,2	77,0
vda	1593	810	572	582	576	36,0	70,6	36,5	71,8	36,1	71,1
x1	661	458	351	336	338	53,1	76,6	50,8	73,3	51,1	73,8
cht	419	262	212	216	223	50,6	81,0	51,5	82,4	53,2	85,1
unreg	209	171	122	127	127	58,3	71,3	60,7	74,3	60,7	74,3
Mittelwert						46,7%	72%	44,9%	70,2%	45,4%	71,4%

Tabelle 4.8: Experimentelle Ergebnisse

2. Das Erreichen einer minimalen Fläche der Ergänzungsschaltung erfordert die Entwicklung eines speziellen Algorithmus. Auf experimentellem Wege wurde gezeigt, dass keine Anhängigkeit zwischen der Anzahl der aufgestellten Invertoren und den Werten der Flächen besteht.

5 Notwendige und hinreichende Bedingungen für die Existenz vollständig selbstprüfender Schaltungen

In diesem Kapitel wird die Aufgabe der Synthese vollständig selbstprüfender kombinatorischer Schaltungen betrachtet. Es wird das Theorem bewiesen, welches die Bedingungen bestimmt und vorgibt, unter denen die Konstruktion vollständig selbstprüfender Schaltungen auf Grundlage der vorgestellten Methode der Logischen Ergänzung garantiert wird. Es werden notwendige und hinreichende Bedingungen für die Konstruktion vollständig selbstprüfender Schaltungen bestimmt.

Am Beispiel traditioneller Methoden zur Konstruktion selbstprüfender Schaltungen wird gezeigt, dass die Synthese selbstprüfender Schaltungen unter Verwendung solcher Methoden aufgrund des Fehlens der notwendigen Selbstprüfungseigenschaft manchmal gänzlich unmöglich ist [83].

5.1 Optimierung komplementärer Schaltungen

In den vorigen Kapiteln dieser Arbeit wurde bewiesen, dass im Vergleich mit traditionellen Methoden der funktionalen Diagnostik die neue Methode eine höhere Fehlerüberdeckung besitzt, dabei eine kleinere Realisationsfläche benötigt.

Am Anfang der Arbeit wurde die neue Methode als eine Methode bezeichnet, welche die Konstruktion vollständig selbstprüfender Schaltungen, deren praktische Wichtigkeit in Abschnitt 2.5 beschrieben wurde sichert.

Die Erschaffung einer beliebigen Methode der funktionalen Diagnostik zur Erhöhung der Fähigkeit digitaler Systeme, Fehler zu erkennen, erfordert heutzutage die Lösung zweier grundlegender Probleme:

- das Erreichen der kleinstmöglichen Realisationsfläche;
- die Sicherstellung eines den Checker prüfenden Testsatzes an den Eingängen der Kontrollschaltung (des Checkers), welcher notwendig ist für die Gewährleistung der Kontrolle der internen Struktur des Checkers.

Mit anderen Worten, die Beschreibung der neuen Methode der Logischen Ergänzung zur Konstruktion selbstprüfender Schaltungen bleibt unvollständig, wenn nicht die Gewährleistung der Selbstprüfungseigenschaften überprüft wird.

1. **Selbsttestend**: Eine Schaltung heißt **selbsttestend** (bezuglich einer Fehlermenge Φ), wenn für jeden Fehler $\varphi \in \Phi$ wenigstens ein Input-Code-Wort existiert, so dass der Output kein Code-Wort ist;
2. **Fehlersicher**: Eine Schaltung heißt **fehlersicher**, wenn es für jeden Fehler $\varphi \in \Phi$ kein Input-Code-Wort gibt, dass zu einem falschen Output-Code Wort führt.

3. **Vollständig selbstprüfend**: Eine Schaltung ist **vollständig selbstprüfend** wenn sie selbsttestend und fehlersicher ist.

Die für die Paritätsbitprüfung konstruierte Fehlererkennungsschaltung gewährleistet die Selbstprüfbarkeit der Schaltung nur, wenn die Selbsttestbarkeit der XOR-Elemente sichergestellt ist. Die Selbstprüfbarkeit der Schaltung wird nur dann gewährleistet, wenn an den Eingängen jedes der XOR-Elemente das Auftreten eines Testsatzes garantiert wird.

Mit dem Ziel der Bestimmung der Fähigkeit der bekannten Fehlererkennungsmethoden, vollständig selbstprüfende Kontrollschaltungen zu konstruieren, wird nun die kombinatorische Schaltung $H(x)$ betrachtet. Die kombinatorische Schaltung $H(x)$ besitzt vier funktionale Ausgänge $f_1(x), f_2(x), f_3(x), f_4(x)$. In der Tabelle 5.1 sind für $H(x)$ die Werte der Ausgänge dargestellt.

x_1	x_2	x_3	f_1	f_2	f_3	f_4	g
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	1
0	1	0	0	0	0	1	1
0	1	1	0	1	0	0	1
1	0	0	0	1	0	0	1
1	0	1	0	0	1	1	0
1	1	0	0	0	1	1	0
1	1	1	1	0	0	0	1

Tabelle 5.1: Beispiel für eine kombinatorische Schaltung

Aus der Tabelle 5.1 folgt, dass an den Ausgängen der Schaltung $H(x)$ vierstellige Vektoren gebildet werden, in welchen die Zahl der Einsen nicht größer ist als zwei. Deshalb ist es möglich, für die Gewährleistung der Kontrolle von $H(x)$ einen systematischen nicht-separierbaren 2-aus-5-Code zu benutzen. Dafür ist es nur erforderlich, die bereits bestehenden funktionalen Ausgänge durch einen zusätzlichen Ausgang zu ergänzen. Die Funktion am ergänzenden Ausgang $g(x)$ muss eine 1 liefern, wenn an den Ausgängen $f_1(x), f_2(x), f_3(x), f_4(x)$ nur eine 1 generiert wird. Und dementsprechend ist die Funktion des Ausgangs $g(x)$ gleich Null, wenn an den funktionalen Ausgängen bereits zwei Einsen sind, so dass ein 2-aus-5 Codewort entsteht. Die Analyse von Tabelle 5.1 liefert für die Überprüfung des Checkers vier 2-aus-5-Code-Eingangsvektoren $\{10001\}$, $\{00011\}$, $\{01001\}$ und $\{00110\}$. D.h., bei der Benutzung dieser Methode unter Verwendung des 2-aus-5 Codes ist die Zahl der Vektoren im Testsatz gleich vier. In [84] wird am Beispiel der Verwendung eines m-aus-n Codes bewiesen, dass für die Gewährleistung der Selbsttestbarkeit des Checkers an seinen Eingängen das Auftreten eines Testsatzes von mindestens $t,\ t > n$ Vektoren notwendig ist, wobei n die Länge der Codewörter ist. Ebenso wird in einer anderen Arbeit [37] bewiesen, dass für ein logisches XOR-Element mit zwei Eingängen das Auftreten eines Testsatzes von mindestens vier Eingangskombinationen garantiert sein muss. Diese Eingangskombinationen sind $\{01\}$, $\{10\}$, $\{00\}$ und $\{11\}$. Damit kann der Checker nicht selbsttestend sein.

Die gleiche Situation entsteht bei der Verwendung separierbarer Fehlererkennungscodes für die Schaltungskontrolle. Bei der Implementierung des Berger-Codes in die Kontrollschaltung ist der Checker aus dem gleichen Grunde nicht selbsttestend. In der Kontrollschaltung wird wieder das Auftreten eines Testsatzes mit der erforderlichen Anzahl von Eingangskombinationen nicht garantiert.

In der Abbildung 5.1 ist die Fehlererkennungsstruktur für die Schaltung $H(x)$ unter Verwendung der Paritätsbitprüfung dargestellt. Im Falle der Verwendung der Paritätsbitprüfung für die Kontrolle der Schaltung muss, um die Eigenschaften der Selbstprüfbarkeit zu gewährleisten, das Auftreten eines Testsatzes an jedem der drei logischen XOR-Elemente garantiert werden.

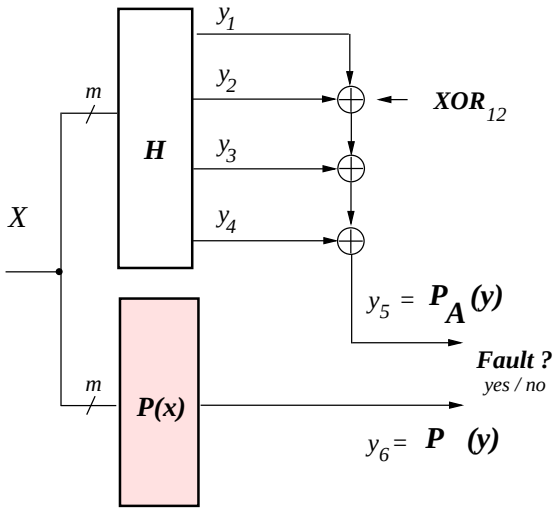

Abbildung 5.1: Fehlererkennungsschaltung unter Verwendung des Paritätscodes

Die Paritätsfunktion $P_A(y)$ der funktionalen Ausgänge, deren Wert mit dem der Funktion $P(y)$ verglichen wird, wird aus der Formel $P_A(y) = (y_1 \oplus y_2 \oplus y_3) \oplus y_4$ bestimmt. Die konstruierte Fehlererkennungsschaltung ist ebenfalls nicht vollständig selbstprüfend. Aus der Tabelle 5.1 folgt, dass an einem beliebigen Paar $f_i(x), f_j(x)$ der funktionalen Ausgänge der Schaltung das Auftreten nur dreier Kombinationen gewährleistet wird. Aus diesem Grunde besteht der Testsatz - zum Beispiel für das Elemente XOR_{12} - nur aus drei Kombinationen $\{01\}$, $\{10\}$, $\{00\}$, und daher ist XOR_{12} kein selbsttestendes Element. Die in Abbildung 5.1 dargestellte Fehlererkennungsschaltung besitzt also nicht die Eigenschaft der Selbstprüfbarkeit. Somit sind die Bedingungen der Selbstprüfbarkeit unter Verwendung der Gruppenparitätsbitprüfung nicht erfüllt.

Die letzte traditionelle Fehlererkennungsmethode, die hinsichtlich der Möglichkeit der Konstruktion selbstprüfender Fehlererkennungsschaltungen untersucht wird, ist die Methode „Verdopplung und Vergleich". Die Schaltung $H(x)$ wird dupliziert, und im Checker werden die Funktionen $f_1(x), f_2(x), f_3(x), f_4(x)$ der Schaltung $H(x)$ mit ihren inversen Werten $\overline{f}_1(x), \overline{f}_2(x), \overline{f}_3(x), \overline{f}_4(x)$ der aus der Verdopplung entstandenen Schaltung H^*(x) verglichen. Wie die zuvor untersuchten Methoden gewährleistet auch die Methode „Verdopplung und Vergleich" in der Struktur nicht das Vorhandensein der Eigenschaft Selbstprüfbarkeit. Für die Sicherstellung der Selbsttestbarkeit fehlt in dem Testsatz noch ein Vektor (Tabelle 5.1). Somit wurde in diesem Abschnitt am Beispiel einer Schaltung gezeigt, dass nicht eine der bekannten Methoden der funktionalen Diagnostik die Konstruktion einer Fehlererkennungsschaltung garantiert, welche die Eigenschaft der Selbstprüfbarkeit besitzt.

In dem nächsten Abschnitt dieses Kapitels wird am Beispiel kombinatorischer Schaltungen untersucht, ob es mit der neuen Methode der Logischen Ergänzung möglich ist, die Eigenschaft der Selbstprüfbarkeit in Fehlererkennungsschaltungen zu garantieren. Wie an der oben beschriebenen Beispiel-

schaltung, so wird an anderen Beispielschaltungen die Implementierung mit der Methode der Logischen Ergänzung aufgezeigt sowie anschließend die Eigenschaft der Selbstprüfbarkeit überprüft.

5.2 Notwendige und hinreichende Bedingungen für die Existenz vollständig selbstprüfender Schaltungen

Nach Kenntnis des Autors werden in dieser Arbeit nach [83] zum ersten Mal notwendige und hinreichende Bedingungen für die Existenz einer vollständig selbstprüfenden Schaltung gegeben. Die Bedingungen werden am Beispiel des 1-aus-n Codes aufgestellt. In Theorem 2 werden Bedingungen für die Gewährleistung der Eigenschaft der Selbstprüfbarkeit in Fehlererkennungsschaltung formuliert, welche mit der neuen Methode der Logischen Ergänzung konstruiert werden. In diesem Abschnitt wird neben der Formulierung der notwendigen und hinreichenden Bedingungen für die Existenz vollständig selbstprüfender Schaltungen ein Algorithmus zur Konstruktion vollständig selbstprüfender Schaltungen vorgestellt.

Es wird nun eine Fehlererkennungsschaltung unter Verwendung der Logischen Ergänzung betrachtet, dargestellt in Abbildung 5.2. In der Schaltung werden die funktionalen Ausgänge $f_1(x), \cdots, f_r(x)$ mit Hilfe der ergänzenden Funktionen $g_1(x), \cdots, g_r(x)$ in die Funktionen $h_1(x), \cdots, h_r(x)$ transformiert. Die Funktionen $f_{r+1}(x), \cdots, f_n(x)$ werden nicht transformiert, d.h., sie bleiben erhalten: $f_{r+1}(x) = h_{r+1}(x), \cdots, f_n(x) = h_n(x)$.

Die Gewährleistung einer großen Anzahl solcher gleichbleibender Funktionen stellt eine effektive Methode bei der Organisation einer Logischen Ergänzungsschaltung dar. In diesem Falle verringert sich die Komplexität der Schaltung, da die Installation von XOR-Elementen und die Realisierung der entsprechenden Ergänzungsfunktionen nicht erforderlich ist. Außerdem vereinfacht die Verringerung der Anzahl der XOR-Elemente die Aufgabe der Bildung von Testvektoren an den ihren Eingängen.

Das folgende Theorem bestimmt die Bedingungen, unter denen die Konstruktion einer vollständig selbstprüfenden Schaltung auf Grundlage der neuen Methode der Logischen Ergänzung garantiert wird.

Theorem 2 *Sei f_C ein kombinatorischer Schaltkreis, welcher an seinen n Ausgängen die Booleschen Funktionen $y_1 = f_1(x), \cdots, y_r = f_r(x)$, $y_{r+1} = f_{r+1}(x), \cdots, y_n = f_n(x)$ implementiert mit einem Satz tatsächlich angelegter Eingangsbelegungen X, wobei alle Booleschen Funktionen $f_i(x)$, $i = 1, \cdots, n$ für $x \in X$ nicht konstant sind.*

Dann ist es möglich, einen vollständig selbstprüfenden Checker für einen 1-aus-n Code unter Verwendung einer komplementären Schaltung g so zu entwerfen, dass die Ausgänge

$y_{r+1} = f_{r+1}(x), \cdots, y_n = f_n(x)$, $n \geq r+1$

von f_C nicht ergänzt und dass die Ausgänge $y_1 = f_1(x), \cdots, y_r = f_r(x)$ mit $r \geq 2$ von f_C ergänzt sind durch die Funktionen $g_1(x), \cdots, g_r(x)$, welche durch die komplementäre Schaltung g genau dann implementiert werden, wenn die folgenden Bedingungen erfüllt sind:

1. *Für $n > r+1$ sind die Paare der nicht ergänzten Funktionen $f_i(x), f_j(x)$ mit $i, j \in \{r+1 \cdots, n\}$, $i \neq j$ paarweise orthogonal,*

$$\sum_{x \in X} f_i(x) f_j(x) = 0, \quad \text{für} \quad i,\ j \in \{r+1, \cdots, n\}, i \neq j.$$

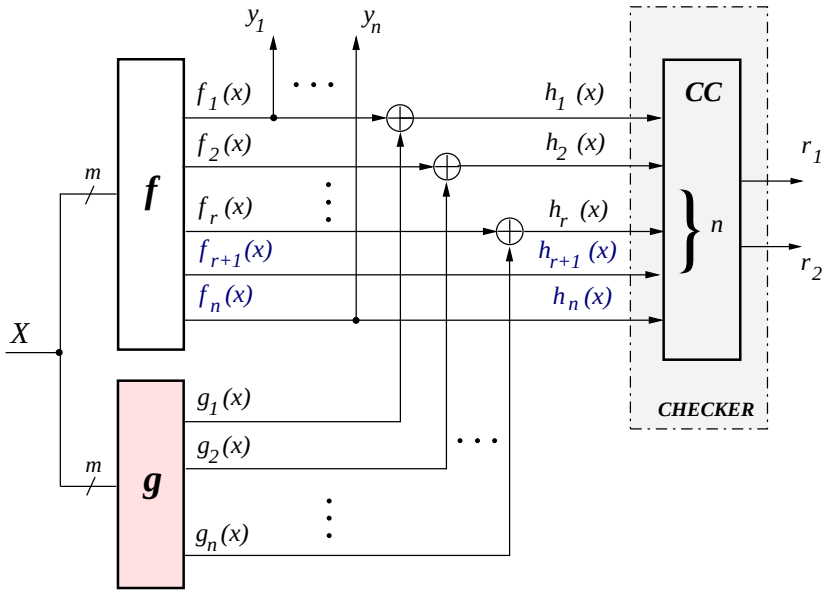

Abbildung 5.2: Allgemeine Struktur der neuen Fehlererkennungsmethode unter Verwendung einer komplementären Schaltung

2. *Sei $\tilde{X}$ eine Teilmenge der Eingänge, für die einer der nicht ergänzten Ausgänge $y_{r+1} = f_{r+1}(x)$, $\cdots, y_n = f_n(x)$ gleich 1 ist,*

   $$\tilde{X} = \{x / \exists i \in \{r+1, \cdots, n\},\ f_i(x) = 1, x \in X\}.$$

   Wenn für jeden ergänzten Ausgang $y_i = f_i(x), i \in \{1, \cdots, r\}$ von f_C eine Teilmenge $\chi_i = \{x_i^1, x_i^0\} \subset X \setminus \tilde{X}$ von Eingängen existiert mit $f_i(x_i^1) = 1$, $f_i(x_i^0) = 0$ und mit $\chi_i \cap \chi_j = \oslash$ für $i, j \in \{1, \cdots, r\},\ i \neq j$.

3. *Für $i = 1, \cdots, r$ ist*

   $$| \mathit{On}\ (f_i)(x) | \geq 2 \quad \textit{und} \quad | \mathit{Off}\ (f_i)(x) | \geq 2.$$

Beweis [83]:
Zuerst wird gezeigt, dass die Bedingungen des Theorems 2 notwendig sind.

5.2.1 Notwendige Bedingungen

Der Checker soll vollständig selbstprüfend sein für einen 1-aus-n Code.

Um alle Einfach-stuck-at-1/0 Fehler des Checkers zu erkennen wird laut [84] ein Testsatz mit $t \geq n$ Test-Eingängen benötigt. Deshalb müssen für einen 1-aus-n Code Checker alle n 1-aus-n Codewörter an die Checkereingänge angelegt werden, und sie müssen durch die n Ausgänge $h_1, \cdots, h_n$ generiert werden. Um die r XOR-Gatter $\mathrm{XOR}_1, \cdots, \mathrm{XOR}_r$ vollständigen zu testen, müssen an jedes dieser XOR-Gatter die Eingangsvektoren $\{00\}, \{01\}, \{10\}, \{11\}$ angelegt werden.

Es ist offensichtlich, dass die modifizierten Ausgänge $y_{r+1} = f_{r+1}(x), \cdots, y_n = f_n(x)$ orthogonal sein müssen, sonst würde für einige Eingangsvektoren ein Nicht-Codewort generiert werden. Es soll zunächst der Fall betrachtet werden, dass $x \in \tilde{X}$.

Fall 1 ($x \in \tilde{X}$): Hier ist genau einer der nicht ergänzten Ausgänge $y_{r+1} = f_{r+1}(x), \cdots, y_n = f_n(x)$ gleich 1. Um ein 1-aus-n Codewort zu generieren, müssen die Ausgänge $y_1 = h_1(x), \cdots, y_r = h_r(x)$ gleich Null sein. Das setzt voraus, dass $h_i(x) = f_i(x) \oplus g_i(x) = 0$ oder $g_i(x) = f_i(x)$ für $i = 1, \cdots, r$ und $x \in \tilde{X}$ ist. Die Gatter $\text{XOR}_1, \cdots, \text{XOR}_r$ werden nicht mit $\{1\ 0\}$ und $\{0\ 1\}$ getestet.

Fall 2 ($x \in X \backslash \tilde{X}$): Angenommen, $x \in X \backslash \tilde{X}$. So sind alle Ausgänge $y_{r+1}(x) = f_{r+1}(x), \ldots, y_n(x) = f_n(x)$ gleich 0. Genau einer der ergänzten Ausgänge, angenommen $y_i = h_i(x) = f_i(x) \oplus g_i(x)$, $i \in \{1, \cdots, r\}$, muss gleich 1 sein, alle anderen Ausgänge $y_j = h_j(x) = f_j(x) \oplus g_j(x)$, $j \neq i$ gleich Null. Das ist nur möglich für $f_i(x) = 1$, oder für $g_i(x) = 0$ oder $f_i(x) = 0$ und $g_i(x) = 1$ und für $f_j(x) = g_j(x)$, $i \neq j$ mit $i, j \in \{1, \cdots, r\}$. Um das XOR-Gatter XOR_i mit $\{1\ 0\}$ und $\{0\ 1\}$ zu testen, sind beide Fälle notwendig, und so kann man schließen, dass zwei verschiedene *Inputs* existieren: $x_i^1, x_i^0 \in X \setminus \tilde{X}$ mit $f_i(x_i^1) = 1$ und $f_i(x_i^0) = 0$.

Für $j \neq i$ mit $i, j \in \{1, \cdots, r\}$ ist $f_j(x_i^1) = g_j(x_i^1)$ und $f_j(x_i^0) = g_j(x_i^0)$. Sei $\chi_i = \{x_i^1, x_i^0\}$. Da alle r 1-aus-n Codewörter

$$\{\underbrace{1\ 0\ \cdots\ 0}_{r}\ 0 \cdots 0\} \quad , \cdots, \quad \{\underbrace{0\ \cdots\ 0\ 1}_{r}\ 0 \cdots 0\}$$

für eine Menge $x \in X \setminus \tilde{X}$ generiert werden müssen und da alle XOR-Elemente $\text{XOR}_1, \cdots, \text{XOR}_r$ vollständig getestet werden müssen, müssen die r Sätze $\chi_j = \{x_j^1, x_j^0\}$ mit $f_j(x_j^1) = 1$ und $f_j(x_j^0)$ für $j = 1, \cdots, r$ existieren. Für $x \in \chi_i$ sind $f_i(x) = \overline{g}_i(x)$ und $f_j(x) = g_j(x)$ für $i \neq j$ und folglich sind χ_i und χ_j disjunkt, $\chi_i \cap \chi_j = \oslash$.

Um die XOR-Elemente XOR_i mit $i = 1, \cdots, r$ auch mit $\{00\}$ und $\{11\}$ zu testen - zusätzlich zu den Eingängen x_i^1 und x_i^0 mit $1 = f_i(x_i^1) = \overline{g}_i(x_i^1)$ und $0 = f_i(x_i^0) = \overline{g}_i(x_i^0)$ müssen zwei weitere Eingänge $\tilde{x}_i^1$ und $\tilde{x}_i^0$ mit $f_i(\tilde{x}_i^1) = g_i(\tilde{x}_i^1) = 1$ und $f_i(\tilde{x}_i^0) = g_i(\tilde{x}_i^0) = 0$ existieren, und es lässt sich schließen:
$|\ \text{On}\ (f_i)(x)\ | \geq 2 \quad$ und $\quad |\ \text{Off}\ (f_i)(x)\ | \geq 2.$
Dies beendet den Beweis, dass die Bedingungen des Theorems 2 notwendig sind.

5.2.2 Hinreichende Bedingungen

Der Beweis ist in [85] enthalten.

Für die verbleibenden Eingänge von f_C können die Funktionen $g_i(x),\ i = 1, \cdots, r$ leicht ermittelt werden, so dass ein beliebiges 1-aus-n Codewort generiert wird. Für den seltenen Fall, dass nur ein Ausgang ergänzt wird ($r = 1$), wird das folgende Theorem 3 vorgestellt.

Theorem 3 *Sei f_C ein kombinatorischer Schaltkreis, welcher an seinen n Ausgängen die Booleschen Funktionen $y_1 = f_1(x), \cdots, y_r = f_r(x)$, $y_{r+1} = f_{r+1}(x), \cdots, y_n = f_n(x)$ implementiert mit einem Satz tatsächlich angelegter Eingänge X, wobei alle Booleschen Funktionen $f_i(x)$, $i = 1, \cdots, n$ für $x \in X$ nicht konstant sind.*

1. *Für $n > 1$ sind die Paare der nicht ergänzten Funktionen f_i, f_j mit $i, j \in \{r+1, \cdots, n\}$, $i \neq j$ paarweise orthogonal;*

$$\sum_{x \in X} f_i(x) f_j(x) = 0, \qquad \text{for } i, j \in \{r+1, \cdots, n\}, i \neq j.$$

2. *Sei $\tilde{X}$ eine Untermenge von Eingängen, für welche eine der nicht ergänzten Ausgänge $y_2 = f_2(x), \cdots, y_n = f_n(x)$ gleich 1 ist,*
   $\tilde{X} = \{x / \exists i \in \{2, \cdots, n\}, f_i(x) = 1, x \in X\}$

3. *Dann existiert eine Untermenge $\chi_1 = \{x_1^1, x_1^0\} \subset X \setminus \tilde{X}$ von Eingängen mit $f_1(x_i^1) = 1$, $f_1(x_i^0) = 0$, und es existieren zwei Eingänge $\tilde{x}_1^1, \tilde{x}_1^0 \in \tilde{X}$ mit $f_1(\tilde{x}_1^1) = 1$ und $f_1(\tilde{x}_1^0) = 0$.*

Beweis:
Zuerst wird gezeigt, dass die Bedingungen notwendig sind.

Für $x \in \tilde{X}$ ist genau einer der Ausgänge von $f_2(x), \cdots, f_n(x)$ gleich 1, und darum ist für $x \in \tilde{X}$ in $h_1(x) = f_1(x) \oplus g_1(x) = 0$ oder $f_1(x) = g_1(x)$. Für $x \in X \setminus \tilde{X}$ ist $f_2(x) = \cdots = f_n(x) = 0$, was $h_1(x) = f_1(x) \oplus g_1(x) = 1$ oder $g_1(x) = \overline{f}(x)$ impliziert.

Um das XOR-Element XOR_1 mit $\{00\}$ und $\{11\}$ zu testen, müssen zwei Eingänge $\tilde{x}_1^1, \tilde{x}_1^0 \in \tilde{X}$ existieren mit $f_1(\tilde{x}_1^1) = 1$ und $f_1(\tilde{x}_1^0) = 0$, und um das XOR-Element XOR_1 mit $\{01\}$ und $\{10\}$ zu testen, müssen $x_1^1, x_1^0 \in X \setminus \tilde{X}$ existieren mit $f_1(x_1^1) = 1$ und $f_1(x_1^0) = 0$. Es ist leicht zu zeigen, dass die Bedingungen hinreichend sind. Sei nun $g_1(x) = f_1(x)$ definiert für $x \in \tilde{X}$ und $g_1(x) = \overline{f}(x)$ für $x \in X \setminus \tilde{X}$. Gemäß der Bedingungen des Theorems 2 werden alle 1-aus-n Codewörter generiert, und das XOR-Element XOR_1 wird vollständig getestet.

Für praktische Anwendungen ist das folgende Theorem 4 von Interesse [83]. In diesem Theorem werden hinreichende Bedingungen für die Existenz eines vollständig selbstprüfenden Checkers für einen 1-aus-n Code gegeben.

Theorem 4 *Sei f_C eine kombinatorische Schaltung mit einer Menge X von anliegenden Eingangsvektoren, welche an ihren n Ausgängen die Booleschen Funktionen $y_1 = f_1(x), \ldots, y_n = f_n(x)$ implementiert.*

Sei für $j = 1, \ldots, n$, $X_j^0 = \{x, x \in X \wedge f_j(x) = 0\}$ ($X_j^1 = \{x, x \in X \wedge f_j(x) = 1\}$) die Untermenge der anliegenden Eingänge, für die der j-te Ausgang $0(1)$ ist. Wenn diese Untermengen $X_1^0, X_1^1, X_2^0, X_2^1, \ldots, X_n^0, X_n^1$ nach der Anzahl ihrer Elemente aufsteigend als $X'_1, X_2', \ldots, X'_{2n}$ geordnet werden und wenn $\mid X'_k \mid \geq k, k = 2, \ldots, 2n$ und $\mid X'_1 \mid \geq 2$ ist, dann existiert ein vollständig selbstprüfender Checker unter Verwendung einer komplementären Schaltung für einen 1-aus-n Code[1].

Beweis:
Da keine orthogonalen Ausgänge betrachtet werden, ist die Bedingung 1. aus Theorem irrelevant. Es wird nun gezeigt, dass die Bedingungen 2. und 3. gültig sind und dass für jeden Ausgang j die Menge χ_j bestimmt werden kann.

Sei $X'_1 = X_{i1}^{c_{i1}}$ mit $c_{i1} \in \{0, 1\}$. Es wird nun ein beliebiges Element $x_{i1} \in X_{i1}^{c_{i1}}$ als ein Element von χ_{i1} gewählt und die Menge $X'_1 = X_{i1}^{c_{i1}}$ aus der Eingangsmenge $X'_1, \ldots, X'_{2n}$ gelöscht. Ebenso wird x_{i1} aus allen verbleibenden Mengen entfernt. Es ist eindeutig $f_{i1}(x_{i1}) = c_{i1}$.

Sei nun $X_{i2}^{c_{i2}}$ die Menge aus den verbleibenden Mengen mit der kleinsten Elementeanzahl. Es wird ein beliebiges Element $x_{i2} \in X_{i2}^{c_{i2}}$ als ein Element von χ_{i2} gewählt. $X_{i2}^{c_{i2}}$ wird aus der Eingangsmenge und x_{i2} aus den restlichen Eingangsmengen gelöscht. So wird fortgefahren, bis jede der Mengen

[1] *Anmerkung:* Durch Zählen der Nullen und Einsen an den verschiedenen Schaltungsausgänge unter Verwendung pseudozufälliger Eingänge können die Bedingungen des Theorems 4 leicht verifi ziert werden.

$\chi_1, \chi_2, \ldots, \chi_n$ bestimmt und keine der Eingangsmengen übrig ist. Es ist eindeutig $\mid \chi_j \mid = 2$ für $j = 1, \ldots, n$ und $\chi_i \cap \chi_j = \oslash$.

Dies schließt den Beweis.

5.3 Experimentelle Ergebnisse: Bestimmung der orthogonalen Ausgänge

In diesem Abschnitt wird am Beispiel von MCNC-Benchmark-Schaltungen gezeigt, wie die Bedingungen des Theorems 2 auf experimentellem Wege überprüft werden können.

Die erste Aufgabe ist hierbei die Bestimmung des Vorhandenseins orthogonaler Paare von funktionalen Ausgängen (oder Paare orthogonaler Ausgänge) in der Menge $f_i(x), f_j(x),\ i \neq j$. Für die Bestimmung wurde der folgende heuristische Algorithmus 5.1. verwendet.

Algorithmus 5.1:

Die einfachste Methode zur Bestimmung der Orthogonalität ist die Simulation der Schaltung mit pseudozufälligen Eingangsvektoren. Diese Möglichkeit wird realisiert durch die Eingabe von 10 000 pseudozufälligen Vektoren an den Eingängen $x_1, \cdots, x_m$ der Schaltung. Zwei Ausgabefunktionen $f_i(x), f_j(x)$ werden in zwei Schritten als orthogonales Paar erkannt. Die vorläufige Erkennung von Ausgangspaaren als Paare mit orthogonalen Ausgängen wird nach der Simulation von 10 000 pseudozufälligen Eingangsvektoren durchgeführt. Nach der Simulation werden die Ausgabewerte aller Paare entsprechend der folgenden Behauptung analysiert:

> „Kandidaten" für orthogonale Ausgangspaare können solche Ausgangspaare $f_i(x), f_j(x)$ sein, bei welchen bei Eingabe von 10 000 Eingangsvektoren die Funktionen $y_i(x), y_j(x)$ nicht gleichzeitig Eins sind. Allerdings ist die Abwesenheit eines Satzes $y_i(x) = 1, y_j(x) = 1$ an dem jeweiligen gefundenen Ausgangspaar kein 100%-iger Garant für die Orthogonalität der Funktionen $f_i(x), f_j(x)$.

Die bestimmten Kandidaten für Paare orthogonaler Ausgänge werden für die Ausführung des zweiten abschließenden Schritts zur Bestimmung der orthogonalen Ausgänge fixiert. Für den zweiten Schritt zur Bestimmung der Orthogonalität der Ausgänge wird die Konstruktion der Schaltung $f_{ij}(x)$ mit einem Ausgang OUT_{ij} realisiert, wobei $f_i(x), f_j(x)$ die Eingänge eines AND-Gatters mit zwei Eingängen darstellen. Die Schaltung $f_{ij}(x)$ wird unter Benutzung des Programmes SIS [75] nach dem einzigen Ausgang OUT_{ij} optimiert. Wenn die Fläche der Schaltung $f_{ij}(x)$ nach der Optimierung gleich Null ist, so werden die Funktionen $f_i(x), f_j(x)$ endgültig als orthogonal festgesetzt. Im Falle, dass die Realisationsfläche der Schaltung ungleich Null ist, werden die entsprechenden Ausgänge $f_i(x), f_j(x)$ als nicht orthogonal fixiert.

Die maximale Anzahl der paarweise orthogonalen Ausgänge wird bestimmt durch die Lösung eines graphentheoretischen Standardproblems. In dieser Arbeit wird für die Beschreibung des Problems die Graphentheorie verwendet, insbesondere der sogenannte Orthogonalitätsgraph G_0 der kombinatorischen Schaltung. Die Knoten des orthogonalen Graphen G_0 sind die funktionalen Ausgänge $y_1(x)$, $\cdots$, $y_n(x)$ der betrachteten kombinatorischen Schaltung. Zwei Knoten des orthogonalen Graphen y_i, y_j werden nur in dem Falle durch eine Kante verbunden, wenn die zugehörigen funktionalen Ausgänge $f_i(x),\ f_j(x)$ orthogonal sind.

Die maximale Untermenge paarweise orthogonaler Ausgänge entspricht dem maximal vollständigen Untergraph des orthogonalen Graphen G_0. Der maximal vollständige Untergraph ist in der Literatur ebenso bekannt als die *maximale Clique* des Graphen [86].

Definition 15:
Die maximale Clique des Graphen G_0 ist die maximale Knotenuntermenge V, in der jeder der Knoten aus V mit allen anderen Knoten der Untermenge V verbunden sind.

In der Abbildung 5.3 ist ein Beispielgraph mit 9 Knoten und der maximalen Clique der Größe vier, $\omega(G_0) = 4$, zu sehen.

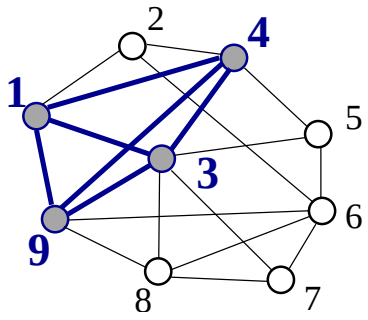

Abbildung 5.3: Orthogonalitätsgraph mit maximaler Clique $\omega(G_0) = 4$

Für sieben LGSynth-89 MCNC-Benchmark-Schaltungen wurden Experimente zur Bestimmung der orthogonalen Paare von Ausgängen und der Größe der maximalen Clique durchgeführt. In der Tabelle 5.2 werden die beschriebenen Parameter in den Spalten 2 und 3 dargestellt. Durch Analyse der experimentellen Ergebnisse lässt sich feststellen, dass die maximale Anzahl orthogonaler Ausgänge einer Clique nicht größer als 5 ist.

Circuits	inputs	outputs	Anzahl der orthogonalen Paare von Ausgängen	Größe der maximalen Clique
cu	14	11	27	3
ttt2	24	21	4	2
vda	17	39	11	4
cmb	16	3	2	1
comp	32	3	3	1
ldd	9	19	70	5
sct	19	15	3	2

Tabelle 5.2: Experimentelle Ergebnisse

5.4 Experimentelle Ergebnisse: Bestimmung der Gruppen orthogonaler Ausgänge

Die Erfüllung der Bedingungen des Theorems 2 kann mit Hilfe des folgenden heuristischen Algorithmus 5.2. verifiziert werden. Die Startdaten für den Algorithmus sind erneut die Werte der Ausgänge nach der Simulierung der Schaltung mit pseudozufälligen Vektoren.

Algorithmus 5.2:

Nach der Simulierung und dem Erhalt der Ausgabewerte an jedem der Ausgänge werden die noch nicht ausgewählten Ausgänge $y_1(x), \cdots, y_r(x)$ als orthogonal betrachtet. Dabei wird aus der Menge X_{ps} eine Menge $X_{otrh,1}, X_{otrh,1}, \cdots, X_{ps}$ von Eingangsvektoren gewählt, für die $y_{r+1}(x) \vee y_{r+2}(x) \vee y_{r+3}(x) \vee \cdots \vee y_n(x)$ gilt.

Für die ergänzten Ausgänge y_i mit $i \in \{1, 2, \cdots, r\}$ werden die Mengen

$$X_i^1 = \{x \in X_{ps} \setminus X_{orth,1},\ y_i(x) = 1\},$$
$$X_i^0 = \{x \in X_{ps} \setminus X_{orth,1},\ y_i(x) = 0\}$$

bestimmt, und es wird vorausgesetzt, dass die Anzahl der Elemente in diesen Mengen nicht kleiner 2 ist. Sei die Menge mit der kleinsten Elementeanzahl $X_{i_1}^{c_1}, c_1 \in \{0, 1\}$. Wenn mehrere Mengen die kleinste Elementeanzahl besitzen, dann wird eine beliebige aus ihnen ausgesucht.

Ein beliebiges Element $x_{i_1} \in X_{i_1}^{c_1}$ wird als ein Element von χ_{i_1} ausgewählt, die Menge $X_{i_1}^{c_1}$ von der Liste der betrachteten Mengen entfernt sowie das Element x_{i_1} aus allen verbleibenden Mengen gestrichen. Sei nun die Menge mit der geringsten Elementeanzahl die Menge $X_{i_2}^{c_2}, c_2 \in \{0, 1\}$. Es wird ein beliebiges Element $x_{i_2} \in X_{i_1}^{c_1}$, als ein Element von χ_{i_2} gewählt und die Menge $X_{i_2}^{c_2}$ aus der Liste der Mengen sowie das Element x_{i_2} aus den restlichen Mengen entfernt. Wenn es möglich ist, so fortzufahren, bis alle Elemente von $\chi_1, \chi_2, \cdots, \chi_r$ bestimmt sind, so sind die Bedingungen des Theorems 2 erfüllt und eine vollständig selbstprüfende Schaltung kann konstruiert werden. Nun soll dieser Algorithmus illustriert werden an einem Beispiel der Schaltung $H(x)$, welche in Tabelle 5.1 beschrieben ist.

Im ersten Schritt werden die orthogonalen Ausgänge von $H(x)$ für die Konstruktion eines selbstprüfenden Checkers nicht verwendet. Die Menge der Eingänge X_i^1, X_i^0, $i = 1, 2, 3, 4$ sind

$$X_1^1 = \{\ 0,\ 7\},\ \ X_1^0 = \{\ 1,\ 2,\ 3,\ 4,\ 5,\ 6\},$$
$$X_2^1 = \{\ 3,\ 4\},\ \ X_2^0 = \{\ 0,\ 1,\ 2,\ 5,\ 6,\ 7\},$$
$$X_3^1 = \{\ 5,\ 6\},\ \ X_3^0 = \{\ 0,\ 1,\ 2,\ 3,\ 4,\ 7\},$$
$$X_4^1 = \{\ 1,\ 2,\ 5,\ 6\},\ \ X_4^0 = \{\ 0,\ 3,\ 4,\ 7\}.$$

Die Mengen mit den kleinsten Eingangsanzahlen sind X_1^1, X_2^1 und X_3^1. Willkürlich wird nun der Eingang 0 von X_1^1 als ein Element von $\chi_1 = \{0^1\}$ gewählt. Der obere Index 1 von 0^1 kennzeichnet, dass $f_1(0) = 1$ ist. Die Eingangsmenge X_1^1 wird entfernt, und der Eingang 0 aus den übrigen Eingangsmengen gelöscht, das heißt, aus X_2^0, X_3^0 und X_4^0. Die entstehende geordneten Menge von Eingängen ist:

$$X_2^1 = \{\ 3,\ 4\},\ X_3^1 = \{\ 5,\ 6\},\ X_4^0 = \{\ 3,\ 4,\ 7\},$$
$$X_4^1 = \{\ 1,\ 2,\ 5,\ 6\},\ X_2^0 = \{\ 1,\ 2,\ 5,\ 6,\ 7\},$$
$$X_3^0 = \{\ 1,\ 2,\ 3,\ 4,\ 7\},\ X_1^0 = \{\ 1,\ 2,\ 3,\ 4,\ 5,\ 6\},$$

und bis hierhin sind die Mengen χ_j bestimmt als $\chi_1 = \{0^1\}$, $\chi_2 = \oslash$, $\chi_3 = \oslash$, $\chi_4 = \oslash$.

Im nächsten Schritt wird der Eingang 3 aus X_2^1 gewählt für χ_2 , $\chi_2 = \{3^1\}$, wobei der obere Index 1 in 3^1 wieder kennzeichnet, dass $f_2(3) = 1$ ist. X_2^1 wird aus der Liste der Eingangsmengen und der Eingang 3 aus all den Mengen, in denen er enthalten ist, entfernt, d.h. aus X_4^0, X_3^0 und X_1^0. Nach diesem zweiten Schritt ist

$$X_3^1 = \{\ 5,\ 6\}, X_4^0 = \{\ 4,\ 7\}, X_4^1 = \{\ 1,\ 2,\ 5,\ 6\},$$
$$X_3^0 = \{\ 1,\ 2,\ 4,\ 7\}, X_2^0 = \{\ 1,\ 2,\ 5,\ 6,\ 7\}, X_1^0 = \{\ 1,\ 2,\ 4,\ 5,\ 6\}$$

mit $\chi_1 = \{0^1\}$, $\chi_2 = \{3^1\}$, $\chi_3 = \oslash$, $\chi_4 = \oslash$, und es wird so fortgefahren, bis die Mengen $\chi_1, \chi_2, \chi_3, \chi_4$ vollständig bestimmt sind als $\chi_1 = \{0^1, 6^0\}$, $\chi_2 = \{3^1, 7^0\}$, $\chi_3 = \{5^1, 2^0\}$, $\chi_4 = \{1^1, 4^0\}$.

In Tabelle 5.3 ist die Wertetabelle für die Funtionen $f_1, \cdots, f_4$, $g_1, \cdots, g_4$, und $h_1, \cdots, h_4$ gegeben. Da $\chi_1 = \{0^1, 6^0\}$ mit $f_1(0) = 1$ und $f_1(6) = 0$, ist $g_1(0) = 0$ $g_1(6) = 0$. Für die anderen Funktionen ist $g_i(0) = f_i(0)$ und $g_i(6) = f_i(6)$ für $i = 2, 3, 4$, und folglich ist $h_1(0) = 1$, $h_2(0) = h_3(0) = h_4(0) = 0$, $h_1(6) = 1, h_2(6) = h_3(6) = h_4(6) = 0$. Mit $\chi_2 = \{3^1, 7^0\}$ und $f_2(3) = 1, f_2(7) = 0$

N	x_1	x_2	x_3	f_1	g_1	f_2	g_2	f_3	g_3	f_4	g_4	h_1	h_2	h_3	h_4
0	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0
1	0	0	1	0	0	0	0	0	0	1	0	0	0	0	1
2	0	1	0	0	0	0	0	0	1	1	1	0	0	1	0
3	0	1	1	0	0	1	0	0	0	0	0	0	1	0	0
4	1	0	0	0	0	1	1	0	0	0	1	0	0	0	1
5	1	0	1	0	0	0	0	1	0	1	1	0	0	1	0
6	1	1	0	0	1	0	0	1	1	1	1	1	0	0	0
7	1	1	1	1	1	0	1	0	0	0	0	0	1	0	0

Tabelle 5.3: Selbstprüfende Schaltung $H(x)$ unter Verwendung der Logischen Ergänzung g

sind die Funktionen $g_2(3)$ und $g_2(7)$ definiert als $g_2(3) = 0$, $g_2(7) = 1$. Für die anderen Funktionen ist $g_i(3) = f_i(3)$ und $g_i(7) = f_i(7)$ für $i = 1, 3, 4$, und daher $h_2(3) = 1$, $h_1(3) = h_3(3) = h_4(3) = 0$, $h_2(7) = 1, h_1(7) = h_3(7) = h_4(7) = 0$.

Auf ähnliche Weise werden die restlichen Reihen der Tabelle 5.3 bestimmt. Es ist leicht zu sehen, dass alle 1-aus-4 Codevektoren h_1, h_2, h_3, h_4 erzeugt werden. In jedem Spaltenpaar f_i, g_i, $i = 1, 2, 3, 4$ werden alle vier Paare von Ausgängen $\{0\ 0\}, \{0\ 1\}, \{1\ 0\}, \{1\ 1\}$ generiert, und deshalb werden die entsprechenden XOR-Gatter vollständig getestet durch komponentenweises XOR-addieren von $f_i \oplus g_i$. Mithilfe der Tabelle 5.3 werden die Komponenten der komplementären Schaltung g bestimmt als $g_1(x) = x_1x_2,\ g_2(x) = x_1(x_2x_3 \vee \overline{x}_2\overline{x}_3)$, $g_3(x) = x_2\overline{x}_3$, $g_4(x) = x_1\overline{x}_2 \vee x_2\overline{x}_3$.

Nun wird die Orthogonalität einiger der Ausgänge betrachtet. Aus der Tabelle 5.3 folgt, dass die folgenden Paare von Ausgängen von $H(x)$ orthogonal sind: f_1, f_2; f_1, f_3; f_1, f_4; f_2, f_3; f_2, f_4. Die maximal vollständigen Subgraphen des Orthogonalitäts-Graphen sind die beiden Graphen G^1 mit den Knoten f_1, f_2, f_3 und G^2 mit den Knoten f_1, f_2, f_4. Für diese vollständigen Subgraphen von orthogonalen Ausgangspaaren sind die Bedingungen aus Theorem 2 nicht erfüllt. Wenn f_1 und f_3 als orthogonale Ausgänge von $H(x)$ gewählt werden, welche nicht ergänzt werden, so sind die Bedingungen aus Theorem 2 erfüllt. Die Untermenge $\tilde{X}$ von Eingängen wird bestimmt als $\tilde{X} = \{0, 5, 6, 7\}$ und

es ist $X \setminus \tilde{X} = \{1,2,3,4\}$, $X_2^1 = \{3,4\}, X_2^0 = \{1,2\}, X_4^1 = \{1,2\}, X_4^0 = \{3,4\}$, und χ_2 und χ_4 können bestimmt werden als $\chi_2 = \{4^1, 1^0\}$, $\chi_4 = \{2^1, 3^0\}$. Die komplementäre Schaltung muss die beiden Booleschen Funktionen $g_2(x) = \overline{x}_1 x_3$ und $g_4(x) = \overline{x}_1 x_3 \vee (x_2 \oplus x_3)$ implementieren. Die Ausgänge f_1 und f_3 von $H(x)$ werden als orthogonale Ausgänge verwendet, und sie werden nicht ergänzt. Beide XOR-Elemente, welche f_2 und g_2 bzw. f_4 und g_4 miteinander XOR-addieren, werden vollständig getestet, somit ist die konstruierte Schaltung vollständig selbstprüfend.

Im allgemeinen Falle kann Algorithmus 5.2. zu Algorithmus 5.3. vereinfacht werden [87].

Algorithmus 5.3:

- Aus der Menge der möglichen Eingänge $A \subseteq X$ wird für jede Funktion $f_i(x)$, $i = 1, \ldots, n$ eine Untermenge $a_i = \{x_{i,a}^1, x_{i,a}^0\} \subset A$ mit $f_i(x_{i,a}^1) = 1$, $f_i(x_{i,a}^0) = 0$ und $a_i \cap a_j = \emptyset$ gewählt, wobei $i \neq j$.

- Für die Eingänge $x_{i,a}^1, x_{i,a}^0 \in a_i$ wird $h_i(x_{i,a}^1) = h_i(x_{i,a}^0) = 1$ und $h_j(x_{i,a}^1) = h_j(x_{i,a}^0) = 0$ gesetzt, wobei $i \neq j$. So sind die Werte $g_i(x_{i,a}^1)$ und $g_i(x_{i,a}^0)$ bestimmt durch:
  $g_i(x_{i,a}^1) = f_i(x_{i,a}^1) \oplus h_i(x_{i,a}^1)$
  und
  $g_i(x_{i,a}^0) = f_i(x_{i,a}^0) \oplus h_i(x_{i,a}^0)$.
  Das XOR-Element XOR_i wird unter dem Eingang $x_{i,a}^1$ mit $\{10\}$ und unter dem Eingang $x_{i,a}^0$ mit $\{01\}$ getestet.

- Nun wird für jede Funktion $f_i(x)$, $i = 1, \ldots, n$ eine zweite Menge $b_i = \{x_{i,b}^1, x_{i,b}^0\} \subset A$ bestimmt mit $f_i(x_{i,b}^1) = 1$ und $f_i(x_{i,b}^0) = 0$ und $a_i \cap b_i = \emptyset$.

- Es wird $h_i(x_{i,b}^1) = h_i(x_{i,b}^0) = 0$ gesetzt und daraus $g_i(x_{i,b}^1) = f_i(x_{i,b}^1) = 1$ und $g_i(x_{i,b}^0) = f_i(x_{i,b}^0) = 0$ geschlossen. Unter diesen Eingängen wird das XOR-Element XOR_i getestet mit $\{11\}$ und $\{00\}$.

- Schließlich werden die verbleibenen Werte der Funktionen $h_i(x)$ definiert, welche bisher nicht bestimmt wurden. Wenn für ein $x \in A$ ein Wert $i \in \{1, \ldots, n\}$ existiert mit $h_i(x) = 1$ für $j \neq i$, wird gesetzt. Dann ist $h_j(x) = 0$.

  Andererseits, wenn für ein $x \in A$ und $i \in \{1, \ldots, n\}$ entweder $h_i(x) = 0$ oder $h_j(x)$ undefiniert ist, so wird einer der noch nicht definierten Werte auf 1, die anderen auf 0 gesetzt, und $g_i(x)$ als $g_i(x) = f_i(x) \oplus h_i(x)$ bestimmt.

5.5 Vollständig selbstprüfender Entwurf

Im ersten Abschnitt dieses Kapitels wurde ein Beispiel einer kombinatorischen Schaltung mit vier Ausgängen gegeben, für welche nicht eine der Methoden der funktionalen Diagnostik die Konstruktion einer selbstprüfenden Fehlererkennungsschaltung zuläßt. In diesem Teil des Kapitels wird der Algorithmus an einem anderen Beispiel einer kombinatorischen Schaltung implementiert, einer Schaltung mit vier identischen Ausgangsfunktionen. Die Tabelle 5.4 beschreibt eine solche Schaltung [87].

Zunächst wird nun die Möglichkeit der Konstruktion einer selbstprüfenden Fehlererkennungsschaltung unter Verwendung traditioneller Methoden untersucht.

N	x_1	x_2	x_3	f_1	g_1	f_2	g_2	f_3	g_3	f_4	g_4	h_1	h_2	h_3	h_4
0	0	0	0	1	1	1	1	1	1	1	0	0	0	0	1
1	0	0	1	1	0	1	1	1	1	1	1	1	0	0	0
2	0	1	0	0	1	0	0	0	0	0	0	1	0	0	0
3	0	1	1	1	1	1	0	1	1	1	1	0	1	0	0
4	1	0	0	0	0	0	1	0	0	0	0	0	1	0	0
5	1	0	1	0	0	0	0	0	1	0	0	0	0	1	0
6	1	1	0	1	1	1	1	1	0	1	1	0	0	1	0
7	1	1	1	0	0	0	0	0	0	0	1	0	0	0	1

Tabelle 5.4: Kombinatorische Schaltung mit vier identischen Ausgängen

Wenn die Kontrolle der Schaltung auf Grundlage der Paritätsbitprüfung konstruiert wird, so werden für die Überprüfung der XOR-Elemente an ihren Eingängen nicht mehr als zwei Kombinationen generiert, was nicht ausreichend ist für einen vollständigen Test dieser Elemente.

Die Konstruktion der Kontrolle dieser Schaltung auf Grundlage der Methode „Verdopplung und Vergleich" führt nicht zu einer selbstprüfenden Schaltung. In dieser kombinatorischen Beispielschaltung treten an den Eingängen des Checkers nur zwei Eingangsvektoren der Form $\{0000, 0000\}$, $\{1111, 1111\}$ (Tabelle 5.5) auf, daher kann der Checker nicht selbsttestend und dementsprechend die gesamte Fehlererkennungsschaltung nicht selbstprüfend sein.

x_1	x_2	x_3	f_1	f_2	f_3	f_4	f'_1	f'_2	f'_3	f'_4
0	0	0	1	1	1	1	0	0	0	0
0	0	1	1	1	1	1	0	0	0	0
0	1	0	0	0	0	0	1	1	1	1
0	1	1	1	1	1	1	0	0	0	0
1	0	0	0	0	0	0	1	1	1	1
1	0	1	0	0	0	0	1	1	1	1
1	1	0	1	1	1	1	0	0	0	0
1	1	1	0	0	0	0	1	1	1	1

Tabelle 5.5: Nicht vollständig selbstprüfender Entwurf für eine Schaltung mit vier identischen Funktionen

Die Analyse der Verwendung systematischer Codes für die Konstruktion einer fehlererkennenden Schaltung aus der Beispielschaltung (insbesondere m-aus-v Codes) liefert ebenfalls ein negatives Resultat. Bei Verwendung des Berger Codes wird das Auftreten nur zweier Codewörter an den Eingängen des Berger Code Checkers gesichert. Bei Verwendung des 4-aus-8 Codes ist die Zahl der Wörter an den Eingängen des Checkers ebenfalls gleich zwei.

Auf diese Weise kann nicht mit einer einzigen der bekannten Methoden der funktionalen Diagnostik eine selbstprüfende Fehlererkennungsschaltung aus einer kombinatorischen Schaltung mit vier identischen Ausgangsfunktionen konstruiert werden.

Weiter wird die Implementierung des Algorithmus 5.3. am Beispiel der Schaltung mit vier iden-

tischen Ausgangsfunktionen aufgezeigt, und am Ergebnis wird bestimmt, ob es möglich ist, für eine solche Schaltung unter Verwendung der neuen Methode der Logischen Ergänzung eine selbstprüfende Fehlererkennungsschaltung zu konstruieren.

In der ersten Spalte der Tabelle 5.6 findet sich für jede binäre Eingangskombination (000, 001 , $\cdots$, 111) das dezimale Äquivalent (0, 1 , $\cdots$, 7). Die dezimale Bezeichnung wird im folgenden zur Verkürzung der Darstellung von Vektorenmengen verwendet, z.B. $a_1 = \{1, 2\}$ statt $a_1 = \{001, 010\}$. Gemäß dem Algorithmus 5.3. ist es erforderlich, die Menge der Eingangsvektoren X aus der Tabelle 5.6 in zwei Gruppen von Untermengen a_i, b_i zu teilen.

N	x_1	x_2	x_3	f_1	f_2	f_3	f_4	g_1	g_2	g_3	g_4	h_1	h_2	h_3	h_4
0	0	0	0	1	1	1	1	1	1	1	0	0	0	0	1
1	0	0	1	1	1	1	1	0	1	1	1	1	0	0	0
2	0	1	0	0	0	0	0	1	0	0	0	1	0	0	0
3	0	1	1	1	1	1	1	1	0	1	1	0	1	0	0
4	1	0	0	0	0	0	0	0	1	0	0	0	1	0	0
5	1	0	1	0	0	0	0	0	0	1	0	0	0	1	0
6	1	1	0	1	1	1	1	1	1	0	1	0	0	1	0
7	1	1	1	0	0	0	0	0	0	0	1	0	0	0	1

Tabelle 5.6: Vollständig selbstprüfender Entwurf für eine Schaltung mit vier identischen Funktionen

Solche Mengen sind:

$$a_1 = \{1, 2\},\ a_2 = \{3, 4\},\ a_3 = \{5, 6\},\ a_4 = \{0, 7\},\ b_1 = \{0, 4\},\ b_2 = \{0, 2\},\ b_3 = \{0, 2\},\ b_4 = \{1, 2\}.$$

Offensichtlich ist $a_i \cap a_j = \emptyset$ für $i \neq j$ und $a_i \cap b_i = \emptyset$.

Die Untermenge der Eingänge $a_1 = \{1, 2\}$ und $b_1 = \{0, 4\}$ impliziert $h_1(1) = h_1(2) = 1$ und $h_1(0) = h_1(4) = 0$. Die Untermenge der Eingänge $a_2 = \{3, 4\}$ und $b_2 = \{0, 2\}$ impliziert $h_2(3) = h_2(4) = 1$ und $h_2(0) = h_2(2) = 0$. Gleichermaßen wird aus $a_3 = \{5, 6\}$ und $b_3 = \{0, 2\}$ und $h_3(5) = h_3(6) = 1$ geschlossen sowie aus $a_4 = \{0, 7\}$ und $b_4 = \{1, 2\}$, $h_4(0) = h_4(7) = 1$ und $h_4(1) = h_4(2) = 0$. Die verbleibenden Werte von $h_i(x)$ sind mit 0 definiert.

Es ist leicht zu sehen, dass die vier Eingangskombinationen $\{00\}$, $\{01\}$, $\{10\}$, $\{11\}$ tatsächlich auf die XOR-Gatter $XOR_i,\ i = (1, 2, 3, 4)$ zur Anwendung kommen. Dies wird gezeigt für das XOR-Gatter XOR_1. Gemäß Tabelle 5.6 ist (entsprechend a_i) $\{f_1(1)\ g_1(1)\} = \{10\}$, $\{f_1(2)\ g_2(2)\} = \{01\}$ und (entsprechend b_i) $\{f_1(0)\ g_1(0)\} = \{11\}$, $\{f_1(4)\ g_1(4)\} = \{00\}$. Für die anderen XOR-Gatter erhält man das gleiche Resultat.

Um die vorgeschlagene Methode auf eine als Gatternetzliste gegebene Schaltung anzuwenden, kann die Schaltung für die Menge X_N von N pseudozufälligen Eingängen simuliert werden. Für jeden Schaltungsausgang $f_i(x)$ wird die Menge geteilt in die Untermengen
$X^1_{N,i} = \{x \in X_N / f_i(x) = 1\}$
und
$X^0_{N,i} = \{x \in X_N / f_i(x) = 0\}$.
Dann können die Untermengen a_i und b_i bezüglich dieser Eingangsuntermengen bestimmt werden.

$Circuit \setminus Outputs$	1	2	3	..	7	..	11	..
frg1 N(0)	3907	1546	5578					
N(1)	6093	8454	4422					
x2 N(0)	1222	2532	8716		3130			
N(1)	8778	7468	1284		6870			
cu N(0)	1259	8741	9922		9629		8111	
N(1)	8741	125	78		371		1889	
sct N(0)	4992	3731	8374		1578		9027	
N(1)	5008	6269	1626		8422		973	
tcon N(0)	5034	4915	5021		5010		5008	
N(1)	4966	5085	4979		4990		4992	
ldd N(0)	4236	916	4132		8749		9422	
N(1)	5764	6084	5868		1251		578	
ttt2 N(0)	7565	6246	7837		7540		7518	
N(1)	2435	3754	2163		2460		2482	
x1 N(0)	3409	6989	6329		5233		1346	
N(1)	6591	3011	3671		4767		8654	
cht N(0)	7449	7547	7483		7542		7516	
N(1)	2551	2453	2517		2458		2484	

Tabelle 5.7: Experimentelle Ergebnisse für LGSynth-89 MCNC-Benchmark-Schaltungen (Teil 1)

$Circuit \setminus Outputs$	15	16	..	19	..	21	..	35	36
frg1 N(0)									
N(1)									
x2 N(0)									
N(1)									
cu N(0)									
N(1)									
sct N(0)	7437								
N(1)	2563								
tcon N(0)	4923	4976							
N(1)	5077	5024							
ldd N(0)	6943	8404		9397					
N(1)	3057	1596		603					
ttt2 N(0)	7481	7519		7434		7498			
N(1)	2519	2481		2566		2502			
x1 N(0)	8593	5070		8736		5012		142	
N(1)	1407	4930		1264		4988		9858	
cht N(0)	7508	7561		7499		7495		7528	7552
N(1)	2492	2439		2501		2505		2472	2448

Tabelle 5.8: Experimentelle Ergebnisse für LGSynth-89 MCNC-Benchmark-Schaltungen (Teil 2)

Die Eingangsuntermengen a_i und b_i können leicht bestimmt werden, wenn die Anzahl $N_i(1)$ von Eingängen $x \in A$, für welche der Ausgang $f_i(x) = 1$ ist, bzw. wenn die Anzahl $N_i(0)$ von Eingängen, für welche die Ausgänge $f_i(x) = 0$ sind, größer oder gleich $2i$ ist.

Für die LGSynth-89 Benchmark-Schaltungen wurden für 10 000 pseudozufällige Eingänge diese Anzahlen $N_i(1)$ und $N_i(0)$ experimentell bestimmt. Die erhaltenen Ergebnisse sind in Tabelle 5.7 und Tabelle 5.8 dargestellt. Neben dem Namen und der Anzahl von Ausgängen der Schaltungen sind für die Ausgänge 1, $\cdots$, 36 die Werte $N_i(1)$ und $N_i(0)$ gegeben. Wie aus Tabellen entnommen werden kann, werden für alle betrachteten MCNC-Benchmark-Schaltungen unter Verwendung von komplementären Schaltungen vollständig selbstprüfende Schaltungen erhalten.

6 Zusammenfassung und Future Work

Die Bedeutung der vorgestellten Arbeit ist angesichts des gegenwärtigen Fortschritts digitaler Systeme offensichtlich. Die unvermeidliche Erhöhung der Anzahl der Transistoren und ihrer Dichte auf dem Kristall, d.h. die Steigerung der Komplexität digitaler Systeme, erfordert eine hohe Fehlertoleranz in den Schaltungen. Gleichzeitig müssen aus ökonomische Sicht die Kosten bei Entwicklung, Verifikation und Test neuer digitaler Systeme möglichst gering gehalten werden.

Diese Arbeit stellt eine neue konkurrenzfähige Theorie einer Methode der Konstruktion vollständig selbstprüfender Schaltungen unter Verwendung einer komplementären Logik vor, genannt *„Logische Ergänzung"*. Es wurde ein neues Prinzip zur On-line Fehlererkennung entwickelt. Diese Methode garantiert eine hohe Fehlerüberdeckung in digitalen Systemen bei geringem Flächeverbrauch. Die Werte der experimentellen Ergebnisse im Vergleich zu den Ergebnissen, welche bei Verwendung der traditionellen Methoden der funktionalen Diagnostik erhalten wurden, sind ein gutes Kriterium, um die Bedeutung dieser Methode in der Praxis zu verstehen.

In dieser Dissertation wurden vier wesentliche Aufgaben bearbeitet:

1. Die Optimierung der komplementären Schaltung $g(x)$.

   Bei der Optimierung der komplementären Schaltung wurde versucht, die Freiheitsgrade bei ihrem Entwurf möglichst optimal auszunutzen.

   Der Flächenaufwand der komplementären Schaltung hängt ganz wesentlich davon ab, welche der Ausgänge der zu überwachenden Schaltung gemeinsam zu einem Codewort des betrachteten Codes komplementiert werden.

   Im Rahmen dieser Arbeit wurden Algorithmen zur Bestimmung der Gruppen von Ausgängen entwickelt. Dabei basiert die Wahl konkreter Ausgänge auf der Analyse der internen Struktur der jeweiligen Schaltung.

2. Im Rahmen der internationalen Zusammenarbeit wurde bei der Entwicklung der neuen Methode der Logischen Ergänzung ein neuer 1-aus-3 Code Checker mit konkurrenzfähigem Flächebedarf entworfen.

3. Es wurden Prinzipien zur Implementation von *Don't-Cares* in der Struktur der neuen Methode unter Verwendung des 1-aus-3 Codes vorgestellt. Es wurde der Versuch unternommen, *Don't-Care* Zustände zur Erreichung eines hohen Optimierungsgrades der komplementären Logik einzusetzen.

4. Die Bestimmung notwendiger und hinreichender Bedingungen für die Existenz vollständig selbstprüfender Schaltungen für eine vorgegebene Schaltung f auf der Basis komplementärer Schaltungen für einen 1-aus-n Code.

Die wesentlichen Ergebnisse der Dissertation sind:

1. Das neue Verfahren ist bezüglich Aufwand und Fehlererkennung vergleichbar mit traditionellen Fehlererkennungsverfahren. Mit dem neuen Verfahren ist es möglich, selbstprüfende Fehlererkennungsschaltungen zu konstruieren, die eine Fehlerüberdeckung von 96% besitzen bei nur 80% Mehraufwand vom Flächebedarf der funktionalen Schaltung.

2. Da sich der erhöhte Flächenaufwand im Vergleich zum Paritätscode wesentlich durch den erforderlichen selbstprüfenden Checker ergibt, wurde ein neuer Code Checker für einen 1-aus-3 Code und für andere Fehlererkennungscodes entwickelt.

3. Für Fehlererkennungsschaltungen auf der Grundlage komplementärer Schaltungen konnten erstmals notwendige und hinreichende Bedingungen für die Existenz einer vollständig selbstprüfenden Fehlererkennungsschaltungen bewiesen werden. Dabei basieren die Fehlererkennungsschaltungen auf dem Prinzip der komplementären Ergänzung für einen 1-aus-n Code. Für traditionelle Fehlererkennungsschaltungen sind derartige Ergebnisse nicht bekannt.

Teile dieser Arbeit sind in internationalen Fachzeitschriften und *Conference Proceedings* veröffentlicht. Diese Publikationen sind im folgenden aufgelistet:

V. Saposhnikov, Vl. Saposhnikov, G. Osadtchi, A. Morozov, M. Gössel: *„Design of Totally Self-Checking Combinational Circuits by Use of Complementary Circuits"*, Proc. of the East-West Design & Test Workshop (EWDTW'04), pp. 30-34, Alushta-Yalta (Ukraine), July-Sept. 2004.

V. Saposhnikov, Vl. Saposhnikov M., A. Morozov, M. Gössel: *„Necessary and Sufficient Conditions for the Existence of Totally Self-Checking Circuits"*, 10th IEEE International On-Line Testing Symposium (IOLTS'04), pp. 25-31, Madeira (Portugal), Juli 2004.

A. Morozov, M. Gössel, V. V. Saposhnikov, Vl. V. Saposhnikov: *„Complementary Circuits for On-Line Detection for 1-aus-3 Codes"*, 17th International Conference on Architecture of Computing Systems (ARCS'04), Organic and Pervasive Computing, pp. 76-83, Augsburg (Germany), 2004.

Vl.V Saposhnikov, V.V. Saposhnikov, A. Morozov, M. Gössel: *„Necessary and Sufficient Conditions for the Existence of Self-Checking Circuits by Use of Complementary Circuits"*, Institute of Computer Science, University of Potsdam, Technical Report, Preprint April/2004, ISSN 0946-7580, 2004.

M. Gössel, A. Morozov, V. V. Saposhnikov, Vl. V. Saposhnikov: *„Logic Complement, a New Method of Checking the Combinational Circuits"*, Automation and Remote Control, Kluwer Academic Publishers, N. 64, Vol. 1, USA, pp. 153-161, January, 2003.

V. V. Saposhnikov, Vl. V. Saposhnikov, A. Morozov, M. Gössel: *„Logisches Komplement, eine neue Methode zur Überwachung kombinatorischer Schaltungen"* (Russ.), Avtomatika i Telemechanika, Number 1, pp.169-178, 2003.

V. V. Saposhnikov, Vl. V. Saposhnikov, A.V. Dmitriev, A. Morozov, M. Gössel: *„On-Line Fehlererkennung kombinatorischer Schaltungen durch komplementäre Schaltungen. Elektronische Modellierung"* (Russ.), Avtomatika & Telemechanika, Vol.24, Number.6, pp.79-94, 2002.

V. V. Saposhnikov, A. Morozov, Vl. V. Saposhnikov, M. Gössel: *„Concurrent Checking By Use of Complementary Circuits for 1-out-of-3 Codes"*, Proc. 5th IEEE Design aund Diagnostics of Electronic Circuit & Systems, 5th International Workshop on IEEE Design and Diagnostics of Electronic Circuits and Systems (DDECS'02), pp. 404-407, 2002.

A. Morozov, V. V. Saposhnikov, Vl. V. Saposhnikov, M. Gössel: *„New Self-Checking Circuits by Use of Berger-Codes"*, Proc. 6th IEEE On-Line Testing Workshop, Palma de Mallorca (Spain), pp. 141-146, 2000.

6.1 Future Work

Insgesamt muss aber eingeschätzt werden, dass das neue Fehlererkennungsverfahren auf der Basis komplementärer Schaltungen nicht den Durchbruch bezüglich Aufwand und Fehlererkennung gebracht hat, wie wir es zu Beginn des Projektes erhofft hatten. Das Ziel war es, tatsächlich *Low-Cost* Fehlererkennungsschaltungen für *Random Logic* zu entwerfen, die einen Zusatzaufwand von ca 10% bis 15% erfordern.

Allerdings ist es in jedem Fall möglich, die erhaltenen Ergebnisse des experimentellen Teils dieser Dissertation weiterhin zu verbessern. Dabei ist es vermutlich von Vorteil, besonderes Augenmerk auf die Weiterentwicklung des Algorithmus zur Implementierung und Verwendung von *Don't-Cares* zu richten. So kann die Modifikation der vorgestellten Algorithmen zur Auswahl und Fixierung der Gruppen von Ausgängen zur weiteren Optimierung der Schaltungen beitragen.

Die zukünftige Arbeit kann die folgenden Methoden zum Erhalt der kleinsten Realisationsfläche der komplementären Logik bei unbedingter Sicherstellung einer hohen Fehlererkennungswahrscheinlichkeit untersuchen.

1. Die Verwendung anderer Codes in der Struktur der neuen Methode der Logischen Ergänzung.

2. Gleichzeitige Verwendung verschiedener Fehlererkennungscodes. Dabei kann die Wahl eines Codes für eine konkrete Gruppe von Ausgängen auf der Häufigkeit der Generierung von Codewörtern des gewählten Codes gegründet sein.

3. Implementation der neuen Methode in typischen digitalen Geräten (Addierer, Register, Zähler, Codierer/Decodierer) und Durchführung von Experimenten.

4. Verbesserung der Parameter der Fehlererkennung in Systemen, welche unter Verwendung der neuen Methode konstruiert wurden, auf Kosten der Fehlererkennungszeit.

5. Die Ausarbeitung der Theorie der funktionalen Diagnostik mit dem Ziel, dass nur jene Fehler erkannt werden, welche zum Auftreten falscher Signale an den Ausgängen der Kontrollschaltung führen.

6. Entwicklung von Fehlererkennungsschaltungen mit gemeinsamer Realisierung der Originalschaltung und der komplementären Logik.

Literaturverzeichnis

[1] G. E. Moore, “Cramming more components onto integrated circuit,” *Electronic Magazine*, vol. 16, pp. 114–117, April 1965.

[2] Y. Zorian, “Testing the Monster Chip,” *IEEE Spectrum*, vol. 36, pp. 54–60, July 1999.

[3] M. Nicolaidis, “Scaling deeper to submicron: On-line Testing to the Rescue,” in *FTCS-28, The Twenty-Eigth Annual International Symposium on Fault-Tolerant Computing*, pp. 299–301, 1998.

[4] P. K. Lala, *Self-Checking and Fault-Tolerant Digital Design*. Morgan Kaufmann Publishers, 2001.

[5] M. Gössel and S. Graff, *Error Detection Circuits*. McGraw-Hill Book Company, McGRAW-Hill International Lim., 1993.

[6] A. Morozov, V. Saposhnikov, V. Saposhnikov, and M. Gössel, “New Self-checking Circuits by Use of Berger-codes,” in *Proc. 6th IEEE International On-Line Testing Workshop*, July 2000.

[7] D. K. Pradhan, *Fault Tolerant Computing: Theory and Techniques*, vol. 1, ch. 5. Englewood Cliffs, NJ: Prentice-Hill, 1986.

[8] D. Bochmann and B. Steinbach, *Logikentwurf mit XBOOLE*. Verlag Technik GmbH Berlin, 1991.

[9] M. Gössel and F. Börner, *Grundlagen digitaler Systeme*. Institut für Informatik, Universität Potsdam, 2001.

[10] A. Morosov, *Entwurf von selbstprüfenden Schaltungen mit monoton unabhängigen Ausgängen*. PhD thesis, Universität Potsdam, November 1997.

[11] W. Moschanin, *Entwurf selbstdualer digitaler Schaltungen zur Fehlererkennung*. PhD thesis, Universität Potsdam, Dezember 1999.

[12] A. Dmitriev, *Entwurf alternierender Signale zum Entwurf von Fehlererkennungsschaltungen und Kompaktoren*. PhD thesis, Universität Potsdam, November 2003.

[13] D. A. Reynolds and G. Metze, “Fault Detection Capabilities of Alternating Logic,” *IEEE Transactions on Computers*, pp. 1093–1098, 1978.

[14] V. V. Saposhnikov, V. V. Saposhnikov, and M. Gössel, *Self-Dual Discrete Devices (russ.)*. St. Petersburg, Energoatomizdat, 2001.

[15] H. J. Zander, *Logischer Entwurf binärer Systeme*. Verlag Technik, 1989.

[16] A. Steininger, “Testing and Built-in-Self-Test - A Survey,” *Journal of Systems Architecture, Elsevier Science Publishers B.V., North Holl*, vol. 46, no. 4, pp. 721–747, 2000.

[17] A. Steininger and C. Scherrer, "On the Necessity of On-Line BIST in Safety-Critical Applications - A Case Study," in *Proc. of the 29th Annual International Symposium on Fault Tolerant Computing (FTCS-29)*, pp. 208 – 215, 1999.

[18] C. Galke, M. Grabow, and H. Vierhaus, "Perspectives of Combining on-line and off-line Test Technology for Dependable Systems on a Chip," in *Proc. Int. On-Line Testing Symposium*, pp. 183–187, July 2003.

[19] M. Pflanz, *On-line Error Detection and Fast Recover Techniques for Dependable Embedded Processors*. Lecture Notes in Computer Science, Springer-Verlag New York, Inc., 2002.

[20] K. Echtle, *Fehlertoleranzverfahren*. Springer-Verlag, 1990.

[21] M. A. Marouf and A. D. Friedman, "Efficient Design of Self-Checking Checker for any m-out-of-n Code," *IEEE Transactionsactions on Computer-Aided Design*, vol. C-27, no. 6, pp. 482–490, 1978.

[22] D. K. Pradhan and J. J. Stiffler, "Error Correcting Codes and Self-Checking Circuits," *IEEE Computer*, pp. 27–37, March 1980.

[23] J.-C. Lo, S. Thanawastien, T. R. N. Rao, and M. Nicolaidis, "An SFS Berger Check Prediction ALU and Its Application to Self-Checking Processor Designs," *IEEE Transactions on Computer-Aided Design*, vol. 11, pp. 525–540, April 1992.

[24] M. Pflanz, K. Walther, and H. T. Vierhaus, "On-line Error Detection Techniques for Dependable Embedded Processors with High Complexity," in *Proc. of International Test Workshop (IOLTW'01)*, pp. 51–53, September 2001.

[25] E. S. Sogomonyan and E. V. Slabakov, *Self-Checking Circuits and Faults-Tolerant Systems (russ.)*. Moscow: Radio i Svyaz, 1989.

[26] J. M. Berger, "A Note on Error Detection Codes for assymmetric Channels," *Informatation and Control*, vol. 4, pp. 68–73, 1961.

[27] H. Dong, "Modified Berger codes for Detection of unidirectional Errors," *IEEE Transactions on Computers*, vol. C-22, pp. 572–572, 1984.

[28] W. C. Carter and P. R. Schneider, "Design of Dynamically Checked Computers," in *Proc. of IFIP Congress*, vol. 2, pp. 878–883, 1968.

[29] M. Abramovici, M. A. Breuer, and A. D. Friedman, *Digital System Testing and Testable Design*. IEEE Press, New York: W. H. Freeman and Company, 1990.

[30] V. P. Nelson, H. T. Nagle, B. D. Carroll, and J. D. Irwin, *Digital Logic Circuit Analysis & Design*. Englewood Cliffs, Prentice-Hall Inc., 1995.

[31] D. A. Anderson and G. Metze, "Design of totally self-checking circuits for m-out-of-n codes," *IEEE Transactions on Computer-Aided Design*, vol. C-22, pp. 263–269, 1973.

[32] D. L. Tao, R. P. Hartmann, and P. K. Lala, “A general Technique for Designing Totally Self-Checking Checker for 1-out-of-n Code with Minimum Gate Delay,” *IEEE Transactions on Computer-Aided Design*, vol. 41, pp. 881–886, July 1992.

[33] T. Haniotakis, A. Paschalis, and D. Nikolos, “Efficient Totally Self-Checking Checkers for a Class of Borden Codes,” *IEEE Transactions on Computer-Aided Design*, vol. 44, pp. 1318–1322, November 1995.

[34] S. W. Burns and N. K. Jha, “A totally Self-Checking Checker for a Parallel Unordered Coding Scheme,” *IEEE Transactions on Computer-Aided Design*, vol. 43, pp. 490–495, April 1994.

[35] V. V. Saposhnikov and V. V. Saposhnikov, *Self-Checking Discrete Devices (in Russian)*. St. Petersburg, Energoatomizdat, 1992.

[36] S. Tarnick, *Data Compression Techniques for Concurrent Error Detection and Built-In Self Test*. PhD thesis, Universität Potsdam, 1995.

[37] G. P. Aksenova, “Necessary and Sufficient Conditions for Sythesis of Completary Testable Modulo-2 Convolution Circuits,” *Automation and Remote Control*, vol. 40, no. 9, pp. 1362–1268, 1979.

[38] E. Fujiwara and K. Matsuoka, “A Self-Checking generalized Prediction Checker and Its Use for Built-In Testing,” *IEEE Transactions on Computer-Aided Design*, vol. C-36, pp. 86–93, January 1987.

[39] J. Kharbaz and E. J. McCluskey, “Self-Testing Embedded Parity Checkers,” *IEEE Transactions on Computer-Aided Design*, vol. C-33, pp. 753–756, August 1984.

[40] M. J. Ashjaee and S. M. Reddy, “On totally Self-checking Checkers for Separable Codes,” *IEEE Transactions on Computer-Aided Design*, vol. C-26, pp. 737–744, August 1977.

[41] T. R. Rao, G. L. Feng, M. S. Kolluru, and J.-C. Lo, “Novel Totally Self-Checking Berger Code Checker Designs Based on Generalized Berger Code Partitioning,” *IEEE Transactions on Computers*, vol. 42, no. 8, pp. 1020–1024, 1993.

[42] T. R. Rao, G. L. Feng, M. S. Kolluru, and J.-C. Lo, “Correction to Novel Totally Self-Checking Berger Code Checker Designs Based on Generalized Berger Code Partitioning,” *IEEE Transactions on Computers*, vol. 43, no. 5, p. 640, 1994.

[43] S. J. Piestrak, “Design of Fast Self-Testing Checkers of a Class of Berger Codes,” *IEEE Transactions on Computer-Aided Design*, vol. C-36, pp. 629–634, May 1987.

[44] A. G. Melnikov, V. V. Saposhnikov, and V. V. Saposhnikov, “Die Konstruktion selbstprüfender Checkers für Codes mit Addition (russ.),” *Problemy peredachi informacii*, vol. 22, no. 2, pp. 85–97, 1986.

[45] C. Halatsis, N. Gaitanis, and M. Sigala, “A new Design Method for m-out-of-n Checkers,” *IEEE Transactions on Computers*, vol. 32, pp. 273–283, March 1983.

[46] S. J. Piestrak, "Design of High-Speed and Coast Effective Self-Testing Checkers for Low-Cost Arithmetic Codes," *IEEE Transactions on Computer-Aided Design*, vol. 39, pp. 360–374, March 1990.

[47] C. Halatsis, N. Gaitanis, and M. Sigala, "Fast and Efficient Totally Self-Checking Checkers for m-out-of-(2m+1) Codes," *IEEE Transactions on Computer-Aided Design*, vol. C-32, pp. 507–511, May 1988.

[48] A. M. Paschalis, D. Nikolos, and C. Halatsis, "Efficient Modular Design of TSC Checkers for M-ut-of-2M Codes," *IEEE Transactions on Computer-Aided Design*, vol. 37, pp. 301–309, March 1988.

[49] P. K. Lala, F. Busaba, and M. Zhao, "Transistor-Level Implementation of Totally Self-Checking Checkers for a Subset m-out-2m Codes," in *Proc. of 2nd IEEE International On-Line Testing Workshop*, pp. 124–131, July 1996.

[50] V. V. Dimakopoulos, G. Soutziotis, A. Paschalis, and D. Nikolos, "On TSC Checkers for m-out-n Codes," *IEEE Transactions on Computer-Aided Design*, vol. 44, pp. 1055–1059, August 1995.

[51] C. Metra, M. Favalli, and B. Ricco, "Highly Testable and Compact Single Output Comparator," in *Proc. of 15th IEEE VLSI Test Symposium (VTS'97)*, pp. 210–215, April 1997.

[52] S. M. Reddy, "A Note on Self-Checking Checkers," *IEEE Transactions on Computers*, vol. 23, pp. 1100–1102, October 1974.

[53] P. Golan, "Design of totally self-checking checker for 1-out-of-3 code," *IEEE Transactions on Computers*, vol. C-33, no. 3, p. 285, 1984.

[54] R. David, "Totally self-checking 1-out-of-3 code checker," *IEEE Transactions on Computers*, vol. C-27, pp. 570–572, 1978.

[55] D. L. Tao, P. K. Lala, and R. P. Hartmann, "A MOS Implementation for totally Self-Checking Checker for 1-out-of-3 Code," *IEEE Transaction on Computers*, vol. 23, pp. 857–877, June 1988.

[56] J. C. Lo and S. Tawanastein, "On the design of combinational totally self-checking 1-out-of-3 code checkers," *IEEE Transactions on Computers*, vol. C-39, no. 3, pp. 387–393, 1990.

[57] C. Metra, M. Favalli, P. Olivo, and B. Ricco, "A Highly Testable 1-out-of-3 CMOS Checkers," *Proc. of IEEE International Workshop on Defect and Fault Tolerance in VLSI Systems*, pp. 279–286, 1993.

[58] T. Haniotakis, Y. Tsiatouhas, C. Efstathiou, and D. Nikolos, "Novel domino-cmos strongly code disjoint and strongly fault secure 1-out-of-3 and 2-out-of-3 code checkers," in *Proc. of 5th IEEE International On-Line Testing Workshop*, pp. 174–178, July 1999.

[59] C. Metra, M. Favalli, and B. Ricco, "Design of TSC Checkers for any 1-out-of-n Code," *Journal of Microelectronics System Integration*, vol. 3, no. 2, pp. 81–91, 1995.

[60] C. Metra, M. Favalli, and B. Ricco, "Embedded 1-out-of-3 Checkers with On-Line Testing Ability," in *Proc. of 2nd IEEE International On-Line Testing Workshop*, pp. 136–141, 1996.

[61] J. Q. Wnag and P. K. Lala, "Partionally Strongly Fault Secure and Partionally Strongly Code Disjoint 1-out-of-3 Checker," *IEEE Transactions on Computers*, vol. 43, pp. 1238–1240, October 1994.

[62] M. Nikolaidis, I. Jansch, and B. Courtois, "Strongly code disjoint checkers," *Proc. of 14th International Symp. Fault-Tolerant Computing*, pp. 16–21, 1984.

[63] V. V. Saposhnikov, A. Morozov, V. V. Saposhnikov, and M. Gössel, "Concurrent Checking By Use of Complementary Circuits for 1-out-of-3 Codes," *Proc. of IEEE DDECS 2002*, April Brno, Chech Republic, 2002.

[64] A. M. Paschalis, C. Efstathiou, and C. Halatsis, "An efficient TSC 1-out-of-3 Code Checker," *IEEE Transactions on Computers*, vol. 23, no. 3, pp. 875–877, 1988.

[65] V. V. Saposhnikov, V. V. Saposhnikov, A. Morozov, and M. Gössel, "Logisches Komplement, eine neue Methode zur Überwachung kombinatorischer Schaltungen (russ.)," *Avtomatika & Telemechanika*, pp. 169–178, 2003.

[66] M. Gössel, A. Morozov, V. V. Saposhnikov, and V. V. Saposhnikov, "Logic Complement, a New Method of Checking the Combinational Circuits," *Automation and Remote Control, Kluwer Academic Publishers (USA)*, vol. 1, no. 64, pp. 153–161, 2003.

[67] V. Saposhnikov and V. Saposhnikov, "Selbstprüfender Komparator mit einem zusätzlichen periodischen Input (Rus)," *Avtomatika & Telemechanika*, vol. 1, no. 6, pp. 200–208, 1997.

[68] A. Steininger and J. Vilanek, "Using Offline and Online BIST to Improve System Dependability - The TTPC-C Example," in *Proc. of International Conference on Computer Design (ICCD2002)*, pp. 277–243, September 2002.

[69] M. Nicolaidis, "Self-Exercising Checkers for Unified Built-In-Self-Test (ubist)," *IEEE Transactions on Computer-Aided Design*, vol. 8, pp. 203–218, March 1989.

[70] S. Kundu, E. S. Sogomonyan, M. Gössel, and S. Tarnick, "Self-Checking Comparator with One Periodic Output," *IEEE Transactions on Computers*, vol. 3, no. 45, pp. 379–380, 1996.

[71] V. Saposhnikov, A. Morozov, V. Saposhnikov, and M. Gössel, "On-line Fehlererkennung kombinatorischer Schaltungen durch komplementäre Schaltungen," *Elektroniesche Modelierung*, vol. 24, pp. 79–94, 2002.

[72] W.-F. Chang and C.-W. Wu, "TSC Berger-Code Checker Design for 2^{r-1}-Bit Information," *Journal of Information Science and Engineering*, vol. 15, pp. 429–441, 1999.

[73] V. Saposhnikov, A. Morozov, V. Saposhnikov, and M. Gössel, "A new Design Method for Self-Checking Unidirectional Combinational Circuits," *Journal of Electronic Testing, Theory and Application*, vol. 12, pp. 41–53, 1998.

[74] K. De, C. Natarjan, D. Nair, and P. Banerjee, "RSYN: A System for Automated Synthesis of Reliable Multilevel Circuits," *IEEE Transactions on VLSI Systems*, pp. 186–195, Juny 1994.

[75] E. Sentovich, K. Singh, I. Lavagno, A. S. C. Moon, H. Savoi, P. Stephan, R. Brayton, and A. Sangiovanni-Vencentelli, "A System for Sequential Circuit Synthesis," Technical Report UCB/ERL M92/41, Berkeley, University of California, 1992.

[76] IWLS'89, "Test benchmark suite," in *International Workshop on Logic Synthesis*, 1989. available from http://www.cbl.ncsu.edu/pub/Benchmark_dirs/LGSynth89/.

[77] A. Morozov, M. Gössel, V. V. Saposhnikov, and V. V. Saposhnikov, "Complementary circuits for on-line detection for 1-out-of-3 codes," *Proc. of 17th International Conference on Architecture of Computing Systems (ARCS'04), Organic and Pervasive Computing*, pp. 76–83, 2004.

[78] D. Brand, "Redundancy and Don't Cares in Logic Synthesis," *IEEE Transactions on Computers*, vol. C-32, pp. 947–952, October 1983.

[79] R. K. Brayton, R. Rudell, A. Sangiovanni-Vincetelli, and A. R. Wang, "MIS: Multiple-Level Logic Optimization System," *IEEE Transactions on Computer-Aided Design*, vol. CAD-6, pp. 1062–1081, Nov. 1987.

[80] G. Hachtel and F. Somenzi, *Logic Synthesis and Verification Algorithms*. Kluwer Academic Publishers, 1996.

[81] D. Brand, R. A. Bergamaschi, and L. Stok, "Don't Cares in Synthesis: Theoretical Pitfalls and Practical Solutions," tech. rep., IBM Research Division, T. J. Watson Research Center, Yorktown Heights, New York, 1995.

[82] I. Wegener, *The complexity of Boolean functions*. U.K.: Wiley, 1987.

[83] V. Saposhnikov, V. Saposhnikov, A. Morozov, and M. Gössel, "Necessary and Sufficient Conditions for the Existence of Totally Self-Checking Circuits," *Proc. of 10th International On-Line Testing Symposium*, pp. 100–106, 2004.

[84] I. S. Vizirev, "Design of Totally Self-checking checkers for Constant-weight Codes," *Automation and Remote Control*, vol. 16, no. 1, pp. 34–40, 1982.

[85] V. V. Saposhnikov, V. V. Saposhnikov, A. Morozov, and M. Gössel, "Necessary and Sufficient Conditions for the Existence of Self-Checking Circuits by Use of Complementary Circuits," tech. rep., University of Potsdam, 2004.

[86] W. J. Cook, W. H. Cunningham, W. R. Pulleyblank, and A. Schrijver, *Combinatorial Optimization*. Wiley-Interscience Series in Discrete Mathematics and Optimization, ISBN 047155894X, 1997.

[87] V. V. Saposhnikov, V. V. Saposhnikov, G. Osadtchi, A. Morozov, and M. Gössel, "Design of Totally Self-Checking Combinational Circuits by Use of Complementary Circuits," *Proc. of the East-West Design & Test Workshop (EWDTW 2004)*, pp. 30–34, 2004.